ABSTRACT

Title of Dissertation: Improving Data Delivery in Wide Area and Mobile Environments

Laura Bright, Doctor of Philosophy, 2003

Dissertation directed by: Professor Louiqa Raschid
Department of Computer Science

The popularity of the Internet has dramatically increased the diversity of clients and applications that access data across wide area networks and mobile environments. Data delivery in these environments presents several challenges. First, applications often have diverse requirements with respect to the latency of their requests and recency of data. Traditional data delivery architectures do not provide interfaces to express these requirements. Second, it is difficult to accurately estimate when objects are updated. Existing solutions either require servers to notify clients (push-based), which adds overhead at servers and may not scale, or require clients to contact servers (pull-based), which rely on estimates that are often inaccurate in practice. Third, cache managers need a flexible and scalable way to determine if an object in the cache meets a client's latency and recency preferences. Finally, mobile clients who access data on wireless networks

share limited wireless bandwidth and typically have different QoS requirements for different applications.

In this dissertation we address these challenges using two complementary techniques, client profiles and server cooperation. Client profiles are a set of parameters that enable clients to communicate application-specific latency and recency preferences to caches and wireless base stations. Profiles are used by cache managers to determine whether to deliver a cached object to the client or to validate the object at a remote server, and for scheduling data delivery to mobile clients. Server cooperation enables servers to provide resource information to cache managers, which enables cache managers to estimate the recency of cached objects.

The main contributions of this dissertation are as follows: First, we present a flexible and scalable architecture to support client profiles that is straightforward to implement at a cache. wireless base station. Second, we present techniques to improve estimates of the recency of cached objects using server cooperation by increasing the amount of information servers provide to caches. Third, for mobile clients, we present a framework for incorporating profiles into the cache utilization, downloading, and scheduling decisions at a We evaluate client profiles and server cooperation using synthetic and trace data. Finally, we present an implementation of profiles and experimental results.

Improving Data Delivery in Wide Area and Mobile Environments

by

Laura Bright

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2003

Advisory Committee:

Professor Louiqa Raschid, Chairman/Advisor
Professor G. Anandalingam, Dean's Representative
Professor Bobby Bhattacharjee
Professor Peter Keleher
Professor Samir Khuller
Professor Nick Roussopoulos

DEDICATION

To my family.

ACKNOWLEDGEMENTS

First, I want to thank my advisor, Louiqa Raschid, who has given me considerable guidance and support throughout my time in graduate school. She was very supportive of my efforts to pursue a research topic outside of her primary area and has given me helpful feedback on my work while encouraging me to collaborate with others. She has consistently believed in me and done everything she can to ensure my success. I also wish to thank Bobby Bhattacharjee for his willingness to co-advise me and work with me on data delivery for mobile clients. His suggestions and expertise made a considerable contribution to my dissertation research. Finally, I wish to thank my other committee members, Anand Anandalingam, Pete Keleher, Samir Khuller, and Nick Roussopoulos for serving on my committee and providing useful feedback.

Avigdor Gal at the Technion has also provided considerable assistance and support as a collaborator, and has provided feedback on

my research. His research ideas and expertise have made a valuable contribution to the work presented in this dissertation. Many past and present members of our research group at Maryland have provided valuable assistance as well. I wish to thank Maria Esther Vidal and Vladimir Zadorozhny for providing feedback on my research ideas, and Hui-Fang Wen, Derek Goldstein and Tao Zhan for their programming support.

I also wish to thank many current and former members of the Maryland database group, including Professors Lise Getoor, Sudarshan Chawathe, and Mike Franklin, students Antonios Deligiannakis, Yannis Sismanis, Dimitrios Tsoumakos, and Feng Peng, and alumni Alex Labrinidis, Ugur Cetintemel, Fatma Ozcan, Mehmet Altinel, Demet Aksoy, Tolga Urhan, and Yannis Kotidis. I have also had the benefit of my numerous past and present colleagues in the CLIP lab, including Mona Diab, Nizar Habash, Okan Kolak, Clara Cabezas, Grazia Russo-Lassner, Rebecca Hwa, Dina Demner-Fushman, Adam Lopez, David Zajic.

Gwen Kaye has provided assistance with meeting deadlines, and has been a helpful mentor throughout my time in graduate school. I had the privilege of working with Gwen first as her teaching assistant and now in her capacity as graduate program coordinator, and she has provided useful advice and support in both roles. I also thank Fatima Bangura for providing administrative support, and Brenda Chick, Adelaide Findlay, and Edna Walker for their assistance. Finally, I thank the UMIACS and CS support staff for their technical

assistance.

I acknowledge Jonathan Eckstein for providing the email trace data used in this disseration, and the National Laboratory for Applied Network Research (NLANR) for providing the proxy trace data.

Last but not least, I need to thank my many friends who have supported me throughout my time in graduate school and have made my stay in Maryland much more enjoyable. In particular, I thank Jaime Spacco for his willingness to listen to me discuss just about any topic, and Selcuk Candan for giving both technical feedback and general advice for surviving graduate school. I also need to acknowledge many other friends, including (but not limited to): Chadd Williams, Debbie Heisler, Reiner Schulz, Aram Khalili, Dave Hovemeyer, Jennifer Baugher, Adam Lopez, Jordan Landes, Kate Swope, Gene Chipman, Sandro Fouche, Rob Sherwood, Omer Horvitz, Brian Postow, Matt Katsouros.

Finally, I thank my parents for their support which made this all possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The popularity of the Internet has led to an increased diversity of applications that access data across wide area networks and in mobile environments. Example applications include news, sports scores, weather reports, E-commerce, auctions, and email. Data delivery on wide area fixed networks is often characterized by high latency due to congestion on the network or heavy loads at remote servers, and data delivery on wireless networks is characterized by low bandwidth and frequent disconnections. Clients may access data on a variety of devices across either fixed or wireless networks, and may have different degrees of connectivity. Many caching and replication technologies have been proposed in these environments to reduce access latencies and bandwidth consumption, and improve data availabilty.

There are many challenges to caching and data delivery in wide area environments. One challenge is keeping cached copies of objects fresh with respect to the objects at the servers. This problem has received considerable attention in the literature, and many different solutions have been proposed. Some solutions are push-based, i.e., servers notify caches when an object is updated. Such solutions can guarantee that cached objects have an acceptable degree of freshness,

but increase the load on servers. Other solutions are pull-based, i.e., the cache must contact servers to validate objects (check for updates) whenever the cached object is estimated to be stale. Pull-based solutions require no cooperation from servers, but their effectiveness is limited by the accuracy of estimates of when objects are updated at remote servers.

A second challenge is that clients may have different preferences with respect to the latency and recency of data for different applications. For some applications, clients may require the most recent data. For other applications, clients may tolerate stale data that can be delivered quickly. Existing solutions do not consider this diversity. For example, pull based solutions may either vaildate cached objects when clients can tolerate some staleness, or deliver stale objects to clients who require the most recent data.

A third challenge is that it is difficult for cache managers to estimate when updates occur at remote servers. Servers typically provide caches with the last time an object was modified, but do not provide any additional information about an object's update patterns. This limits the effectiveness of heuristic estimates of when updates occur. Estimates that are too conservative may cause too many validations at remote servers, which can increase the latency of requests. On the other hand, estimates that are too optimistic may result in stale data being delivered to clients. Thus, the inability of cache managers to estimate when updates occur severely limits the effectiveness of pull-based policies, and makes it difficult for cache managers to meet the recency preferences of clients.

Finally, a challenge to data access for mobile clients is maintaining the desired level of latency for different applications in the presence of limited wireless bandwidth. Mobile clients typically have different Quality of Service (QoS) re-

quirements for their different applications. For some applications, e.g., instant messaging, they may require low latency, but for other applications, e.g., casual web browsing or email, they may tolerate higher latencies. Many techniques have been proposed in the literature for end to end QoS deployment, i.e., providing QoS guarantees to different applications. However, these techniques may have high overhead and may require changes to both the wireless network and the underlying fixed network, which is not always feasible in practice. We note that while QoS is also a challenge on fixed networks, we focus on mobile clients in this dissertation. This is because we emphasize solutions that can be implemented with minimal changes to existing architectures, and QoS deployment on fixed networks typically requires changes to the underlying network.

The goal of this dissertation is to provide flexible, scalable data delivery solutions that can meet client latency and recency preferences with minimal changes to existing architectures and protocols. Existing solutions fail to meet one or more of these criteria. Specifically, our objectives are as follows:

1. Enabling clients to specify and communicate their latency and recency preferences to cache managers and wireless base stations.

2. Enabling cache managers to determine if a cached object meets the client's preferences.

3. Enabling cache managers to estimate the recency of cached objects with greater accuracy than existing solutions.

4. Providing solutions that are flexible (i.e., allow clients to easily express and change their preferences), scalable (i.e., support a large number of clients

with minimal overhead for cache managers and base stations) and that require minimal changes to existing protocols.

1.1 Contributions

In this dissertation we present two complementary solutions to address these challenges and meet the above objectives: client profiles and server cooperation. These two mechanisms improve pull-based consistency policies by enabling clients and servers, respectively, to provide additional information to caches. They provide a scalable framework for customized data delivery to clients on both fixed and mobile networks.

To improve pull-based consistency policies, profiles enable *clients* to communicate to caches their application specific preferences with respect to both latency and recency of data. Cache managers can use this client information to better meet the needs of diverse clients and applications. Similarly, server cooperation enables *servers* to provide caches with resource information. This may include either an individual or aggregated history of updates to objects at the server. This history is used by cache managers to estimate when objects are likely to be updated in the future. This reduces the number of times the cache manager needs to validate objects at remote servers, which improves access latencies compared to existing pull-based policies without the heavy server overhead of push-based policies. Together, this framework for both clients and servers to provide additional information to a cache can reduce the latency of client requests while still providing fresh data in many cases. This dissertation shows that client profiles and server cooperation are flexible and scalable ways to customize data delivery to diverse clients in wide area environments with reduced latency, bandwidth con-

sumption and server overhead compared to existing push-based and pull-based solutions.

Profiles can improve data access for mobile clients without the overhead of end to end QoS deployment, i.e., changing the underlying fixed network to provide QoS guarantees for different applications. Profiles leverage proxy caching at or near a wireless base station to reduce the effects of fixed network latencies. In addition, mobile client profiles enable clients to specify the relative priority of each application. These priorities are used at the base station to schedule data delivery on the wireless downlink. In this dissertation we show that using mobile client profiles for both caching and scheduling decisions can effectively differentiate services for mobile applications.

Specifically, this dissertation makes the following contributions:

- We present a framework to support two complementary techniques, client profiles and server cooperation, to improve pull-based caching and data delivery in wide area and mobile environments.

- We show that for fixed network data access, using client profiles in caching decisions can effectively differentiate services for diverse clients and applications. Profiles are flexible: they can be specified by clients and stored locally, so clients can adjust profiles without any additional communication overhead. They can also be tuned to control the latency-recency tradeoff, or to provide an upper bound with respect to either recency or latency. We present an architecture for communicating profiles to caches that is scalable to a large number of clients. Client profiles can be deployed with no overhead for servers, and low overhead for clients and caches. They can be supported by clients and caches with only minor changes in the communi-

cation protocol.

- Experimental results with trace data over a 5-day period show that profiles can reduce the total validations by 39%, nearly all of which are unnecessary validations. Profiles reduce the number of unnecessary validations (freshness misses) by up to 45%, without requiring any additional contacts with remote servers. This is a significant improvement over existing approaches to reducing freshness misses. Existing approaches use either push-based strong consistency [78], which requires servers to push update information to clients, or pre-validation [39], which requires clients to prefetch expired objects before they are requested.

- We present a server cooperation scheme that complements client profiles by enabling servers to provide caches with resource information about objects at servers. We present techniques for servers to model update paterns to objects at servers using either individual or aggregated history information. While our models do not provide a statistical fit, we show that our techniques are more accurate than existing approaches using three distinct datasets. We show that server cooperation can improve a cache manager's estimates of the freshness of cached copies, which can reduce bandwidth consumption and communication overhead. Server cooperation can be implemented with only minor changes in communication protocols. It can also scale better than push-based policies that require servers to store information about individual clients, and has less implementation overhead for servers. Experiments with three datasets show that using an aggregated history can reduce the number of validations by 10%-16% compared to using only the last update, and individual history can reduce the number of

validations by 20%-36% compared to using the last update, while providing a comparable level of freshness. We also present an adaptive policy that can respond to unexpected changes in an object's update patterns.

- We use profiles to improve data delivery and differentiate services for mobile clients. Our scheme uses profiles for caching decisions at or near the wireless base station. We present a scheduling scheme for data delivery to mobile clients that can provide different levels of service for different mobile applications. Our results show that using profiles for both caching and scheduling decisions at the wireless base station can provide low latency for certain classes of applications without the overhead of end-to-end QoS deployment. We also enable applications to use handoff profiles, which can mitigate the effects of delays when clients migrate to a neighboring wireless cell.

- We present an implementation of profiles on both fixed and mobile networks using the Squid proxy cache [24]. We describe the design of the system, and present an experimental evaluation of profiles. Our implementation shows that profile deployment is feasible in both fixed and mobile environments, and our implementation results validate the effectiveness of using profiles. Further, our results show that validations can significantly increase access latencies, even when an object has not changed, which further motivates the need to reduce unnecessary validations. Our implementation results over a three-hour period show that profiles can reduce the total number of validations by up to 16%, and nearly all of these are unnecessary validations.

1.2 Organization of Thesis

This dissertation is organized as follows. In Chapter 2 we provide an overview of wide area caching and data delivery technologies and formally define our problem. In Section 2.2 we present several examples of caching architectures that are widely deployed on fixed networks. We discuss the advantages and disadvantages of each, and challenges to maintaining data consistency. We then present several commonly used policies for keeping cached data consistent with data at remote servers in Section 2.3. We consider both pull-based and push-based policies, and both strong and weak consistency guarantees. We present architectures for mobile data access in Section 2.4 and discuss the unique challenges for data delivery for mobile clients. We formally define the problem addressed in this dissertation in Section 2.5.

We survey related work in Chapter 3. This chapter includes related work in databases, web caching, and networking. We classify this work broadly into two areas, caching and scheduling. We present work in caching in Section 3.1. This includes techniques in both web caching and databases to maintain data copies and reduce access latencies. We consider work in web cache consistency and view materialization, and present the state of the art in both push based and pull based consistency policies. We also discuss prefetching. In Section 3.2 we present work in scheduling. This work includes packet scheduling on both wireless and fixed networks, broadcast scheduling, and real time scheduling. We survey schemes to differentiate services on wireless and fixed networks, as well as work on adaptive applications and resource allocation.

In Chapter 4 we present our framework for clients to provide profile information to caches using Latency-Recency Profiles. Profiles are a set of application-

specific parameters that reflect client preferences with respect to the latency of their requests and the recency of their data. We present a flexible, scalable architecture for clients to configure and communicate their profiles to caches. The profile parameters can be tuned to control the latency-recency tradeoff or provide an upper bound with respect to either latency or recency. We evaluate profiles using both synthetic and trace data. Our results show that using profiles can reduce bandwidth consumption and latency compared to existing policies while still providing fresh data in many cases.

In Chapter 5 we present techniques for modeling updates at remote servers to improve the effectiveness of using profiles. We show how to model updates to either an individual object (individual history) to multiple objects (aggregated history), and present multiple levels of server cooperation depending on how much information servers provide. Depending on the level of cooperation, cache managers can choose different policies to estimate the freshness of cached objects for different levels of server cooperation. We also consider heuristics to detect bursts, i.e., periods where the number of updates to an object exceeds the expected number of updates and is not consistent with the object's past update history. We present an adaptive policy that can choose between different policies depending on an object's behavior. We evaluate these different policies using several data traces from diverse applications.

In Chapter 6 we show how profiles can improve data delivery for mobile clients. We show that using profiles for both caching decisions at the base station and scheduling decisions on the wireless downlink can improve end-to-end latencies for different classes of applications. Using profiles can also mitigate the effects of handoff delays when clients migrate to a neighboring cell. We present simulation

results that show the effectiveness of profiles in mobile environments.

We present our implementation of profiles in Chapter 7. We have extended the Squid Proxy Cache [24] to support profiles, and we have modeled a low bandwidth wireless link to show the effectiveness of both caching and scheduling for mobile clients. We describe the design of the system, challenges, and lessons learned. We also present experimental results from the implementation that show the effects of caching and scheduling on latency and recency of data.

We conclude and discuss future research directions in Chapter 8.

Chapter 2

Background and Problem Definition

In this chapter we present the state of the art in data access in wide area and mobile environments. We first present motivating examples that illustrate challenges of data access across wide area fixed and wireless networks. We then discuss existing caching technologies that are currently used to address these challenges, and describe policies for keeping cached data fresh. We classify these policies on two dimensions: amount of server cooperation (pull-based or push-based) and consistency guarantees (strong or weak). We consider the advantages and disadvantages of each policy, and discuss limitations of current solutions. Next, we discuss issues and challenges for clients who access data from mobile devices. Finally, we formally define the problem addressed in this dissertation.

2.1 Motivation

We present several motivating examples of clients with diverse latency and recency preferences on both fixed and wireless networks, and discuss challenges to modeling updates at remote sources.

2.1.1 Diverse client preferences

Clients accessing data in wide area environments often have diverse latency and recency preferences. There are many factors that can influence client preferences, including latency, cost, and connectivity.

Latency refers to the amount of time it takes to deliver data to clients. Validating cached objects can add significant latency to requests, even when clients have a high bandwidth connection to the Internet [40]. For some applications, e.g., stock quotes, clients may be willing to tolerate this extra latency if it guarantees that they receive the most recent data. For other applications, e.g., news, weather, clients may be willing to validate the object less often (and risk receiving stale data) to improve latencies. *Cost* refers to other costs associated with remote data access. For example, if a source charges money for data access, or if it must regenerate dynamic objects, clients may be willing to explicitly accept stale data to save money or reduce overhead. A third factor that may influence client latency and recency preferences is *connectivity*. Clients with a high speed connection to the Internet may be more interested in receiving the most recent data because they have abundant bandwidth, and they may be less tolerant of staleness. On the other hand, clients with a low bandwidth connection may much more willing to tolerate stale data to reduce access latencies and other costs. For example, a mobile client with data cached locally on their PDA may wish to minimize the number of contacts with remote servers to conserve bandwidth and battery power.

2.1.2 Modeling updates

Estimating the recency of cached objects is useful in many contexts, for example keeping cached data copies consistent with objects and remote servers, or determining if a cached copy meets a client's recency requirements. However, there are several challenges to modeling update patterns to objects at remote servers. Sources have considerable heterogeneity in update patterns, so a single model may not be appropriate for all sources. We outline some of the challenges below.

Sources can vary considerably with respect to their update frequency, predictability, and burstiness. *Frequency* refers to how often a source is updated. Some sources may be updated many times every day, e.g., a news source. Others may be updated less frequently, e.g., daily or weekly. *Predictability* refers to how easily one can predict when the next update will occur. At one extreme are sources with completely deterministic update patterns, for example a source that is updated every hour on the hour. At the other extreme are sources with completely random updates. Many data sources lie in between these two extremes, for example a source that is updated once every morning, but not necessarily at the same time each day. *Burstiness* refers to periods of bursts of updates that are not consistent with the object's update patterns. We define a burst as a period where the number of updates is significantly larger than the expected number based on the object's update patterns. For example, a news source may normally be updated at regular intervals, but may experience a burst of updates when a breaking news event occurs.

Existing techniques for estimating updates to sources do not consider this heterogeneity. To accurately estimate when updates occur, we need modeling techniques that can adapt to sources with different degrees of update frequency,

predictability, and burstiness.

2.1.3 Mobile data access

Like clients on fixed networks, mobile clients typically have diverse QoS requirements for their different applications. Applications such as instant messaging typically require low latency, while applications such as email may tolerate higher latencies. As discussed in Section 2.4, many mobile clients may share a low bandwidth wireless connection to the fixed Internet. Further, clients in this environment may experience frequent disconnections, both voluntary and involuntary, and handoffs (discussed further in Section 2.4). Thus, some clients may wish to reduce the end-to-end latency of their requests. We motivate two ways to improve data delivery to mobile clients and present examples below.

The first way to improve data delivery is to use an intelligent scheduling scheme on the wireless downlink. A naive scheduling scheme would delivery data in a first come, first served manner. When there is a heavy workload, the latency of all requests will increase. This is unacceptable for applications such as instant messaging that require low latency. Thus, an intelligent scheduling scheme that can effectively differentiate services for different applications is needed.

A second way to improve data delivery for mobile clients is by caching data near the wireless base station. While there is typically more bandwidth on the fixed network than on the wireless downlink, caching can still reduce end to end latencies and can improve data delivery for many mobile clients. For example, if a mobile client may be involuntarily disconnected, it may tolerate stale cached data that can be delivered quickly.

2.2 Caching Architectures

Caching is a widely used technology to reduce access latencies and improve data availability on wide area networks. A challenge to caching is that cached data becomes stale as updates are made at remote servers. Many cache consistency solutions have been proposed, both push-based and pull-based, with varying levels of server cooperation. In this section, we present examples of widely used caching architectures. These architectures make different assumptions about the level of server cooperation, including whether or not the server is aware of the cache and how much information the server provides to the cache. In Section 2.3, we discuss issues and challenges to keeping caching data fresh with respect to data at remote servers, and present pull-based and push-based solutions that are widely used in practice.

Caching can be performed at many different points on the Internet, either close to clients (e.g., browser caches, proxy caches), close to a server (e.g., application server caches), or in between (e.g., CDNs). In this section we present examples of caching architectures at different locations on the Internet. We categorize each of these architectures by the types of benefits they provide to both clients and servers, as well as by the amount of cooperation required between servers and caches and their ability to meet client preferences.

Client/Browser An architecture that can improve data access for a single client is caching data locally on a client's machine, e.g., as part of their web browser. This architecture is shown in Figure 2.1. Caching at a client's browser allows clients to access frequently referenced objects without contacting remote servers, which can significantly reduce access latencies. It can also improve data

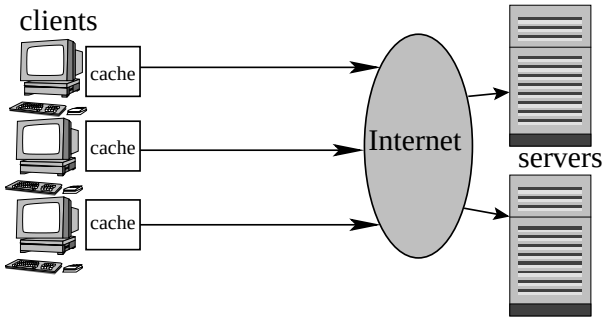

Figure 2.1: Browser Cache Architecture

availability when a client is temporarily disconnected from the network. Since a browser cache serves a single client, clients can configure the cache according to their preferences. However, there is currently no way for clients to explicitly tradeoff data recency for reduced latency.

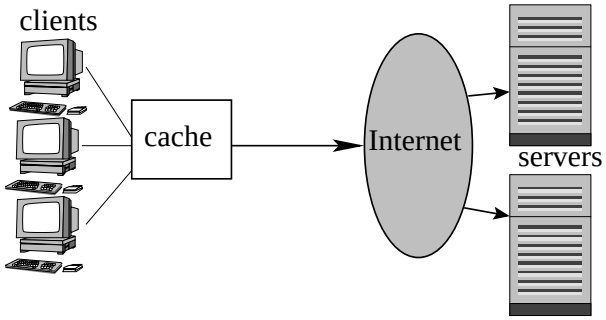

Figure 2.2: Proxy Cache Architecture

Proxy Caches Client-side proxy caching is another example of caching close to the client, and is a widely used technique to reduce access latencies on the Web [26, 41, 72]. In this architecture, a cache resides between a group of clients, e.g.,

a company or university campus, and the Internet. This architecture is shown in Figure 2.2. A proxy cache stores objects previously requested by clients, and these cached objects may be used to serve subsequent requests. Since multiple clients access objects through the proxy, a proxy cache can leverage commonalities in client requests and reduce access latencies. Proxy caches typically treat all client requests alike, so they may not meet the preferences of individual clients. We note that servers are typically unaware of the existence of both browser and proxy caches and no server cooperation is assumed.

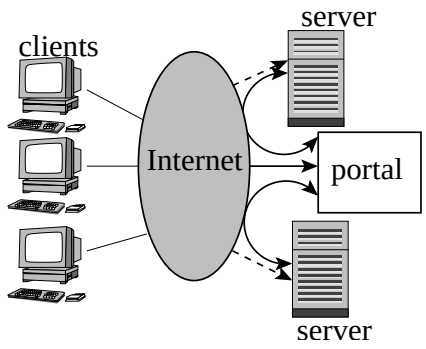

Figure 2.3: Portal Architecture

Web Portals Portals are sites that make it easier for clients to locate and access relevant information. They cache data gathered from other data sources, so clients can easily access all relevant information from a single site. This can reduce the latency and overhead of accessing multiple sources. The web portal architecture is shown in Figure 2.3. In this architecture, some servers may be aware of and cooperate with the portals (i.e., notify them of updates), but this is not required. Therefore, an important challenge for portals is keeping cached data up to date, otherwise they will provide no benefit to the clients.

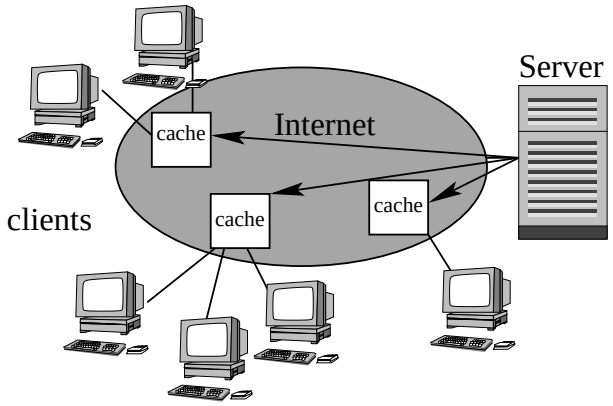

Figure 2.4: CDN Architecture

Content Delivery Networks (CDNs) Content Delivery Networks (CDNs) distribute frequently accessed content from a web server to multiple locations throughout the Internet. Servers redirect client requests for an object to a cached copy that resides close to the client or has the lightest load. CDNs serve a large amount of static content that rarely changes, e.g., images. However, CDNs may also serve content that changes regularly. In this case, servers notify the copies of changes [40], so the CDN will always deliver fresh data. This differs from portals that may not be able to rely on receiving updates from servers. The CDN architecture is shown in Figure 2.4. CDNs benefit servers by reducing the number requests for frequently requested objects, and work particularly well for large objects, e.g., images. They also improve access latencies for clients by serving requests with a copy that is geographically close to the client, rather than the copy at the server.

We note that in this dissertation, we do not study the performance of CDNs because they are limited to cases where the server fully cooperates.

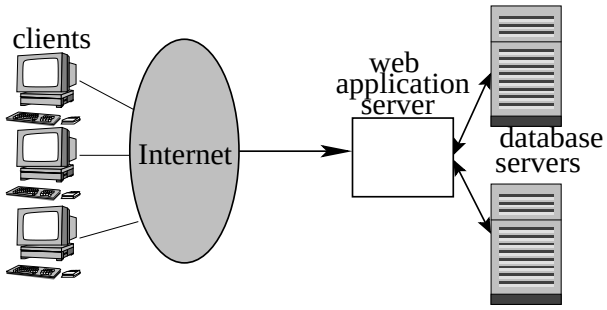

Figure 2.5: Application Server Architecture

Application Server Caches/Reverse Caches An example of caching technologies closer to a server are application server caches and reverse proxy caches. Application servers improve the performance of data intensive web sites by offloading some functionality from web servers. Many commercial products are available, e.g., Oracle9iAS Web Cache[23], IBM WebSphere[111], and BEA WebLogic[110]. Application servers are well-suited for large scale data handling, and can perform caching to further improve performance. For example, an application server cache can reside between database server and the Internet, and cache components of dynamically generated web pages. The application server can then automatically deliver pages without contacting the database server. The Oracle Application Server Web Cache [23] is an example of a product with this functionality. Similarly, reverse proxy caches reside close to a server and cache popular objects to reduce the load on the web server. For servers, the advantage of a reverse proxy cache is reducing the load on the server, which improves performance. Unlike proxy caches, which aim to improve latencies for clients, reverse proxy caches aim to reduce loads on servers. However, they also benefit clients who access the server by reducing the latencies of their requests. We note that there is typically

some cooperation between the database server and the cache, and there may be a high-bandwidth link connecting them. The architecture is shown in Figure 2.5.

2.3 Cache Consistency

An important challenge to any of the above caching technologies is that cached data becomes stale as updates are made at remote servers. Many types of cache consistency policies have been proposed to address these challenges. These policies can be categorized by both the amount of server cooperation required (push-based vs. pull-based) and the types of guarantees they provide (strong vs. weak). We define each of these below and discuss the tradeoffs, then present example of policies in each category.

Categorization of Policies We categorize consistency policies by the amount of server cooperation required. *Push-based* policies require servers to notify caches whenever a cached object is updated. Cache managers will typically mark such objects as invalid, and it is their responsibility to request an updated object from the server (either on the next client request for the object or earlier). Servers must store information about the contents of all clients caches, which may add significant overhead at the server. Push-based policies may work well when the number of caches is small (e.g., an application server cache), but may not scale well to a large number of caches.

In contrast, *Pull-based* policies do not require servers to store any information about caches. With pull-based policies, it is the responsibility of the cache manager to contact the server whenever it estimates that a cached object is stale. Inaccurate estimates may cause either fresh objects to be validated or stale ob-

jects to be delivered to clients, both of which reduce the benefits of caching. Since no server cooperation is required, pull-based policies work well when there are a large number of caches, e.g., browser or proxy caches. We note that while pull-based policies do not require servers to store any information about clients, they may allow servers to provide additional information to clients as part of their responses to client requests, e.g., [41, 92, 70, 112].

Policies can also be categorized by the consistency guarantees they provide, either strong or weak. *Strong* consistency means that clients are guaranteed to receive fresh data on every request. *Weak* consistency means that clients are not guaranteed to receive fresh data. We note that the choice of push vs. pull is orthogonal to strong vs. weak consistency.

2.3.1 Example Policies

We now consider several widely used policies. We categorize each policy according to both the amount of server cooperation (push-based or pull-based) and freshness guarantees (strong or weak). We discuss which caching architectures most commonly use each policy, and consider the advantages and disadvantages of each policy, as well as the ability of each to meet client preferences.

Time-to-Live (TTL) TTL is a pull-based weak consistency policy that requires no cooperation from remote servers. Each object is assigned a time-to-live (TTL) [30, 47, 57], i.e., the estimated length of time the object will remain fresh. If the TTL of a requested object has expired, the cache must *validate* the objects (check for updates) at the remote server. We discuss the details of TTL in Chapter 3. While TTL can deliver fresh data to clients in many cases, validation

adds overhead to client requests and reduces the benefits of caching. Further, it is difficult to accurately estimate an object's TTL. An estimate that is too conservative will improve freshness but result in many unnecessary validations at remote servers, while an estimate that is too optimistic reduces contact with remote servers but may result in many clients receiving stale data. Further, since cache managers typically control the TTL parameters, TTL treats all clients and applications alike and does not consider clients with diverse preferences. TTL is the most commonly used mechanism in browser and proxy caches, and may also be used by web portals.

Polling-Every-Time Polling-Every-Time [27, 78] is a pull-based strong consistency policy. On every request, a cache must validate the cached object before delivering the object to the client. Thus, it is equivalent to TTL with every object expiring immediately after being cached. While this policy guarantees that clients will receive fresh data, it adds extra latency to every request and reduces the benefits of caching. It is useful when strong consistency is required and downloading objects is costly (e.g., the objects are very large), but is not widely used in practice due to its high latency for clients and heavy load on remote servers. It also does not consider clients who may tolerate stale data.

Always-Use-Cache (AUC) A pull-based weak consistency policy that minimizes latency is to serve all requests from a cache, and perform prefetching in the background to keep cached objects up to date. We refer to this approach as Always Use Cache (AUC). Prefetching strategies to maximize the overall recency of a cache are described in [29, 35, 36]. This approach has several limitations. First, in the general case where there is no cooperation from remote servers, the

cache has no knowledge of when updates occur. Therefore, AUC typically must poll remote servers to keep cached data up to date. This may consume large amounts of bandwidth and does not scale well to large numbers of objects. Also, there may be some delay between when the update occurs and when the cache manager checks for updates. During this time, the cache will return stale data. Therefore, while AUC minimizes the latency of client requests, it may perform poorly with respect to recency and consume large amounts of bandwidth, since it does not scale well to large caches or frequently updated objects. Thus, it may not meet the recency requirements of some clients. AUC is commonly used by web portals and other technologies that maintain copies of objects from many web sites, e.g., web search engines [35].

Server-Side Invalidation (SSI) A push-based strong consistency policy is to have servers maintain information about objects stored in client caches, and send invalidation messages to caches when an object is updated. Alternatively, a server may push the updated object to caches. We refer to both of these approaches as SSI [78]. When a server sends only invalidation messages to the cache, SSI has performance comparable to TTL assuming TTL estimates are accurate. This approach was shown to be feasible in terms of bandwidth and server load in [27, 78]. However, if servers instead send the updated objects after each update, as is the case with many application server caches[23], the overhead of SSI may be considerably greater. Also, if clients can tolerate stale data, this places an unnecessary load on the server and consumes excessive bandwidth.

SSI is often used in application server caches, CDNs, or web portals. It may be feasible in these environments because the number of servers and caches is typically fixed, which facilitates cooperation and reduces scalability concerns. It

is unlikely to be a viable alternative in proxy caches because many web servers are either unable or unwilling to implement it.

Approximate Caching Finally, a push-based policy that does not require strong consistency is approximate caching, e.g., [8, 65, 88]. This policy allows cached data values to deviate from the values at the server in a controlled way. For example, a client could accept cached stock quotes that deviate from the actual values by no more than 5%. Servers keep track of the values stored in client caches as well as the client specified bounds, and notify clients whenever their cached value exceeds the bounds. Approximate caching can reduce the amount of contact between caches and servers compared to SSI, and still provides freshness guarantees. However, like SSI, is requires servers to store information about the contents of caches, so it works best when the number of caches is relatively small.

Summary To summarize, the above policies treat all clients and applications alike and may not meet the needs of diverse applications. Some policies (e.g., TTL, Polling-Every-Time, SSI) may increase the latency of requests, consume excessive bandwidth, or do both. This overhead may be unnecessary in cases where clients will tolerate stale data that can be delivered quickly. Similarly, some policies (e.g., AUC) can minimize the latency of requests but may not meet client recency preferences. A scalable solution that can handle clients and applications with diverse latency and recency requirements is needed.

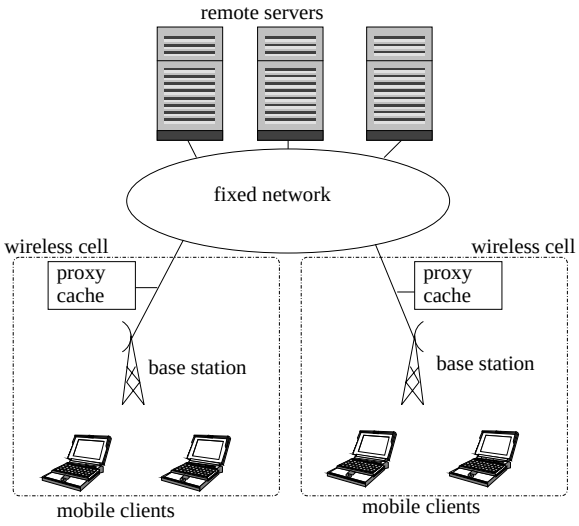

Figure 2.6: Mobile Architecture

2.4 Mobile Data Access

In the previous section, we discussed challenges to remote data access on wide area fixed networks, and existing caching technologies and consistency policies to address some of these challenges. We now consider additional characteristics and challenges to mobile data access on wireless devices.

2.4.1 Architecture

We consider a set of mobile clients in neighboring wireless cells. Clients access the fixed Internet through the wireless base station in their cell. Clients may migrate to a neighboring cell before all their requests are served, and need to connect to the base station in the new cell to receive their data. When objects become available at the base station, a scheduling algorithm determines the order that they are delivered to clients.

A proxy cache may be located at or near the base station to reduce fixed network latencies. We assume that each base station maintains a separate cache. This architecture is shown in Figure 2.6.

We assume that base stations are equipped with functionality to make caching and scheduling decisions. Another feasible alternative is to implement this functionality at a host colocated with the base station. We do not consider such implementation-specific issues further in this dissertation; instead, we use the generic term base station to refer to the entity with the caching and scheduling functionality.

2.4.2 Challenges

Data delivery in mobile environments presents several challenges in addition to those in fixed network data delivery. First, the available bandwidth on the wireless downlink is typically much lower than on the fixed network. Therefore, clients may experience delays due to congestion on the wireless downlink. We note that congestion is also a challenge on fixed networks, and many scheduling algorithms, e.g., [15, 43, 54, 103, 117] have been proposed to allocate bandwidth fairly to clients. We discuss these further in Chapter 3. Data delivery on fixed networks typically requires multiple hops between the source and the destination, so implementing these algorithms requires changes to the entire network. In contrast, data delivery on wireless cellular networks consists of a single hop from the wireless base station to the client, so wireless scheduling schemes can be deployed locally at a base station without any changes to the fixed network. Thus, in this dissertation we focus on scheduling only for mobile clients.

Second, clients in this environment typically have different preferences with

respect to the latency of their applications. For some applications, e.g., instant messaging, they may require low latency. For other applications, e.g., email, they may tolerate higher latency. Scheduling objects for delivery in a first-come first served manner may not meet the requirements of applications that require low latency, thus, a more intelligent scheduling scheme is needed.

