PH.D. THESIS

Air-Caching: Adaptive Hybrid Data Delivery

by Konstantinos Stathatos
Advisor: Nick Roussopoulos and John S. Baras

CSHCN Ph.D. 99-1
(ISR Ph.D. 99-2)

Sponsored by: NASA and Hughes Network Systems

ABSTRACT

Title of Dissertation: Air-Caching: Adaptive Hybrid Data Delivery

Konstantinos Stathatos, Doctor of Philosophy, 1998

Dissertation directed by: Professor Nick Roussopoulos
Department of Computer Science

With the immense popularity of the Web, the world is witnessing an unprecedented demand for on-line data services. A growing number of applications require timely data delivery from information producers to thousands of information consumers. At the same time, the Internet is evolving towards an information superhighway that incorporates a wide mixture of existing and emerging communication technologies, including wireless, mobile, and hybrid networking.

For this new computing landscape, this thesis advocates creating highly scalable data services based on adaptive hybrid data delivery. It introduces air-caching, a technique that effectively integrates broadcasting for massive dissemination of popular data, and unicasting for upon-request delivery of the rest. It describes the special properties, performance goals, and challenges of air-caching. Then, it presents adaptive cache management techniques for three different settings: servicing large numbers of data requests over a heavily accessed databases, propagating data updates to mobile clients intermittently connected to information sources, and implementing publish/subscribe services again in the context of mobile computing. In all cases, performance experiments demonstrate the scalability, efficiency, and versatility of this technique, even under rapidly changing data access patterns.

Air-Caching: Adaptive Hybrid Data Delivery

by

Konstantinos Stathatos

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
1998

DEDICATION

To my Family

ACKNOWLEDGEMENTS

During the last few years, I was really looking forward to the moment I would have to write the acknowledgments for my dissertation. Now that this moment has at last arrived, I feel full of gratitude for several people, but short of words to accurately express it.

I am indebted to my research advisors, Professors Nick Roussopoulos and John Baras, for their guidance and continuous encouragement throughout my graduate school life. They offered me both the inspiration and the freedom to choose the research topic. Prof. Roussopoulos was always a dependable source of technical advice and moral support. He believed in my work and had confidence in my abilities. Prof. Baras helped me widen my research horizon, and managed to keep my engineering background alive. His interest in hybrid networking was a major motivation for my work, which he strongly supported right from the outset. Overall, it has been an honor and a great pleasure to know and collaborate with both of them.

My sincere gratitude is due to Prof. Michael Franklin. Not only his involvement in the Broadcast Disks project inspired my research, but as the “broadcast expert” of the database group, he significantly contributed to my work with many stimulating technical questions and comments. I would also like to thank Prof. Christos Faloutsos for his useful comments as well as all the insightful and enthusiastic discussions we had. In retrospect, I truly regret the fact that our collaboration did not extend beyond a small “independent study”. In addition, I would like to express my appreciation to Prof. A. Udaya Shankar for serving on my dissertation committee.

I was very fortunate to work among a group of talented colleagues in the Database Group of the University of Maryland, including Demet Aksoy, Mehmet Altinel, George Apostolopoulos, Ugur Cetintemel, Sandeep Gupta, Björn T. Jónsson (aka Thakis Yannatos), Flip Korn (aka Philipos Kornoutsos), Yannis Kotidis, Alex Labrinidis, David Lin, Nelson Padua-Pérez, Manuel Rodriguez-Martinez, Yannis Sismanis, Tolga Urhan, and Byoung-Kee Yi. They all contributed to the stirring and fun working environment I enjoyed for the last few years. Björn, Yannis Kotidis, and Alex deserve special thanks for taking the extra time to discuss about and help me with any problem I had. Björn also deserves extra credit for providing his “semi-automatic results compiler” that proved to be a great time (and sanity) saver.

I have to also thank all my friends that filled my Maryland days with nothing but good memories. The members of the Greek community in and around this university were a great and fun group of people to spend a few years with. Among all these, I would like to name and especially thank a few. First, my close friend Fotis Stamatelopoulos for all the wonderful student years we had together, starting long before we came to Maryland. Then, my roommates Christos Seretis, and Yannis Manolopoulos. Finally, the special group of friends that I share the most memories with, including Pavlos Christofilos, George Ioannou, Eleni Kassotaki, Yannis Kotidis, Alex Labrinidis, and Spyridoula Varlokosta.