Data access on wireless networks is also characterized by varying signal strength. The rate that a base station can deliver data to a client can vary according to the location of the mobile client. If the base station must deliver a large amount of data to a client with low signal strength, it may increase the latency of other client's requests.

Another challenge in this environment is that clients typically disconnect frequently. Disconnections can be either *voluntary* or *involuntary*. Before a voluntary disconnection, clients may wish to download all the data they will need during the disconnection period. They may have deadlines because they need to receive all the data before they will disconnect. Involuntary disconnections also present a challenge because a client may become disconnected before receiving their data. Clients may also experience *handoffs*, when they migrate to a neighboring cell. In this case they may not be disconnected, but the data needs to be re-routed to the new base station. Thus, clients may experience additional delays during handoffs.

A final challenge is that mobile devices typically have very low battery power. Further, sending data consumes more power than receiving data, so it is important to minimize the number of times that clients contact remote servers. For example, if a client has data cached locally, it may want to minimize the number of times it checks for updates at remote servers.

2.5 Problem Definition

We now formally state the problem addressed in this dissertation. We first state the problem for clients accessing data on fixed networks, then consider the additional challenges for clients accessing data on wireless networks.

2.5.1 Fixed Networks

We consider two problems on fixed networks. Given a client request for a cached object, as well as client preferences with respect to latency and recency of data, to (1) determine the recency of the object and (2) determine whether to serve the client request from the cache without validation, or to validate the object at the remote server.

We state the problem formally as follows: We are given a cache containing a set of n objects, $O_1, O_2, \ldots, O_n$, and corresponding latencies $L_1, L_2, \ldots, L_n$ and update patterns $U_1, U_2, \ldots, U_n$ (described in Chapter 5).

Given a client request for object i, $1 \leq i \leq n$ at time T, and given client preferences r_i and l_i with respect to the recency of the object and the latency of their request, the problem is to

(1) Determine the recency R_i of object i. This is a function of T and U_i.

(2) Determine whether to validate object O_i at the remote server before delivering the object to the client, or deliver object O_i without validation. This is a function of r_i, l_i, R_i, and L_i.

2.5.2 Wireless Networks

On wireless networks, given a request for an object (which may or may not be cached at the wireless base station, the problem is (1) if the object is in the cache, determine whether or not to validate the object before delivering to the client, and (2) when the object becomes available for delivery at the base station, determine when to deliver it to the client. Clients may experience handoffs before all of their data is delivered, so we want to mitigate the effects of handoff delays when clients migrate to neighboring cells. The key challenge in this environment is scheduling the delivery of objects to meet the desired latencies of different applications.

We state the problem formally as follows: We are given a set of m clients, $C_1, C_2, \ldots, C_m$ in a single wireless cell (some clients may have migrated from a neighboring cell), a cache containing a set of n objects, including k cached objects $O_1, O_2, \ldots, O_k$, as well as a set of $n - k$ objects $O_{k+1}, \ldots, O_n$ not in the cache, and corresponding latencies of all objects $L_1, L_2, \ldots, L_n$ and recencies of cached objects $R_1, R_2, \ldots, R_k$, and a queue Q containing y pending requests $Q_1, \ldots, Q_y$, where Q_1 is the next object the base station will deliver.

Given a client request for object i, $1 \leq i \leq n$, and given client preferences r_i, l_i, and p_i with respect to the recency of the object, the fixed network latency of their request, and the priority of the request, the problem is to determine (1) if $i \leq k$ (i.e., the object is in the cache) whether to validate object O_i at the remote server before delivering the object to the client, or deliver object O_i without validation, and (2) where to insert object O_i in Q, once it becomes available for delivery.

Chapter 3

Related Work

There has been a considerable amount of research in the database, web caching, and networking communities addressing challenges to data delivery in wide area and mobile environments. In this chapter, we survey relevant research in all of these areas. We classify this research broadly into two categories, caching and scheduling. In Section 3.1 we discuss research related to caching for both web and database applications to improve performance and policies for data consistency. In Section 3.2 we present scheduling algorithms for data delivery on fixed and mobile networks, including adaptive techniques to handle varying bandwidth and battery power in mobile environments, broadcast scheduling, and real-time scheduling.

3.1 Caching and Consistency

Research in caching and cache consistency addresses challenges to keeping data copies consistent with data at servers. This includes providing data within an acceptable degree of consistency while meeting server, client, and bandwidth constraints. In databases, related research considers caching approximate values, view materialization, and synchronizing large collections of objects. In web

caching, related research includes cache consistency policies and caching dynamic content. There is also related research in caching in other contexts, e.g., caching on mobile devices, shared memory, and distributed filesystems. We survey research in all of these areas below.

3.1.1 Web Cache Consistency

Research in web cache consistency aims to keep cached data consistent with data at remote servers (see [109] for a survey). In Chapter 2 we gave a brief overview of both push-based and pull-based consistency policies. We discuss details of research on both types of policies below.

Pull-Based Consistency The pull-based cache consistency mechanism currently used in most web proxy caches is to assign each object a time-to-live (TTL) [30, 47, 57], either using heuristics or simply using a TTL value assigned by a server. A TTL is an a priori estimate of how long a cached object will remain valid. If a requested object's TTL has expired, it must be validated at the remote server. This increases the latency of the request and reduces the benefit of caching. When servers do not provide TTL estimates for an object, the TTL is estimated as a function of the object's last-modified time. The TTL is typically a percentage of the time elapsed since the object was last modified. This heuristic is based on the intuition that objects that have been modified the most recently are likely to change again in the near future, and was shown to work better than assigning a constant TTL value in [57].

While TTL is straightforward to implement, it has several limitations. In practice, TTL estimates tend to use conservative estimates of when an object

will be updated. Therefore, it may cause many unnecessary validations (freshness misses). It also treats all objects alike and does not consider that different types of objects may have different update frequencies or update patterns. Many solutions have been proposed to address these limitations of TTL.

Some research, e.g., [35, 53, 75], has proposed techniques for modeling updates to sources to improve the accuracy of estimating when objects are updated. Research reported in [75] estimates TTL values based on the probability that an object will be updated within a time interval, rather than considering only the time the object was last modified. The technique suggested in [75] is identical to the First Arrival policy suggested in [53]. [35, 53, 75] all suggest modeling updates as a Poisson model. Research reported in [35, 75] assumes a model that is homogeneous over time, while the model we present in Chapter 5 assumes a time varying update intensity, which was shown to work better in [53].

Research reported in [87] aims to reduce the number of freshness misses by using different TTL values depending on the type of objects. For example, images are generally updated less frequently than HTML objects, so the number of validations could be reduced by using less conservative TTL estimates for images. This research shares similar goals to our research in modeling updates presented in Chapter 5. However, in our research we show that considering the update patterns of individual objects can improve the accuracy of freshness estimates compared to considering the aggregate behavior of similar objects.

There has been research on reducing the number of validations of TTL by having servers piggyback information about related objects on their responses to client requests, e.g., [41, 70, 112]. This research shares our goal of improving pull-based consistency by improving server cooperation, and having servers

piggyback information on the responses to client requests. However, rather than providing history information, servers provide information on which objects have been updated since the cache last contacted the server. This research is orthogonal to ours and is not concerned with estimating the freshness of cached objects, but rather how to efficiently refresh cached objects. In [70], clients piggyback a list of potentially stale cached objects when they contact a server. Servers piggyback the subset of those objects that have been updated on their responses. Research reported in [41] groups related objects into server volumes based on the likelihood that objects will be accessed together, and presents methods for proxies to filter and customize this information. Research reported in in [112] views HTML pages as containers, and piggybacks information about relationships between containers and embedded objects on responses to client requests. For example, if a container needs to be validated frequently, the server could piggyback information about embedded objects on its responses and eliminate the need to validate each embedded object.

Another proposed solution to reduce the client-perceived latency of the TTL approach is to send a (possibly stale) cached copy of the data quickly, and send update information as soon as the remote server has been contacted [12, 47]. In [47], the authors propose sending a stale page to clients and replacing it with a more recent page when it becomes available. Research reported in [12] aims to reduce this latency even further by calculating the difference between the new page and the stale page, and sending only the delta to the client. This reduces latency and guarantees that the client eventually receives a fresh copy, but may consume excessive bandwidth and would be particularly costly in mobile environments. Further, calculating the delta may be non-trivial.

Research reported in [40, 37] considers some of the limitations of TTL consistency when there are multiple levels of caching. It studies the *age penalty*, which occurs when cached data is obtained from another cache, e.g., a reverse proxy cache close to a server. If cached data is obtained, its TTL will expire earlier than if it is obtained from the remote server, which increases the likelihood that the object will need to be validated. This research shows that the age penalty increases the probability of unnecessary validations, or freshness misses. This research shows another limitation of using TTL consistency and further motivates the need to improve mechanisms to keep cached data fresh.

Finally, research reported in [39] considers pre-validation policies to proactively validate expired cached objects before clients request them, which can reduce the client-perceived latency caused by freshness misses. This work shows that up to 30-50% of cache hits may result in freshness misses. The best policy in [39] eliminates 25% of freshness misses (useless validations). However, they replace each online request with up to two offline requests, so the overhead on the server increases. In contrast, in Chapter 4 and Chapter 7 we will show that using profiles can reduce 16%-45% of freshness misses, without any additional contacts with remote servers. Therefore, it can reduce both latency and bandwidth consumption, in contrast with [39] which reduces latency but increases bandwidth consumption.

Push-Based Consistency There has also been much research in push-based cache consistency [45, 78, 114, 116]. Research reported in [78] studies techniques for strong cache consistency, i.e. guaranteeing fresh data. The authors show that server-side invalidation is effective for maintaining strong cache-consistency, however this technique must be implemented by remote servers. Research reported

in [78, 114] shows that push-based freshness is feasible and works well in many cases. In[27] the authors evaluate existing strong consistency schemes and conclude that strong consistency policies such as server invalidation have comparable communication overhead to weak policies such as TTL. However, this requires additional storage and monitoring overhead for servers, and many servers may be unwilling or unable to implement it.

More recently, research reported in [45, 114, 115, 116] proposes techniques to improve the scalability of strong consistency policies. Research reported in [116, 115] proposes using a hierarchical scheme. In [115] the authors study how different workloads affect the scalability of strong consistency policies, and propose using adaptive hierarchies that can adjust to changes in workloads. Research reported in [116] uses application-level multicast to communicate invalidations. This improves the scalability of hierarchical cache consistency policies.

Research reported in [114] shows how servers can limit the amount of information they can store without significantly impacting data consistency. The authors show that maintaining *leases* on objects can improve scalability. Before a lease on an object expires, it is the server's responsibility to notify clients of updates, but after the lease expires clients must contact servers. The authors of [114] show that maintaining short leases can reduce storage overhead without a negative impact on performance. They also show that delaying invalidations, i.e. sending invalidation messages to clients only when the server has sufficient resources, can improve performance and scalability without significantly impacting the recency of data delivered to clients. However, this research does not consider clients with diverse preferences.

Research reported in [45] proposes an adaptive push-pull scheme where servers

can adaptively push updates to some clients and require others to use a pull-based policy depending on the available resources and the update frequencies of objects. Servers can adaptively switch from push to pull to improve scalability when necessary.

Research reported in [85] proposes scalable push-based consistency for content delivery networks (CDNs). This research proposes cooperation among caches to reduce the overhead of pushing updates from the server. For example, servers can grant a single lease to multiple caches, which improves scalability. This research also considers the diverse recency requirements of different types of data, and can guarantee that data is consistent within a time Δ. This is useful for placing a bound on the amount of time by which an object is out of date. However, this research does not consider objects with varying update frequencies or update semantics.

Research reported in [100] also considers cooperation among caches. This research considers both when servers and caches should push updates, and how much cooperation (i.e., sharing update information) there should be between caches. The goal is to maintain the desired level of consistency among caches with minimal overhead in terms of both network delays and processing delays. The authors show that when network delay is high, a high degree of cooperation between caches improves consistency. However, when processing delay is high, increasing cooperation between caches can negatively impact performance because processing delays at caches add excessive overhead.

Cache Replacement We note that there has been much research in web cache replacement policies, e.g., [26, 67, 71, 96, 97, 98]. While this research is orthogonal to this dissertation, much of this research takes into account latency and recency

of data to improve the effectiveness of caching. For example, there is typically greater benefit to caching objects with higher latency and less frequent updates. Thus, ncorporating profiles into cache replacement policies could further improve the ability of caches to meet client preferences.

Research reported in [96, 113] incorporates the latency of objects into the replacement decision, and shares our goal of reducing access latencies for clients. Research reported in [97, 98] combines cache replacement with cache consistency and aims to improve the recency of cached objects. However, this research does not consider clients or applications that may tolerate stale data.

3.1.2 Approximate Caching

Research in approximate caching [8, 65, 88, 89] allows cached data values to differ from values at the remote server within a client-specified bound. Like the research in push-based consistency (e.g., [78]) described above, this research requires servers to store information about clients and the objects in their caches, as well as information about the client-specified bounds for each data value. This requires a considerable amount of storage and monitoring overhead at servers and may not scale well. However, this research can reduce the number of communications between caches and servers, which can reduce bandwidth consumption while still providing data within an acceptable degree of recency.

Research reported in [8] introduces the term *quasi-copy*, a cached value that is allowed to deviate from a server value in a controlled way. For example, a client querying stock prices may be satisfied with cached stock prices that are within 5 percent of actual prices. Research reported in [65] aims to reduce the number of transmissions of an object from a server to a client. The authors

propose a dynamic algorithm that optimizes the refresh rate between the client and server based on the client's tolerance for stale data, the frequency of updates to the object, and the frequency of requests for the object. This research uses the number of updates as its recency metric, and does not consider the amount that a value changes on each update. Research reported in [88] generalizes this research to consider the precision of data values, and adaptively adjusts the degree of precision of cached data values to achieve optimal performance under varying workloads. Finally, research reported in [89] studies policies to prioritize refreshes to minimize divergence between server data values and cached values. These policies exploit server cooperation by considering the available bandwidth and resources at both the server and the cache.

Approximate caching is useful when clients can tolerate staleness within certain bounds, but requires servers to push updates to clients and may not scale. In contrast, the research we present in Chapter 5 on modeling updates at servers can deliver data within client staleness bounds with a high probability, but does not require the high server overhead of approximate caching.

3.1.3 Materialized Views

Research in the area of materialized views, e.g., [13, 56, 60] precomputes answers to database queries to reduce query execution time. Queries can be answered using these precomputed views, which is faster than querying the underlying database. As in web caching, a key challenge in materialized view research is keeping the views fresh when updates are made to the underlying database. However, the challenge is to reduce the computational overhead of recomputing views, rather than to reduce network latency. This research typically assumes full

knowledge of updates to the underlying database, i.e., push-based consistency. Therefore, they do not address the issue of how often to check for updates.

One relevant problem in research on materialized views is when to incorporate updates into a view [59, 119]. Unlike our research, the issue is not when to check for updates, but to determine the most efficient strategy for recomputing the view given the update information [59]. Research reported in [52] allows stale data to be incorporated into materialized views by adding an obsolescence cost, and shares our goal of allowing clients to accept stale data in exchange for lower latencies. Another problem in materialized views is view selection [13, 56, 60] , i.e., choosing a subset of views to materialize to minimize query response time and/or the cost of maintaining the views. This is related to research in cache replacement that caches web objects to minimize latency or bandwidth consumption.

3.1.4 Caching Dynamic Content

A related problem in the context of web-accessible databases is caching dynamically generated web content. As in materialized views, this research typically assumes full knowledge of updates to the underlying database, i.e., push-based consistency. Challenges include efficiently propagating update information, determining which pages are affected by updates, and efficiently recomputing pages.

Research reported in [74, 73] addresses the problem of computing materialized views for web-accessible databases. It differs from related research in materialized views in that it considers the problem of where to materialize views, i.e. in the underlying database or at the web server. As in other research in materialized views, the web server is aware of all updates to the underlying database. This

research determines when materializing a view improves performance, and where to materialize it, assuming full knowledge of updates to the underlying database. Efficient strategies for the database to propagate updates to the WebView are presented in [73].

Research reported in [25, 33, 118] cache dynamic data at the page level. This research proposes techniques to determine which pages the server should invalidate when updates occur to the underlying database. The goal is determining when to propagate updates to cached data, assuming full knowledge of updates to the underlying database. In contrast, the research presented in this dissertation aims to determing whether cached data meets client preferences.

In [25], pages are invalidated by two modules, a sniffer which maps the relationship between dynamic pages and the underlying queries that generate them, and an invalidator which maps the relationship between queries and changes to the database. In [33] a dependency graph maintains information about relationships between dynamic pages and the underlying data, and a graph traversal algorithm determines which pages need to be invalidated when the underlying data is updated. Research reported in [118] proposes several techniques to reduce the overhead of invalidation for dynamically generated web pages, and shares our goal of reducing the overhead of maintaining cache consistency. It partitions dynamic pages into classes that share similar patterns, so servers can invalidate pages in groups rather than individually. It also proposes a lazy invalidation scheme that does not invalidate a page until it is requested, to reduce the overhead of computing which pages must be invalidated. Finally, it proposes precomputing predictable pages that are updated frequently, to reduce the overhead of generating new pages after each update.

There is also research in caching database tuples or page components rather than entire pages. Research reported in [80] caches components of dynamically generated web pages to exploit overlap in queries. However, this research does not consider updates to the underlying databases. Research reported in [9] also proposes techniques to cache database tuples to answer queries. Servers send periodic refresh messages to notify caches of tuples that have changed, which guarantees that cached data is consistent with a past database state within some constant time limit. Research reported in [42] proposes dynamic proxy-based caching. This combines the benefits of reverse caches and proxy caches by allowing proxies to cache components of dynamically generated pages and generate them on the fly. They determine page layout on demand by contacting remote servers. The focus of this research is on enabling proxies to cache components of dynamic pages. This research assumes that server invalidates the cached components when they are updated, and does not consider client preferences.

3.1.5 Prefetching

Previously we have discussed both pull-based consistency policies that refresh objects on-demand, i.e., when they are requested by a client, and push-based policies where servers notify caches when updates occur. In addition, there is much research, e.g., [35, 36, 29, 34] that considers *prefetching* objects before they are requested by clients to improve the availability and recency of cached objects. These policies are all pull-based.

Research reported in [29, 35, 36] considers the problem of refreshing a large set of objects, e.g., crawling pages for a web crawler. This is the AUC policy described in Chapter 2. This research assumes that all requests are served from

the cache, and does not consider client preferences for recency or latency. Cache managers periodically sample servers to detect updates, so they may have incomplete update histories. Research reported in [35] considers prefetching locally cached objects to improve the overall recency of a cache. It determines which objects to sample based on their observed update frequencies, and does not consider object popularity. Research reported in [36] samples a subset of objects at each server to detect which servers change most frequently. Research reported in [29] incorporates object popularity into the decision of which objects to prefectch. Since all this research assumes that requested objects are served from the cache, these solutions may not meet client recency preferences. Further, these techniques for modeling update patterns do not consider that an object's update frequency may vary at different times, as we do in Chapter 5.

Research reported in [34] considers profile-driven cache management. The goal is to refresh a collection of cached objects for a client who is connected for a limited time. There may be insufficient time or bandwidth to refresh all objects, so the decision on what to refresh is made based on a client's profile. The profile exploits both client preferences for recency and semantic relationships between objects, and the goal is to refresh the subset of objects that will maximize utility for the client. This research differs from the research in this dissertation because its goal is to select a set of objects to refresh to maximize client utility. In contrast, the goal of our profiles is to meet a client's preferences with respect to latency and recency of an individual object, and to reduce unnecessary communications with servers whenever possible.

Prefetching for proxy caches to refresh objects before they are requested was proposed in [41, 39, 47]. This research proposes validating cached objects when

their TTL expires. To further reduce latency, research reported in [38] proposes prefetching the means to document transfer. This can reduce the overhead of DNS lookups and connecting to servers, without prefetching the actual document. Research reported in [72] investigates the performance benefits of both caching and prefetching and concludes that proxy caching can reduce latency up to 26%, while prefetching can reduce latency by 57%, and a combined caching and prefetching proxy can reduce latency by up to 60%.

Research reported in [66, 72, 90] uses predictive prefetching to prefetch objects that are likely to be requested by clients in the near future. This research typically exploits the relationships between links on a page. This research is orthogonal to the research in this dissertation because the emphasis is on predicting client's access patterns rather than improving the recency of data in the cache.

3.1.6 Caching in Other Contexts

There is a considerable amount of research in caching and cache consistency in other contexts, e.g., caching on mobile devices, caching in distributed memory and filesystems. This research shares our goals of reducing latency and improving availability, but makes different assumptions about available bandwidth and connectivity than our research. These soutions are also designed for different applications, so they may have different consistency requirements.

Mobile Environments Research in data caching on mobile devices aims to maintain data consistency and improve data availability in the presence of limited connectivity. Some of this work is for client-server environments, e.g., [1, 14, 63]. In [1, 2, 14] clients cache data on their mobile devices, and a server periodically

broadcasts data to clients. These works aim to optimize use of the wireless bandwidth for all clients and do not consider client preferences with respect to latency and recency of data. In contrast, our research in mobile profiles presented in Chapter 6 focuses on caching on the fixed network to reduce client-perceived latency. We discuss pull-based policies that can improve client-server data access on mobile devices in Chapter 5.

Some research proposes proxy caching on fixed networks to improve wireless web access. WebExpress [63] is a system that aims to reduce the latency of Web access for mobile clients by caching data on both client devices and on the fixed network. However, the focus of this research is on reducing wireless traffic volume and protocol overheads. Research reported in [58] considers mobility issues by pushing portions of a proxy cache to neighboring cells, based on predictions of clients' movement patterns. However, this research does not consider keeping cached objects fresh, and does not consider the latency-recency tradeoff we study in this dissertation.

There is also research in caching on mobile devices for peer to peer applications [32, 44]. In this research, there is no centralized server, and updates to objects can occur in multiple locations. The goal of this research is to improve data availabilty while managing conflicting updates in the presence of limited connectivity. This research emphasizes maintaining consistency while reducing communication costs between devices, rather than meeting client latency and recency preferences as in our research.

Distributed Filesystems and Databases There is a considerable amount of research in caching and cache consistency in other contexts, e.g., distributed filesystems [64, 83], client-server databases [51], and distributed shared memory

[68]. This research differs from research in web caching because it assumes clients can both read and write to cached copies; in contrast, client caching in web environments is read-only. Thus, this research typically requires stronger consistency than in a web environment. Further, this research may assume higher bandwidth between copies than in a web environment, and may not scale well to the web where there may be many cached copies of an object.

3.1.7 HTTP Protocol

Finally, we note that the HTTP/1.1 Protocol [92] includes two cache control mechanisms that allow clients to express recency requirements. Specifically, there are two header fields supported by HTTP/1.1 that can be used by clients who wish to control the freshness of their data. The first is the `max-age` field. This field can be used by either the client or server to indicate the maximum age that a cached object is valid, where age is defined as the number of seconds elapsed since the cached object was delivered or validated by the remote server. When both the client and the server specify a `max-age` value, the smaller of the two values is used. We note that a client who uses the `max-age` header will not accept stale data unless a `max-stale` header (described below) is also present.

HTTP/1.1 also includes a `max-stale` field that can be used by clients to indicate that they will accept stale data. It allows clients to specify the number of seconds after the object's TTL expires that they will still accept a cached object. `max-stale` can also be used with no value to indicate that a client will accept a stale object of any age. When both `max-age` and `max-stale` values are set, or when the client and server specify different values for `max-age`, the smaller value is used. However, most browsers do not provide an interface for clients to

easily specify a `max-stale` value, and not all servers support this header.

While the `max-stale` header field is useful for clients who can tolerate stale data, it has several limitations compared to the research presented in this dissertation. First, it supports only one recency metric, the number of seconds elapsed since the object became stale as in [35]. This metric may not be appropriate in all situations because it does not consider update frequency or the effects of updates on the value of the cached data. Further, it may be difficult for a client to specify the exact number of seconds after the expiration time that they will accept stale data. For objects that are updated frequently, clients may prefer a smaller `max-stale` value than for objects that are updated infrequently. Without any knowledge of the update frequencies of the source, the client cannot choose the appropriate values. Therefore, it may be more natural for a client to express their profiles in terms of the expected number of updates to the source, or some other recency metric, e.g., obsolescence [52]. Since the proxy cache stores information about the last time that the object was modified at the remote server, the proxy can make an informed estimate of the number of updates to the object, and make an appropriate decision based on the client profile.

We note that as with the above HTTP/1.1 header fields, our proposed framework does not override the `no-cache` and `must-revalidate` fields. These header fields are important for applications where both servers and clients require strong consistency, e.g., for client-server transactions.

3.2 Scheduling

Related research in scheduling in both networking and systems shares our goal of meeting the latency requirements of diverse applications. This includes re-

search in packet scheduling on both fixed and wireless networks to provide fairness or QoS guarantees. There is also research in supporting diverse applications on wireless networks to adapt to limited resources including battery power and bandwidth. Related research in mobile computing also considers how to improve data delivery during handoffs. Finally, there is relevant research in scheduling for data broadcast and scheduling in other contexts, e.g., real time systems.

3.2.1 Packet Scheduling and Bandwidth Allocation

There has been a considerable amount of research in the networking community in packet scheduling to allocate bandwidth fairly among multiple clients, e.g., [15, 43, 54, 103, 117]. For example, Weighted Fair Queueing (WFQ) [43] guarantees each client receives a fair share of the available bandwidth. However, it does not consider the diverse bandwidth and scheduling requirements of different applications. Research reported in [79] considers fair queueing on wireless networks and will be discussed further below.

Providing support for diverse applications sharing a single network has been considered for both fixed networks, e.g.,[15, 28, 48, 54, 62, 82, 93, 103] and wireless networks, e.g.,[5, 16, 76, 77, 86]. The emphasis of this research is on allocation of sufficient bandwidth to support certain applications, e.g., real time and multimedia. These applications require a continuous bandwidth stream over a period of time. In contrast, the goal of our profiles is to ensure timely delivery of individual objects rather than allocating bandwidth for streams of data. This ensures efficient use of the available bandwidth and reduces the implementation overhead. We discuss related research for both fixed and wireless networks below.

Fixed Networks Research reported in [15, 54, 103] present fair queueing algorithms for integrated services networks that must support a variety of applications such as multimedia, ftp, telnet, WWW, etc. Hierarchical Fair Service Curve Scheduling (H-FSC) [103] describes a hierarchical bandwidth sharing model that considers both fairness and QoS guarantees for diverse applications sharing bandwidth. It aims to schedule packet delivery to meet requirements of real-time applications while allocating the remaining bandwidth fairly among multiple clients and organizations. This scheme shares our goal of supporting diverse applications. However, it would be difficult to implement on wireless networks where the clients and applications sharing the bandwidth is constantly changing.

There has also been research in bandwidth allocation on fixed networks to meet the requirements of an application. Research reported in [82] introduces a QoS broker which allocates bandwidth as well as application and operating system resources to provide QoS guarantees to multimedia applications. This research uses profiles to allocate resources. While these are similar in spirit to the profiles in this dissertation, the emphasis is on QoS parameters for multimedia applications. Research reported in [28] considers fixed-network bandwidth allocation to maximize the utility of diverse applications. This research shares our goal of provisioning limited resources according to the needs of applications.

The service differentiation scheme described in Chapter 6 is similar in spirit to the relative service differentation scheme described in [48]. This research differentiates services for different classes without the overhead of admission control and resource reservation mechanisms. Clients can select the service class that best meets their quality of service and pricing constraints. However, in our research we specifically take cache and mobility issues into account, and do not provide

the same strict set of guarantees as in [48].

Wireless Networks There is also much research in bandwidth allocation on mobile networks which aims to adapt to variances in bandwidth availability and reliability typical in these environments. This research typically considers tradeoffs in QoS or data quality that result from varying bandwidth availability. These tradeoffs differ from the latency-recency tradeoff that we consider in this dissertation. Research reported in [76] describes a framework for adaptive service for mobile multimedia applications. The goal is to provide consistent QoS in the presence noise disturbance, varying distance between the client and the base station, and handoffs. Utility-based adaptive bandwidth allocation is presented in [16, 77].

Research reported in [79] describes packet scheduling on wireless networks. This research aims to approximate fair queueing algorithms, e.g., [15, 43, 54] while taking into account both mobility issues and varying signal strengths. Research reported in [5] describes algorithms for scheduling data delivery for requests with deadlines at a wireless base station. This research considers requests with varying utility per byte delivered to client as well as clients with varying signal strength. Online approximation algorithms are presented to maximize overall utility. These algorithms are effective when requests have no utility after their deadlines, however, starvation is possible with this scheme. In contrast, the scheduling scheme we present in Chapter 6 avoids starvation.

Adaptive support for mobile applications is also presented in Odyssey [86]. The aim is to adapt to changing network characteristics that are typical in mobile environments. When available bandwith becomes limited, applications can trade data quality for reduced resource consumption. For example, applications that

download images may tolerate smaller images that consume less bandwidth. This tradeoff differs from the latency-recency tradeoff that we consider in this dissertation. Finally, in [69, 102] the authors consider application-specific power saving techniques for mobile devices. Research reported in [69] presents an application-specific transport layer protocol that can suspend and resume communication to save power while still meeting the latency requirements of an application.

3.2.2 Handoffs

There has also been a considerable amount of research in mobility support in the presence of handoffs. This research shares our goal of reducing the overhead of handoff during data delivery to mobile clients. Mobile IP[91] is a widely used protocol that routes packets to mobile clients through a home agent. However, packets may be lost during handoffs, so clients may experience some delays. Many schemes aim to improve upon this by multicasting packets to neighboring base stations, e.g., [11, 81, 94, 99]. However, implementing such techniques may add excessive overhead on the fixed network, and multicast may not be available. ICEBERG [108] aims to support mobility in the presence of diverse networks and applications.

In contrast to the above solutions, the caching and scheduling scheme we present in Chapter 6 can reduce latencies during handoffs without requiring multicast or other changes to the underlying network infrastructure.

3.2.3 Broadcast Scheduling

The goal of research in broadcast scheduling, e.g.,[1, 3, 6, 104, 107] is to minimize the average latency of client requests. Some of this research is online or on-

demand, e.g., [3, 6]. In this research, clients requests data objects and a server broadcasts objects to clients based on these requests. Research reported in [6] presents an online scheduling algorithm that can be tuned to trade off average and worst case latency assuming uniform object sizes. Research reported in [3] proposes online algorithms that consider varying object sizes and aims to minimize stretch, i.e., the ratio of the latency of the requests to its size. This is shown to perform better when objects have varying sizes. This research does not consider the varying latency requirements of different applications and are most appropriate in a broadcast setting. In contrast, this disseration considers a unicast model.

Other research in broadcast scheduling is offline, e.g., [1, 104, 107]. This research uses a priori knowledge of object's popularity to develop a broadcast schedule. Since they rely on offline knowledge of an object's popularity, they cannot adapt to varying workloads and are not appropriate for the types of applications we consider in this dissertation.

Most research in broadcast scheduling assumes that all data is available for broadcast at the server, and does not consider updates to requested data objects. Research reported in [7] considers the problem of data staging, i.e. bringing requested objects into main memory so they can be broadcast to clients. However, this research does not consider how to keep the cache fresh in the presence of updates, and does not consider latency/recency tradeoffs.

3.2.4 Real-Time Scheduling

Finally, there is a considerable amount of research in scheduling for real-time systems, e.g.,[21, 22, 101]. The goal of this research is to schedule a set of jobs

such that each job will complete before its deadline. The Earliest Deadline First (EDF) scheduling algorithm [101] has been shown to be optimal in the sense that it is guaranteed to find a solution such that all jobs complete before their deadlines, if such a solution exists. When a system is overloaded, i.e., it is impossible to schedule all jobs to complete before their deadlines, some jobs must be rejected. Utility functions are used to determine which jobs should be rejected to maximize the overall utility of the system. For example, a job that has no utility to the system if it completes after the deadline would have a utility value of a constant if it completes before the deadline, and 0 if it completes after the deadline. Research reported in [20] studies value functions for real-time systems. In Chapter 6, we present a best-effort service differentiation scheme that maps profiles to deadlines and uses EDF scheduling at the wireless base station.

Chapter 4

Latency-Recency Profiles

We now present our framework to support Latency-Recency profiles. Latency-Recency profiles are a set of application-specific parameters that allow clients to specify their latency and recency preferences for different applications [19]. Our framework for profiles enables clients to communicate this information to caches and improve pull-based cache consistency. We first discuss issues important to successfully deploying profiles, and present the parameters and scoring function used by the profiles. We describe our profile based downloading policy (labelled Profile). Finally, we present experimental results using both synthetic and trace data.

Our main results are as follows:

- Using profiles can significantly reduce access latencies for clients who can tolerate stale data.
- Using profiles can significantly reduce the number of unnecessary validations (freshness misses) while still providing fresh data in many cases.
- Profiles can exploit increased cache size better than TTL or AUC. AUC may deliver very stale data when the cache is large, and TTL cannot utilize

a larger cache to reduce latency.

- During surge periods, using profiles can reduce latencies for all clients, even those that require fresh data.
- We present extensive sensitivity analysis that shows the effects of tuning profile parameters on both latency and recency, and shows that the performance does indeed meet the preferences specified by the profile parameters.
- Profiles can be tuned to provide performance anywhere between the extremes of TTL and AUC, and can provide guarantees with respect to either latency or recency of data.

4.1 Profiles: Overview and Parameters

Latency-Recency Profiles allow clients to express their preferences for their applications using a few parameters. Profiles are set individually by each client, and a single client can specify either a single profile or different profiles for different applications. In this section we present Profile, a profile-based downloading policy that is a generalization of the TTL and AUC policies presented in Chapter 2. We first discuss several key issues that are crucial to successfully implementing and using profiles. We then describe how clients can choose target latency and recency values, and present a parameterized decision function that can capture the latency-recency tradeoff for a particular client or application. Finally, we discuss upper bounds provided by our function, and describe how the parameters can be tuned to meet client requirements with minimal overhead for the clients.

4.1.1 Specifying Profiles

There are several issues that are important to successfully implementing and using client profiles at a cache. The first issue is *scalability*. An implementation of profiles that requires a cache to store detailed information about each client would add considerable overhead because clients would need to register profiles with the cache, and the cache would need to keep the information up to date. This does not scale well to large numbers of clients. Our solution to this problem is to implement a parameterized function at the cache, which is sensitive to profiles but does not require the cache to store any profile information. In our framework, browsers append the profile parameters to client HTTP requests, and Profile uses these parameters in the decision function. Thus, Profile can easily scale to a large number of clients, with no additional communication overhead between the client and the cache. This scalability is a key benefit to using a parameterized function.

A second issue is *flexibility*. Clients should be able to specify profiles that are appropriate for each of their applications, and they should be able to easily adjust their profiles as needed. To allow clients to use different profiles for different applications, clients can choose a default profile which they can override for specific domain names or URLs. For example, a client requiring the most recent stock quotes may specify that all requests to the domain `finance.yahoo.com` [49] require the most recent data, but that all other requests can tolerate up to 1 update. Clients can easily change their profiles using their browser, without communicating with the cache.

The third issue is *ease of implementation*. It is straightforward to modify a cache to implement Profile. Profile allows clients with diverse profiles to share a cache without adding any overhead to each other's requests. For each individual

request, the cache will use Profile to choose how to serve the request based on that client's profile. If clients with different profiles request the same object simultaneously, the cache could serve one client's request from the cache while downloading a fresh copy for the other client.

The final issue relates to *guarantees*. Profile is a generalization of TTL, which aims to provide fresh data (assuming TTL estimates are accurate) but may have high latency, and AUC, which guarantees low latency but may delivery stale data. Profile can be tuned to provide performance anywhere between these two extremes. In addition, Profile can support upper bounds on either latency or recency, which other approaches do not support.

4.1.2 Parameters of Profiles and Profile-Based Downloading

Recall from Section 2.5 that our problem is, given a request for an object O_i and client preferences for the recency and latency of the object r_i and l_i, to determine whether or not the object meets the client preferences. We now present the corresponding profile parameters and algorithms.

Profiles include the following parameters:

Target Latency: The first parameter is a *target latency* (T_L), which is the desired end-to-end latency to download an object. For an object O_i, this corresponds to l_i in Section 2.5. We note that the cache can estimate the latency L_i of downloading an object using techniques described in [4, 55], which have been shown to be reasonably accurate in practice.

Target Recency (Age): Clients specify a target recency T_A. For an object O_i, this corresponds to r_i in Section 2.5.

There are many possible recency metrics that could be chosen. We choose as our recency metric the number of times the object has been updated at the remote server since it was cached. We refer to this metric as *age*. We briefly discuss our choice of recency metric. There have been many different metrics described and used in the literature, e.g., [8, 35, 52, 65]. One metric is the amount of time elapsed since the cached object became stale [35]. Obsolescence measures age in terms of the number of insertions, deletions, and modifications [52]. Research reported in [65] considers *age*, the number of times an object has been updated at the remote server. The choice of recency metric depends on the semantics of the application and the types of updates that occur, so each of the above metrics is useful in different circumstances. We selected *age* as the recency metric [65] because we believe this metric is useful for a variety of applications. In the remainder of this dissertation, we use the terms *recency* and *age* interchangeably.

4.1.3 Profile: Parameterized Decision Function and Profile-Based Downloading

Given a request for an object O_i, to determine if it needs to be downloaded we must do the following:

1. Estimate the age of the cached copy of the object (R_i), and the latency of downloading a fresh object (L_i).

2. Compute scores for both using the cached object and downloading a fresh object from a remote server. These scores are a function of R_i, r_i, L_i, and l_i.

3. Choose whether to download a fresh object or deliver the cached object to

the client.

We note that latency can be estimated using cost models such as those in [4, 55]. We present a heuristic for estimating age R_i in Section 4.2.4, and we consider more sophisticated policies for estimating age in Chapter 5. We now describe how to compute scores and determine whether or not to download an object O_i, given latency and recency estimates L_i and R_i.

Scoring Function Profile uses a parameterized function that incorporates client profiles into the decision of whether to download a requested object or to use a cached copy. First, we describe the decision function. We note that there are many different functions that could be used. We chose this particular function because it has several desirable properties. First, it can be tuned to provide an upper bound with respect to latency or recency. Second, when it is impossible to meet both targets, two parameters can be set to reflect a tradeoff, i.e., the relative importance of meeting each of the targets.

Our function first calculates a score for both recency and latency as follows:

$$\texttt{Score}(\texttt{T}, \texttt{x}, \texttt{K}) = \begin{cases} 1 & \text{if } \texttt{x} \leq \texttt{T} \\ K/(x - T + K) & \text{otherwise} \end{cases}$$

T is the target value of recency or latency, x is the actual value, and K is a constant ≥ 0 that is used to tune the rate at which the score decreases. Let $\texttt{K}_L$ be the K value used to control the latency score, and let $\texttt{K}_A$ be the K value used to control the recency score. Note that the K values are set automatically by the browser based on client preferences, using a graphical interface (described in Section 4.1.4).

Combined Weighted Score The decision function is a separable function that combines the scores for recency and latency. It can also be tuned to capture the latency-recency tradeoff for a client or application. This is done by assigning (relative) weights to the importance of latency and recency. The sum of the weights must equal 1. For some applications it may be more important to meet the recency target; for others it may be more important to meet the latency target. Let `w` be the weight assigned to meeting the latency target, and let `(1 - w)` be the weight assigned to meeting the recency target. Given `Age`, the estimated age of object O_i (corresponding to R_i), and `Latency`, the estimated latency of O_i (corresponding to L_i), we compute the *combined score* of an object as follows:

$$\texttt{CombinedScore = (1 - w)*Score(T}_A\texttt{, Age, K}_A\texttt{) + w*Score(T}_L\texttt{,Latency, K}_L\texttt{)}$$

Profile-Based Downloading Our algorithm Profile uses the combined scoring function to make the decision of whether or not to download an object. When an object is requested, we compute the score of either downloading the object (`DownloadScore`) or using the cached copy (`CacheScore`). The Profile policy is as follows: *When an object is requested, if* `DownloadScore` $>$ `CacheScore`*, the object is downloaded from the remote server. Otherwise the cached copy is used.*

We compute `DownloadScore` for an object as follows: Recall that when an object is downloaded, its `Age` is 0 because the remote server always provides the most recent data. Therefore, `Score(`T_A`, Age, `K_A`)` is always 1.0. `Latency` is the estimated latency of downloading the object from a remote server. We note that latency can be estimated using cost models such as those in [4, 55]. Thus, `DownloadScore`, the combined score of downloading an object, is

$$\texttt{DownloadScore} = (1 - \texttt{w}) * 1.0 + \texttt{w} * \texttt{Score}(\texttt{T}_\texttt{L}, \texttt{Latency}, \texttt{K}_\texttt{L}) \quad (4.1)$$

We now consider CacheScore. Recall that when an object is read from the cache, its Latency is 0. Therefore, Score(T_L,Latency, K_L) is always 1.0. Age is the estimated age of the cached object. CacheScore, the combined value of using a cached copy of an object, is

$$\texttt{CacheScore} = (1 - \texttt{w}) * \texttt{Score}(\texttt{T}_\texttt{A}, \texttt{Age}, \texttt{K}_\texttt{A}) + \texttt{w} * 1.0 \quad (4.2)$$

4.1.4 Choosing a Profile

The success of latency-recency profiles depends on the ease of creating a profile. If setting the parameters is complicated and time consuming, clients will be less inclined to use profiles. We describe an interface that allows clients to express the most appropriate profiles for their applications.

Default Profiles The default profile has its targets set to provide identical performance to TTL. This corresponds to settings of w=0 and T_A=0. Note that with these settings, the T_L, K_A, and K_L values are irrelevant. This TTL setting is what many caches currently provide, e.g., proxy caches [24]. For those clients who wish to explicitly trade recency for improved latency, the browser will present a small number of parameter settings to the client, and let the client choose the settings that best suit their needs for each application. We describe how this choice can be made using the graphical interfaces of Figure 4.1 and Figure 4.2.

Figure 4.1 illustrates the latency-recency tradeoffs of three possible parameter settings. In these graphs, we plot the recency (age) of a cached object as x

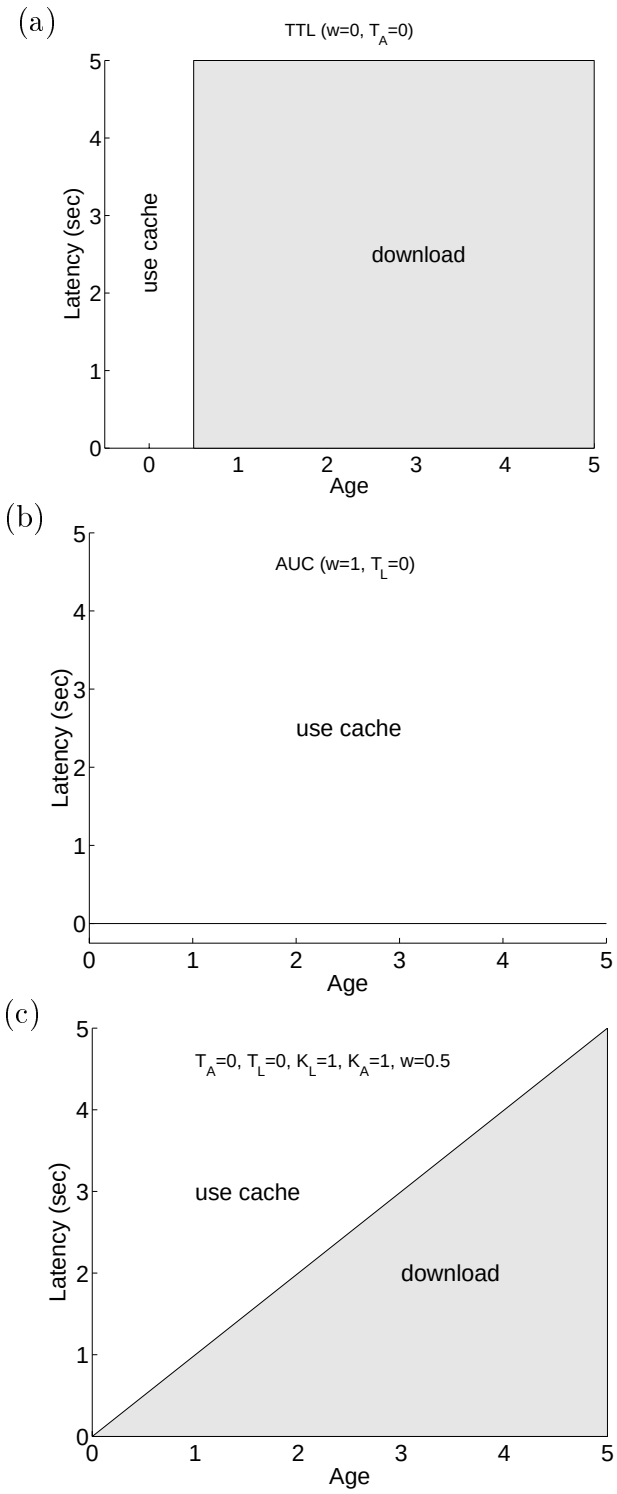

Figure 4.1: Behavior of (a) TTL (b) AUC (c) Profile with T_A=T_L=0, w=0.5, and K_A=K_L=1

number of updates on the x axis and the latency of downloading the object as `y` seconds on the y axis. If a point `(x,y)` lies in the shaded area, then the object is downloaded. If `(x,y)` lies in the white area, then the object is read from the cache. Figure 4.1(a) displays the behavior of the default TTL profile (`w`=0, $\mathtt{T}_A$=0) to the user. Note that when `w`=0, the values of T_L, K_A, and K_L are irrelevant. Any object with 0 updates is served from the cache, while any object with 1 or more updates is downloaded. Figure 4.1(b) displays the behavior of AUC (`w`=1, $\mathtt{T}_L$=0), where the client will tolerate any amount of staleness to minimize access latency. AUC always uses the cached object (no shaded area), regardless of the number of updates.

Tuning Profiles For clients who desire performance between the two extremes of TTL and AUC, there are many profiles that can be chosen. An example of a profile between these extremes has parameters (`w` = 0.5, $\mathtt{K}_A$ =$\mathtt{K}_L$=1, and $\mathtt{T}_A$ = $\mathtt{T}_L$=0). Figure 4.1(c) displays the behavior of this profile. We see that the decision function captures the latency-recency tradeoff. When objects have higher access latencies, users may tolerate older cached objects (white area). Conversely, as the cached object becomes more stale, users are willing to wait longer to download a fresh object (gray area).

The profiles illustrated in Figure 4.1 can be tailored further. This is straightforward to do in our framework. For example, consider a client who wishes to receive data with recency of no more than 1 update. Such a client could choose the default TTL as in Figure 4.1(a), but change the T_A value from 0 to 1, i.e., (w=0, T_A = 1). This would result in any object with 2 or more updates being downloaded, rather than 1 or more updates as shown in Figure 4.1(a).

Upper Bounds The profiles of Figure 4.1 do not provide any upper bounds on latency or recency. For clients who desire even greater control over the settings of their profiles, the values for w and K_L and K_A can be chosen to provide upper bounds. Clients do not need to manually choose w and K values. Instead, clients can choose an upper bound for either latency or recency. They are then aided by a graphical interface (similar to Figure 4.2) that illustrates the tradeoff for settings of w and K values, and allows them to make the appropriate choice.

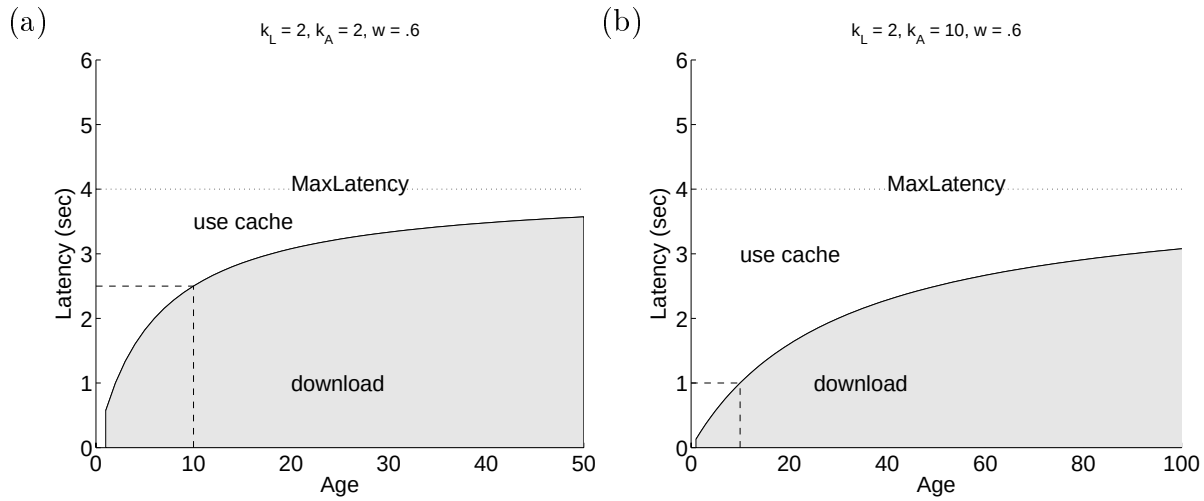

Figure 4.2: Upper Bounds on the Latency-Recency Tradeoff

An upper bound for either latency or recency can be chosen. In particular, assigning a higher weight to latency (w > 0.5) places an upper bound on the latency of a downloaded request, and assigning a higher weight to age (w < 0.5) places an upper bound on the age of an object delivered to the client from the cache.

We illustrate with an example. Suppose a client has a profile of (w = 0.6, T_A=0, T_L=0). This means the weight of latency (w) is 0.6 and the weight of

recency $(1 - \mathtt{w})$ is 0.4.

The combined score of downloading the object is:

$$0.6 * \mathtt{DownloadScore} + 0.4 * 1.0$$

The combined score of using the cached object is:

$$0.6 * 1.0 + 0.4 * \mathtt{CacheScore}$$

Note that when DownloadScore=0, the combined score of downloading the object is 0.4, and when CacheScore=0, the combined score of using the cached object is 0.6. These are lower bounds on the combined score. Therefore, when the combined score of downloading the object is less than 0.6, it will always have lower score than using the cached object. Therefore, the cached object will be used, regardless of the value of CacheScore.

We solve for DownloadScore as follows:

$$0.6 * \mathtt{DownloadScore} + 0.4 < 0.6$$

If $\mathtt{DownloadScore} < 1/3$, the object will always be read from the cache, regardless of the age of the cached data. This property allows clients to specify a bound on latency. Let MaxLatency be the maximum acceptable latency. The value of K_L can be set such that $\mathtt{Score}(\mathtt{T_L}, \mathtt{MaxLatency}, \mathtt{K_L}) = 1/3$.

We solve for K_L when x_L=MaxLatency such that:

$$\mathtt{K_L}/(\mathtt{x_L} - \mathtt{T_L} + \mathtt{K_L}) = 1/3$$

When we solve the equation, we have:

$$3K_L = x_L - T_L + K_L$$

$$K_L = (x_L - T_L)/2 \tag{4.3}$$

Therefore, when x_L=MaxLatency, we have K_L=(MaxLatency-T_L)/2. This value of K_L gives a scoring function that guarantees the latency is $\leq$ MaxLatency.

In Figure 4.2, $w = 0.6$ and T_L, T_A=0. MaxLatency is 4 seconds, so by Equation 4.3 we have K_L=2. The choice of K_A in Figures 4.2 (a) and 4.2 (b) illustrate the latency-recency tradeoff that the clients can select that controls how the latency asymptotically approaches the upper bound of 4 seconds. The choice of values makes Profile more aggressive to download data as reflected by the larger shaded area.

4.2 Experiments

We use both trace data and synthetic data to compare Profile against three algorithms, TTL, AUC, and SSI (described in Chapter 2. Our simulation models the proxy cache architecture of Figure 2.2. These results also apply to browser caches, and to CDNs and portals, if we do not consider the additional time to send data from the cache to a client. We first describe the details of these algorithms. We then describe the details of both the trace and synthetic datasets. Finally, we present our results. Our key results are as follows:

- Profile significantly reduces bandwidth consumption compared to all approaches for both trace and synthetic data. Compared to TTL, Profile reduces bandwidth consumption with only a slight increase in the amount

of stale data delivered to clients (trace data). Profile also provides better recency than AUC (trace and synthetic data).

- Profile can benefit from an increased cache size more than either TTL or AUC (trace data). AUC cannot deliver recent data when the cache size is large, while TTL cannot utilize a larger cache size to reduce latency. Profile can exploit increasing cache size to reduce both age and latency.

- In the presence of surges, Profile improves latencies for *all* clients, even for clients who require the most recent data.

- Our sensitivity analysis shows the effects of tuning profile parameters on both latency and recency. When T_A increases, Profile becomes less sensitive to changes in T_L because fewer objects need to be downloaded. When T_L increases, Profile becomes more sensitive to changes in T_A because more objects can be downloaded to meet T_A. We also show how changing K_A and K_L values affects the sensitivity of Profile to T_A and T_L values. Finally, we show that Profile can provide upper bounds specified by the profile parameters.

4.2.1 Algorithms

We consider the following algorithms:

- TTL: This is the cache consistency mechanism currently used in most proxy caches [24, 30, 47, 57]. Cached objects are assigned a TTL value which is an estimate of how long they will be fresh in the cache. The TTL approach guarantees that all cached objects are up-to-date if the TTL estimate is

accurate. It uses two parameters, `UpdateThreshold` and `DefaultMax`; these are explained in Section 4.2.2.

- AUC: We implemented a modified version of the prefetching strategy presented in [35]. We relax their assumption that all objects must be in the cache. Instead, we refresh objects that are currently in the cache in a round robin manner. On a cache miss, objects are downloaded from a remote server. This strategy has the advantage of being straightforward to implement at the cache, and was shown to be near optimal in [35]. Objects are validated in the background at a specified `PrefetchRate`, and only validated objects that have been updated at the remote server are downloaded.

- Profile: This is implemented as was described in Section 4.1.3. The decision function uses the *estimated latency* of downloading objects, and the *estimated age* of cached objects. We describe how to compute these for the NLANR trace data in Section 4.2.2. The settings of the profile parameters are described with the results in Section 4.2.4.

- SSI-Msg: We consider two variations of SSI. In the first, *SSI-Msg* the server sends invalidation *messages* to a cache whenever an object is updated, but does not send the actual object to the cache. If the cached object is subsequently requested, the updated object is downloaded from the server. Note that this approach is comparable to TTL with accurate expiration times. This approach was shown to consume a comparable amount of bandwidth to TTL in [78].

- SSI-Obj: In the second variation, *SSI-Obj*, the server sends all updated objects to the cache. This consumes more bandwidth than *SSI-Msg* but

guarantees that all cached objects will be up-to-date, which reduces the latency of requests.

4.2.2 Data

We now describe the trace and synthetic data used in our experiments.

NLANR Trace

We used trace data from NLANR [50]. We describe the details of the preparation of this data in Appendix A. This data was gathered from a proxy cache in the United States in January 2002. We considered approximately 3.7 million requests made over a period of 5 days. We performed preprocessing on the NLANR trace data to prepare it for the experiments. Specifically, the trace data did not report on the times objects changed, which we need to make downloading decisions and to determine the recency of cached objects. Our solution to this problem was to create an "augmented" trace using the workload from the original NLANR trace data. Over a period of 5 days, we replicated the trace workload by sending requests to the servers in the traces at (approximately) the same time of day as in the original workload. The requests were made from the domain `umiacs.umd.edu` which is connected to its ISP via a high speed DS3 line with a maximum bandwidth of 27 Mbps. When each requested object arrived, we logged the latency of the request and the time the object was last modified (when available). We used the logging mechanism provided by the Squid cache[24] to create the augmented trace, but did not cache any objects. This augmented trace data provided the information we needed for this study. We describe additional properties of this data in Appendix A. We summarize some key parameters in

Parameter	Value
trace duration	129 hours
mean request arrival	8 requests/sec
mean object size	2.1 KBytes
total objects	1365K
total requests	3707K

Table 4.1: Parameters in NLANR trace data

Table 4.1.

In our trace-based experiments, we cached only objects that had last modified information available and were not labelled uncacheable. For the TTL algorithm, to estimate the TTL of an object, we use the policy implemented in Squid [24]. When an object's last-modified timestamp is available, Squid estimates the lifetime of an object using the adaptive TTL technique [30, 57]. In adaptive TTL, an object's `TTL` is estimated to be proportional to the age of the object at the time it was cached. The exact value depends on a parameter `UpdateThreshold`. We used an `UpdateThreshold` of 0.05, which is representative of values used in practice [24].

We calculate an object's `TTL` as follows:

$$\texttt{TTL} = (\texttt{CurrentTime} - \texttt{LastModifiedTime}) * \texttt{UpdateThreshold} \qquad (4.4)$$

Given the value of `TTL`, an object is no longer valid after its `ExpirationTime`. We compute an object's `ExpirationTime` as:

$$\texttt{ExpirationTime} = \texttt{CurrentTime} + \texttt{TTL} \tag{4.5}$$

If the estimated `ExpirationTime` exceeds a default maximum value `DefaultMax`, then an object's `TTL` is estimated as `DefaultMax`. As in the Squid cache implementation we use a `DefaultMax` of 3 days.

For the Profile algorithm, we need estimates of the latency and recency of objects to make a downloading decision. We estimated the latency of an object as the average latency over all previous requests, which was shown to perform well in [4]. We estimate the age of cached objects as follows: we first estimate an `UpdateInterval`, the estimated length of time between updates. We define `UpdateInterval` as:

$$\texttt{UpdateInterval} = \texttt{ExpirationTime} - \texttt{LastModifiedTime} \tag{4.6}$$

We defined the age of a cached object as:

$$\textit{Age} = (\texttt{CurrentTime} - \texttt{LastModifiedTime})/\texttt{UpdateInterval}. \tag{4.7}$$

We describe more sophisticated policies to improve the accuracy of estimating the age of objects in Chapter 5.

For AUC, all cache hits were served directly from the cache, and we validated objects in the background at a specified `PrefetchRate`. We considered AUC with two different prefetch rates, 60 objects per minute (AUC-60) and 300 objects per minute (AUC-300). Note that for TTL and Profile we did not perform any prefetching in this study.

On a cache hit, we need to determine if an object is fresh or stale. We determined an object's freshness as follows:

Parameter	Value
mean request arrival	8 requests/sec
mean latency	500 msec
median latency	200 msec
mean object size	2-10 Kbytes
update interval	10 min - 2 hours
total requests	172800
world size	100000
α	0.7

Table 4.2: Parameters in synthetic trace

1. For all schemes, an object was fresh if the object's `last-modified` time was unchanged since the previous request.

2. For TTL and Profile, an object was stale if its `last-modified` time had changed.

3. For AUC, we also need to consider the effects of prefetching. If the object's `last-modified` time had changed and was more recent than the time the object was last prefetched, the object was stale. Otherwise it was fresh.

Synthetic Trace

To complement our trace results and study the performance of profiles, we also performed simulation studies using synthetic data, where we control updates at remote servers, and use more accurate age information. We used the following parameters to generate the synthetic data: they are summarized in Table 4.2.

- *Update Interval* is the average length of time between consecutive updates. In our simulation this value ranged from once every 10 minutes to once every 2 hours.

- *Estimated Latency* is the expected end-to-end latency of downloading the object from the remote server. We modeled the latencies of objects using latency distributions from NLANR traces [50]. To reduce the effects of network and server errors in this data we considered only requests with latencies of less than 5000 msec. The distribution of these values was highly skewed, with a median of approximately 200 msec and a mean of approximately 500 msec. 90% of the requests had latencies less than 1400 msec.

- *Workload* is the average number of requests per minute. We report on a workload of 8 requests/sec (480 requests/minute), which is representative of many cache workloads[50]. We ran simulations for 6 hours of simulation time for a total of $\approx$172800 requests.

- *World Size:* We considered a world of 100,000 objects with a popularity following a Zipf-like distribution. The ith most popular object had popularity proportional to $1/i^{\alpha}$, where α is a value between 0 and 1.0. We generated a distribution with α= 0.7, which was typical of traces analyzed in [17].

We note that for TTL and Profile, for the synthetic data we assumed that the cache had accurate expiration times (TTL estimates) for all objects. We use the trace data to compare TTL, AUC, and Profile in the real world case where estimates are often inaccurate. We use the synthetic trace to compare the performance of TTL, AUC, Profile, and SSI because it provides information about when updates occur at servers.

4.2.3 Setup and Metrics

We implemented our simulation environment in C++. We ran simulations and experiments with trace data on a Sparc 20 workstation running Solaris 2.6. We assumed the cache was initially empty.

For the synthetic trace, we ran simulations for 2 hours of simulation time to warm up the cache, then ran them for an additional 6 hours. For the trace data, we used the first 12 hours of the trace to warm up the cache, then collected data on the remainder of the trace. We repeated each simulation 10 times to verify the accuracy of our results, and validated that our results satisfied the 95% confidence intervals. For both the NLANR and synthetic traces, we consider cache sizes ranging from 1% of the world size to an infinite cache. We first report on results for an infinite cache. We then consider the effects of varying cache size on the performance of all approaches. We used the Least Recently Used (LRU) policy to replace objects when the cache was full; this is commonly used in practice [24].

We report on the following metrics:

- Validation messages (vals): This is the number of messages that were sent between cache and remote servers. For TTL, AUC, and Profile, a validation message is sent from the cache to a server to check for updates. The requested object was only downloaded if it had actually been updated. For SSI-Msg, a validation message is sent from a server to a cache to invalidate cached objects. Messages are typically much smaller than the actual objects. We note that for SSI-Obj, the server sends the actual objects to the cache, so no messages are sent.

- Downloads (Useful Validations): This is the number of requested objects

that were validated and subsequently downloaded because they were stale in the cache. For SSI-Obj, this includes all objects that were updated at remote servers and sent to the cache. For SSI-Msg,TTL and Profile, this includes requested cached objects that were not sufficiently recent in the cache. For AUC, this includes objects that were prefetched (validated) in the background and were downloaded because they were stale.

- Freshness misses: For the trace data, we also report on freshness misses [40]. These are objects that were in the cache and were validated at the remote server, but had not actually been modified since they were cached. Since freshness misses add unnecessary latency to requests, it is important to minimize this number. In many cases the latency of a freshness miss can be comparable of that to a cache miss [40].

- Stale Hits: For the trace data, this is the number of objects served from the cache (without validation), but that had actually been updated at the remote server.

- Age: This is the average age of objects delivered to clients, i.e., the number of times they were updated at the remote server. Objects that were downloaded from a server always had an age of 0.

- Latency: This is the average latency of the requests in msec.

4.2.4 Comparison of Profile to Existing Policies

Our first set of results shows the benefits of using Profile for an infinite cache. We first show simulation results using the synthetic trace. We then use the NLANR trace to compare Profile to TTL and AUC. The NLANR trace reflects

	SSI-Obj	SSI-Msg	TTL
Val. Msgs	0	161768	67170
Downloads	161768	67170	67170
AvgAge	0	0	0
StaleHits	0	0	0
	AUC-60	AUC-300	Profile
Val. Msgs	21600	100797	15932
Downloads	16158	75833	15932
AvgAge	2.09	0.61	1.38
StaleHits	112492	74548	96281

Table 4.3: Results for Experiments with Synthetic Trace

the situation when TTL estimates are inaccurate, which is often the case in practice. We do not study SSI on the NLANR trace since the trace does not provide a complete history of updates at remote servers. Objects at servers may have been updated multiple times between two consecutive requests for the object in the trace.

In these experiments, all clients used a single profile = (w = 0.5 T_A =1 update, T_L=1 second, K_L, K_A = 1). Recall that with w=0.5, neither latency nor recency is favored in the tradeoff. We consider the effects of varying w, T_A, T_L, K_A, and K_L in Section 4.2.7.

Synthetic Trace The number of validations and downloads for the simulation study with synthetic trace is shown in Table 4.2.4. The first observation is that SSI-Obj consumes the most bandwidth because it sends a large number of objects

	TTL	AUC-60	AUC-300	Profile
Validation Messages	151367	378312	1891560	92943
Useful validations	24898	933	2810	22896
Freshness Misses	122074	279349	327776	67601
Avg Est.Age	0	18.4	11.1	0.87
Stale Hits	4282	31285	22897	7704

Table 4.4: Results for Experiments with NLANR Trace

to the cache to keep the cache up to date. This is shown in the Downloads row. SSI-Msg and AUC-300 also consume significant amounts of bandwidth compared to Profile. While they download fewer objects than SSI-Obj, they still send many validation messages. In contrast, Profile performs fewer validations *and* fewer downloads than all other approaches. We will use the NLANR trace data to further quantify the bandwidth savings of Profile relative to TTL and AUC.

The average ages of objects and number of stale hits are also shown in Table 4.2.4. These results show that while AUC-60 and Profile have a comparable number of downloads, AUC-60 does so at the cost of delivering significantly less recent data. AUC-60 delivers objects with an average age of 2.09 updates compared to 1.38 updates for Profile. AUC-60 also provides nearly 20% more stale hits than Profile. AUC-300 provides better recency (0.61) than Profile. However, it does so at the cost of validating 600% more objects than Profile (100797 vs. 15932) and downloading nearly 500% more objects(75833 vs. 15932).

NLANR Trace Our NLANR trace results further compare TTL, AUC, and Profile in the real-world case where TTL estimates are often inaccurate. Table

2 shows the number of validations for TTL, AUC, and Profile, for an infinite cache. The first observation is that both variants of AUC validate significantly more objects than either TTL or Profile. Recall that AUC validates objects at the specified `PrefetchRate`. TTL also validates many more objects than Profile.

The number of useful validations and freshness misses are shown in the second and third lines of Table 4.2.4[1]. We note that for AUC, for a fair comparison we measured useful validations and freshness misses only for prefetched objects that were subsequently requested.

A key observation is that TTL has nearly twice as many freshness misses as Profile (122074 vs. 67601). In these cases, TTL adds latency to requests without improving the recency. In contrast, Profile can significantly reduce the number of freshness misses by 45% ($\approx$ 60,000) with only a small increase in the number of stale hits ($\approx$ 4000 more than TTL). We note that these results do not include the approximately 196,000 requests (described in Appendix A where we could not accurately determine the cached object's freshness from the augmented trace. Based on the original NLANR trace data, many of these appear to have been freshness misses. Thus, the potential reduction in freshness misses from using profiles may be even greater than 45%.

Another key observation is that both variants of AUC perform many more freshness misses than either TTL or Profile. Further, AUC performs very few useful validations for objects that are subsequently requested (less than 3000 for AUC-3000 vs. 24898 for Profile). Thus, AUC can consume large amounts of bandwidth to keep the cache refreshed while doing little to improve the recency

[1]In some cases the trace did not contain a last modified date to determine if a validation was useful, therefore the sum of these values is less than the validation messages.

of data delivered to clients. We note that a more intelligent prefetching policy described in [39] can reduce the total number of freshness misses by 25%. However, this prefetching policy replaces each online request with up to two offline requests, so total bandwidth consumption and server loads increase. In contrast, Profile provides a greater reduction in the number of freshness misses (45%) and reduces total bandwidth consumption and server load.

Since the NLANR trace does indicate how many times servers actually modified objects, we must estimate the age, i.e., number of times a stale object was updated at a server since it was cached. We compute age= (`CurrentTime - LastModifiedTime`)/`UpdateInterval` (equation 4.7), where `UpdateInterval` is estimated as defined in equation 4.6. While this is an estimate, it gives an idea of how out of date the stale objects were.

Both variants of AUC prefetch a large number of objects, while still delivering many stale objects to clients. The average estimated age of the stale hits is shown in the last line of Table 2, and show that AUC can deliver very out of date objects. This is because the prefetching strategy for AUC prefetches all objects with equal frequency, which may cause frequently updated objects to become very out of date. While this prefetching strategy is near optimal for minimizing the number of stale hits[35], our results clearly show that AUC may nevertheless result in very stale data. Thus, prefetching may not be appropriate for applications that cannot tolerate stale data, especially when the data is updated frequently.

Summary To summarize, our main results are as follows:

- Profile validates significantly fewer objects than TTL, AUC, or SSI.
- Profile provides fresh data to clients in many cases due to conservative TTL

estimates. The number of stale hits is not significantly higher than for TTL.

- Profile provides more recent data than AUC. In contrast, AUC may not be approriate when fresh data is required because it may deliver objects that are significantly out of date.

4.2.5 Effect of Cache Size

We now use the NLANR trace to measure the performance of TTL, AUC, and Profile for varying cache sizes. We varied our relative cache size from 1% of the world size to 100% of the world size (i.e., an infinite cache). We show that Profile can better utilize cache size to reduce latency (compared to TTL) and to reduce age (compared to AUC).

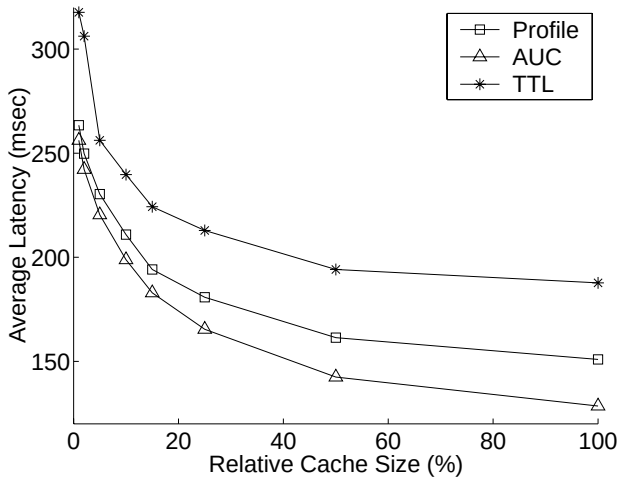

Figure 4.3: Effect of Cache Size on Average Latency

The average latency for Profile and the baseline algorithms are plotted in Figure 4.3. The first observation is that both Profile and AUC better utilize increased cache size to reduce latency. While increasing the cache size increases

the number of objects that can be cached, objects that expire in the cache must always be validated for TTL. Increasing the cache size does not decrease the number of stale objects in the cache, so TTL does not benefit significantly from a larger cache. In contrast, Profile and AUC can benefit more from an increased cache size. While objects in the cache may be stale, they may still be useful to some clients.

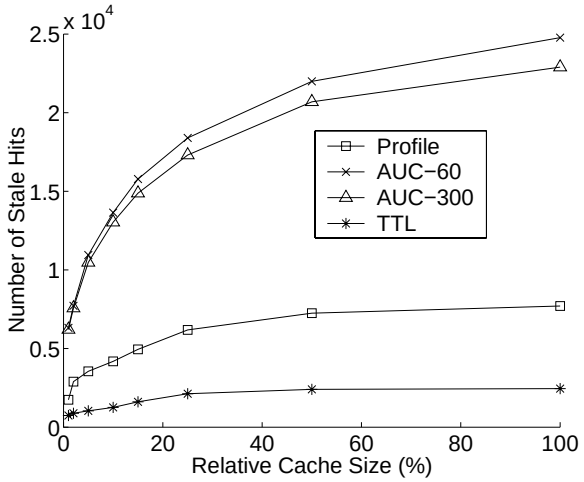

Figure 4.4: Effect of Cache Size on Number of Stale Hits

Figure 4.4 shows the number of stale hits. As the cache size increases, the stale hits for both AUC-60 and AUC-300 increase dramatically. This is because prefetching for AUC does not scale well and a greater number of client requests are being serviced by (possibly stale) cached objects, so the number of stale hits increases. This shows that the reduced latency of AUC comes at the high cost of delivering very stale data. In contrast, the number of stale hits for Profile increases by a much smaller amount. In summary, AUC cannot utilize a large cache size to reduce age and delivers very stale data. Similarly TTL cannot

utilize a larger cache size to reduce latency. In contrast, Profile is flexible and can exploit increasing cache size to reduce both age and latency.

4.2.6 Effect of Surges

Under normal workloads, there is typically sufficient bandwidth and server capacity to handle all requests. However, from time to time networks or servers may experience "surges", i.e., a period of time during which the available resource capacity exceeds the demand. During surges, many request will be backlogged and their processing may be delayed significantly. As an example, we consider the case where there is insufficient bandwidth between a cache and the servers. In this case, the servers will attempt to deliver many objects simultaneously, which will cause delays delivering the objects to the cache. This could occur in a proxy cache if a surge in remote requests saturates the bandwidth between the Internet and the cache. It could also occur in an application server cache if many clients make requests to the server simultaneously.

Our next experiment is a simulation using a synthetic trace that compares Profile to TTL in the presence of surges. A surge is represented by a capacity ratio. The capacity ratio is the ratio of available resources per second to the resources required per second. For example, during a surge period, if a server can handle 10 requests per second and requests arrive at the rate of 20 requests per second, then the capacity ratio during this period is 1/2. A capacity ratio of 1 means there are sufficient resources to handle all requests, and requests will incur no extra delay as a result of the surge. However, if this ratio is less than 1, performance can severely degrade.

We consider two groups of clients. The first group, `MostRecent`, has Profile =

(w = 0.5, $T_A = 0$ updates, $T_L = 1$ sec, K_L, $K_A = 1$). The second, LowLatency, has Profile = (w = 0.5,$T_A = 1$ update, $T_L = 0$ sec, K_L, $K_A = 1$). In our simulation, we considered a surge with duration 30 seconds. The request rate is 100 requests per second. We vary the available capacity from 20 to 100 objects per second, i.e., the capacity ratio varies from 0.2 to 1.0. For simplicity, we assume no requested objects are evicted from the cache during the surge period. We warmed up the cache for 10000 requests at a non-surge workload of 8 requests/sec, then began the surge period and gathered data.

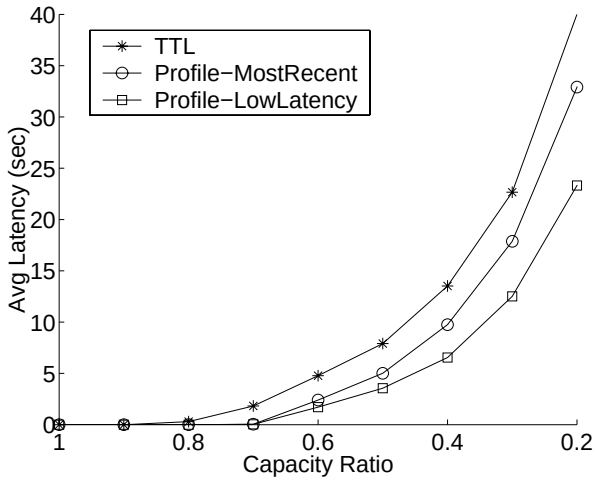

Figure 4.5: Average. Latency during a 30-sec. surge period

Figure 4.5 plots the average latencies of all requests. For TTL, all requests are treated equally, and all objects that have expired in the cache are downloaded. As expected, the latency is very high, especially when the capacity ratio is below 0.5. However, Profile can distinguish between the two groups of clients and better serve their requests. As expected, the latency for some LowLatency clients is significantly lower than for TTL. This is because, when a requested object is in

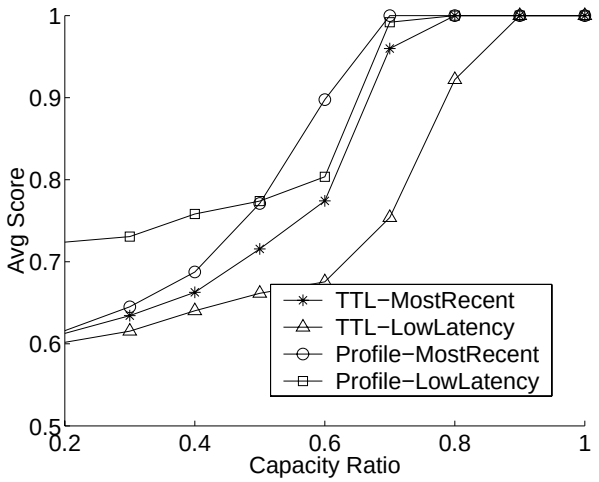

Figure 4.6: Avg. scores during a 30-sec. surge period

the cache, stale data can be delivered to the `LowLatency` clients. Consequently, there is more available bandwidth to serve the other clients. Thus, the latency for the `MostRecent` clients also decreases compared to TTL. Thus, using Profile during a surge can significantly improve access latencies for *all* clients, not just those that can tolerate stale data.

Figure 4.6 plots the scores for both groups of clients, using the scoring functions presented in Section 4.1.3. This gives a measure of client satisfaction with the data and service they receive. As expected, using either TTL or Profile the scores of both groups of clients increase as the capacity ratio approaches 1. The key observation is that using Profile improves not only the score of the `LowLatency` clients, but also the `MostRecent` clients. Profile reduces bandwidth consumption under all workloads, which can reduce the effects of surge periods and improve performance for all clients during surges.

4.2.7 Sensitivity Analysis

In the previous experiments we have considered client profiles with constant T_A and T_L values and with w=0.5 and K_A=K_L=1. In this section, we explore the effects of varying the T_A and T_L values on the average latency of requests and average recency of data. We also consider the effects of adjusting the K_A and K_L values to control the latency-recency tradeoff. Finally, we consider the effects of varying w to provide an upper bound with respect to either recency or latency.

Setup We perform our analysis using the synthetic trace described in Section 4.2.2. In all experiments, we varied the cache size from 1% of the world size of 100,000 objects up to 35% of the world size. Increasing the cache size beyond 35% had little impact on performance for this trace. We ran all experiments for 40000 requests to warm up the cache, and then ran them for an additional 80000 requests and gathered data. Other simulation parameters are identical to those in Table 4.1.

To measure the latency-recency tradeoff, in these experiments we set the recency unit metric to one update, and the latency unit to 100 msec. This means that when K_L=K_A, and w=0.5, clients would trade off 100 msec of latency for every update (rather than 1 second per update as in Figure 4.1).

In the following experiments we report on both average latency and average number of updates for different settings of T_A, T_L, K_A, K_L, and w.

Effect of Varying T_A Values We first consider the effects of varying the T_A and T_L values with $K_A = K_L$ and w = 0.5. This shows the sensitivity of profiles to T_A and T_L when there is no upper bound on either latency or recency. Figures 4.7 and 4.8 plot the latencies and recencies, respectively, for three different T_L

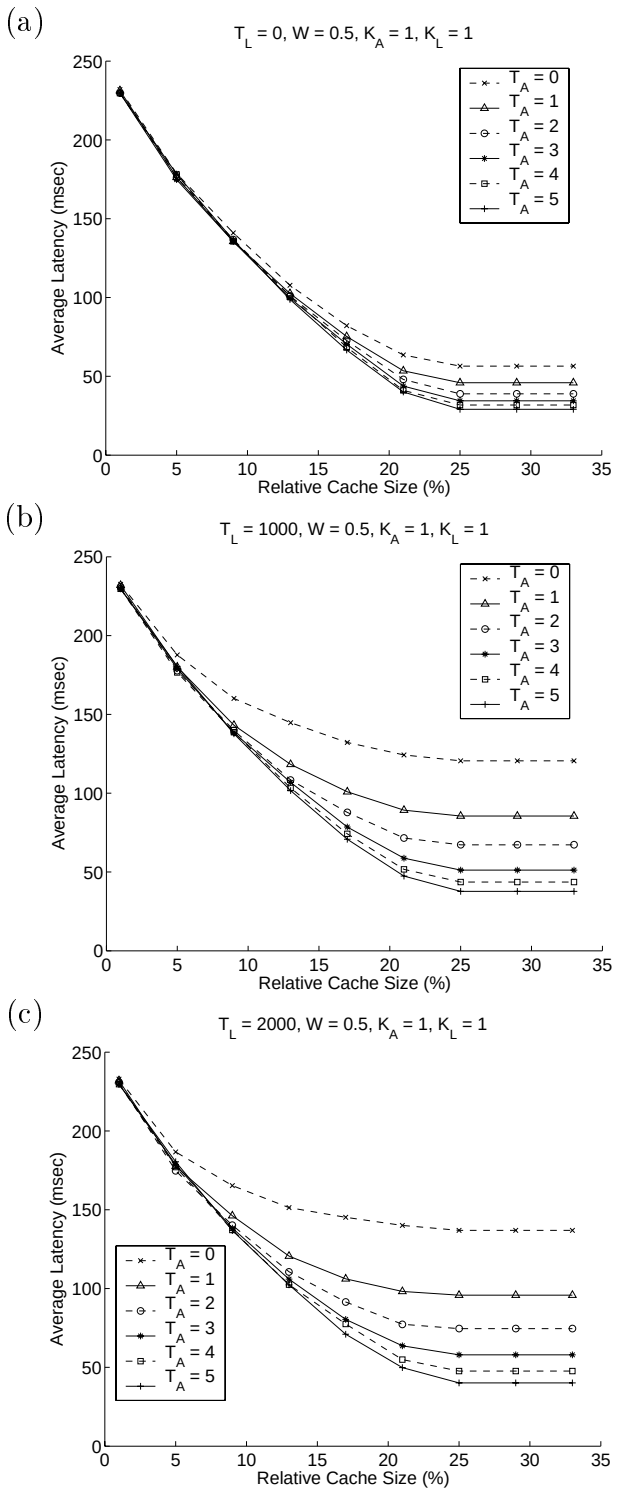

Figure 4.7: Effect of varying T_A values on average latency for (a) $T_L = 0$ (b) $T_L = 1000$ (c) $T_L = 2000$, w=0.5, and K_A=K_L=1

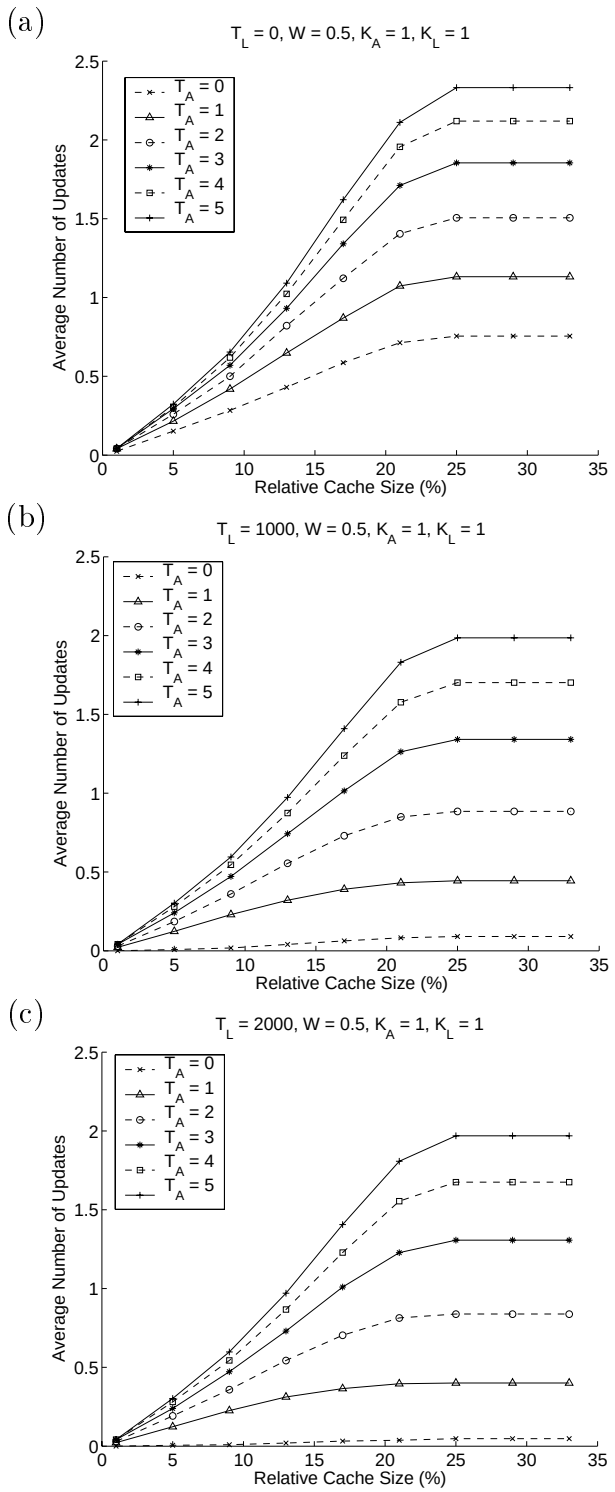

Figure 4.8: Effect of varying T_A values on average number of updates for (a) T_L = 0 (b) T_L = 1000 (c) T_L = 2000, w=0.5, and K_A=K_L=1

values. Figure 4.7(a) plots the average latencies for different T_A values when $T_L = 0$. In this case, minimizing latency is important, thus the T_A values have only a small impact on the average latency. For all T_A values the average number of updates is greater than 0.5 as shown in Figure 4.8(a). In contrast, when T_L is 1000 msec (Figure 4.7(b)), Profile is more sensitive to the T_A values because it can tolerate higher latencies. The T_A values determine when an object needs to be validated. Figure 4.8(b) shows the recency of these requests. When $T_A = 0$, the average number of updates is near 0 because clients can tolerate higher latencies to download fresh data. However, when clients can tolerate stale data, Profile is able to significantly reduce average latencies as shown in Figure 4.7 (b). We observe a similar trend in Figures 4.7(c) and 4.8(c). However, increasing T_L from 1000 msec to 2000 msec has a smaller impact on the average latencies and recencies, because there are relatively few objects with latencies over 1000 msec.

Effect of Varying T_L Values Next, we consider the sensitivity of Profile to varying T_L values when $K_A = K_L$ and w = 0.5. Figures 4.9 and 4.10 plot the respective latencies and recencies (average number of updates) for three different T_A values. When $T_A = 0$ (Figure 4.9(a)), Profile is very sensitive to the T_L values when determining when to download fresh data. This is because clients will not tolerate stale data, so it will always validate any object whose estimated latency is within the T_L value. Thus higher T_L values significantly increase the average latency. When T_L is 0 the average latency is about 50 msec, when T_L is 3500 msec the average latency is about 150 msec. In contrast, for higher T_A values (Figures 4.9(b) and 4.9(c)), Profile is less sensitive to the choice of T_L because clients can tolerate stale data in many cases. The corresponding recencies are plotted in Figure 4.10. When $T_L = 0$, the average number of updates is high because

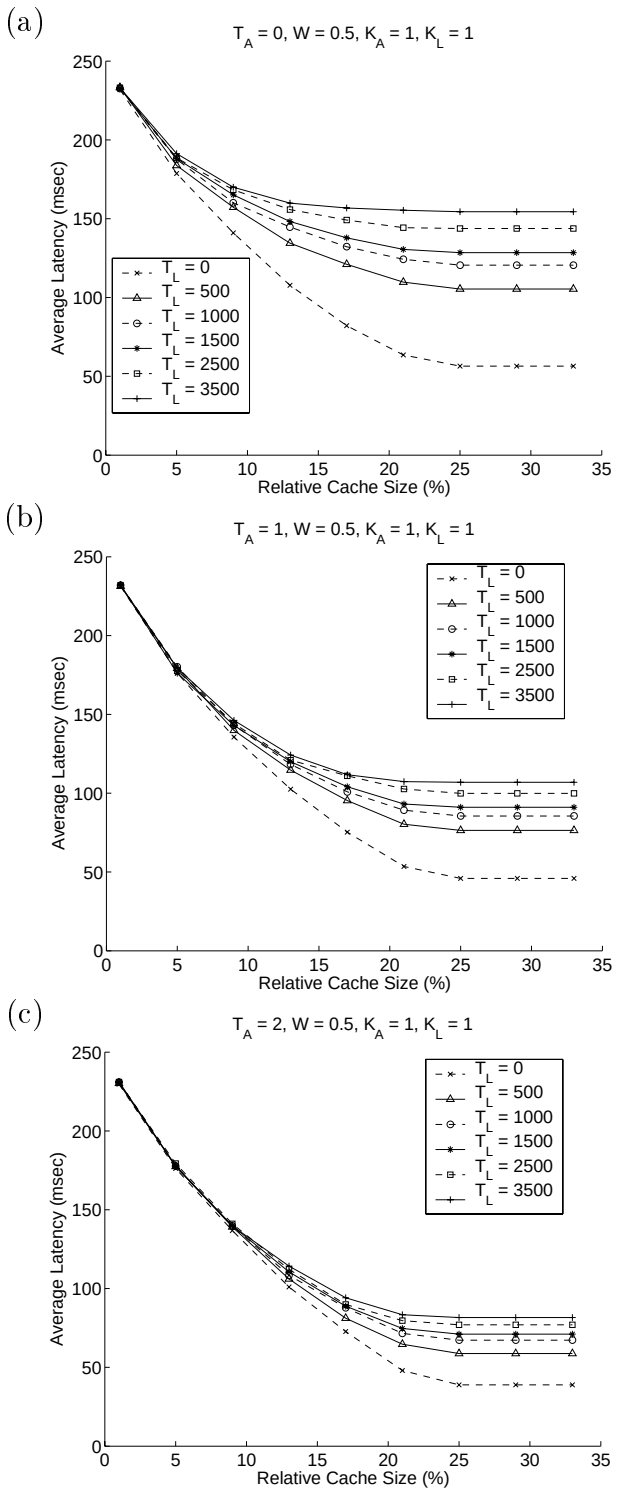

Figure 4.9: Effect of varying T_L values on average latency for (a) $T_A = 0$ (b) $T_A = 1$ (c) $T_A = 2$, w=0.5, and K_A=K_L=1

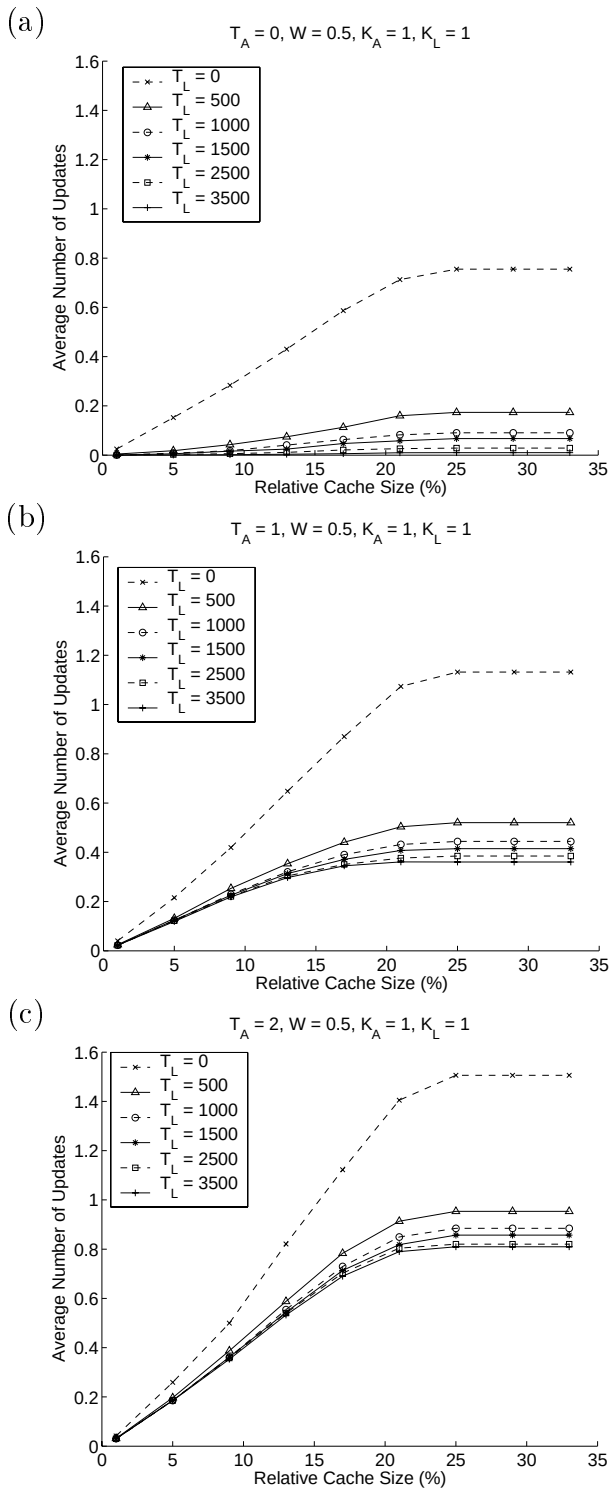

Figure 4.10: Effect of varying T_L values on average number of updates for (a) $T_A = 0$ (b) $T_A = 1$ (c) $T_A = 2$, w=0.5, and K_A=K_L=1

Profile aims to minimize latency, independent of the value of T_A. However, for higher T_L values, Profile aims to meet the T_A values. Since most requests have latency less than 500 msec, higher T_L values have only a small impact on the average number of updates.

Effect of Varying K Values We now consider how varying the K_A and K_L values controls the latency-recency tradeoff and impacts both the latency and recency of client requests. Figure 4.11 shows the effect of varying T_L values when T_A=0 for three different pairs of K values. Note that Figure 4.11(a) is identical to Figure 4.9(a), it is shown here for comparison purposes. When $K_L = 10$ and $K_A = 1$ (Figure 4.11 (b)), the latency score approaches 0 more slowly than the age score, so the average latencies are high for all values of T_L. In contrast, when $K_A = 10$ and $K_L = 1$ (Figure 4.11 (c)), the age score approaches 0 more slowly than the latency score, so Profile can tolerate higher ages. Thus, for lower T_L values, the latency is significantly lower in Figure 4.11 (c). Figure 4.12 plots the recencies of the data. Note that the y-axis in Figure 4.12 (c) is 10 times that of Figure 4.12(b). These graphs show that when K_L is high the average age of the data is low (Figure 4.12 (b)) and when K_A is high the average age of the data is high (Figure 4.12(c)).

We observe a similar trend in Figures 4.13 and 4.14. These graphs consider the effects of varying the K values when T_A=2. The key observation is that when T_A is higher, Profile is less sensitive in variations to T_L values. When $K_L = 10$ (Figure 4.13(b)), the performance is nearly identical for all values of T_L.

Effect of Varying w Values We now consider the effects of changing the w values to provide upper bounds with respect to either recency or latency.

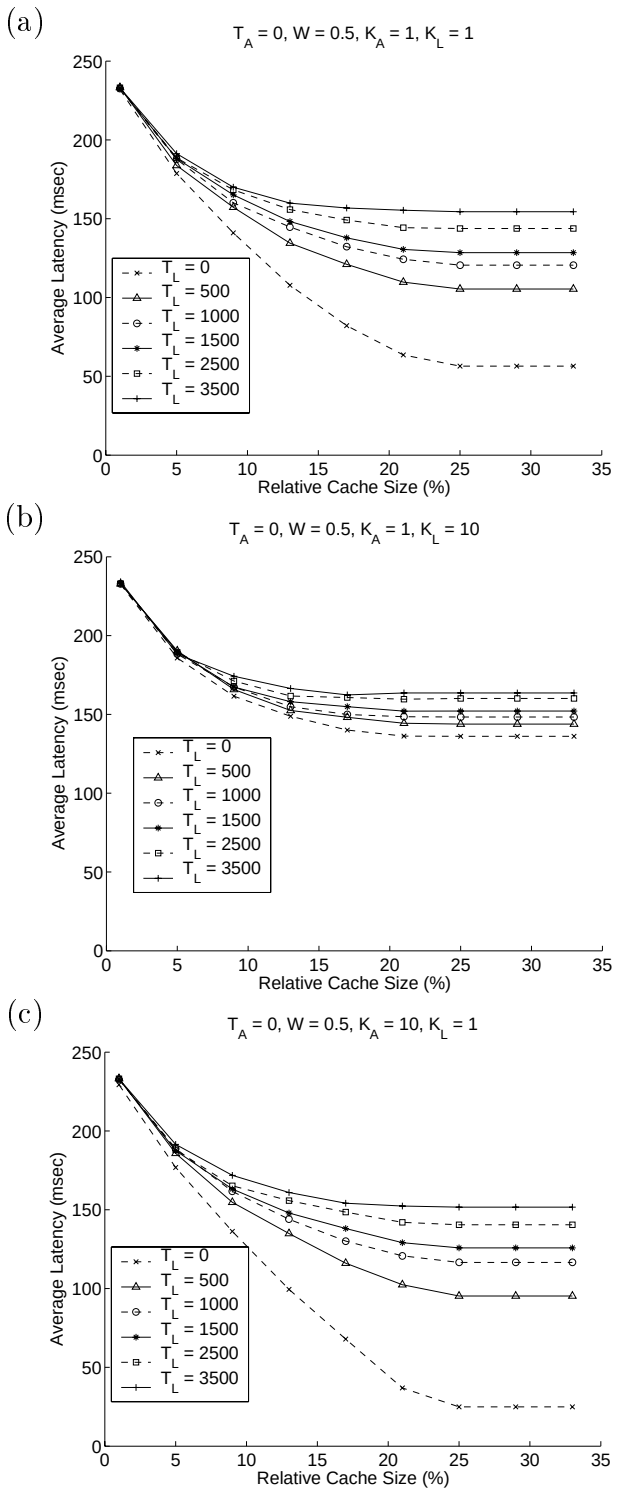

Figure 4.11: Effect of varying T_L values on average latency for T_A =0 (a) K_A=K_L=1 (b)K_A=1, K_L=10 (c) K_A=10, K_L=1, T_A = 0, w=0.5

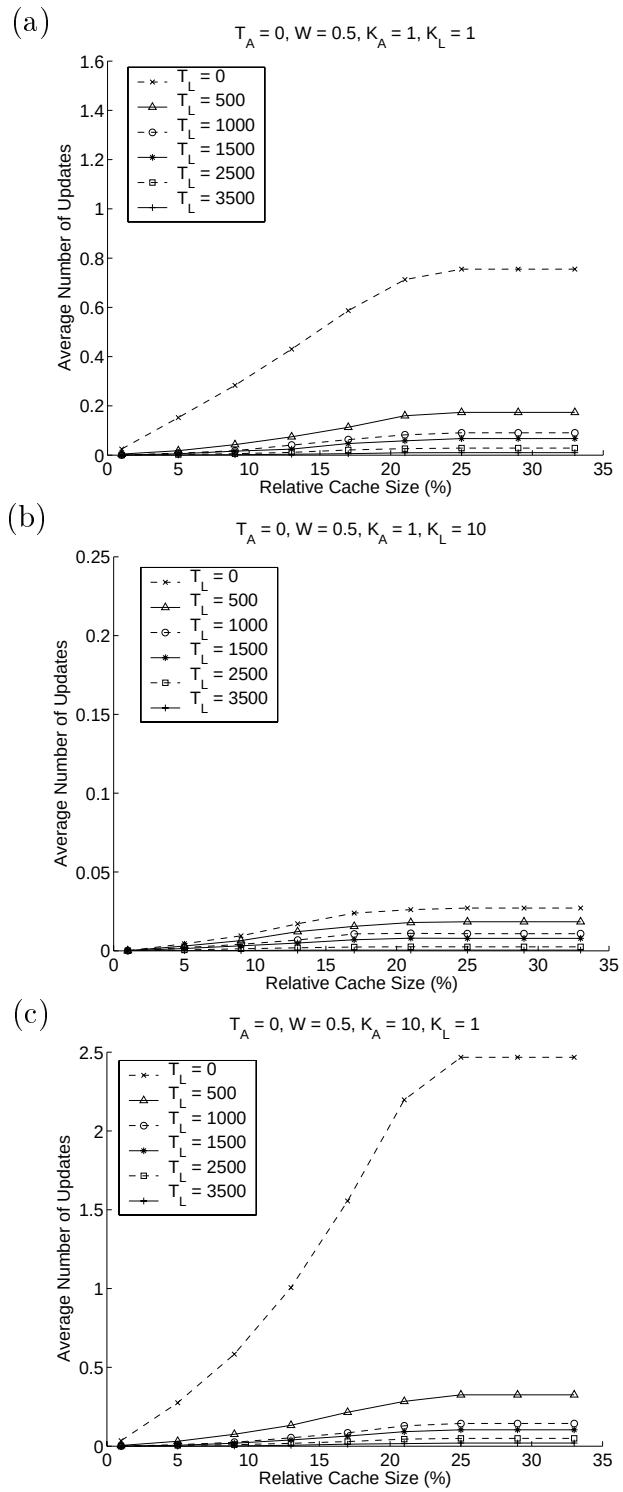

Figure 4.12: Effect of varying T_L values on average number of updates for T_A =0 (a) K_A=K_L=1 (b)K_A=1, K_L=10 (c) K_A=10, K_L=1, T_A = 0, w=0.5

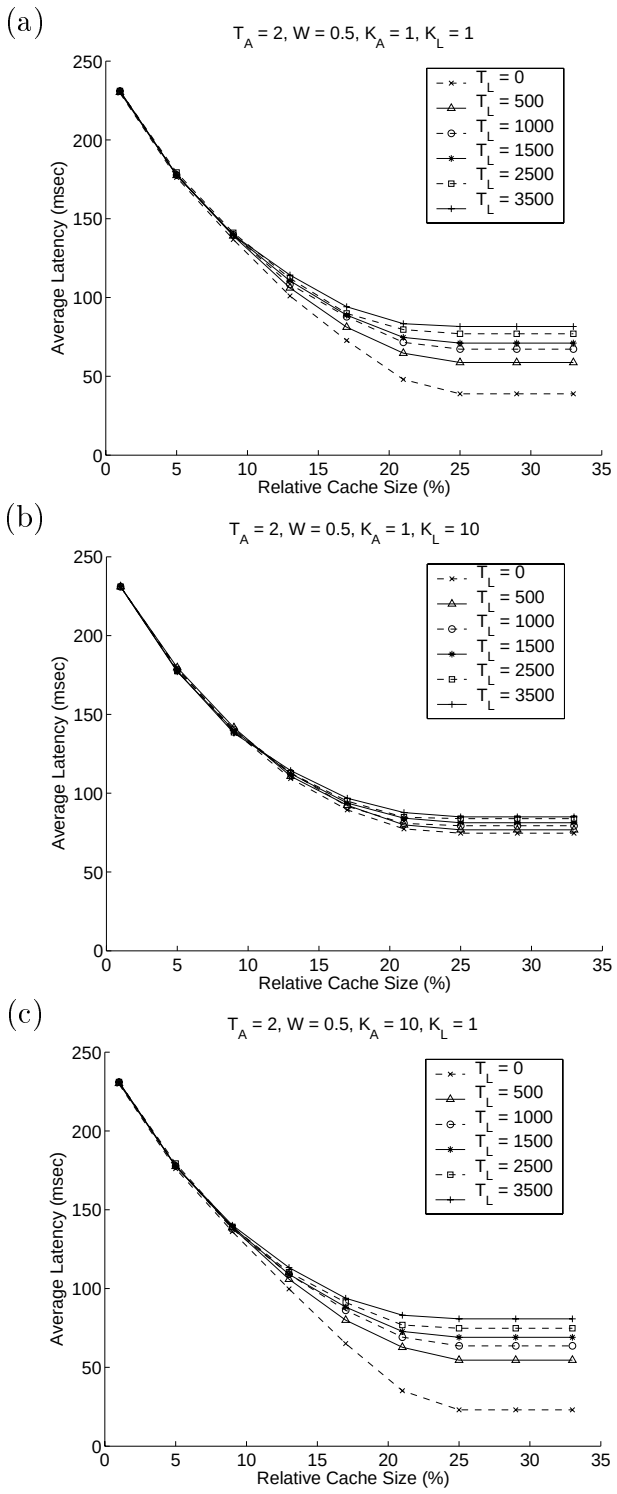

Figure 4.13: Effect of varying T_L values on average latency for T_A =2 (a) K_A=K_L=1 (b)K_A=1, K_L=10 (c) K_A=10, K_L=1, T_A = 0, w=0.5

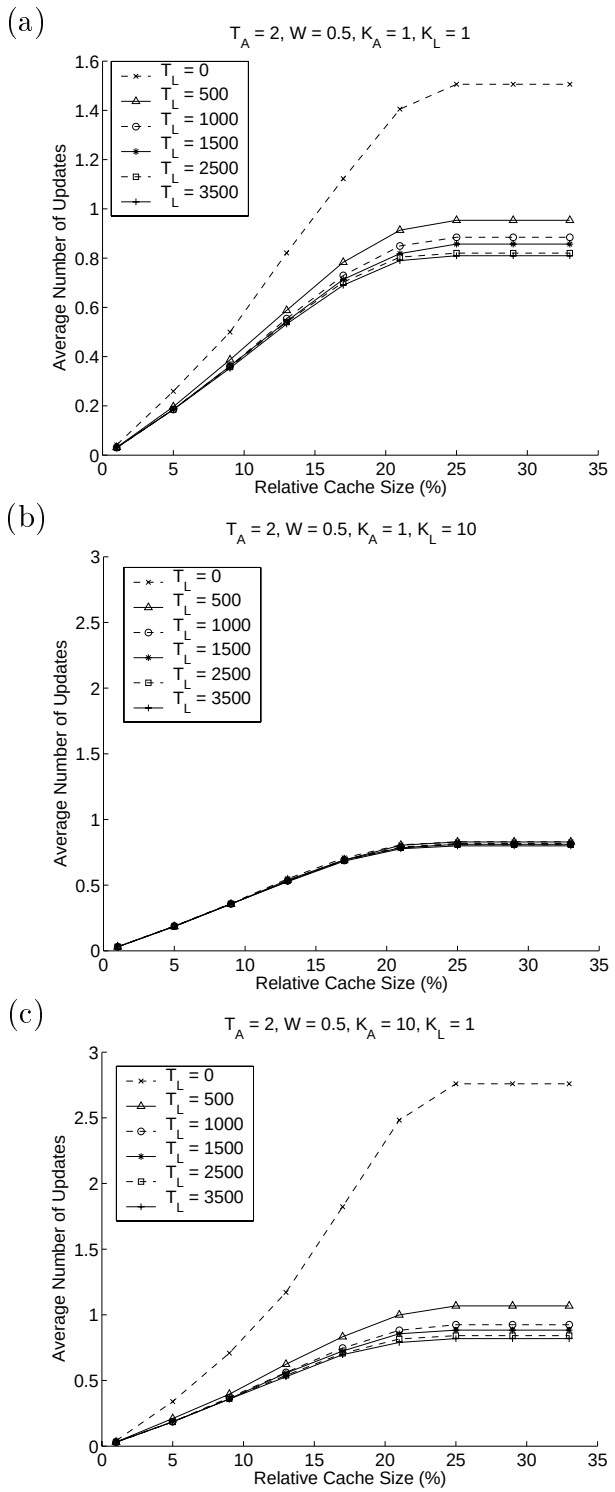

Figure 4.14: Effect of varying T_L values on average number of updates for T_A =2 (a) K_A=K_L=1 (b)K_A=1, K_L=10 (c) K_A=10, K_L=1, T_A = 0, w=0.5

Figure 4.15 plots the average latency when w = 0.4, i.e., there is an upper bound on the age. In this case, when $K_A = 1$, there is an upper bound of 2 updates. Figure 4.15(a) shows the average latencies when $K_L = 1$, and Figure 4.15(b) show the average latencies when $K_L = 10$. Since these parameter settings require profile to download any object with more than 2 updates, the K_L values and T_L values have little effect on the average latencies.

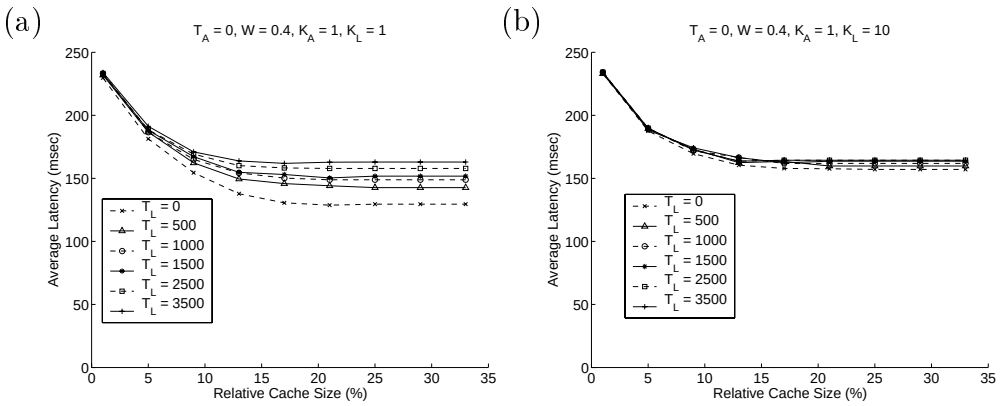

Figure 4.15: Effect of varying T_L values on average number of updates for T_A =0 (a) K_A=K_L=1 (b)K_A=1, K_L=10, w=0.4

Figure 4.16 plots the average latencies when w = 0.6, i.e., there is an upper bound on the latency. Figure 4.16 plots the average latencies for K_A=K_L=1, and Figure 4.16 plots the average latencies for K_A=1 and K_L=10. By Equation 4.3, when w = 0.6 and K_L=1, the latency has an upper bound of 2 units, i.e., 200 msec (Figure 4.16 (a)). In this case, the average latency is very low for all T_A values because of this upper bound. In contrast, when K_L=10, the latency has an upper bound of 20 units, i.e., 2000 msec (Figure 4.16 (b)). In this case, Profile is more sensitive to the T_A values because it can download any object with latency up to 2000 msec.

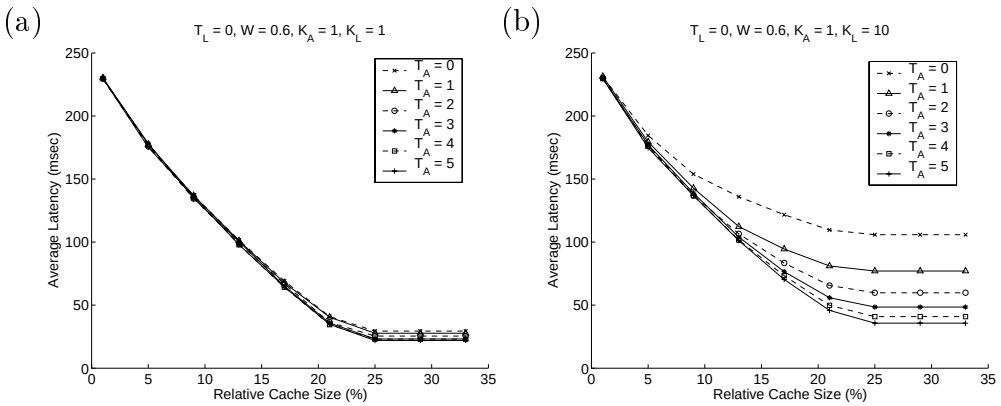

Figure 4.16: Effect of varying T_A values on average number of updates for T_L =0 (a) K_A=K_L=1 (b)K_A=1, K_L=10, w=0.6

Figure 4.17 (a) plots the distribution of the latencies of validated objects for a cache 30% of the world size when K_A=K_L= 1 and w = 0.6 (corresponding to Figure 4.16(a)). In this case, the latency has an upper bound of 200 msec, so all validated objects have latency $\leq$200. For comparison purposes, Figure 4.17 (b) plots the distribution for the same K values when w=0.5, i.e., there is no upper bound. In this case there is a similar latency-recency tradeoff. However, many more objects are validated because meeting the target latency is less important, and many requests have latency >200 msec because there is no firm upper bound.

Figure 4.18(a) plots the distribution of latencies for a 30% cache when w=0.6, K_A=1, and K_L=10. This corresponds to 4.16(b). In this case, the upper bound on latency is 2000 msec, and all requests have latency below this upper bound. In contrast, Figure 4.18(b) shows the distribution for the same K values when w = 0.5. While the distributions are similar, some requests have latency higher than the upper bound of 2000 msec.

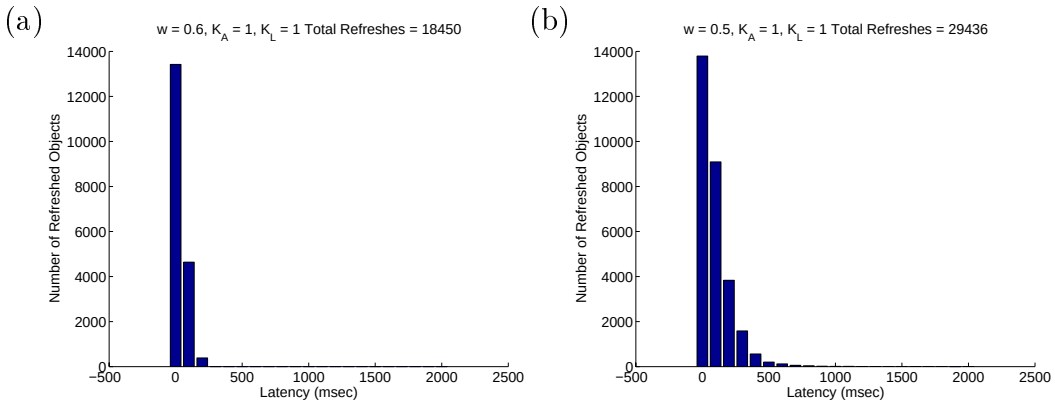

Figure 4.17: Effect w on latencies of validations for T_L =0, K_A=K_L=1 (a) w=0.6 (b) w=0.5

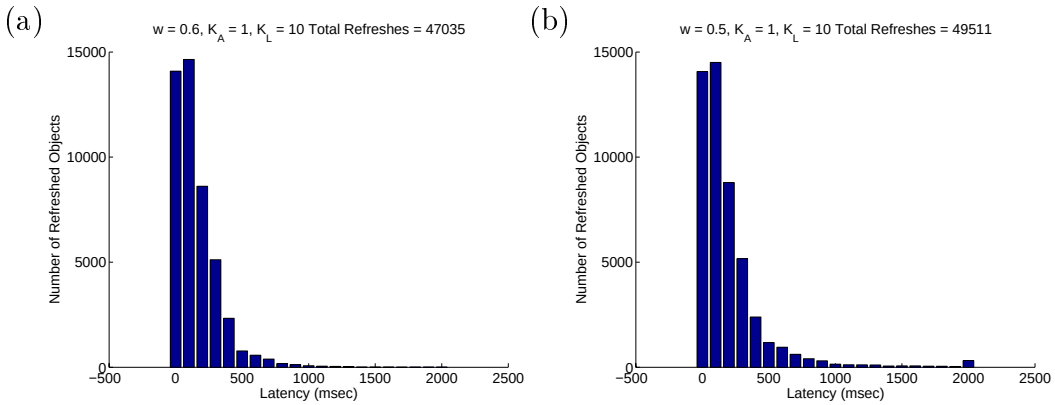

Figure 4.18: Effect w on latencies of validations for T_L =0, K_A=1, K_L=10 (a) w=0.6 (b) w=0.5

Summary To summarize, our sensitivity analysis shows the following:

- For higher T_A values, Profile is less sensitive to changes in T_L, and for higher T_L values, Profile becomes more sensitive to changes in T_A.
- For w=0.5, tuning K values can significantly impact the sensitivity of Profile to T_A and T_L. When $K_A > K_L$, the age score approaches 0 more slowly than the latency score, so Profile is more sensitive to changes in T_L values. Similarly, when $K_A < K_L$, profile is less sensitive to changes in T_L values and more sensitive to changes in T_A values.
- Tuning K and w values is an effective way to control the latency-recency tradeoff and provide upper bounds with respect to either recency or latency.

4.3 Summary and Open Problems

In this chapter, we have shown the following:

- When clients can tolerate stale data, profiles can significantly reduce the latencies of their requests compared to using TTL.
- Due to conservative TTL estimates, profiles can reduce the number of freshness misses while still delivering fresh data in most cases.
- Using profiles provide better recency than AUC because prefetching cannot keep cached data sufficiently fresh.
- During surge periods, profiles can reduce latencies for all clients.
- For higher T_A values, Profile is less sensitive to changes in T_L, and for higher T_L values, Profile becomes more sensitive to changes in T_A. Tuning

K and w values is an effective way to control the latency-recency tradeoff and provide upper bounds with respect to either recency or latency.

There are several areas for further exploration not covered in this chapter. These include:

- Developing interfaces to help clients specify and choose the appropriate profiles for their different applications. In Section 4.1.4 we presented preliminary work in this direction, but more work is needed in developing an interface that allows clients to easily specify and use profiles.

- Studying what profiles clients choose in practice, and the effects of these profiles on performance. In this chapter we have presented a framework to specify and use profiles for caching decisions. However, we have not determined what values are most appropriate for different clients and applications, and how much they improve the latency or recency of different applications.

- Learning profiles based on client behavior, network conditions, and object update patterns. This could improve the choice of default profiles for clients, and aid clients in choosing the appropriate profiles for their different applications.

- Studying the effects of different clients having different profiles for the same object. If some clients prefer the most recent data while others prefer low latency, an open question is how much will the low latency clients benefit because the cached data is fresh. Evaluating the impact of different profiles on performance and developing schemes to ensure fairness is an area of future work.

Chapter 5

Modeling Updates

We now present our work in modeling updates patterns at remote servers. Our work in client profiles relies on knowledge of when updates occur at remote sources. Clearly it is impossible to know exactly when an update occurs without either contacting remote servers (i.e., poll-every-time) or being notified by the server (i.e., server side invalidation), and there are many challenges to accurately modeling update patterns as discussed below. However, we show the exploiting knowledge of update histories can improve existing consistency policies and improve the effectiveness of using client profiles.

There are many challenges to modeling update patterns and using this knowledge to determine an appropriate consistency policy. The first challenge is predicting updates. As discussed in Chapter 2, sources can vary considerably with respect to their update frequency, predictability, and burstiness. In addition, an object may have a unique update pattern, or it may have similar update patterns to other objects at the same server. Thus, an important challenge is determining whether to model objects individually or aggregate similar objects. Aggregating objects may reduce accuracy but also has lower storage overhead. Another challenge is identifying the length of update cycles. For example, some objects

may have cyclic patterns that repeat daily, while others may have patterns that repeat weekly. Finally, object update patterns must be continually monitored to detect changes or bursts and change policies accordingly. To summarize, we need efficient techniques to both identify and exploit update patterns to objects.

A related challenge is determining how much servers should cooperate with clients and caches. By server cooperation we refer to how much information servers provide to clients. As discussed in Chapter 2, existing pull-based consistency policies assume that servers provide either the time the object was last modified or no information. At the other extreme are push-based policies where the server notifies clients of updates. In this chapter, we explore server cooperation to improve pull-based policies. If servers provide clients and caches with update histories, clients and cache managers can significantly improve the accuracy of their freshness estimates of objects compared to existing pull-based policies, while scaling better than push-based policies. However, a single server may contain thousands of objects, so server cooperation schemes must be scalable at servers with respect to both storage and computational overhead.

A final challenge is for clients and caches to determine the appropriate consistency policy to use, based on the level of server cooperation. Clients must determine whether to use all the information a server provides or only a subset. As discussed in Section 5.1, some objects can be modeled using their update histories, while others can be better modeled using only the time of the last update. Also, some policies may have a high computational overhead which is undesirable when a policy with less overhead will provide a good approximation.

In this chapter, we address the above challenges to modeling updates and server cooperation. We show that modeling updates and server cooperation can

improve the effectiveness of using profiles by increasing the probability that clients will receive data that meets their acceptable degree of recency, and can also reduce the number of unnecessary contacts with remote data sources. This is useful for many of the caching architectures presented in Chapter 2. In particular, minimizing contact with remote servers is important when there is limited bandwidth between the cache and the server and remote accesses are costly, e.g., a browser cache or a cache on a mobile device.

We first present a categorization of update patterns that often occur in practice, and give examples from real datasets. We then describe two policies for modeling updates. The first is using *individual* history (i.e., a single object), which may improve accuracy for some objects but has high computational overhead. The second is an *aggregate* history (i.e., multiple objects), using a Poisson process as in [53]. This may be less accurate for some objects but has lower computational overhead. We note that while this model does not provide a statistical fit, our results show that it predicts updates more accurately than TTL on three distinct datasets. We consider advantages and disadvantages of different policies, and discuss different options for clients and cache managers depending on what information servers provide. We also consider architectures and implementations to support server cooperation and discuss tradeoffs for both clients and servers. We evaluate our policies using data sets from two very different applications, web caching and email.

Our main results are as follows:

- For objects with cyclic update patterns, using either individual or aggregate history information can significantly improve the accuracy of estimating when it is updated compared to using TTL.

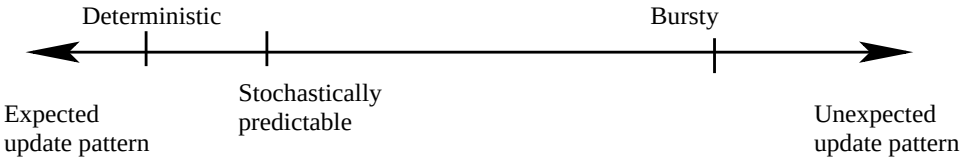

Figure 5.1: Degrees of Predictability of Update Patterns

- An adaptive policy that chooses between using update histories and using TTL depending on the update behavior of the object can generalize well to both more predictable (cyclic) and less predictable (bursty) objects.

5.1 Update Patterns

In general, objects can be classified by the regularity and predictability of their update patterns. At one extreme are objects updated at regularly scheduled times; at the other extreme are objects with completely unpredictable updates. In this section we present examples of real objects between these extremes. We illustrate these varying degrees of predictability in Figure 5.1. Note that this is not an exhaustive list. We consider more predictable (cyclic) objects that are updated at similar times each day, but not necessarily at the exact same time each day, so they are not completely predictable. We also consider less predictable (bursty) objects that experience periods with a large number of updates that are not consistent with earlier update patterns. While updates to these objects are not completely random, the bursts of updates are difficult to predict. We analyzed data from the 1998 World Cup website [10] as well as two email logs. The World Cup log consists of all requests made to the World Cup website. The two email logs each consist of all messages that arrived in a single client's mailbox. We

report on the details of these datasets in Section 5.4.

5.1.1 World Cup

Our analysis of the World Cup data shows that many objects exhibited cyclic or bursty update patterns, which are two examples of the different degrees of predictability. In the analysis below we classified objects as either cyclic or bursty. To identify objects in each category, we classified objects offline using the update histories from all 15 days of the trace. We note that there are many techniques to define bursts, e.g., using variance. We use the following straightforward technique to classify objects: For each object, we counted the number of updates that occurred on each day in the trace. We subtracted the average number of updates per day from the maximum number of updates that occurred on any day. If this difference was greater than 5, we assumed a burst had occurred on at least one day and classified the object as bursty. We classified all other objects as cyclic.

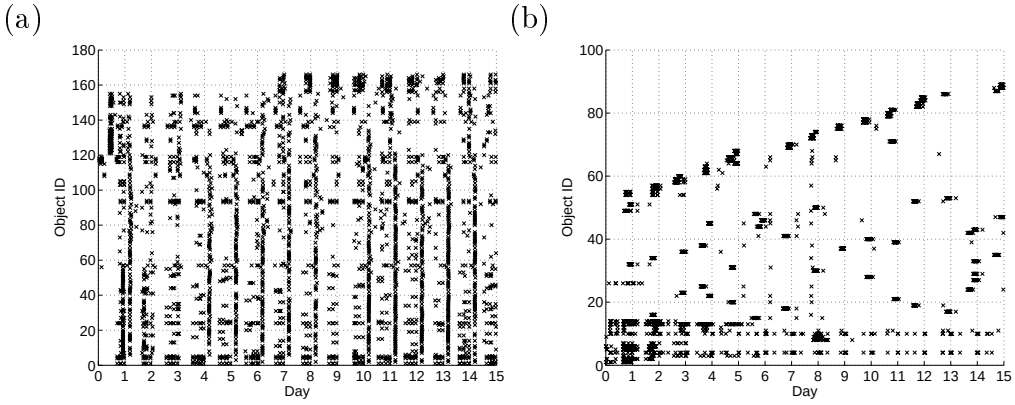

Figure 5.2: Updates to (a) Cyclic and (b) bursty objects in the World Cup trace

Figure 5.2 plots the updates to cyclic and bursty objects in the World Cup trace that were updated at least 10 times in a 15-day trace period. In these figures, the x-axis is the time of day within a 15 day window, and each value on

the y-axis represents a distinct object. An $\times$ in the graph at point `(x,y)` denotes an update to object `y` at time `x`.

Figure 5.2(a) shows objects that exhibit cyclic behavior that is repeated daily. For example, we observe in Figure 5.2(a) that many of the objects are updated at the beginning of each day, although not necessarily at the same time. These objects may correspond to pages that provided daily updates on World Cup scores and events. Cyclic update patterns commonly occur at websites, for example a weather site that updates the temperature at regular times every day.

Figure 5.2(b) shows objects with bursts of updates. In this trace, these are objects where most of the updates occurred on the same day, and few updates occurred before or after the burst. These objects may correspond to a specific World Cup event such as the score of a match. Many updates to the object occur on the day of the match, but few updates occur on other days. Bursty updates also occur at other web sites, such as news web site that frequently updates an article on the day of a breaking news event.

5.1.2 Email Traces

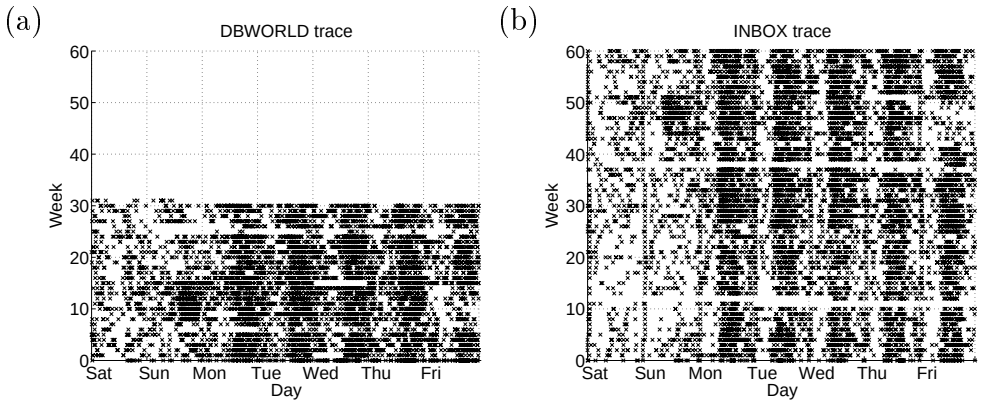

Figure 5.3: Updates to two email traces

We also considered two different email traces (labelled DBWORLD and INBOX). Figure 5.3 plots the arrival of email messages in these two traces. In each graph, the x axis shows the day of the week (and relative time of day), and each value on the y axis is a distinct week of the trace. An $\times$ value at point (x,y) indicates that an email message arrived in the client's mailbox at time x during week y. The first observation is that both mailboxes exhibit fairly regular behavior from week to week, again showing cyclic update patterns. However, both traces also exhibit occasional bursts of updates (for example, on Sunday around week 50 of the INBOX trace). Another important observation is that the update patterns repeat weekly, unlike the World Cup traces where most patterns repeat daily. This shows that different objects can have different periodicity, and illustrates another challenge to modeling updates.

5.2 Modeling Update Patterns

We model update patterns based on *recurrent piecewise constant update intensities*, as suggested in [53]. The underlying assumption of such models is that there is a time period, e.g., a day, whose update pattern is repetitive. Therefore, one can partition an update history into equal time periods with similar update pattern. To represent update patterns we use a time-varying parameter $\lambda(t)$, representing the intensity of updates over time.

A basic model of update patterns that assumes a *homogeneous* update intensity (λ) over time is inadequate for many applications [53]. This is because, as shown in the examples in Section 5.1, many objects have different update intensities at different times of day or different days of the week. Therefore, we present a more refined analysis of λ. Within a given repetitive time cycle, λ may

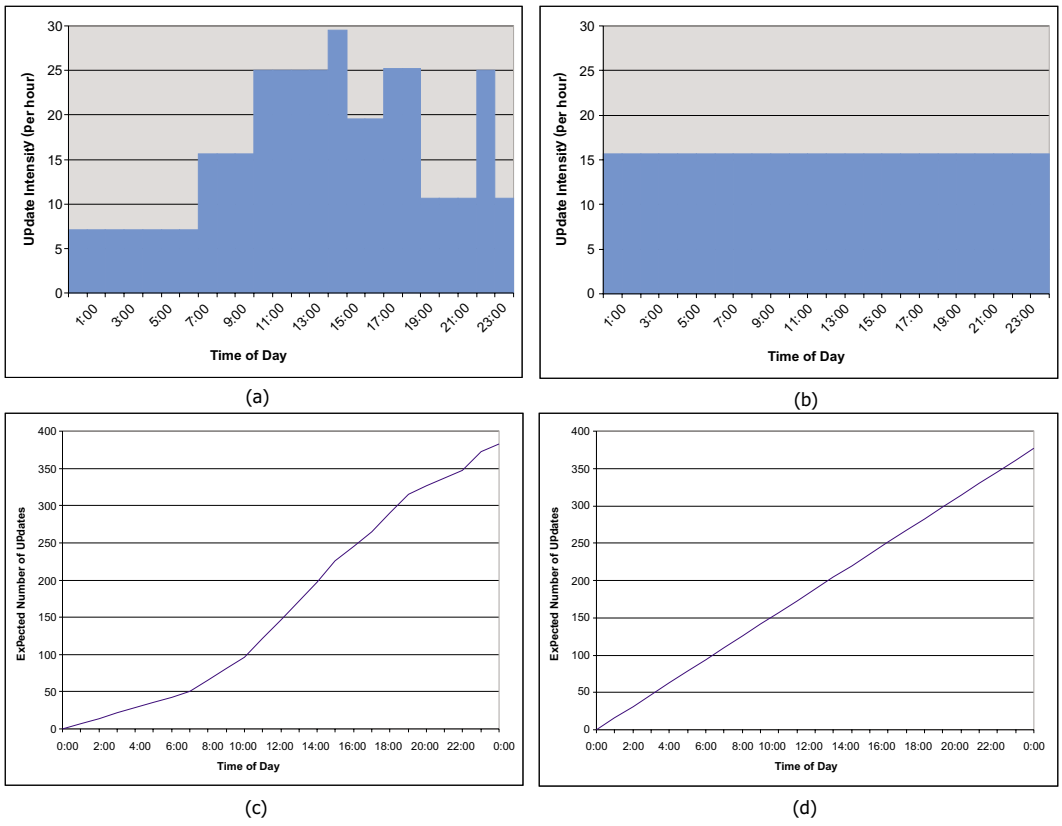

Figure 5.4: Homogeneous vs. nonhomogeneous update patterns

vary, representing, for example, change of intensities between work hours and after hours. Therefore, λ becomes time-dependent. To simplify calculations, one may assume that while λ changes over time, it may be represented as a combination of intervals, in which λ is constant, hence the term *piecewise constant*. To demonstrate the differences between homogeneous and time-dependent λ, consider Figure 5.4. Figure 5.4(a) shows the changes to the intensity of updates over a period of one day, using a piecewise-constant model. Figure 5.4(b) corresponds to a constant arrival rate of updates. Figure 5.4(c) and Figure 5.4(d) demonstrate the accumulation of λ (representing, in the case of a Poisson model, the expected number of updates in the corresponding time period) over a period of one day for the time-dependent and homogeneous λ, respectively. While the accumulation for the homogeneous model is linear over time, the accumulation rate of the time-dependent λ changes with fluctuations in the update intensity $\lambda(t)$.

Formally, given a time interval Q, suppose that the update rate $\lambda(t)$ repeats every Q time units, that is, $\lambda(t) = \lambda(t+Q)$ for all t. Furthermore, the interval $[0, Q)$ is partitioned into a finite number of subsets $J_1, \ldots, J_K$, with $\lambda(t)$ constant throughout each J_k, $k = 1, \ldots, K$. Finally, each J_k is in turn composed of a finite number of half-open intervals of the form $[s, f)$. For instance, in Figure 5.4(a) $k = 7$, with $J_1 = [0{:}00, 7{:}00)$, $J_2 = [7{:}00, 10{:}00)$, etc.

We next define a specific recurrent piecewise constant model. The model is stochastic since the repetitive nature of updates in a distributed autonomous environment cannot be modeled in a deterministic fashion. We use a nonhomogeneous Poisson process [95, 106] with instantaneous update rate $\lambda : \Re \rightarrow [0, \infty)$ to model the occurrence of *update events*. Each *update event* possibly consists of

multiple updates (possibly to different data objects) aggregated over an interval in time. The number of update events occurring in any interval $(s, f]$ is a Poisson random variable with expected value $\Lambda(s, f) = \int_s^f \lambda(t)\, dt$. When λ is constant over time $\Lambda = \lambda \cdot (f - s)$.

Using the notation given above, each interval J_i will be modeled by a homogeneous Poisson process with its own λ_i.

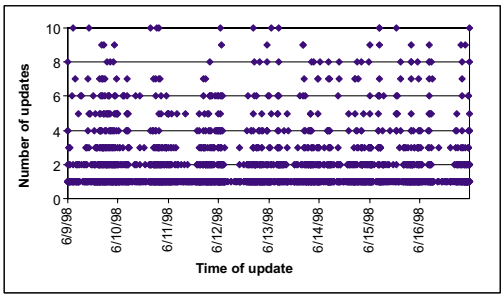

Figure 5.5: An example of a bulk insertion process.

To make the model applicable to modeling simulataneous updates, we consider a *bulk update*, the simultaneous update of objects. At update event i, a random number of objects Δ_i are updated. Figure 5.5 provides a pictorial example of a bulk insertion process for the World Cup trace. The vast majority of the updates arrive in quanta of 1, while some of the updates arrive in bigger bulks. Assuming that the $\{\Delta_i\}$ are independent and identically distributed (IID), then the stochastic process $\{B(t), t \geq 0\}$ representing the cumulative number of updates through time t is a *compound Poisson* process (*e.g.*, [95]). We let $B(s, f)$ denote the number of updates falling into the interval $(s, f]$. The expected number of updated objects during $(s, f]$ may be computed as:

$$\mathrm{E}[B(s,f)] = \int_s^f \lambda(t)\, \mathrm{E}[\Delta]\; dt = \mathrm{E}\left[\Delta\right] \int_s^f \lambda(t) dt = \mathrm{E}[\Delta]\, \Lambda(s,f) \tag{5.1}$$

Here, Δ represents a generic random variable distributed like the $\{\Delta_i\}$.

Time	λ per hour
0-7	7.13
7-10	15.59
10-14,22-23	24.97
14-15	29.50
15-17	19.53
17-19	25.23
19-22,23-24	10.60

Table 5.1: Aggregate Update History ($\vec{H}_{ag} = (\vec{T}, \vec{\lambda})$) for the World Cup trace

5.2.1 Individual and Aggregate History

We now consider two ways to model update patterns using individual history (updates to a single object) or aggregate history (updates to a set of objects with similar update patterns). The use of individual history assists in forecasting future updates more accurately, but may be costly in terms of storage overhead. A server must maintain the individual history for a potentially large number of objects, as well as indices to rapidly access the history. In addition, accumulating sufficient training data for modeling individual object history may take much longer than the time required for modeling aggregate history. Aggregate history is a less costly alternative in which aggregated data of the *update pattern* of *multiple objects* with common update patterns at a site is used to obtain a model of aggregate update pattern that is sent to the client. This provides a more compact representation and is more scalable.

Next, we introduce models for aggregate history and individual history.

Aggregate History We illustrate how to model aggregate history using the World Cup trace data. We constructed a nonhomogeneous compound Poisson

process to model the update pattern aggregated over all cyclic objects in a training set of eight days of data (from June 10, 1998 to June 17, 1998). 10,074 update times of 4,405 objects were analyzed. While this is a relatively low number of objects, these objects were requested by many clients, and pushing updates to all these clients could be very expensive.

We note that the log does not explicitly indicate the time an object is updated. In Appendix B, we describe how we detect updates. Assuming a cyclic behavior that repeats daily, we have identified seven distinct segments. Table 5.1 provides the aggregate history ($\vec{H}_{ag} = (\vec{T}, \vec{\lambda})$) (corresponding to U_i in Section 2.5), a vector representing the effective λ value for each time interval. To interpret this history, each row gives the expected number of events per hour during the time interval. For example, between 0:00 and 7:00, there are 7.13 expected events every hour. Each event corresponds to a bulk update to all the objects. For the bulk update part of the model, we aggregated updates within 30 seconds of one another into a single update event. With this data set, there were an average of 3.34 updates for each update event. We use this value in our aggregate update estimation technique described in Section 5.3.3. To estimate the number of updates (recency R_i) to an individual object O_i in each interval, we scale the λ value by f_o, the fraction of *all updates* at the server that occurred to object O_i.

In general, models are an idealized representation of a process. It is well known that Poisson processes model a world where updates are independent from one another. Therefore, models such as the one presented above need to be verified. Using verification methods, as suggested in [61], it becomes clear that the World Cup data cannot be accurately modeled using a Poisson model, most likely due to correlations of update events. However, as we show in Section 5.4, even an

"inaccurate" model that considers aggregation over multiple objects can provide a benefit over using only the last modified times of an individual object, and performs on average almost as good as using individual object's history with less overhead.

Individual History We construct individual history ($\vec{H}_{ind} = (\vec{T}, \vec{\lambda})$) (corresponding to U_i) for object O_i in the same manner as we construct aggregate history. However, the relatively small number of updates per object makes any segment analysis error prone. For individual object history we partition the day into 24 equal size intervals, and assume a constant λ within each one hour interval. As an example, we consider updates to a single object in the World Cup trace over the 8 day period from June 10- June 17. During this 8 day period, the object had 4 updates in the time period [10:00, 11:00) (which corresponds to λ=0.5, i.e., 0.5 updates/day), 1 update in the time period [11:00, 12:00) (λ=0.125), 1 update in the time period [12:00, 13:00), 3 updates in [13:00, 14:00) (λ=0.375), 2 updates in [14:00, 15:00) (λ=0.25). No updates occurred between [17:00, 22:00). Thus, this object experienced a period of high update activity in the morning, moderate activity around noon, another period of high activity in the afternoon, and no activity in the evening.

5.3 Server Cooperation and Pull-Based Consistency Policies

We now consider different levels of server cooperation and corresponding pull-based policies that can be used by clients and caches depending on how much in-

formation a server provides. Recall that in Chapters 2 and 3 we gave an overview of existing pull-based consistency policies. These policies typically assume that servers provide only the time an object was last modified, and do not provide any other update history information about the object.

In this section we present different levels of server cooperation that provide more detailed history information, and present new pull-based policies corresponding to each level of cooperation. We discuss implementation issues and architectures to support server cooperation in Section 5.3.2.

There are many reasons why a server would cooperate. First, by reducing the number of times an object needs to be validated, it can reduce the workload at the server and improve performance. Also, if the server and cache belong to the same organization (e.g., a reverse cache or a CDN as described in Chapter 2), server cooperation can improve consistency without the high server overhead of push-based solutions.

5.3.1 Architectures for Server Cooperation

Server cooperation is a term that describes multiple activities of servers. This includes the storage and computational overhead to maintain history information, and the amount of data the server must send to clients.

Figure 5.6 shows the different levels of server cooperation. Levels 0-3 correspond to pull-based policies. Pull-based policies do not require servers to store any client information, and vary by how much information servers provide on past updates to objects. This information is typically piggybacked on a client's request for the object. Levels 0 and 1 correspond to the cases where a server give no information or only the time that the object was last modified. These two

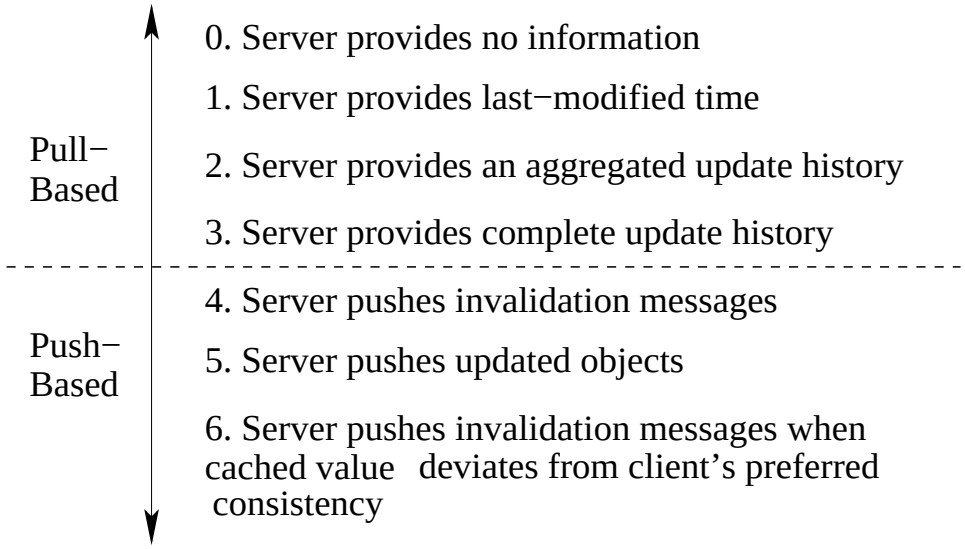

Figure 5.6: Levels of Server Cooperation

levels are what most web servers currently provide. Levels 2 and 3 increase the amount of information the server provides. At level 2, servers provide an update history aggregated over all objects rather than individual update histories. This can reduce bandwidth consumption and server and client overhead at the cost of providing less accurate information to the clients. At level 3, servers provide a complete update history of the object. We discuss the details and tradeoffs of these different policies further in Section 5.3.3.

Levels 4-6 are push-based techniques that require the server to store information about what objects each client has in their cache. Level 4 requires servers to send invalidation messages to clients whenever an object is updated (e.g., [78, 114]). Clients can then request an updated object from the server. Level 5 can further improve consistency by having servers push the updated object to clients. However, this policy consumes significantly more bandwidth than level

4. Finally, level 6 allows an object cached at the client to deviate from the object at the server within a specified bound (e.g., [8, 65, 88, 89]). This can reduce the amount of data a server needs to push to a client compared to level 5. However, this also requires servers to store information on the level of staleness a client will tolerate, and to compute when the server value exceeds this level. Thus, there is more overhead compared to level 5.

5.3.2 Implementation Issues

We have assumed that servers can either compute an aggregate history over multiple objects, or provide clients with a history of updates to an individual object, and clients can use either the individual or aggregate history to compute the expected number of updates to an object. However, this ignores practical implementation challenges such as the storage and computational capacity of clients. Depending on the power and storage capacity of the client, it may be more appropriate to have a server or intermediate proxy use the update history to compute an expiration time for a client. For example, if an object is cached on a mobile device, doing the computation at the server or proxy would conserve the limited battery power of the mobile device.

In this section we briefly consider implementation issues concerning where computations are performed and tradeoffs in terms of both performance and flexibility. We discuss levels of server cooperation in terms of how much computation is performed at a server or proxy vs. how much is performed at the client, and discuss scenarios where each level of cooperation is most appropriate. We present different options for servers/proxies and clients, in order of increasing amount of overhead for servers or proxies and decreasing overhead for clients:

1. Server provides history to client.

2. Server provides λ values to client, either for individual object or aggregated over multiple object.

3. Server provides expiration time to client, based on server-defined θ value (i.e., time that expected number of updates to the object will exceed θ.)

4. Server provides expiration time to client, based on client-defined θ value.

5. Server provides individual or aggregate history to an intermediate proxy, and proxy computes expiration time based on client-defined θ value.

Levels 1 and 2 are most appropriate when the server has many objects and the client has sufficient capacity to compute the expected number of updates, and they give the client the flexibility to determine when to validate cached objects based on their tolerance for stale data. Level 3 requires no computational overhead for clients and minimal overhead for the server, however, it does not give clients any flexibility. Level 4 gives clients greater flexibility but requires more cooperation and computation at the remote server. Clients can pass their desired θ value to servers in the header of their requests, as described in Chapter 4. Finally, performing the computation at an intermediate proxy may be most appropriate when the client desires greater flexibility but the server is unable or unwilling to perform the computation, e.g., mobile clients. A mobile service provider can perform the computation at an intermediate proxy to compute an expiration time before delivering the data to clients.

5.3.3 Pull-Based Policies

Next, we provide the details of different levels of server cooperation and describe policies that clients can choose based on the level of cooperation. Clients must develop appropriate policies depending on the server's level of cooperation as well as the expected benefit. We present four categories of policies based on the level of server cooperation. We summarize these levels in Figure 5.3.3, and describe each level and its corresponding policies below.

S1: Last Modified If servers provide only the time an object was last modified (S1 in Figure 5.3.3), the client can use TTL, or if they are willing to tolerate stale data, they can use Last Modified Staleness Estimation. We describe each of these below.

The advantage of these policies is that they require minimal compuational and storage overhead at the client. The disadvantage is that they do not consider previous updates to the object, and cannot exploit knowledge of update patterns to cyclic objects. As we will show in Section 5.4, this may lead to less accurate estimates.

C1a: TTL Using only the time an object was last modified, clients or caches can use TTL, a pull-based policy widely used in practice. TTL estimates how long an object remains fresh in the cache as a function of its *last modification time*. Any object that is estimated to be stale must be validated. TTL can be tuned using a parameter α, which is typically a real number between 0 and 1. If an object is cached at time t_{cache} and was last modified at time $t_{lastmod}$, its TTL is estimated as:

$$TTL = t_{cache} + \alpha * (t_{cache} - t_{lastmod})$$

Server options:

- S1: Last Modified
- S2: Aggregate History
- S3: Individual History

Client options:

- S1: Server provides Last Modified Time
  - Client can choose:
    - C1a: TTL
    - C1b: Last Modified Staleness Estimation
- S2: Server provides Aggregate History
  - Client can choose:
    - C2a: Aggregate Based Staleness Estimation (AggHist)
- S3: Server provides Individual History
  - Client can choose:
    - C3a: TTL
    - C3b: Last Modified Staleness Estimation
    - C3c: Individual History Based Staleness Estimation (IndHist)
    - C3d: An adaptive strategy

Figure 5.7: Options for client and server cooperation

The TTL policy works as follows: *If a cached object is requested before the TTL time expires, it is served from the cache without validation (i.e., contact with the remote server). If the object is requested after the TTL time expires, the cache validates the object at the remote server before delivering the object.*

Note that smaller values of α generate more conservative TTL estimates, which improve data freshness, but increase the number of validations.

C1b: Last Modified Staleness Estimation (LMSE) Another pull-based policy that caches can use if servers provide the last modified time is Last Modified Staleness Estimation (LMSE). This policy can be used if clients can tolerate some staleness, e.g., they will accept an object with no more than two updates. LMSE uses a heuristic to estimate the expected number of times an object has been updated since it was cached as in [19]. If an object is cached at time t_{cache} and was last modified at time $t_{lastmod}$, we compute $ExpUpdates(t_{cache}, t)$, the expected number of updates at time t as:

$$ExpUpdates(t_{cache}, t) = \frac{t - t_{lastmod}}{(TTL - t_{lastmod})} = \frac{t - t_{lastmod}}{(1+\alpha)(t_{cache} - t_{lastmod})}$$

where TTL is calculated using t_{cache}. If the expected number of updates exceeds a threshold θ, then the object is validated, otherwise it is served from the cache without validation. θ represents the client's tolerance towards staleness. The higher θ is, the more willing the client will be to accept stale data.

S2: Aggregate History (AggHist) If a server provides aggregate history information (S2 in Figure 5.3.3), the client can compute the expected number of updates to object O_i using the aggregated intervals and λ values U_i=($\vec{H_{ag}}$ = ($\vec{T}, \vec{\lambda}$)) provided by the server (AggHist). Recall that each λ value corresponds to the expected number of bulk events per hour in interval T. The server scales

the aggregate history by the fraction of updates of each object with respect to the total number of updates at the server. This requires less computational overhead for the clients than history-based estimation. It can also reduce both bandwidth consumption and storage overhead for clients if clients cache many objects from the same server. If a client already has the aggregate history for a server, the server needs to provide only the fraction of updates to each object. This can provide significant savings if a client accesses many objects from the same server. However, it also provides less accurate estimates. We note that servers could also provide aggregate histories for individual objects; however, this would significantly increase computational overhead at the server and may not scale to a large number of objects. A good compromise would be for servers to group objects with similar update patterns and compute aggregate lambda values for every group. We discuss this further in Chapter 8.

If servers provide an aggregate history but do not provide an individual history or a last modified time, clients cannot use TTL or LMSE as shown in Figure 5.3.3. We note that if servers provide both an individual update history and aggregate history, clients could use an adaptive strategy that switches between multiple options, e.g., aggregate and TTL (or LMSE). We do not consider the adaptive aggregate policy in this dissertation.

C2a: AggHist The aggregate based policy (AggHist) uses the aggregate history ($\vec{H}_{ag} = (\vec{T}, \vec{\lambda})$) that is learned from the past updates to a set of objects. Table 5.1 gives an example aggregated over all objects in the World Cup trace. Recall that for a given interval, λ denotes the update intensity in that interval. According to Equation 5.1 the expected number of updates accumulated to an

object in an interval (s,f) is

$$\mathrm{E}[B(s,f)] = \mathrm{E}[\Delta] \int_s^f \lambda(t)dt \tag{5.2}$$

To estimate the update pattern of an individual object from the aggregate update history, servers scale the aggregate λ values by the relative fraction of server updates that occurred to that object. Without loss of generality, assume that time s falls in interval 0 and time f falls in interval n. Therefore, whenever $n > 0$, we can rewrite Equation 2 to be

$$\mathrm{E}[B(s,f)] = \mathrm{E}[\Delta] \left(\begin{array}{c} (Upper\,(T\,(0)) - s)\,\lambda\,(0) + \\ \sum_{i=1}^{n-1} (Upper\,(T\,(i)) - Lower\,(T\,(i)))\,\lambda(i) + \\ (f - Lower\,(T\,(n)))\,\lambda(n) \end{array} \right) \tag{5.3}$$

where $Upper\,(T\,(i))$ and $Lower\,(T\,(i))$ represent the upper bound and lower bound of $T\,(i)$, respectively.

The AggHist policy works as follows: *Given an initial time t_m, an aggregate update history $\vec{H}_{ag} = (\vec{T}, \vec{\lambda})$, the fraction of updates of an object O_i with respect to the total number of updates at the server f_o, and the expected number of updates per bulk update event Δ, calculate the expected number of updates (recency R_i). If R_i exceeds a threshold θ, then validate the object at the server.*

We illustrate with an example from the World Cup trace. Suppose an object O_i is cached at 1:00 and requested at 8:00. We use the λ values from Table 5.1. If 1% of all updates at the World Cup site occur to object O_i, i.e. f_o= 0.01, the corresponding λ values for the intervals from 1:00 to 8:00 in Table 5.1 are scaled as follows:

Time [1:00 - 7:00]: 7.13 * 0.01 = 0.0713

Time [7:00 - 8:00]: 15.59 * 0.01 = 0.1559

Recall from Section 5 that for each bulk update event at the World Cup site, there were an average of Δ=3.34 updates per hour. Therefore, for object O_i, the expected recency R_i is:
$ExpUpdates(1,7) + ExpUpdates(7,8)$ = (0.0713 * 3.34 * 6 hours) + (0.1559 * 3.34 * 1 hour) = 1.95

S3: Individual History If a server provides the complete individual update history of an object, the client has several options. First, it can use TTL (C3a) or LMSE (C3b) as described above, because the individual history includes the time of the last update. This is straightforward to compute for the clients and reduces the overhead of storing update histories, at the cost of possibly less accurate update estimates. Alternately, clients could use the history to perform history-based staleness estimation. This has the advantage of using the individual update history of an object, but requires more overhead for clients.

C3c: IndHist The individual history policy (IndHist) uses the individual history U_i= ($\vec{H}_{ind} = (\vec{T}, \vec{\lambda})$) to estimate the recency R_i of a cached object O_i based on its update history. There are many methods to estimate the number of updates using the individual history. In our evaluation, we calculate $\vec{H}_{ind}$ by partitioning all past updates to an individual object into constant intervals as described in Section 5.2.1.

The IndHist policy works as follows: *First, use $\vec{H}_{ind}$ to estimate the expected number of updates to the cached object. Use formula 5.3 to compute the expected number of updates, with $E[\Delta] = 1$. If the expected number of updates exceeds a threshold θ, validate the object.*

Using our example object from Section 5.2.1, if the object was cached at 11:30

and is requested at 14:00, its expected number of updates R_i is:

$\frac{1}{2}ExpUpdates(11,12)+ExpUpdates(12,13)+ExpUpdates(13,14) = \frac{1}{2}*0.125+$ $0.125 + 0.375 = 0.5625$.

C3d: Adaptive Policy Finally, if servers provide individual history information, clients can use an adaptive policy and adaptively choose between TTL or LMSE and history based estimation. Adaptive policies require individual histories in order to compare the actual number of updates to an object in a given time window to its expected behavior.

We hypothesize that the AggHist and IndHist policies may perform poorly during bursts because the bursts are inconsistent with the update histories. In contrast, TTL may perform well during bursts because it estimates that an object that was recently updated is likely to be updated again soon.

Given an individual history, an adaptive policy uses heuristics to detect bursts online and dynamically choose between TTL and IndHist. Thus, it can generalize well to different types of update patterns, and requires no prior knowledge of whether an object is cyclic or bursty. We describe the details of detecting bursts and the adaptive policy in Section 5.4.4.

5.4 Experiments

We now evaluate the AggHist, IndHist, and TTL policies on data traces that exhibit both cyclic and bursty behavior. We use trace data from two different applications, web caching and email, to show that using history information can improve the accuracy of estimating the freshness of cached objects and significantly reduce the number of validations. We first compare the effectiveness of

the AggHist and IndHist policies against TTL on objects with cyclic behavior in the two email traces and the World Cup trace. We then present an adaptive history policy (IndHist-AD) that can generalize to different types of objects, and evaluate its performance on both cyclic and bursty objects in the World Cup trace.

5.4.1 Data Traces

World Cup Data The trace data from the 1998 World Cup Web Site [10] contains a log of all requests to the site. The World Cup site had servers in four different geographical locations: Paris, France; Herndon, VA; Santa Clara, CA; and Plano, TX. The entire trace consists of 1.3 billion requests made from May 1, 1998 to July 23, 1998. In our experiments we used a 15-day subset of this trace from June 10, 1998 to June 25, 1998. This corresponds to the first 15 days of the World Cup event and includes about 333 million requests. In our experiments, we report separate results for cyclic and bursty objects. To identify objects in each category, we classified objects offline using the update histories from all 15 days of the trace, using the techniques described in Section 5.4.4.

For each request, the trace contains the following:

- *ClientID:* Unique ID of the client making the request. Note that this may be a proxy.
- *ObjectID:* Unique ID of the requested object.
- *Timestamp:* The time the request was made.
- *Size:* Size of the object in bytes.

The trace does not explicitly give information on updates to objects, however, we can infer updates when an object changes size as described in Appendix B.

In the 15-day trace, 42 million requests were for cyclic objects and 11 million requests were for bursty objects. The remaining 280 million requests were for objects that did not change during the 15 days, most of which were static images. If all clients had sufficient cache space (see below), 9 million of the requests for cyclic objects would be cache hits, as would be 1 million of the requests for bursty objects. Note that the percentage of requested bursty objects that are in the cache ($\approx$11%) is smaller than for cyclic objects ($\approx$21%). This is because the bursty objects are most interesting to clients during a short interval (during the bursty period), so they are less likely to be cached prior to the update burst.

Email Data Our first email trace (**DBWORLD**) includes email notifications of postings to the DBWORLD electronic bulletin board and other messages. The data were collected over seven months and consists of more than 6400 insertions, from November 9, 2000 through June 17, 2001. Our second email trace (**INBOX**) is taken from messages to a client's inbox from March 3, 2001 - December 24, 2002 and consists of about 10,000 insertions. We collected the data for both these traces using a capture program (similar to the way the vacation program works on Unix) to capture messages and process them.

5.4.2 Setup

World Cup Experiments Our experiments with the World Cup trace model a traditional web caching scenario. We compared the TTL, IndHist, and AggHist policies. For each of these policies, when a client requests a cached object, the

cache uses the policy to determine whether or not to validate the object. Using TTL, an object is validated if it is requested after it expires. Using IndHist and AggHist, it is validated if the expected number of updates exceeds a specified threshold θ.

We maintained separate caches for each client ID, which may correspond to either an individual client or a proxy. For each client ID, we assumed an initially empty cache. To simplify our presentation, we assume all clients had sufficient space to cache their objects and no objects were evicted from client caches during the trace period. This is a reasonable model because cache size affects only the hit rate of the cache. Therefore, a limited cache would have equal impact on the performance of all estimation policies, and would not change their relative accuracy. Each experiment included a training period to gather object update history information, followed by a test period during which we collected data. We give the length of the training and test periods when reporting the results of each experiment.

Email Experiments Our experiments with the email traces model a scenario where a client has a locally cached mailbox, e.g., on their mobile device, that needs to be refreshed in the background to promptly notify the client of new messages. The goal is to minimize the time elapsed between when a new message arrives and when it appears in the client's mailbox while maintaining a reasonable network resource consumption. This differs from the above web caching application where objects are refreshed only when they are requested (and the cached copy is not sufficiently fresh).

For the email application, we compare the TTL and IndHist policies.[1] After

[1]Note that since a mailbox corresponds to a single object, we do not consider the AggHist

each refresh, for the TTL policy, we computed the time of the next refresh as a function of the time the last message arrived. For the IndHist policy, after each refresh we computed the time of the next refresh as the time that the expected number of updates (i.e., new messages) would exceed some threshold θ. We used the first week of each trace as a training period to gather a history, and continuously updated the history during the experiments.

Metrics We use the following metrics:

- **Total Validations**: This is the number of times requested objects that were in the cache needed to be validated at the remote server.
- **Stale Hits**: For the World Cup trace, this is the number of objects that were served from the cache without validation but had actually been updated at the remote server.
- **Average Delay**: For the email traces, this is the average amount of time elapsed between the arrival of a new message and the time it appears in the client's mailbox.

5.4.3 Results for Cyclic Objects

Our experiments show that using either aggregate histories or individual histories of cyclic objects can *significantly improve the accuracy of estimates of an object's freshness*. In web caching, this can increase the number of objects served from the cache without validation, which reduces costly remote server accesses for clients and reduces the load on servers. In email applications, this can reduce the delay

policy.

of new messages appearing in a client's mailbox without increasing the mailbox refresh rate, which is of particular importance to mobile devices.

Accuracy of Estimates

World Cup Trace We first compare the accuracy of estimating the number of updates to cyclic objects in the World Cup Trace using LMSE, IndHist, and AggHist. Each time a client requests a cached object, we compare the actual number of updates to the object against the estimated number using each policy. Using LMSE, we estimate the number of updates to an object at time t as $(t - t_{lastmod})/(TTL - t_{lastmod})$ [19], where $t_{lastmod}$ is the last modified time of the object, and use an α value of 0.05, which is commonly used in practice [24].

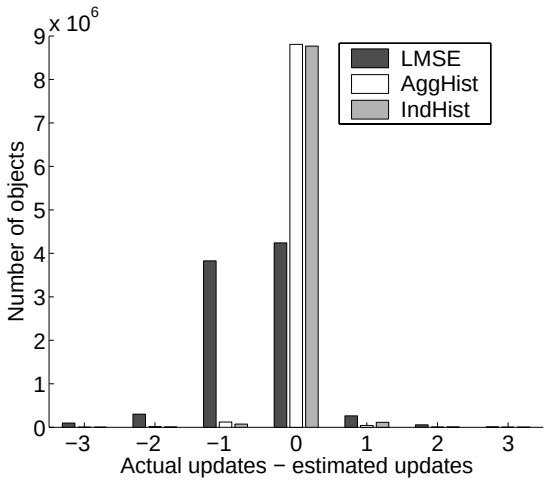

Figure 5.8: Comparison of three policies for Cyclic objects in the World Cup Trace

Figure 5.8 compares the estimated updates to the actual value for each policy. A value of 0 means the estimate was accurate. A positive error value means the actual value exceeded the estimated value, and a negative value means the actual

θ (expected)	actual (DBWORLD)	actual (INBOX)
0.01	0.010	0.011
0.05	0.049	0.057
0.1	0.098	0.114
0.2	0.196	0.229
0.3	0.295	0.344
0.4	0.394	0.457
0.5	0.493	0.575
0.6	0.592	0.687
0.7	0.691	0.799
0.8	0.790	0.913
0.9	0.886	1.034
1.0	0.980	1.145

Table 5.2: Comparison of the expected and actual number of updates using IndHist

value was less than estimated. AggHist and IndHist have more than twice as many accurate estimates as TTL. This shows that using histories can significantly improve the accuracy of freshness estimates for cyclic objects.

Email Traces We next consider the accuracy of the IndHist policy for the email application. Recall that for the email application, using IndHist we refreshed the mailbox whenver the expected number of updates (new messages) exceeded θ. In Table 5.2 we compare the expected number of updates per validation (θ) against the actual number of updates per validation. We report results for both the DBWORLD trace and the INBOX trace. For all values of θ, the expected number and actual number are very close, which shows that IndHist is accurately estimating the expected number of updates.

Number of Validations

We now report on the number of validations required to maintain a given level of freshness using IndHist, AggHist and TTL for the World Cup trace and the DBWORLD and INBOX traces.

World Cup Trace We first compare the TTL, IndHist, and AggHist policies in terms of both number of validations and data freshness of cyclic objects in the World Cup trace. In these experiments, we used all 15 days of trace data. We used the first 8 days to construct histories, and ran the experiments on the next 7 days.

We varied the α parameter for TTL, and the θ parameter for IndHist and AggHist. For TTL, we varied α from 0.05 to 0.55. This range is typical of what is used in practice [57]. For IndHist, we varied θ from 0.05 to 0.30. For AggHist, we varied θ from 0.07 to 0.50. We report on the number of useful and useless validations and stale hits for each of the three policies.

Policy	Total Vals	Useful Vals	Useless Vals	Stale Hits
TTL α=0.10	3345399	880977	2464422	129048
Agg θ=0.10	3028941	888252	2140689	97931
Hist θ=0.10	2297665	873456	1424209	129078
TTL α=0.20	2603697	840095	1763602	288646
Agg θ=0.20	2249506	864662	1384844	215165
Hist θ=0.20	1819490	851922	967568	229994

Table 5.3: Validations and Stale Hits for Cyclic objects

We show the results for selected α and θ values in Table 5.3. The first observation is that AggHist and IndHist perform significantly fewer validations than TTL, without a significant increase in the number of stale hits. When $\theta = 0.20$, AggHist and IndHist have both fewer validations *and* fewer stale hits than TTL α =0.20. AggHist reduces the number of validations by 14%, and IndHist reduces the number of validations by 30%. When θ =0.10, AggHist reduces the number of validations by 9% and IndHist reduces the number of validations by 31% compared to TTL with $\alpha = 0.10$. Thus, policies that consider past updates to objects can significantly reduce bandwidth consumption compared to TTL while providing fresher data to clients.

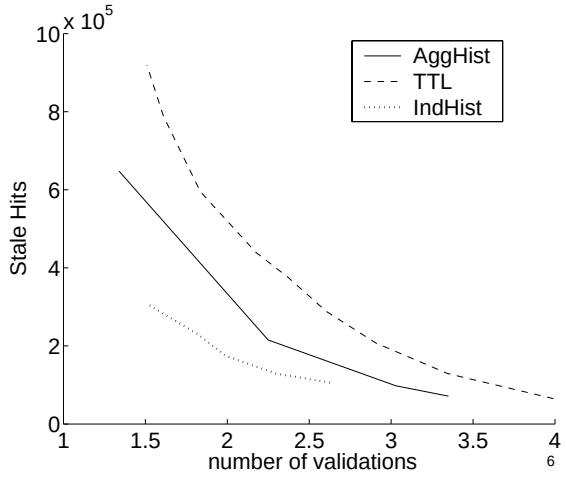

Figure 5.9: Effect of Tuning TTL, AggHist, and IndHist on data freshness and validations

In Figure 5.9 we report on the number of stale hits given similar levels of total validations. Note that the lines for IndHist and AggHist do not extend beyond 2,500,000 and 3,500,000, repectively, because both of these policies perform fewer validations than TTL even for small values of θ.

For each of the three policies, this graph shows the benefit of increasing the total number of validations. Given a number of validations, both AggHist and IndHist deliver significantly fewer stale objects than TTL. This is because the improved accuracy of the freshness estimates of objects reduces the number of unnecessary validations. This is especially true when there are relatively few validations, i.e., higher values of α and θ. For example, when each of the policies has about 1,500,000 total validations, TTL (α $\approx$0.5) provides $\approx$800,000 stale hits while AggHist (θ $\approx$0.5) provides $\approx$500,000 stale hits and IndHist (θ $\approx$0.3) provides $\approx$300,000 stale hits. Another important observation is that the IndHist policy offers an improvement over the AggHist policy because it can model the individual update patterns of objects that may differ from the average behavior.

To summarize, for objects with cyclic update patterns, the AggHist policy can offer significant improvements over TTL. Therefore, the AggHist policy is a good alternative to TTL. It can reduce computational overhead (compared to IndHist) while still providing reasonable estimates of the freshness of cached data.

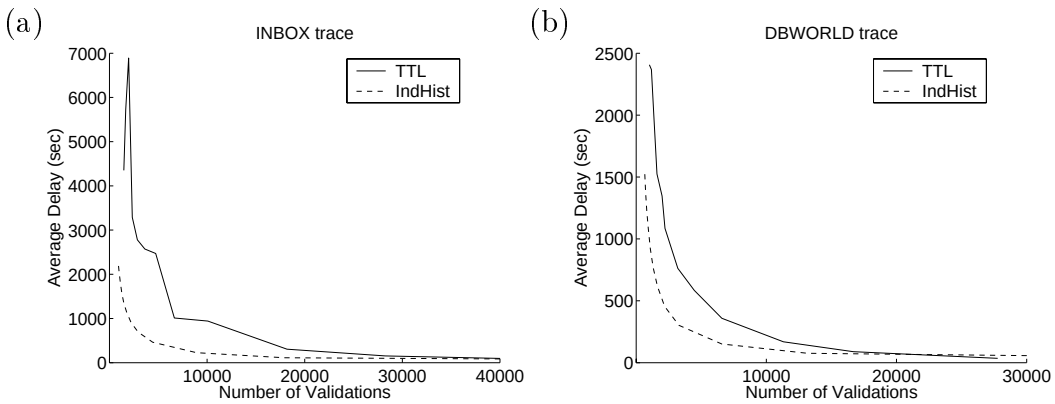

Figure 5.10: Effect of Tuning TTL and IndHist on average delay and validations

Email Traces We now consider the DBWORLD and INBOX traces. Recall that we use the average delay as our metric. For TTL, we varied α between 0 and 1 and for IndHist, we varied θ between 0 and 1. We plot the number of validations against the average delay for TTL (for different α values) and IndHist (for different θ values) for both traces in Figure 5.10. As expected, as the number of validations increases, the average delay decreases for both. The key observation is that for a given average delay, IndHist performs significantly fewer validations than TTL. For example, in Figure 5.10(a), to provide an average delay of about 500 seconds, TTL must perform about 170,000 refreshes while IndHist performs about 80,000. Similarly, for the dataset in Figure 5.10(b), to provide an average delay of 500 seconds TTL performs about 50,000 validations while IndHist performs about 20,000. Thus, IndHist can reduce the total number of refreshes by more than half. This can provide significant savings in terms of both power and bandwidth to clients who read email on their mobile devices.

5.4.4 Adaptive Policy

In Section 5.1 we presented update patterns observed in several traces. In the World Cup trace, we observed both cyclic and bursty objects. The email traces also generally experience cyclic update patterns but may experience occasional bursts. In this section, we present an adaptive policy that uses heuristics to detect bursts online and dynamically choose between IndHist and TTL. We evaluate its performance on both cyclic and bursty objects, and show that the adaptive policy can generalize well to both.

IndHist-AD We now describe an adaptive policy (IndHist-AD) that can (a) detect bursts and (b) dynamically choose between policies. Thus, it can generalize well to different types of update patterns, and requires no prior knowledge of whether an object is cyclic or bursty. We first describe how we identify bursts. We then describe the adaptive policy (IndHist-AD) which dynamically chooses between the IndHist and TTL policies depending on whether or not an object exhibits bursty behavior. Thus, for objects with no bursts, it has comparable performance to the IndHist policy.

Identifying Bursts

We use the term burst to refer to the case where the number of actual updates to an object is considerably higher than that approximated by IndHist. Consider an object that is cached at time t and created at time t_0. We estimate that a burst occurs when the *actual number of updates* in a window of size W prior to t, i.e., all updates in the interval $[t-W, t]$, exceeds the *expected number of updates*. The *expected number of updates* is estimated by the IndHist policy, using only updates that occurred in $[t_0, t-W]$ prior to the current cycle.

The adaptive history policy works as follows: *Given an intensity function λ for the interval (t_0, t - W), and λ^* for the interval ($t-W$, t), a distance function $f(\lambda, \lambda^*)$, and a threshold T, IndHist-AD identifies a burst if $f(\lambda, \lambda^*) \geq T$. On each request, if $f(\lambda, \lambda^*) \geq T$, IndHist-AD assumes a burst is occuring and uses TTL. Else, if $f(\lambda, \lambda^*) < T$, IndHist-AD assumes a burst is not occuring and uses the IndHist policy.*

We next provide a distance measure f. This was empirically evaluated to provide a good estimation of bursty periods in the World Cup trace data. We

note that more research is needed to identify distance measures that will work on several traces. Let the *expected number of updates* (Λ) in $[t-W,\ t]$ with respect to time t be $\Lambda(W,t)$, and the *actual number of updates* (Λ^*) in $[t-W,\ t]$ be $\Lambda^*(W,t)$. Then,

$$f(\lambda,\lambda^*) = \begin{cases} \frac{\Lambda^*(W,t)}{\Lambda(W,t)} & \text{if } \Lambda(W,t) > 0 \qquad (a) \\ T & \text{if } \Lambda(W,t) = 0 \text{ and } \Lambda^*(W,t) > 0 \ (b) \\ 0 & \text{otherwise} \qquad (c) \end{cases}$$

Intuitively, condition (a) covers the case when at least one update was expected ($\Lambda(W,t) > 0$). A burst occurs when the ratio of *observed* updates to *expected* updates exceeds T. Condition (b) covers the case when no updates were expected ($\Lambda(W,t) = 0$) and at least one update occurs.

Bursty Objects in the World Cup Trace

Most bursty objects in the World Cup trace had a "burst" of updates on a single day, and few (if any) updates on other days, as shown in Figure 5.2(b). For these objects (or any object with no history available), TTL is likely to provide more accurate freshness estimates compared to the IndHist based policy. A more interesting case occurs when an object that normally has cyclic update patterns experiences a burst in updates. This could occur at a news web site that is normally updated at regular intervals but experiences a burst of updates during a breaking news event. For these objects, IndHist is likely to do well during cyclic periods, but TTL may do better during a burst. This requires an adaptive policy that could choose between different policies, such as our IndHist-AD.

Few objects in the World Cup trace exhibited this behavior of cyclic patterns and bursts. We modified the trace data as follows to generate such objects.

We randomly selected 55 of the most popular bursty objects with respect to client requests and mapped them to 55 of the most popular cyclic objects. In our experiments, we treated each bursty/cyclic pair as a single object. These 55 merged objects exhibited cyclic update patterns for most of the 8 days, but experienced bursts of updates on one day.

Policies

We compare TTL, the non-adaptive IndHist policy (IndHist-NA), and the adaptive IndHist (IndHist-AD) policy described above. We evaluate the policies on a "combined" trace of 55 merged objects and the cyclic objects. We ran these experiments on the first 8 days of our 15 days of trace data. We used the first 4 days to gather history information, and report results on the remaining 4 days. For comparison purposes, we also report on results for the cyclic objects during the same period.

For IndHist-AD, recall that we estimate when a burst occurred by considering the number of updates in a window W. IndHist-AD will use TTL whenever $f(\lambda, \lambda^*)$ in a window of size W exceeds the threshold T. In our experiments, we report results for $W = 1$ hour and $W = 24$ hours, and $T = 2$.

Results

We show the number of validations and stale hits for selected values of α (TTL) and θ (IndHist) when $W = 1$ in Table 5.4. In the first 3 rows of Table 5.4, the number of useful validations are similar for each of the three techniques, while TTL has many more total validations. IndHist-AD has fewer stale hits than IndHist-NA. In the next 3 rows, the total number of validations of all three

Policy	Total Vals	Useful Vals	Useless Vals	Stale Hits
TTL α=0.05	1368170	371353	996817	14936
Hist-AD θ=0.05	988331	357034	631297	76115
Hist-NA θ=0.05	932469	351317	581152	94010
TTL α=0.30	755100	350135	404965	136092
Hist-AD θ=0.30	798578	348986	449592	116855
Hist-NA θ=0.30	764919	341186	423733	139349

Table 5.4: Validations and Stale Hits for Bursty objects when W=1 hour

techniques are similar. We note that IndHist-AD has fewer stale hits than TTL. It also has fewer stale hits than IndHist-NA, and is thus better suited to bursts.

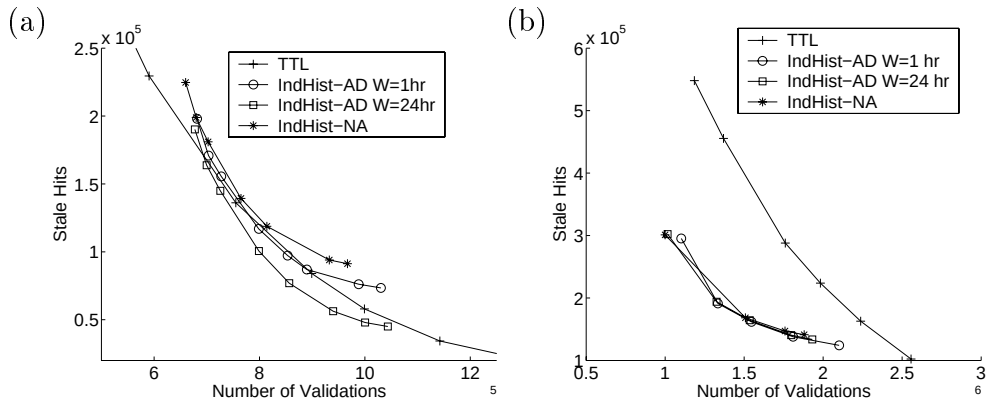

Figure 5.11: Effect of Tuning TTL, IndHist-AD, and IndHist-NA on data freshness for (a) bursty and (b) cyclic objects

We compare the performance of TTL, IndHist-NA, and IndHist-AD. For all policies, we varied the tuning parameter from 0.02 to 0.7. We plot the number of stale hits versus the total number of validations in Figure 5.11(a). As expected,

TTL outperforms IndHist-NA for the bursty objects. This is because TTL assumes that objects that have been updated recently are more likely to be updated in the near future, so it is well-suited for bursty data. In contrast, IndHist-NA assumes that an object's update patterns will be consistent with its past update history, so it cannot handle bursts as well. However, IndHist-AD offers some improvement over IndHist-NA, especially as the total number of validations increases. This shows that IndHist-AD can detect some bursts in updates and chooses TTL when appropriate. IndHist-AD with W=24 can provide fewer stale hits, and in most cases even provides fresher data than TTL for the same number of validations. This suggests that larger values of W may be more effective at detecting bursts.

We also compare the performance of IndHist-NA and IndHist-AD on the cyclic objects. Our goal is to ensure that IndHist-AD performs well on cyclic objects as well as IndHist-NA. We plot these results in Figure 5.11 (b). The key observation is that IndHist-AD has comparable performance to IndHist-NA for cyclic objects, so it can generalize to both cyclic and bursty objects.

5.5 Summary and Open Problems

In this chapter we have shown how to improve pull-based cache consistency using server cooperation. Specifically we have shown the following:

- For cyclic objects, using either individual or aggregate update history information improves the accuracy of the estimates of the number of updates to a cached object.

- For cyclic objects, for a given level of freshness, using either individual or

aggregate histories performs significantly fewer validations than TTL.

- An adaptive policy that can choose between individual history and TTL performs well for both cyclic and bursty objects.

There are several areas for future work. These include the following:

- **Grouping Objects:** An open problem is determining the best way to group objects for modeling aggregate history. Smaller groups of objects may improve the accuracy of the approximation but also increase storage overhead, so there is a tradeoff between storage overhead and accuracy.

- **Identifying Bursts:** In this chapter we presented heuristics to identify bursts of updates, however, additional research is needed to fine tune these heuristics. Different heuristics and parameter settings may be most effective depending on the degree of frequency, predictability, and burstiness of object update patterns.

- **Distributed Updates:** In many systems updates may occur in multiple locations rather than at a single server. These updates may be independent, or they may be related, so modeling updates introduces new challenges. Effective techniques to model updates in these contexts is an area of future work.

- **Data Recharging:** Another problem that relates to modeling updates is data recharging. If a client is connected to the Internet for a limited period of time, a challenge is determining which objects are most likely to be refreshed during the connection period and schedule refreshes accordingly. For example, if an object is most likely to be updated near the end of

the connection period, it should be scheduled to be refreshed at this time. Conflicts could occur if many objects are likely to be updated near the end of the connection period, in this case techniques to prioritize objects based on both client preferences and the accuracy of the model are needed.

- **Server Cooperation Issues:** There are many open problems related to implementations and architectures for server cooperation. One area of future work is studying the effect of different server cooperation schemes on the performance of both clients and servers. If a policy requires too much work at the server, it could have a negative impact on performance and increase latencies for all clients. Similarly, if a policy requires too much work at the client, it could reduce the benefits of caching. A related area of future work is developing server cooperation architectures that place an appropriate load on both clients and servers.

Chapter 6

Profiles in Mobile Environments

In Chapter 4 we showed the benefits of using client profiles for caching decisions on fixed networks. In this chapter we present a framework for mobile clients to communicate profiles to proxy caches and base stations, and for base stations and caches to support diverse mobile applications [18]. Profiles can improve mobile data access in two ways. First, using profiles in caching decisions at a proxy cache at the wireless base station can reduce fixed network latencies by increasing the number of requests served from the cache. Second, using profiles for scheduling decisions at the wireless base station can provide different priorities to different classes of applications, and can reduce the latency of high priority applications.

We first describe our framework for profile-based data delivery in mobile environments, including possible deployment scenarios, parameters, and our profile-based data delivery algorithm (labelled Profile). Finally, we present simulation results (we present implementation results in Chapter 7).

Our main results are as follows:

- Using profiles for both caching and scheduling decisions at a wireless base station can reduce end-to-end latencies and differentiate services for mobile clients.

- Clients who have too many high priority applications increase the latencies of all applications, which gives all clients an incentive to cooperate.
- Using handoff profiles can mitigate the effects of delays when clients migrate to a neighboring cell by giving outstanding requests higher priority in the new cell and/or using cached data.

6.1 Profile-Based Data Delivery

Our framework for mobile client profiles allows clients or service providers to provide different levels of service for different applications. It is based on the observation that mobile clients typically have different QoS requirements for different applications. Some applications, e.g., instant messaging, require low latency, while others, e.g., file transfer, email, will tolerate higher latencies. In addition, our framework includes *handoff profiles*, which can give higher priority to requests during handoffs and mitigate the effects of handoff delays. Unlike prior work in this area, e.g., [81, 94, 99], our profile-based scheme does not require any changes to the network layer.

We first describe potential deployment scenarios and granularities of profiles. We then describe the profile parameters, and how to choose values for them. Finally, we describe the details of the profile-based data delivery algorithm.

6.1.1 Profile Deployment and Granularity

Profiles can be set by either clients or wireless service providers. If clients set their profiles, parameter values can be appended to requests and passed to the base station, assuming that profiles can be encoded suitably succinctly (see Sec-

application	site	T_A	T_L	Priority
1.E-mail	-	-	-	Low
2.IM	-	-	-	High
3.Stock	`finance.yahoo.com`	0	2 sec	High
4.Sports	`www.espn.com`	0	2 sec	Low
5.News	`www.cnn.com`	2	1 sec	High
6.Weather	`www.weather.com`	2	1 sec	Low
7.Default	-	0	10 sec	Low

Figure 6.1: An Example Profile

tion 6.1.4). Thus, the base station does not need to store any profile specific information, and the profiles themselves do not add extra storage or retrieval overhead at the base station. Also, clients can set and tune their profiles locally, without the overhead of communicating with the base station.

We present an example profile in Figure 6.1.1. In this example, a client accesses both instant messages and email on her PDA (rows 1 and 2 in Figure 6.1.1), as well as several cacheable web applications. We note that, as in Chapter 4, clients can distinguish between different applications of the same request type, e.g. HTTP requests, by setting different profiles for different domain names. We will discuss the details of the profile parameters in Section 6.1.3.

Alternatively, service providers could set profiles. In this case, the service provider could map profiles to either clients or applications. If the service provider maps clients to profiles, the scheme is pricing-based, with higher paying clients receiving relatively better service. If the service provider maps applications to profiles, the scheme is performance based, and the aim is to provide low latency to time sensitive applications. In the remainder of this dissertation, we assume

that clients set their own profiles.

For each application, clients set two profiles, regular and handoff. Initially, a client makes a request using the regular profile for the application. However, if the client migrates to a neighboring cell before the request is served, the client's mobile device can communicate the handoff profile to the new base station. The request will be served using the handoff profile in the new cell. We describe how to choose parameters for handoff profiles in Section 6.1.3.

6.1.2 Assumptions and Restrictions

We make the following assumptions:

- Clients share the available wireless downlink bandwidth, and objects are unicast from the base station to the clients. This is how cellular providers currently serve mobile clients on a data channel.

- We assume that base stations are equipped with functionality to make caching and scheduling decisions based on profiles. Another feasible alternative is to implement the profile specific functionality at a host colocated with the base station. We do not consider such implementation-specific issues further in this dissertation; instead, we use the generic term base station to refer to the entity with the caching and scheduling functionality.

- We assume that every object has a Time-to-Live (TTL) [57], either assigned by a remote server or estimated using heuristics. We do not consider the effects of inaccurate TTL estimates on our results. Our implementations of the different downloading polices rely on the same estimates of the update

frequencies of objects; therefore the effects of inaccurate TTL estimates on all policies are comparable.

- We assume that the time to read an object from the cache is negligible; therefore all cached objects are available to be sent to clients immediately. This assumption is reasonable because latency on a wide area fixed network is typically much higher than the latency of accessing a local cache.

- We do not consider the cost of sending requests on the wireless uplink, as the sizes of these requests are typically much smaller than the objects transmitted on the downlink.

6.1.3 Profile Parameters

Our profiles include three parameters: a target latency T_L and a target age T_A as described in Chapter 4, and a *priority*. Given a request for an object O_i, T_L corresponds to l_i from Section 2.5, T_A corresponds to r_i, and *priority* determines the object's position in the service queue Q (described further in Section 6.1.5). We describe how to choose these parameters for specific applications.

Choosing Priority

The first parameter is a *priority* which is used to schedule objects that are available for delivery on the wireless downlink. We specify the *priority* for applications as either *LowPriority* (e.g., casual web browsing, email) or *HighPriority* (e.g., instant messaging). In our example in Figure 6.1.1, the client prefers to receive her instant messages as quickly as possible, but for her email she will tolerate some delay. Therefore, she sets her email to *LowPriority* and her instant messages to

HighPriority. She assigns priorities to her cacheable web applications in a similar manner. The stock and news are *HighPriority*, and the sports and weather are *LowPriority*. Intuitively, *HighPriority* should be used for requests where it is important to deliver data as quickly as possible, and *LowPriority* should be used for requests that can tolerate higher latencies if necessary. We describe how use these priorities for scheduling in Section 6.1.5.

Choosing Target Latency

Clients assign a target latency T_L to each request as in Chapter 4. T_L indicates how long a client is willing to wait for fresh data, and is used to determine when to download an object and when to use a cached copy. It differs from *priority* which is used for scheduling objects that are available for delivery to clients. The T_L parameter is used only to determine when to deliver a cached object to clients and when to download a fresh copy.

Note that both *HighPriorty* and *LowPriority* applications can have either high or low T_L values. For example, consider the weather application on row 6 of Figure 6.1.1. The weather application has its T_L set to 1 second, (i.e., lower latency), but a priority of *LowPriority* (i.e., higher latency). Intuitively, *LowPriority* means that the request has lower priority than requests such as instant messaging where low latency is critical. A lower T_L value means that the client will tolerate stale cached data if it can be delivered quickly.

Choosing Recency

In addition to specifying a target latency T_L, clients specify a target age T_A. Recall that age is defined as the number of times an object has been updated at

the remote server since it was cached. Clients who require the most recent data set T_A to 0.

In our example, the client sets the T_A of the stock and news applications to 0 to indicate that she prefer more recent data, and set the T_A of sports and weather to 2 updates to indicate that she will tolerate some staleness.

Choosing a Handoff Profile

Clients can choose handoff profiles for each application. These allow requests to be served more promptly in the new cell and mitigate the effects of handoff delays.

Clearly, the choice of handoff profile is linked to the choice of profile for non-handoff requests to be effective. For example, a handoff profile may choose *HighPriority* for an application whose non-handoff profile was *LowPriority*. Alternatively, one may choose a higher T_A value to indicate that stale cached data is acceptable, or both. We present example handoff profiles in Section 6.2.2.

6.1.4 Configuring and Communicating Profiles

As described in Section 6.1.1, profiles can be specified at a granularity determined by the client and/or service provider. Clients can communicate their profiles to the base station assuming the profile parameters can be encoded succinctly, for example by including them in a request's HTTP header or in the IP TOS byte [84]. This eliminates the overhead of storing profiles at the base station and gives greater flexibility to the client.

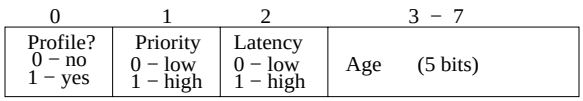

Figure 6.2: Using an IP TOS byte to communicate profiles

A Possible Implementation

Profile could be implemented by adapting the IP TOS byte as in DiffServ[84]. Setting this byte in each request packet would allow profile parameters to easily be communicated to a base station. A possible scheme for doing so is shown in Figure 6.2. The first bit indicates whether or not this byte contains profile information. The second bit indicates whether the request is *HighPriority* or *LowPriority*. The third bit indicates whether T_L is *HighLatency* or *LowLatency*. The remaining 5 bits indicate the value of T_A.

Clearly this scheme would not be feasible in a DiffServ[84] network. However, such a network would provide QoS beyond what we are proposing. Further, in this scheme, we use a network header field to specify application preferences, However, recall that the TOS byte was initially designed to specify service classes for IP datagrams[46], which is essentially what we are using it for. If layer violations are a concern, an application layer shim encoder can be used to carry profile information.

6.1.5 Profile-Based Data Delivery Algorithm

The profile-based data delivery algorithm (Profile) combines making a caching decision at the base station proxy cache with scheduling data delivery on the

wireless downlink. Recall that in Chapter 4 we presented a profile-based downloading scheme on fixed networks that relies on scores of how well the latency and the age of an object meet the client's target values. However, in mobile environments, the latency score of an object must consider the delay on the wireless link in addition to the fixed network latencies. In this section we define a modified profile-based downloading scheme for mobile clients that incorporates traffic on the wireless downlink into its latency scores.

We first describe the profile-based downloading and profile-based scheduling schemes, then present the combined algorithm.

Profile-Based Downloading

When a request arrives for an object O_i, the base station must first make a decision about whether to validate a cached object at the remote server or deliver a possibly stale cached copy without validation.

We define a combined decision function that considers latency on the wireless downlink as the weighted average of `Score(`T_L`, Latency, `K_L`)`, where T_L corresponds to l_i in Chapter 2 and `Latency` corresponds to L_i, and `Score(`T_A`, Age, `K_A`)`, where T_A corresponds to r_i and `Age` corresponds to R_i. We modify the formulas for `DownloadScore` and `CacheScore` from Equations 4.1 and 4.2 as follows:

We define `Svc(`Q`)` as the expected amount of time it will take to serve outstanding requests in Q with higher priority on the wireless downlink. We use this information, combined with the object's size and downlink bandwidth, to estimate the transmission time of an object. Since downloading always provides the most recent data, `Score(`T_A`, Age, `K_A`)` is always 1.0. Therefore, we modify

Equation 4.1 to compute `DownloadScore` as:

$$\texttt{DownloadScore} = (1-w) * 1.0 + w * \texttt{Score}(\texttt{T}_\texttt{L}, \max(\texttt{Svc}(\texttt{Q}), \texttt{Latency}), \texttt{K}_\texttt{L}) \quad (6.1)$$

We estimate the latency to be the maximum of the service time of objects remaining in the queue ahead of this object and the fixed network latency of downloading the object.

We now consider `CacheScore`. Recall that when an object is read from the cache, its fixed network latency is 0. However, in mobile environments we must also consider the latency on the wireless downlink. Therefore, we modify Equation 4.2 and compute `CacheScore` as:

$$\texttt{CacheScore} = (1-w) * \texttt{Score}(T_A, \texttt{Age}, K_A) + w * \mathit{Score}(\texttt{T}_\texttt{L}, \texttt{Svc}(\texttt{Q}), K_L) \quad (6.2)$$

Note that when `Svc(queue)` is greater than the fixed network latency, `DownloadScore` > `CacheScore`. This means that when the downlink is congested, more recent data can be downloaded in response to client requests because clients have to wait for their data anyway.

Profile-Based Scheduling

When a requested object becomes available at the base station, it is added to the service queue Q and is scheduled to be sent to the client via the wireless downlink. In our profile-based scheme, the base station transmits objects sequentially using a scheduling algorithm that considers priorities. We first discuss some issues in developing a priority-based scheduling algorithm. We then present our algorithm.

Issues and Challenges In general, there are several desirable qualities for scheduling algorithms. First, we want to avoid starvation and ensure that all requests will eventually be served. Second, any scheduling scheme should be deployable with minimal overhead at the wireless base station. A third desirable quality is fairness. All clients should have a fair share of *HighPriority* requests. In addition, the performance should degrade gracefully in heavy workloads, i.e., when it is impossible to meet the desired latencies of all requests, we should be able to control how the algorithm behaves. A final desirable quality is flexibility. In this chapter we present a scheme that allows clients to map their applications to two different classes of priority, and provides the same level of service to all applications in each class. However, a scheme that allows clients to set their own deadlines for different classes of applications and to have more than two levels of priority would be useful. We discuss preliminary work in this direction at the end of this chapter.

Scheduling Algorithm Our scheduling algorithm is based on EDF scheduling [101]. This is straightforward to implement and guarantees that all requests will eventually be served. We discuss fairness issues in Section 6.2.2.

Our EDF based scheduling scheme maps *HighPriority* and *LowPriority* requests to deadlines, then uses Earliest Deadline First (EDF) scheduling [101] to schedule objects for delivery at the base station. The object is inserted into the service queue Q in position Q_j such that for all $i < j$, $deadline(Q_i) \leq deadline((Q_j)$, and for all $i > j$, $deadline(Q_i) > deadline((Q_j)$. This scheme assures that *LowPriority* requests will not get starved.

We use two parameters, Δ_1 and Δ_2, to set the deadlines of *HighPriority* and *LowPriority* requests. Let t be the time that a request for an object arrives

PROFILE-BASED DATA DELIVERY(Object O, Profile P)

```
if O is in cache
    use P to compute CacheScore and DownloadScore
    { using equations 6.2 and 6.1 }
    if DownloadScore > CacheScore
        Request O from remote server
        When O arrives:
            ENQUEUE_EDF(O, Q)
            Refresh O in cache
    else
        ENQUEUE_EDF(O, Q)
    end if
else      { O is not in cache }
    Request O from remote server
    When O arrives:
        ENQUEUE_EDF(O, Q)
        Insert O in cache
        if cache is full
            replace using LRU
        end if
end if
```

Figure 6.3: Profile-Based Data Delivery Algorithm

at the base station. We map requests to deadlines as follows. If the request is *HighPriority*, we set its deadline to $t+\Delta_1$. If the request is *LowPriority*, we set its deadline to $t+\Delta_2$. Note that the different between Δ_1 and Δ_2 controls the relative latencies of each type of request. Under heavy workloads, when Δ_2 is much larger than Δ_1, *HighPriority* latencies will stay low while *LowPriority* latencies will increase. When $\Delta_1 = \Delta_2$, *LowPriority* and *HighPriority* requests have identical priorities and are processed in a first come, first served manner.

Under lighter workloads, most requests will meet the deadlines set by their Δ_1 and Δ_2 values. Under heavier workloads, these values will affect the relative latencies of HighPriority and LowPriority requests and determine how many of each group will meet their deadlines. We study the affects of varying Δ values in Section 6.2.2.

All of the components described above are integrated into a single algorithm at the base station. This algorithm is shown in Figure 6.1.5.

6.2 Simulation Results

Our simulation results show that using profiles for both caching and scheduling can reduce end-to-end latencies and differentiate service for *HighPriority* and *LowPriority* applications. We first describe our simulation environment and parameters. We then present our results. (We present implementation results in Chapter 7). Our key results are as follows:

- Profiles are effective at differentiating service of different applications.
- Effective use of the proxy cache can improve the latency of requests, even when there is contention on the wireless downlink.

- Too many *HighPriority* requests increase the latency of *all* requests. This gives clients or service providers an incentive to differentiate applications into *HighPriority* and *LowPriority*, and ensure that there are not too many *HighPriority* applications.

- Using handoff profiles can significantly improve the latency of handoff requests compared to a naive approach that does not use profiles. The handoff profile can benefit from both using cached data *and* assigning higher priority to handoff requests.

6.2.1 Simulation Model and Environment

We modeled a wireless downlink of 128 kbps, which is representative of emerging Third Generation (3G) [105] wireless networks. This is a dedicated data channel shared by multiple clients at a base station. Client requests were uniformly distributed between the 4 applications shown in Table 6.2.1, with about 25% of requests for each type of application. We discuss the effects of varying distributions later in Section 6.2.2.

Parameters

We considered a world of 100,000 objects. We used the following parameters in our simulation; they are summarized in Table 6.2.1.

- *Object Size and Popularity*: We ran experiments with both variable and uniform object sizes. For simplicity, we report on the results for the uniform object size; results for variable sizes were comparable. We considered objects with size 12.8 kbits. These are representative of data that web sites currently provide to wireless web clients [31].

We skewed object popularity using a Zipf-like distribution. Work reported in [17] showed that web accesses typically follow a Zipf-like distribution, where the ith most popular object has popularity proportional to $1/i^{\theta}$, where θ is a value between 0 and 1.0. We generated a distribution with θ= 0.7, which was typical of the web traces analyzed in [17].

- *Request Arrival Rate*: We generated requests with exponential interarrival times, and considered mean request arrival rates between 0 - 10 requests per second. This corresponds to 0 - 100% utilization on the wireless downlink, assuming no channel error. Studying the effects of varying signal strengths and channel errors is an area of future work and will be discussed in Chapter 8.

- *Fixed Network Latency*: This is the estimated time to download an object from a server on the fixed network. To model fixed network latencies, we used trace data from NLANR [50]. The data was gathered from client-side web proxy caches at several sites on June 27, 2001 and consisted of approximately 1.3 million requests. To reduce the effects of network and server errors we considered only requests with latencies of less than 5000 msec. The distribution of these values was highly skewed, with a median of approximately 200 msec and a mean of approximately 500 msec. 90% of the requests had latencies less than 1400 msec.

- *Priority*: We considered two different priority values, *HighPriority* and *Low-Priority*. We present the corresponding Δ_1 and Δ_2 values in Table 6.2.1.

- *Update Rate*: This is the frequency with which an object is updated. In our simulation this value was uniformly distributed in the range of once every

10 minutes to once every 2 hours.

- *Cache Size*: We considered a default cache size of 160 MB (about 10% of the world size).

Setup

We implemented a request-level simulation environment in C++. We ran all simulations on a Sun Sparc 20 workstation running Solaris 2.6. We assumed an initially empty cache. When the cache was full, we evicted objects using the Least Recently Used (LRU) policy. To keep cached objects up to date, we refreshed objects in the background at a rate inversely proportional to the workloads. This ensured that even at low workloads, the cache was reasonably fresh and increased the number of requests that could be served from the cache. The cache was refreshed in a fixed order in a round-robin manner; this strategy was shown to be near-optimal in [35].

We ran each simulation for 40000 requests to warm up the cache, then ran the simulation for an additional 80000 requests to collect measurements. We repeated each experiment 10 times and ran 95% confidence intervals. These intervals were very small in all cases and we do not show them in our plots. The settings of the profiles are shown in Table 6.2.1. Note that for the uncacheable requests, there is no need to set values for T_A and T_L, because these objects will always be downloaded.

Algorithms and Metrics

We compare *Profile*, our profile-based downloading approach combined with EDF scheduling, against a profile-unaware scheme (*No Profiles*), which uses traditional

Parameter	Range
Request Rate	0 - 10 requests/sec
Downlink Bandwidth	128 kbps
Object Size	12.8 Kbits
Downlink Latency	100 msec
Workload	0 - 10 requests/sec
Update Rate	10 min - 2 hrs
Cache Size	160 MB
Fixed Netwk Latency	0 - 5000 msec
Median Fixed Netwk Lat	200 msec
Δ_1	300 msec
Δ_2	900 msec

Table 6.1: Simulation Parameters

Application	T_A	T_L	priority
1 (stock)	0	900 msec	HighPriority
2 (news)	2	300 msec	LowPriority
3 (IM)	N/A	N/A	HighPriority
4 (email)	N/A	N/A	LowPriority

Table 6.2: Application Profiles

TTL[30, 57] cache consistency and FIFO scheduling. For the No Profiles approach, if a requested object is in the cache, it is delivered to clients only if its TTL has not expired. Otherwise a fresh copy is downloaded from the remote server.

We report on the following metrics:

- **Average End-to-End Latency** This is the time elapsed from when a request arrives at the base station to when the last bit is delivered to the client.
- **Average Wireless Latency** This is the time elapsed from when the data becomes available for delivery at the base station to when the last bit is delivered to the client.

6.2.2 Results

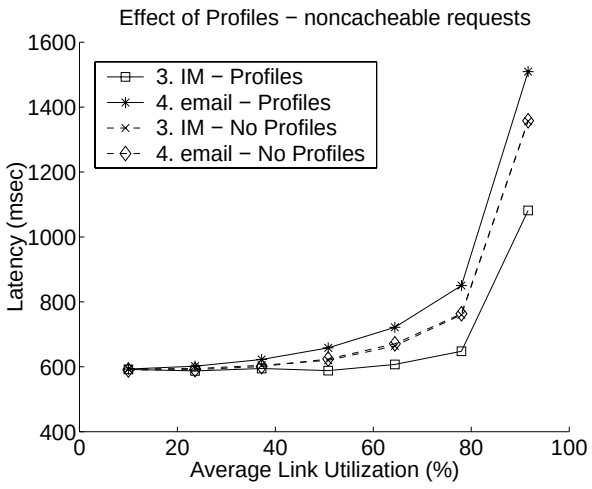

Figure 6.4: Effect of Using Profiles

Effect of Using Profiles

Our first set of results compares the latency of the two non-cacheable applications (IM and Email) both with and without profiles. Recall that when profiles are not used, objects are scheduled for delivery using FIFO scheduling. The results are plotted in Figure 6.4. The key observation is that Profile effectively differentiates services for the two applications. Under light workloads, there is little contention for the wireless downlink and all applications have the same average latency regardless of the use of profiles. However, as the workload increases, the latency of both the Email and the IM requests increase at the same rate when FIFO scheduling is used. In contrast, Profile trades off the email response time to maintain a low latency for IM requests. The important observation is that at up to 80% utilization, the latency of the IM requests does not increase significantly. When the utilization increases beyond 80%, even Profile cannot maintain constant response time, but the relative gain from using profiles *increases*.

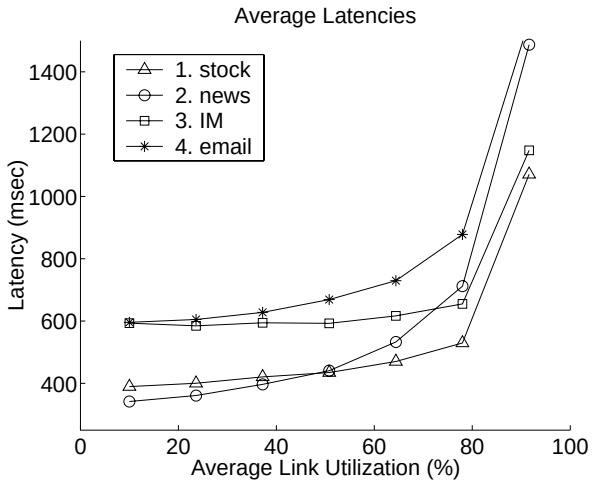

Figure 6.5: Average Latencies for Different Applications

We now consider how Profile differentiates services for all four applications. Figure 6.5 plots the latencies for all applications as a function of utilization on the downlink. The first observation is that at less than 50% utilization, the cacheable applications (stock and news) have lower latencies than the non-cacheable applications (IM and Email) because the cache reduces the fixed network latencies. Thus, under lighter workloads, caching data at the base station can significantly reduce the latencies of cacheable objects. A second observation is that using cached data improves the latencies of the cacheable applications (stock and news), independent of priority. At less than 50% utilization, the news requests have a slightly lower average latency than the stock requests. This is because the news application can tolerate stale cached data, while the stock requests require the most recent data.

As the workload increases, however, Profile is able to maintain the low latency of stock requests due to their *HighPriority* deadline. In contrast, the latency of the news requests increases. Thus, Profile can differentiate services and keep the latencies of the *HighPriority* stock requests relatively constant while the latencies of the *LowPriority* requests increase. At up to 80% utilization, both cacheable applications (stock and news) have lower latencies than both non-cacheable applications (IM and email). Thus, caching is beneficial even for *LowPriority* applications.

We now consider the benefits of using Profile priorities in EDF scheduling. Figure 6.6 plots the wireless downlink latencies for all four applications, i.e. the amount of time to deliver data to clients *after* the data becomes available at the base station. For comparison, we plot the latency of No Profiles (TTL cache consistency and FIFO scheduling) for all requests. Recall that we use a fixed file

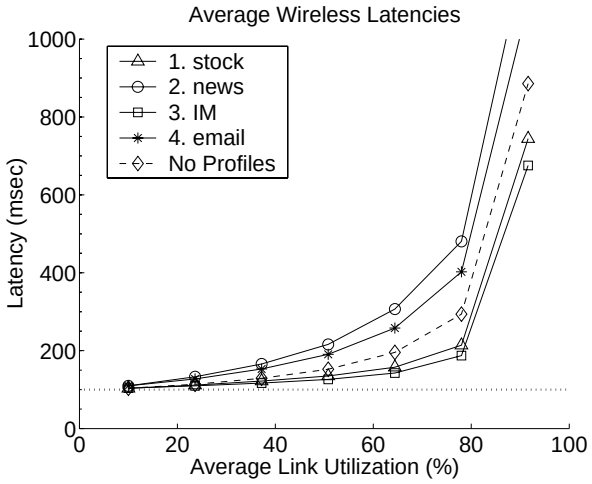

Figure 6.6: Average Wireless Downlink Latencies for Different Applications Using Profiles

size of 12.8 kbits and a downlink bandwidth of 128 kbps, so the minimum latency on the wireless downlink is 100 msec, shown by the horizontal dotted line. The key observation is that the *HighPriority* requests (stock and IM) have latencies within 25% of 100msec at up to 50% utilization. Thus, scheduling delay at the wireless base station has a minimal effect on the latency of these requests. We conclude that under reasonable workloads, EDF scheduling at the base station can control the wireless downlink latencies of different applications.

Effect of Percentage of *HighPriority* requests

Previously we assumed that all clients are cooperative and assign *LowPriority* to applications such as email and news that can tolerate higher latencies. We now consider the effects of the number of *HighPriority* requests on the performance of the system. Our results show the need for either clients or service providers to assign different priorities to different applications to fully exploit the benefits of

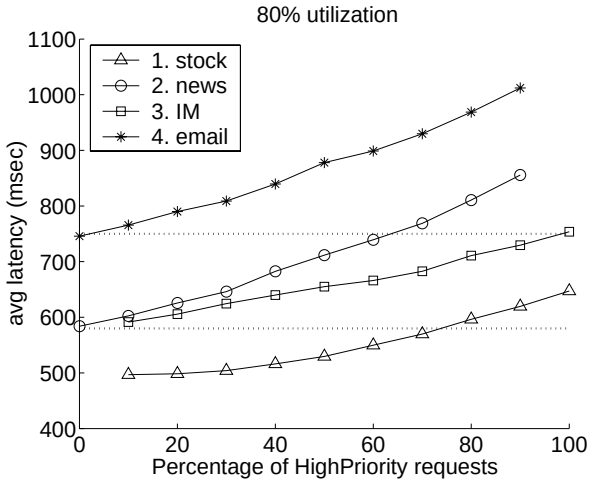

Figure 6.7: Effect of the Percentage of *HighPriority* Requests on Latencies

profiles.

We vary the percentage of *HighPriority* requests from 0-100%, and plot the latencies of all four applications when the downlink is at 80% utilization in Figure 6.7. For a given percentage of *HighPriority* (or *LowPriority*) requests, there was an equal number of cacheable and non-cacheable requests. For example, when 40% of requests were *HighPriority*, this corresponds to 20% IM, 20% stock, and the 60% *LowPriority* requests were 30% email, and 30% news.

We plot the results in Figure 6.7. As the percentage of *HighPriority* requests increases, the latency of all applications increases. We first compare the performance of the non cacheable applications (IM and Email). When all requests are *LowPriority* (i.e., 0% *HighPriority*), the latency of Email is 750 msec. When 0 - 40% of requests are *HighPriority*, the average latency of the IM requests is in the range 600-650 msec. As the percentage of *HighPriority* requests increases, the latency of the IM requests reaches the lowest latency of email.

We observe similar behavior for the cacheable requests. Consider the lowest latency of the *LowPriority* news requests. When all requests are *LowPriority*, the latency is 600 msec. If we consider the latency of the *HighPriority* stock requests, as the percentage of *HighPriority* requests increases, the latency of the stock requests exceeds 600 msec.

These results indicate that when there is a high percentage of *HighPriority* requests, profiles can no longer provide service differentiation and meet application requirements. This is because profiles trade off latencies of *LowPriority* requests to better serve *HighPriority* requests. This implies that in a network with only *HighPriority* requests, a relative service differentiation scheme such as profiles is not useful and some other mechanism such as per flow scheduling is required.

Effect of Varying Δ Values

We now consider how changing the Δ_1 and Δ_2 values affects the latencies of both *HighPriority* and *LowPriority* requests, for varying percentages of *HighPriority* and *LowPriority* requests. In these experiments we consider only wireless latencies, i.e., the amount of time it takes to deliver an object to a client after it becomes available at the base station. We consider workloads with different percentages of *HighPriority* and *LowPriority* deadlines and study the effects of varying Δ values. We do not consider the effects of fixed network latencies or caching.

Figure 6.8 plots the HighPriority and LowPriorityLatencies for three different distributions of HighPriority and LowPriority requests. In all experiments $\Delta_1 = 0$ and we considered the effect of increasing Δ_2 values. Figure 6.8 (a) plots the latencies when 25% of the requests are HighPriority. In this case, increasing the

value of Δ_2 has a significant impact on the average latencies of the *HighPriority* requests (solid lines) but little impact on the latencies of the *LowPriority* requests (dashed lines). This is because there are relatively few *HighPriority* requests, so it is easier to maintain low latencies for them. Increasing the Δ_2 value decreases the latencies of *HighPriority* requests under heavy workloads without a significant impact on the latencies of the *LowPriority* requests. For larger values of Δ_2, the latencies of the *HighPriority* requests decrease. For higher percentages of *HighPriority* requests (Figures 6.9 (b) and 6.9 (c)), increasing the value of Δ_2 has a significant impact on the latencies of *LowPriority* requests, but less impact on *HighPriority* requests. This shows that when there is a high percentage of *HighPriority* requests, the average latency of *HighPriority* requests is high for all Δ values. Further, increasing Δ_2 values can significantly increase the latency of the *LowPriority* requests.

Next, we consider the effects of varying Δ_1. Figure 6.9 plots the latencies of HighPriority Requests for varying Δ values. The key observation is that the difference between the two Δ values controls their relative latencies. For example, the latencies are identical when $\Delta_1 = 0$ and $\Delta_2 = 500$, and when $\Delta_1 = 100$ and $\Delta_2 = 600$. When the difference between the two values is larger the latencies of the *HighPriority* requests decrease and the latencies of *LowPriority* requests increase.

Effect of Profiles on Handoff Requests

Finally, we consider the effect of using Profiles to improve the latencies of handoff requests. For this experiment, we consider a single cacheable application. All non-handoff requests have priority set to *LowPriority*, T_A=0, and T_L=900 msec.

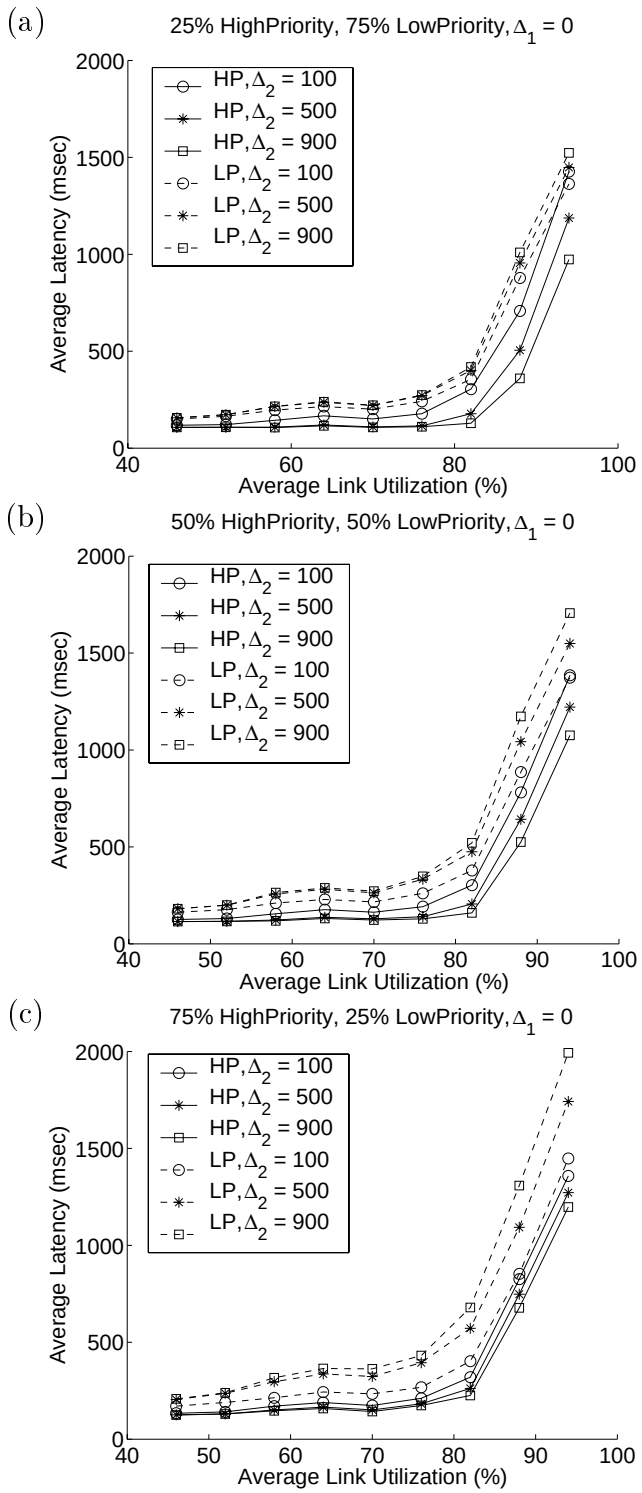

Figure 6.8: Effect of Varying Δ values for (a) 25% HighPriority Requests (b) 50% HighPriority Requests (c) 75% HighPriority Requests

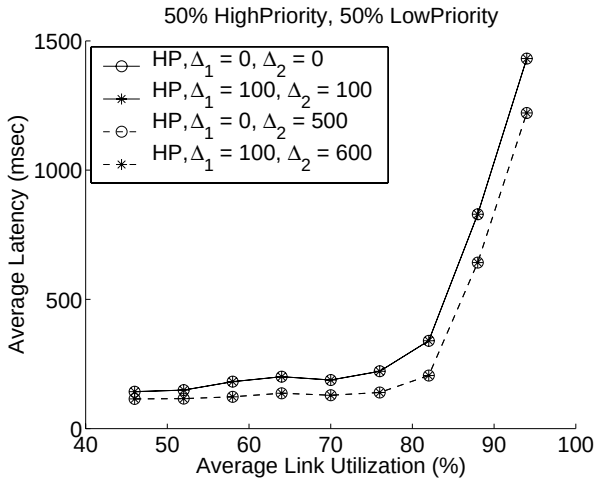

Figure 6.9: Effect of Varying Δ_1

This corresponds to casual web browsing where the client prefers the most recent data.

To study the potential benefits of exploiting both *HighPriority* and T_A (serving handoff requests with possibly stale cached data), we consider several possible handoff profiles as follows:

- *HighPriority, LowAge:* This handoff profile has priority=*HighPriority*, T_A=0, and T_L=900 msec. This profile has the benefits of giving handoff requests higher priority in a new cell and always uses fresh data.

- *LowPriority, HighAge:* This handoff profile has priority=*LowPriority*, T_A=2, and T_L=300 msec. This profile possible has the benefit of serving handoff requests with stale cached data in the new cell.

- *HighPriority, HighAge:* This handoff profile has priority=*HighPriority*, T_A=2, and T_L=300 msec. This profile has the combined benefits of serving handoff

requests with possibly stale data *and* giving them higher priority in the new cell.

- *naive:* In this case, no handoff profile is used. Handoff requests are treated as new requests when they enter the new cell, i.e., they do not receive higher priority than other requests, and use the non-handoff profile. Note that we make no assumptions about the underlying fixed network and do not assume the availability of multicast, e.g., [81, 94, 99] to reduce the effects of handoff delays.

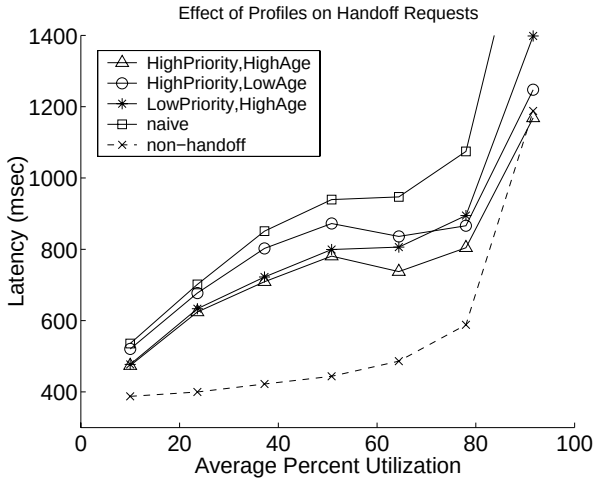

Figure 6.10: Effect of Profiles on Latencies of Handoff Requests

We plot the results in Figure 6.10. We show non-handoff average latency for comparison. As expected, the naive approach performs worse than all three handoff profiles. In contrast, profiles can offer up to 25% improvement over the naive approach. At up to 50% utilization, the *LowPriority, HighAge* has similar latency to the *HighPriority, HighAge*. Thus, at lower utilization, serving handoff requests from the cache significantly improves their latency. Recall that in

our simulation, in flight packets were lost during handoffs and were re-sent from the remote server. Thus, assigning higher priority to handoff requests does little to mitigate the effects of this delay, so *HighPriority, LowAge* does not benefit because it cannot serve stale data from the cache. In contrast, using the cache can mitigate this delay, allowing handoff requests to be served promptly. When the load on the downlink is greater than 60%, there is a greater benefit to giving handoff requests higher priority due to increased congestion on the wireless downlink. However, there is still some benefit to using the cache. A final observation is that the *HighPriority,HighAge* profile continues to have lower latency than the other handoff profiles under all workloads, which shows the benefits of combining both caching and higher priority to improve the latency of handoff requests.

6.3 Summary and Open Problems

In this chapter we have shown how profiles can improve data delivery for mobile clients. Specifically, we have shown the following:

- At lower workloads, caching can significantly reduce end to end latencies, independent of priority. At higher workloads, using priorities for scheduling can differentiate services for different classes of applications, but caching is still helpful under all workloads.

- Uncooperative clients that have too many *HighPriority* applications increase the latency of *all* requests. This motivates the need for a scheme to ensure that clients do not have more than their fair share of *HighPriority* requests.

- Using handoff profiles can mitigate the effects of delays when clients migrate to a neighboring cell before all their requests are served. Combining caching and higher scheduling priority improves the latencies of handoff requests under all workloads.

There are several open areas for future work. These include:

- **Fairness Issues:** As we showed in Section 6.2.2, clients who have a large percentage of *HighPriority* requests have a negative impact on the latencies of *all* requests. Thus, a mechanism to enforce fairness could improve latencies for cooperative clients who have a small number of *HighPriority* requests and punish clients who have a large number of such requests, which would improve latencies of all requests and give all clients an incentive to cooperate.

- **Adapting to Varying Workloads:** Under heavier workloads, the latencies of different applications may increase. To maintain the desired latency of the high priority applications, the deadlines of different applications may need to be adjusted according to the current workload.

- **Multiple Classes:** The scheduling scheme presented in this chapter provides only two classes of service, *HighPriority* and *LowPriority*, for all clients. This scheme does not consider clients who desire more than two classes. It also does not take into account that different clients may prefer different latencies for their different classes. For example, one client may prefer a higher Δ_2 value for their *LowPriority* applications in exchange for more *HighPriority* applications, while other clients may prefer a lower Δ_2

value (lower latency for the *LowPriority* applications) and fewer *HighPriority* applications. A scheme that takes into account these types of preferences could improve profile-based scheduling. We discuss preliminary work in this direction below.

- **Implementation Issues:** In this chapter we have not considered some common challenges on wireless networks such as varying signal strengths, disconnections, and packet loss, all of which could affect the performance of our algorithms. For example, a scheduling algorithm that takes into account both priority and the signal strength of the client could improve link utilization.

We briefly discuss preliminary work on a scheduling scheme that can enforce fairness, adapt to changes in workload, and support multiple classes. The idea is to allow clients to specify the desired *stretch* (i.e., ratio of the actual completion time of a request to its length) for different applications, then attempt to schedule data delivery to meet these target values.

Research reported in [3] presents a scheme to minimize the average stretch of all requests in a broadcast setting, but does not consider applications with different requirements. In [3], the scheduler guesses a target stretch value for all requests, and maps each request to a deadline based on this value. It then tries to schedule the request using EDF scheduling [101]. If it is impossible to meet all deadlines, the algorithm picks a new target using binary search, and continues until all deadlines are met.

This scheme could be adapted to accomodate requests with different target stretch values as follows. For each request, clients can choose a target stretch. Thus, each client can choose the latency most appropriate for each application,

and can have more than two different classes of applications. These values are mapped to deadlines and scheduled using EDF scheduling as in [3]. When it is impossible to meet all deadlines, the scheduler adjusts the deadlines proportional to the average latency of prior requests by that client. Thus, clients who have had many requests with low stretch will have their deadlines increased more. This can enforce fairness while aiming to meet the target stretch values of different applications.

Chapter 7

Implementation

In this chapter we report on our implementation of profiles using the Squid Proxy Cache [24]. Our implementation allows us to measure the impact of caching on end-to-end latencies for clients on both fixed and mobile networks. We consider the impact of caching on both fixed network latencies and recency of data, and study the effects of both caching and scheduling on data delivery to mobile clients.

Our main results are as follows:

- Using profiles for caching decisions can significantly improve cache utilization and reduce the number of freshness misses, while still providing fresh data in most cases.
- Using profiles for caching can significantly reduce end-to-end latencies for clients, even when there is congestion on the wireless downlink.
- As in our simulation results in Chapter 6, we show that using profiles for scheduling is effective at differentiating services under heavy workloads.

7.1 Setup

In our implementation, we consider a set of mobile clients in a single cell. We have implemented a prototype of our system consisting of two processes: a *client* to generate requests, and a *base station* to receive requests, access data from the fixed network or the cache, and schedule data delivery to the clients. The client and base station communicated using TCP. To simulate the low bandwidth wireless downlink between the base station and the clients, we controlled the rate that the base station sent data to the client. In our experiments we modeled a wireless downlink bandwidth of 128 Kbps. We ran all experiments on a Sun Blade in the domain `umiacs.umd.edu`. This machine was connected to its ISP via a high speed DS3 line with a maximum bandwidth of 27 Mbps.

We augmented the popular Squid [24] cache to include profiles. We modified the Squid refreshing mechanism to use profiles to make downloading decisions. To communicate the profiles to Squid, we created three new HTTP header fields: `Target-Age`, `Target-Latency`, and `Priority`. These parameters were appended to each request and used by Squid and the base station to make caching and scheduling decisions.

7.1.1 Parameters

We use the following parameters in our implementation (summarized in Table 7.1.1).

- *Request Arrival Rate:* We considered varying request rates from 1 request/sec to 25 requests/sec. We used the same trace for all experiments, and adjusted the workload by using a subset of the requests.

Parameter	Value
Request Rate	1-25 requests/sec
Downlink Bandwidth	128 Kbps
Avg. Object Size	4.8 Kbits
Cache Size	1 GB
Δ_1	1 sec
Δ_2	3 sec

Table 7.1: Implementation Parameters

- *Priority:* We considered two different priority values, *HighPriority* and *Low-Priority*. The corresponding Δ values are shown in Table 7.1.1.

- *Cache Size:* We considered a default cache size of 1 GB. When the cache was full, we evicted objects using LRU replacement. We do not consider the effects of cache size and cache replacement policies.

7.1.2 Comparing Alternative Approaches

An important challenge to evaluating our implementation was providing a fair comparison between Profile and a profile-unaware policy. In particular, when both policies download the same object, they should have the same fixed network latency for the object. Since an object's fixed network latency may vary considerably depending on factors such as network traffic and server workloads, it is impossible to reproduce fixed network latencies across multiple experiments.

Our solution to this problem was to run two different experiments simultaneously. We ran two base stations in parallel. For each requested object, the client generated two requests simultaneously, one for each base station. This

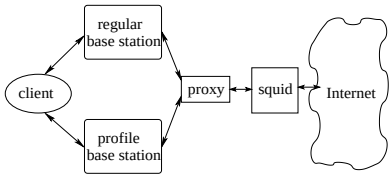

Figure 7.1: Prototype Architecture

architecture is shown in Figure 7.1.

To ensure that a downloaded object would have the same fixed network latency in both experiments, we implemented a *proxy* between the base stations and Squid. All communications between the base stations and Squid went through the proxy. If a requested object was downloaded in both experiments, the proxy would simultaneously deliver identical copies of a single downloaded object to both base stations. This ensured they would have comparable latencies. In our results, the fixed network latencies of objects downloaded in both experiments usually differed by less than 10 msec. This implementation also allowed us to compare the relative freshness of data delivered by the different policies. If an object was validated using TTL but served from the cache using Profile, the Squid logs indicated whether or not the object had actually changed at the remote server as described in Section 7.2.3

7.1.3 Workload Generation

We generated a workload using trace data from NLANR [50]. This data contains HTTP requests to a proxy cache in the United States in June 2002. We used a 3 hour portion of the trace that contained about 500,000 requests. The average file size was small: 75% of requests were for objects smaller than 5 Kbytes, 57%

% util	objs downloaded		objs from cache		% downloaded		% from cache	
	Profile	No Prof.	Profile	No Prof.	Profile	No Prof.	Profile	No Prof.
15%	17338	20746	8911	5773	66%	79%	34%	21%
57%	39098	44772	25429	19755	61%	69%	39%	31%
75%	54584	64051	52270	42803	51%	60%	49%	40%

Table 7.2: Comparison of Hit Rates with and without profiles

of requests were under 2 Kbytes.

We adapted this trace to model a wireless workload as follows. First, we considered only objects that were smaller than 3.2 Kbytes (about 68% of the requests in the trace). The average object size was about 600 bytes (4.8 Kbits). This ensured that large objects would not cause congestion on the limited downlink bandwidth, and this size is representative of objects from existing WML-enabled sites [31]. The trace had a fairly heavy workload of about 25 requests/sec (for objects under 3.2 Kbytes), which was near 100% utilization on the downlink. To model lighter workloads, we used a subset of the trace chosen uniformly at random over the same 3 hour period. We assumed an initially empty cache. We ran the experiments for 1 hour to warm up the cache, then collected measurements for the remaining two hours.

7.2 Results

We consider the effect of profiles on both *HighPriority* and *LowPriority* cacheable applications. We compared using profiles that could tolerate some staleness against using the traditional TTL approach as implemented in Squid. When profiles were used, we used parameters `Target-Age`= 1 update and `Target-Latency`= 1000 msec.

7.2.1 Effect of Profiles on Cache Utilization

Table 7.2 shows the number of requests downloaded and served from the cache at varying levels of downlink utilization. At lower utilization levels, objects in the cache get refreshed less frequently, and more objects need to be downloaded. As the utilization level increases, the percentage of requests served from the cache also increases. At both high and low utilization levels, profiles can increase the number of requests served from the cache by more than 20%. Thus, using profiles can potentially reduce the latency of a significant number of requests at both high and low workloads. Further, these results show that using profiles can reduce the total number of validations by up to 16%. As we will discuss in Section 7.2.3, nearly all of this reduction is due to useless validations.

7.2.2 Effect of Caching and Scheduling on Latency

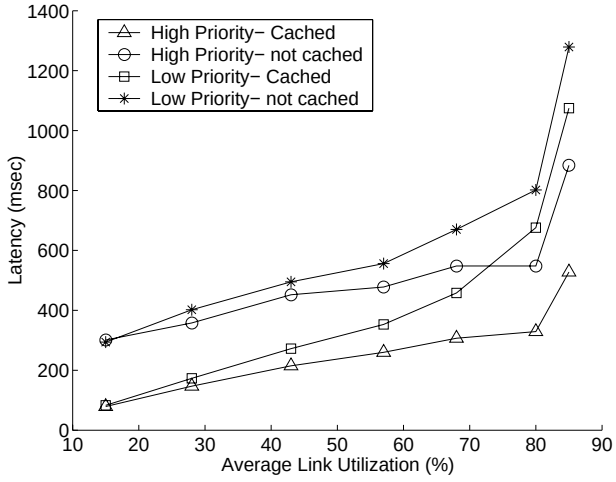

Figure 7.2: Effect of Using Profiles

We consider objects that were validated using TTL but served from the cache

using profiles. This shows the potential improvement in the latencies of cacheable requests when clients will tolerate stale data. Figure 7.2 plots the latencies as a function of utilization on the downlink. The first observation is that at less than 75% utilization, the objects served from the cache have lower latencies than the validated objects because the cache reduces the fixed network latencies. Thus, under lighter workloads, caching data at the base station can significantly reduce the latencies of cacheable objects. A second observation is that using cached data improves the latencies of the cacheable applications independent of priority.

As the workload increases, however, profiles are able to maintain the relatively low latency of the cached *HighPriority* requests. In contrast, the latency of the cached *LowPriority* requests increases. At higher levels of utilization, *HighPriority* requests have lower latency than the *LowPriority* requests, regardless of whether or not caching is used. This shows that EDF scheduling is effective at differentiating the latencies of different applications when there is congestion on the wireless downlink. However, for both *HighPriority* and *LowPriority* applications, cached objects have lower latency than validated objects. Thus, caching is beneficial even for *LowPriority* requests.

We now consider the latencies of individual requests, both with and without profiles. We ran an experiment where the request rate changed every 60 seconds. The request rate alternated between low (5 requests/sec, about 20% utilization), medium (10 requests/sec, about 35% utilization), and high (15 requests/sec, about 55% utilization). We consider a 3-minute period of the experiment starting at 7200 seconds (2 hours).

We plot the latencies of the *HighPriority* cacheable requests in Figure 7.3, and the *LowPriority* cacheable requests in Figure 7.4. In both of these graphs,

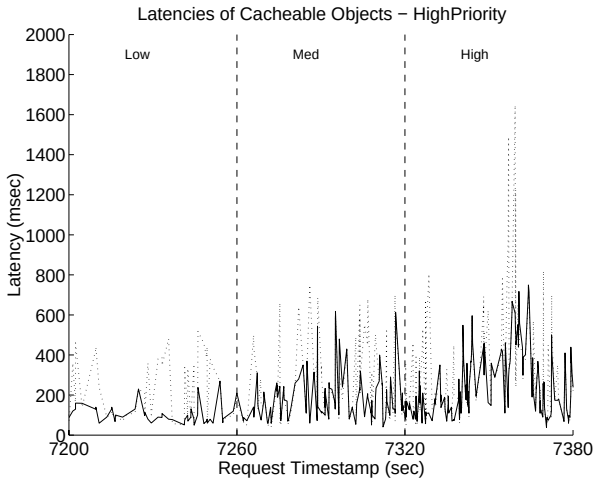

Figure 7.3: Comparison of Latencies of Cacheable HighPriority Requests

the x axis shows the request timestamp, i.e., the time the request was made, and the y axis shows the latency of the request. The first 60 seconds correspond to the low workload, the second 60 seconds the medium workload, and the final 60 seconds the high workload. The solid line indicates the latencies of the requests that used profiles, and the dotted line indicates the requests that used TTL.

We first consider the *HighPriority* requests in Figure 7.3. The first observation is that EDF scheduling is effective at maintaining low latency for the *HighPriority* requests. Even at high utilization, most requests have latencies under 500 msec. Another important observation is that caching significanly improves the latency of many requests under all workloads.

We now consider the *LowPriority* requests in Figure 7.4. As expected, fewer *LowPriority* requests benefit from caching due to congestion on the wireless downlink. However, some requests still benefit from caching, especially under lower workloads. Another observation is that while profiles reduce latencies on average,

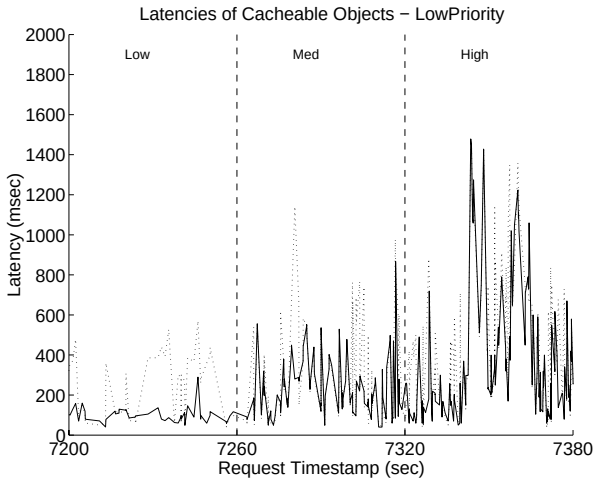

Figure 7.4: Comparison of Latencies of Cacheable LowPriority Requests

they may increase the latency of some individual requests, especially *LowPriority* requests. However, on average, profiles reduce the latency of most requests.

7.2.3 Effect on Age

We used the Squid logs to measure the age of the data returned to clients using profiles. Clearly we cannot know exactly how many times an object was updated at the remote server. However, the Squid logs indicate whether a validated object had actually changed at the remote server between two requests. Therefore, we can use these logs to measure the number of times profiles returned stale data to clients.

In the above experiment, over the 3-hour trace period, 10020 objects were validated using TTL but served from the cache using profiles. Of these objects, only 748 had actually changed at the remote server. Thus, even when profiles were used, clients received stale data only 7% of the time. This shows an additional

benefit to using profiles: they can reduce the overhead of freshness misses due to conservative TTL estimates, and is consistent with the trace data results from Chapter 4. This significantly improves latency while still providing fresh data in many cases.

Another key observation is that most of the validations using TTL were freshness misses. This means that the objects were not actually downloaded from the servers because they had not changed. However, our results in Figures 7.3 and 7.4 show that validating objects can increase access latencies by several hundred milliseconds in many cases. This supports the observation in [40] that validating objects may have latency as high as downloading the objects in many cases, and further motivates the need to reduce the number of times cached objects are validated.

7.3 Summary

In this chapter we have presented our implementation of profiles and evaluated their performance using a real web cache and real fixed network latencies. Specifically, we have shown the following:

- Using profiles for caching decisions can significantly reduce the number of freshness misses and increase cache utilization by more than 50%.
- Using profiles for caching decisions can reduce fixed network latencies, even though most validations are for objects that did not change and therefore do not need to be downloaded. This shows that validations can significantly increase access latencies, *even when the object hasn't changed*, and further motivates the need to reduce the number of validations.

- Using profiles for both caching and scheduling reduces end-to-end latencies for mobile clients, even when there is congestion on the wireless downlink. This is especially true for *HighPriority* requests, but caching can improve the latency of *LowPriority* requests as well.

Chapter 8

Conclusions and Future Work

8.1 Conclusion

The increased number and diversity of applications and services available on the Internet requires data delivery technologies that can be customized to meet the needs of clients while scaling to a large number of clients. This dissertation has addressed these challenges through client profiles and server cooperation. Client profiles provide a flexible, scalable framework for clients to communicate application-specific latency and recency preferences to a cache, and caches use these profiles to determine when to validate cached objects, and to determine the relative priority of delivering the data to mobile clients. Server cooperation enables servers to provide resource information to caches about the update histories of objects. This improves the ability of caches to estimate when objects will be updated at servers and improves the effectiveness of using profiles. Together, these two complementary techniques provide a framework for scalable customized data delivery for clients on both fixed and mobile networks.

Specifically, we have made the following contributions:

- We presented a flexible, scalable framework to support client profiles and

server cooperation.

- We have shown that using profiles for caching decisions on fixed networks can significantly reduce the number of object validations, while still delivering fresh data to clients in most cases. Using trace data we have shown that profiles can reduce the number of freshness misses by 16%-45%.

- We have evaluated the effects of server cooperation on maintaining data consistency and shown that using either individual or aggregated history information can significantly reduce the number of validations required to keep cached data fresh. This can provide bandwidth savings when connectivity is limited. We have also presented an adaptive policy to detect unpredictable bursts in object update patterns and choose between TTL or a history-based policy depending on the observed object behavior. While our model is not a statistical fit, experiments with trace data from 3 distinct datasets have verified the effectiveness of history-based policies. Using an object's update history to estimate freshness can reduce the total number of validations compared to using only the last update by 10%-36%, while providing a comparable level of freshness.

- We have shown that using profiles for both caching and scheduling data delivery on wireless networks can differentiate mobile applications without the overhead of end to end QoS deployment. Profiles can also mitigate the effects of handoff delays without requiring multicast or other changes to the underlying fixed network.

- We have presented an implementation of profiles for clients on both fixed and wireless networks using the Squid Proxy Cache [24]. Our implemen-

tation results show that profiles can significantly improve cache utilization and reduce fixed network latencies, while still delivering fresh data in most cases. This shows that validations can add significant overhead, even when a new object does not need to be downloaded, and provides further motivation to reduce the number of unnecessary validations. Further, our implementation results show that using profiles for both caching and scheduling can differentiate service for mobile clients, even when there is congestion on the wireless downlink.

8.2 Future Work

We plan to explore the following directions for future work:

8.2.1 Profiles

There are several areas of future work related to enabling clients to successfully use profiles. These include:

- **Usability:** In Section 4.1.4 we briefly discussed how a graphical interface can help users choose the appropriate settings for their profiles. However, more work is needed to develop an interface that is easy to use and allows clients to choose the most appropriate profiles for their applications.

- **Profiles used in Practice:** A related area of future work is studying what profiles clients use in practice, and their effect on performance. Determining what settings are most appropriate for different clients and applications and how much they improve latency and recency would help quantify the benefits of using profiles in practice.

- **Learning Profiles:** Another interesting area of future work is learning profiles based on past client behavior, network conditions, and object update frequencies. This could improve the choice of default profiles for clients, and aid clients in choosing the appropriate profiles for their different applications.

- **Fairness:** A final area of future research in profiles is studying fairness issues. For example, if there are bandwidth constraints, a challenge is meeting the requirements of as many clients as possible subject to these constraints. Another problem related to fairness is studying the effects of clients having different profiles for the same object. If some clients prefer the most recent data while others prefer low latency, an open question is how much will the low latency clients benefit because the cached data is fresh. Evaluating the impact of different profiles on performance and developing schemes to ensure fairness is another area of future work.

8.2.2 Modeling Updates

Our work in modeling updates presented in Chapter 5 also presents several interesting areas of future work. We summarize each of these below.

- **Grouping Objects:** In Chapter 5 we showed that aggregating the update patterns of mutliple objects with similar behavior can provide an approximation of individual object update patterns. However, an open problem is determining the best way to group objects. Smaller groups of objects may improve the accuracy of the approximation but also increase storage overhead, so there is a tradeoff between storage overhead and accuracy.

Determining when there is a benefit to having smaller groups of objects, and how to identify similar objects that should be grouped together is an interesting area of future work.

- **Identifying Bursts:** Another area of future research is in identifying bursts. In Chapter 5 we presented heuristics that can improve upon non-adaptive policies, however, additional research is needed to fine tune these heuristics. Different heuristics and parameter settings may be most effective depending on the degree of frequency, predictability, and burstiness of object update patterns.

- **Distributed Updates:** A related problem is studying update patterns in distributed environments, for example peer to peer systems or related objects at multiple servers. In Chapter 5 we studied techniques to model update patterns when objects are updated at a single location. However, in many systems updates may occur in multiple locations, and updates in different locations may or may not be correlated. One example is a peer to peer systems where different clients may update the same object. Modeling update patterns in this context could reduce the overhead of maintaining consistency. Another example is when related objects reside on multiple servers. For example, several news sources may update related objects, e.g., a story about a news event. These updates may be independent, or they may be related, so modeling updates introduces new challenges. Effective techniques to model updates in these contexts is an area of future work.

- **Data Recharging:** Another problem that relates to modeling updates is data recharging. This problem has received a considerable amount of atten-

tion in the literature, e.g., [34]. If a client is connected to the Internet for a limited period of time, a challenge is determining which objects are most likely to be refreshed during the connection period and schedule refreshes accordingly. For example, if an object is most likely to be updated near the end of the connection period, it should be scheduled to be refreshed at this time. Conflicts could occur if many objects are likely to be updated near the end of the connection period, in this case techniques to prioritize objects based on both client preferences and the accuracy of the model are needed. This problem futher motivates the need for modeling updates, and introduces new problems in this area.

8.2.3 Server Cooperation

In Chapter 5 we showed that using update histories to estimate the freshness of cached objects can significantly reduce the number of validations at a remote server, which can potentially provide significant savings in terms of power and bandwidth for mobile clients. However, this increased accuracy may come at the cost of additional computational overhead for clients. Thus, an important area of future work is evaluating the computational overhead of different policies for both servers and clients, and developing architectures appropriate for different environments.

Specific areas include the following:

- **Performance:** One area of future work is studying the effect of different server cooperation schemes on the performance of both clients and servers. If a policy requires too much work at the server, it could have a negative impact on performance and increase latencies for all clients. Similarly, if a

policy requires too much work at the client, it could reduce the benefits of caching.

- **Architectures:** A related area of future work is developing server cooperation architectures that place an appropriate load on both clients and servers. Different architectures for server cooperation may be appropriate in different contexts. For example, a cache on a desktop machine has considerably more power than a cache on a mobile device. The desktop machine may be able to perform aggregations or compute expiration times without a negative impact on performance, but this may not be true for the mobile device. For the mobile device, it may be more appropriate to perform the computations at a server or intermediate proxy. Developing flexible and scalable architectures that meet the computational requirements of different servers and clients is an important area of future work.

8.2.4 Scheduling

Our work in mobile client profiles in Chapter 6 showed that using profiles for scheduling decisions at a wireless base station is an effective way to differentiate services for mobile clients. However, there are several open problems to improve scheduling and service differentiation in this environment.

- **Fairness:** As shown in Chapter 6, clients who have a large number of high priority applications increase the latency of all requests. This is unfair to clients who have fewer high priority applications. Thus, a fairness mechanism that can limit the number of high priority requests per client is needed.

- **Adapting to Varying Workloads:** Under heavier workloads, the latencies of different applications may increase. To maintain the desired latency of the high priority applications, the deadlines of different applications may need to be adjusted according to the current workload.

- **Multiple Classes:** In Chapter 6 we presented a scheme that maps applications to two classes, HighPriority and LowPriority. A useful extension to this would be to allow more than two classes of applications, and to allow clients to specify the desired latency for each of their classes. However, such a scheme needs to enforce fairness to ensure that clients do not have a large number of applications requiring low latency.

- **Implementation Issues:** Finally, an important area of future work is how to adapt the proposed scheduling scheme to handle features of wireless networks such as varying signal strengths, disconnections, and packet loss. For example, clients who have good signal strength can receive their data more quickly than clients with weaker signals, so an algorithm that considers signal strength can make more efficient use of the available bandwidth and ensure fairness to all clients.

Appendix A

Preparation of NLANR Trace Data

We performed preprocessing on the NLANR trace data to prepare it for the experiments. Specifically, the trace data did not report on object modification or expiration times, which we need to make downloading decisions and to determine the recency of cached objects. Our solution to this problem was to create an "augmented" trace using the workload from the original NLANR trace data. Over a period of 5 days, we replicated the trace workload by sending requests to the servers in the traces at (approximately) the same time of day as in the original workload. The requests were made from the domain `umiacs.umd.edu` which is connected to its ISP via a high speed DS3 line with a maximum bandwidth of 27 Mbps. When each requested object arrived, we logged the latency of the request and the time the object was last modified (when available). We used the logging mechanism provided by the Squid cache[24], but did not cache any objects. This augmented trace data provided the information we needed for this study. In our trace-based experiments, we cached only objects that had last modified information available and were not labelled uncacheable.

We gathered the augmented trace data from 16:48 on January 21, 2002 to 1:53 on January 26, 2002. We note that there were several gaps in our augmented

Start time	End time
Jan 22, 13:22	Jan 22, 13:27
Jan 23, 14:58	Jan 23, 15:22
Jan 24, 18:39	Jan 24, 19:13
Jan 24, 22:34	Jan 24, 22:46
Jan 25, 7:11	Jan 25, 10:11
Jan 25, 18:00	Jan 25, 18:13

Table A.1: Gaps in Augmented NLANR trace (GMT)

trace when errors occured and data was not collected. Most of these gaps lasted less than 30 minutes and did not significantly impacte our results. We report on the dates and times of these gaps in Table A.

A.1 Classification of Objects

In Table A.1 we show the number of requests made to different types of objects in the trace. Our NLANR trace contains 3707K requests total. In the trace we observed about 308K requests to objects that changed at least once during the trace period. Of these requests, about 196K (5% of all requests) were requests to objects that appeared to change when the last modified time changed to a time before the time they were cached. This could occur for many reasons, possibly due to an inaccurate clock at the remote server. We refer to such objects as *invalid objects*. A challenge to the trace analysis in Chapter 4 was how to handle invalid objects. Clearly, the last modified time of such objects is inaccurate, so we had no way of knowing when an update had actually occured. Any update

estimation policy that uses the last-modified time is likely to perform poorly for such objects. However, to ensure that our results are accurate, we analysed these objects in greater detail.

Table A.1 describes the content types of the 192 requests to invalid objects. The first observation is that most of these requests (96K) were to small GIF and JPG images. It is unlikely that these objects actually changed between consecutive accesses, thus, maintaining consistency of these objects is not a concern. Similarly, of the 30K accesses to HTML documents, at least 27K were requests to the domain `web.icq.com`. These objects were not accesses to traditional HTML data, but were linked to advertisements that changed on every access. The same holds for the 37K requests to `ads.web.aol.com` and the 19K JavaScript requests.

To summarize, most of the 196K requests to invalid objects were either dynamic advertisements or images that rarely changed. Since most of the perceived "updates" to these 196K objects did not actually change the objects, we believe that omitting these objects did not significantly impact the results in Chapter 4.

Object Type	Entire Trace	Invalid Objects
GIF	1606K	90K
JPG	748K	6K
HTML	406K	30K
TXT	113K	<1K
JS	82K	22K
aol	180K	37K
Other	572K	10K
Total	3707K	196K

Table A.2: Characterization of Requests in NLANR trace

Appendix B

Preparation of World Cup Trace Data

We performed some preprocessing on the World Cup trace data [10] to detect updates to objects. We assumed an update occurred whenever an object's size changed in the trace. However, there were several challenges to detecting changes to objects. First, some changes to an object's size were not due to updates. Many apparent changes in an object's size were caused by temporary inconsistencies at servers in different geographic locations. Recall that the trace contains requests to 33 servers in four locations. When an update occurred at one server, it often took several minutes to propagate to other servers at all locations. During this time, clients would receive different objects depending on which server handled their request. Therefore, our update detection technique needs to avoid these "false" changes due to temporary inconsistencies at different server locations.

Our solution to this problem was to only consider an object changed when the majority of requests to the object had the new size, and when the object had this size for at least two minutes. This allowed enough time for updates to propagate to servers in all four locations, to eliminate the effects of false changes due to server inconsistencies.

BIBLIOGRAPHY

[1] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast Disks: Data Management for Asymmetric Communication Environments. *Proceedings of ACM SIGMOD Conference*, 1995.

[2] S. Acharya, M. Franklin, and S. Zdonik. Disseminating Updates on Broadcast Disks. *Proceedings of Conference on Very Large Data Bases (VLDB)*, 1996.

[3] S. Acharya and S. Muthukrishnan. Scheduling On-demand Broadcasts: New Metrics and Algorithms. *Proceedings of MOBICOM*, 1998.

[4] S. Adali, K.S. Candan, Y. Papakonstantinou, and V.S. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. *Proceedings of ACM SIGMOD Conference*, 1996.

[5] M. Agarwal and A. Puri. Base Station Scheduling of Requests with Fixed Deadlines. *Proceedings of IEEE INFOCOM*, 2002.

[6] D. Aksoy and M. Franklin. Scheduling for Large-Scale On-Demand Data Broadcasting. *Proceedings of IEEE INFOCOM Conference*, 1998.

[7] D. Aksoy, M. Franklin, and S. Zdonik. Data Staging for On-Demand Broadcast. *Proceedings of Very Large Data Bases (VLDB)*, 2001.

[8] R. Alonso, D. Barbara, and H. Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM. TODS Vol. 15, no. 3*, 1990.

[9] K. Amiri, R. Tewari, S. Park, and S. Padmanabhan. On Space Management in a Dynamic Edge Data Cache. *Proceedings of WebDB*, 2002.

[10] M. Arlitt and T. Jin. 1998 World Cup Web Site Access Logs. *Available at http://www.acm.org/sigcomm/ITA/*, 1998.

[11] H. Balakrishnan, S. Seshan, and R. H. Katz. Improving Reliable Transport and Handoff Performance in Cellular Wireless Networks. *ACM Wireless Networks Journal (WINET)*, December 1995.

[12] G. Banga, F. Douglis, and M. Rabinovich. Optimistic Deltas for WWW Latency Reduction. *Proceedings of USENIX Technical Conference*, 1997.

[13] E. Baralis, S. Paraboschi, and E. Teniente. Materialized View Selection in a Multidimensional Database. *Proceedings of VLDB*, 1997.

[14] D. Barbara and T. Imielinski. Sleepers and Workaholics: Caching Strategies in Mobile Environments (Extended Version). *VLDB Journal Special Issue of the best of SIGMOD System-Oriented Papers*, 1995.

[15] J. Bennett and H. Zhang. Hierarchical Packet Fair Queueing Algorithms. *Proceedings of SIGCOMM*, 1996.

[16] G. Bianchi, A. Campbell, and R. Liao. On Utility-Fair Adaptive Services in Wireless Networks. *Proceedings of 6th International Workshop on Quality of Service (IEEE/IFIP IWQOS98)*, 1998.

[17] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-Like Distributions: Evidence and Implications. *Proceedings of IEEE INFOCOM*, 1999.

[18] L. Bright, S. Bhattacharjee, and L. Raschid. Supporting Diverse Mobile Applications with Client Profiles. *Proceedings of ACM Workshop on Wireless Mobile Multimedia (WoWMoM)*, 2002.

[19] L. Bright and L. Raschid. Using Latency-Recency Profiles for Data Delivery on the Web. *Proceedings of Very Large Data Bases*, 2002.

[20] A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, and L. Strigini. The Meaning and Role of Value in Scheduling Flexible Real-Time Systems. *Journal of Systems Architecture*, 46:305–325, 2000.

[21] G. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1997.

[22] G. Buttazzo, M. Spuri, and F. Sensini. Value vs. Deadline Scheduling in Overload Conditions. *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, 1995.

[23] Oracle9iAS Web Cache. *http://otn.oracle.com/products/ias/web_cache/content.html*.

[24] Squid Proxy Cache. *http://www.squid-cache.org*.

[25] K.S. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal. Enabling Dynamic Content Caching for Database-Driven Web Sites. *Proceedings of SIGMOD*, 2001.

[26] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. *Proceedings of USENIX Symposium on Internet Technologies and Systems*, 1997.

[27] Y. Cao and M. T. Ozsu. Evaluation of Strong Consistency Web Caching Techniques. *WWW Journal*, 51(15), July 2002.

[28] Z. Cao and E. Zegura. Utility Max-Min: An Application-Oriented Bandwidth Allocation Scheme. *Proceedings of INFOCOM*, 1999.

[29] D. Carney, S. Lee, and S. Zdonik. Scalable Application-Aware Data Freshening. *Proceedings of ICDE*, 2003.

[30] V. Cate. Alex - A global filesystem. *Proceedings of USENIX File System Workshop*, 1992.

[31] Cellmania. *http://www.cellmania.com.*

[32] U. Cetintemel, P. Keleher, and M. Franklin. Support for Speculative Update Propagation and Mobility in Deno. *Proceedings of ICDCS*, 2001.

[33] J. Challenger, A. Iyengar, and P. Dantzig. A Scalable System for Consistently Caching Dynamic Web Data. *Proceedings of IEEE INFOCOM*, 1999.

[34] M. Cherniack, E. Galvez, M. Franklin, and S. Zdonik. Profile-Driven Cache Management. *Proceedings of 19th International Conference on Data Engineering (ICDE)*, 2003.

[35] J. Cho and H. Garcia-Molina. Synchronizing a Database to Improve Freshness. *Proceedings of ACM SIGMOD Conference*, 2000.

[36] J. Cho and A. Ntoulas. Effective Change Detection Using Sampling. *Proceedings of VLDB Conference*, 2002.

[37] E. Cohen, E. Halperin, and H. Kaplan. Performance Aspects of Distributed Caches Using TTL-based Consistency. *Proceedings of ICALP*, 2001.

[38] E. Cohen and H. Kaplan. Prefetching the Means for Document Transfer: A New Approach for Reducing Web Latency. *Proceedings of IEEE INFOCOM*, 2000.

[39] E. Cohen and H. Kaplan. Refreshment Policies for Web Content Caches. *Proceedings of IEEE INFOCOM*, 2001.

[40] E. Cohen and H. Kaplan. The Age Penalty and its Effect on Cache Performance. *Proceedings of USITS*, 2001.

[41] E. Cohen, B. Krishnamurthy, and J. Rexford. Improving End-to-End Performance of the Web Using Server Volumes and Proxy Filters. *Proceedings of SIGCOMM*, 1998.

[42] A. Datta, K. Dutta, H. Thomas, D. VanderMeer, Suresha, and K. Ramamritham. Proxy-Based Acceleration of Dynamically Generated Content on the World Wide web: An Approach and Implementation. *Proceedings of ACM SIGMOD Conference*, 2002.

[43] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *Proceedings of SIGCOMM*, 1989.

[44] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou Architecture: Support for data sharing among mobile users.

Proceedings of IEEE Workshop on Mobile Computing Systems and Applications, 1994.

[45] P. Deolasee, A. Katkar, P. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive Push-Pull: Disseminating Dynamic Web Data, journal =.

[46] A.L. DeSchon. A Survey of Data Representation Standards. *RFC 971*, 1985.

[47] A. Dingle and T. Partl. Web Cache Coherence. *Proceedings of 5th WWW Conference*, 1996.

[48] C. Dovrolis and P. Ramanathan. A Case for Relative Differentiated Services and the Proportional Differentiation Model. *IEEE Network*, 1999.

[49] Yahoo! Finance. *http://finance.yahoo.com*.

[50] National Laboratory for Applied Network Research. *ftp://ircache.nlanr.net/Traces*.

[51] M. Franklin, M. Carey, and M. Livny. Transactional Client-Server Cache Consistency: Alternatives and Performance. *ACM Transactions on Database Systems*, 22(3):315–363, 1997.

[52] A. Gal. Obsolescent Materialized Views in Query Processing of Enterprise Information Systems. *Proceedings of CIKM*, 1999.

[53] A. Gal and J. Eckstein. Managing periodically updated data in relational databases: a stochastic modeling approach. *Journal of the ACM*, 48(6):1141–1183, 2001.

[54] P. Goyal, H. Vin, and H. Cheng. Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. *Proceedings of SIGCOMM*, 1996.

[55] J.R. Gruser, L. Raschid, V. Zadorozhny, and T. Zhan. Learning Response Time for Web Sources Using Query Feedback and Application in Query Optimization. *VLDB Journal 9(1):18-37*, 2000.

[56] H. Gupta. Selection of Views to Materialize in a Data Warehouse. *Proceedings of ICDT*, 1997.

[57] J. Gwertzman and M. Seltzer. World Wide Web Cache Consistency. *Proceedings of USENIX Technical Conference*, 1996.

[58] S. Hadjiefthymiades and L. Merakos. Using Proxy Cache Relocation to Accelerate Web Browsing in Wireless/Mobile Communications. *Proceedings of WWW10*, 2001.

[59] E. Hanson. A Performance Analysis of View Materialization Strategies. *Proceedings of ACM SIGMOD Conference*, 1987.

[60] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing Data Cubes Efficiently. *Proceedings of ACM SIGMOD Conference*, 1996.

[61] R.V. Hogg and E.A. Tanis. *Probability and Statistical Inference*. MacMillan, New York, second edition, 1983.

[62] C. Hoover, J. Hansen, P. Koopman, and S. Tamboli. The Amaranth Framework: Probabilistic, Utility-Based Quality of Service Management for High-Assurance Computing. *Proceedings of IEEE International High-Assurance Systems Engineering Symposium (HASE 99)*, 1999.

[63] B. Housel and D. Lindquist. WebExpress: A System for Optimizing Web Browsing in a Wireless Environment. *Proceedings of MOBICOM*, 1996.

[64] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[65] Y. Huang, R. Sloan, and O. Wolfson. Divergence Caching in Client-Server Architectures. *Proceedings of PDIS*, 1994.

[66] Z. Jiang and L. Kleinrock. Prefetching Links on the WWW. *Proceedings of IEEE Int'l Conference on Communications*, 1997.

[67] S. Jin and A. Bestavros. GreedyDual* Web Caching Algorithm. *Fifth International Web Caching and Content Delivery Workshop*, 2000.

[68] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. *Proceedings of the 19th International Symposium of Computer Architecture (ISCA)*, 1992.

[69] R. Kravets and P. Krishnan. Application-Driven Power Managements for Mobile Communication. *Proceedings of MOBICOM*, 1998.

[70] B. Krishnamurthy and C. Wills. Study of Piggyback Cache Validation for Proxy Caches in the World Wide Web. *Proceedings of USENIX Symposium on Internet Technologies and Systems*, 1997.

[71] B. Krishnamurthy and C. Wills. Proxy Cache Coherency and Replacement-Towards a More Complete Picture. *Proceedings of ICDCS*, 1999.

[72] T. Kroeger, D. Long, and J. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. *Proceedings of USENIX Symposium on Internet Technologies and Systems*, 1997.

[73] A. Labrinidis and N. Roussopoulos. Update Propagation Strategies for Improving the Quality of Data on the Web, journal =.

[74] A. Labrinidis and N. Roussopoulos. WebView Materialization, journal =.

[75] J.-J. Lee, K.-Y. Whang, B. S. Lee, and J.-W. Chang. An Update-Risk Based Approach to TTL Estimation in Web Caching. *Proceedings of Conference on Web Information Systems Engineering (WISE)*, 2002.

[76] K. Lee. Adaptive Network Support for Mobile Multimedia. *Proceedings of MOBICOM*, 1995.

[77] R. Liao and A. Campbell. A Utility-Based Approach for Quantitative Adaptation in Wireless Packet Networks. *Wireless Networks*, 7(5):541–557, 2001.

[78] C. Liu and P. Cao. Maintaining Strong Cache Consistency on the World Wide Web. *Proceedings of ICDCS*, 1997.

[79] S. Lu, V. Bharghavan, and R. Srikant. Fair Scheduling in Wireless Packet Networks. *IEEE/ACM Transactions on Networking*, August 1999.

[80] Q. Luo and J. Naughton. Form-Based Proxy Caching for Database-Backed Web Sites. *Proceedings of VLDB*, 2001.

[81] A. Misra, S. Das, A. Dutta, A. Mcauley, and S. K. Das. IDMP-based Fast Handoffs and Paging in IP-Based Cellular Networks. *Proceedings of 3G Wireless*, 2001.

[82] K. Nahrstedt and J.M. Smith. The Qos Broker. *IEEE Multimedia Magazine*, 2(1):53–61, 1995.

[83] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, 1988.

[84] K. Nichols et al. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. *RFC 2474*, 1998.

[85] A. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham, and R. Tewari. Cooperative Leases: Scalable Consistency Maintenance in Content Distribution Networks. *Proceedings of WWW Conference*, 2002.

[86] B. Noble and M. Satyanarayanan. Experience with Adaptive Mobile Applications in Odyssey. *Mobile Networks and Applications*, 4, 1999.

[87] M. Nottingham. Optimizing Object Freshness Controls in Web Caches. *Proceedings of 4th International Web Caching Workshop*, 1999.

[88] C. Olston, B.T. Loo, and J. Widom. Adaptive Precision Setting for Cached Approximate Values. *Proceedings of ACM SIGMOD Conference*, 2001.

[89] C. Olston and J. Widom. Best-Effort Cache Synchronization with Source Cooperation. *Proceedings of ACM SIGMOD Conference*, 2002.

[90] V. Padmanabhan and J. Mogul. Using Predicitive Prefetching to Improve World Wide Web Latency. *ACM SIGCOMM Computer Communication Review*, July 1997.

[91] C. Perkins. IP Mobility Support for IPv4. *RFC 3220*, 2002.

[92] HTTP/1.1 Protocol. *RFC 2616, http://www.w3.org.*

[93] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A Resource Allocation Model for QoS Management. *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, 1997.

[94] R. Ramjee, T. La Porta, S. Thuel, K. Varadhan, and S.Y. Wang. HAWAII: A Domain-Based Approach for Supporting Mobility in Wide-Area Wireless Networks. *Proceedings of ICNP*, 1999.

[95] S. Ross. *Stochastic Processes*. Wiley, second edition, 1995.

[96] P. Scheuermann, J. Shim, and R. Vingralek. A Case for Delay-Conscious Caching of Web Documents. *Proceedings of 6th WWW Conference*, 1997.

[97] P. Scheuermann, J. Shim, and R. Vingralek. A Unified Algorithm for Cache Replacement and Consistency in Web Proxy Servers. *Proceedings of WebDB*, 1998.

[98] P. Scheuermann, J. Shim, and R. Vingralek. Proxy Cache Design: Algorithms, Implementation, and Performance. *IEEE Trans. on Knowledge and Data Engineering*, 1999.

[99] S. Seshan, H. Balakrishnan, and R. H. Katz. Handoffs in Cellular Wireless Networks: The Daedalus Implementation and Experience. *Wireless Personal Communications*, January 1997.

[100] S. Shah, A. Bernard, V. Sharma, K. Ramamritham, and P. Shenoy. Maintaining Temporal Coherency of Cooperating Dynamic Data Repositories. *Proceedings of VLDB Conference*, 2002.

[101] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms.* Kluwer Academic Publishers, 1998.

[102] M. Stemm and R. Katz. Reducing Power Consumption of Network Interfaces in Hand-Held Devices. *Proceedings of Third International Workshop on Mobile Multimedia Communications*, 1996.

[103] I. Stoica, H. Zhang, and T.S.E. Ng. A Hierarchical Fair Service Curve Algorithm for Link-Sharing, Real-Time and Priority Services. *Proceedings of SIGCOMM*, 1997.

[104] C. Su and L. Tassiulas. Joint Broadcast Scheduling and User's Cache Management for Efficient Information Delivery. *Proceedings of MOBICOM*, 1998.

[105] Third Generation Wireless Systems. *http://www.fcc.gov/3G.*

[106] H.M. Taylor and S. Karlin. *An Introduction to Stochastic Modeling.* Academic Press, 1994.

[107] N.H. Vaidya and S. Hameed. Data Broadcast in Asymmetric Wireless Environments. *Workshop on Satellite Based Information Services*, 1996.

[108] H.J. Wang et al. ICEBERG: An Internet-core Network Architecture for Integrated Communications. *IEEE Personal Communications*, 2000.

[109] J. Wang. A Survey of Web Caching Schemes for the Internet. *ACM Computer Communication Review*, 29(5), October 1999.

[110] BEA WebLogic. *http://www.bea.com.*

[111] IBM WebSphere. *http://www-3.ibm.com/software/webservers/.*

[112] C. Wills and M. Mikhailov. Studying the Impact of More Complete Server Information on Web Caching. *Proceedings of 5th Workshop on Web Caching and Content Delivery*, 2000.

[113] R. Wooster and M. Abrams. Proxy Caching that Estimates Page Load Delays. *Proceedings of 6th WWW Conference*, 1997.

[114] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. Engineering Server-Driven Consistency for Large Scale Dynamic Web Services. *Proceedings of 10th WWW Conference*, 2001.

[115] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Hierarchical Cache Consistency in a WAN. *Proceedings of USITS.*, 1999.

[116] H. Yu, L. Breslau, and S. Shenker. A Scalable Web Cache Consistency Architecture. *Proceedings of SIGCOMM*, 1999.

[117] L. Zhang. Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks. *Proceedings of SIGCOMM*, 1990.

[118] H. Zhu and T. Yang. Class-based Cache Management for Dynamic Web Content. *Proceedings of IEEE INFOCOM*, 2001.

[119] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. *Proceedings of ACM SIGMOD Conference*, 1995.