However, this endeavor would not have been possible without the unconditional love and support of my family. First and foremost, I need to express my deepest gratitude to my parents. Ever since I can remember, they have been offering me limitless encouragement and support. They have taught me (and keep reminding me) that no goal is too high when you are determined to succeed. I feel that I cannot thank enough my wife, Evi, for being by my side and enduring with me the hardships of this long and difficult journey. Often, when I was looking at her, I could not help but wonder whether it is more difficult doing a PhD, or being married to someone doing a PhD! I also want to thank my son, Andreas, whose smile was enough to carry us through the last and hardest part of the journey. Last, I am grateful to the rest of my big and loving family. Each and every one of them has contributed in his/her own special way to my success. This thesis is dedicated to all of them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

At the doorstep of the 21st century, the world is witnessing an unprecedented demand for on-line data services. With the pervasive role of the Internet in our lives and the immense popularity of the World Wide Web, there is an exponentially growing interest in electronic access to all sorts of information for many different purposes. This creates a tremendous request load, which often well exceeds the capabilities of information servers and the capacity of network resources. On top of that, the increasing availability of powerful, inexpensive portable computers combined with the proliferation of wireless communication services has enabled mobile computing and the access of information virtually from any place and at any time. This convenience of mobility imposes even more stringent requirements on the supporting infrastructure.

At the same time, the Internet is evolving towards an Information Superhighway that incorporates a wide range of existing and emerging communication technologies, including wireless and hybrid networking [ABF$^+$95, KB96, BG96, Kha97]. These technologies provide users with a variety of options for connecting to information sources and retrieving the desired information.

This new, fast evolving computing and communications environment presents several interesting challenges to the deployment of large scale data services. But, it also creates opportunities for exploring alternative approaches to address these challenges. New types of information services are surfacing as practical solutions to the anticipated explosion of user demands [FZ96]. Among these, broadcast-based services have the potential of meeting such workload requirements, as they can efficiently disseminate information to any number of receivers.

This thesis advocates the development of efficient, adaptive, and highly scalable information systems based on hybrid data delivery. The main theme is the dynamic integration of data broadcasting for massive information dissemination with traditional interactive data services for selective upon-request data delivery.

1.1 Motivation

The size and the population of the Internet are growing at an exponential rate. More and more people are using information technology at work, school, and at home. The amazing number of advertisements, articles, and business cards that include a URL on the Web is clearly attesting to the significance of electronic access to information sources. Reports estimate that both the number of hosts and the number of users almost doubled in the last year alone. For example, [IS98] reports that from January 1997 to January 1998 the number of people accessing the Internet worldwide rose from 57 to 102 million. This number is estimated to reach 707 million by 2001. At the same period, the number of hosts connected to the Internet grew by about 88%.

This phenomenal growth rate is by far surpassing the growth of Internet resources, i.e., data server capacities and the network bandwidth. The huge workloads are exposing the scalability limitations of the currently employed data distribution mechanisms. Often, users experience long delays when accessing information sources, and occasionally fail to retrieve the desired information all together. A source for these problems is that, typically, data services use connection-oriented point-to-point protocols, which scale at best linearly with server capacity and network bandwidth.

These problems are severely aggravated in the course of special events when several million of requests are made during peak periods. For example, in the 1998 Winter Olympics, the peak request rate was reported to exceed 100 thousand per minute [Las98]. Similar bursty workloads may occur in crisis management applications, both civilian and military, which rely on rapid information distribution for responding to emergencies, natural disasters, and other "panic situations". In such cases, the demand during peak periods can be much higher than the average. This means that expanding the infrastructure to meet peak demand is wasteful and uneconomical, as it requires big investments in equipment that would be underutilized and wasted for most of the time. Instead, an alternative data dissemination approach is needed that can adapt and scale to unpredictable and bursty user demands.

1.1.1 Information Consumption

Along with the Web, a new class of information-centered applications is attracting a lot of interest [FZ96]. These applications follow the **information consumption** model, where there is a set of **consumers** retrieving and using information made available by one or more **producers**. Examples of such applications include news delivery [Poi98], financial market information, commuter and traveler services [SFL96], software distribution, as well as entertainment delivery.

An interesting property of this kind of applications is that they exhibit two types of **asymmetry**. First, there is almost exclusively a one-way information flow from the producers to the consumers. Typically, consumers send small requests (e.g., a get URL request) and in response may receive large amounts of data, possibly including graphics and video. Because of this asymmetry, the careful management of the downstream bandwidth (i.e., the network bandwidth from the producers to the consumers) is much more critical to the successful deployment of such services. Second, the number of consumers is significantly higher than the number of producers. It is often the case where a single provider provides information which is of interest to a very large audience (e.g., CNN and news updates). As a result, the server providing the information may receive enormous amounts of requests that it is unable to service. This phenomenon is particularly grave during peak periods following, for example, important breaking news or special announcements.

1.1.2 Emerging Communication Technologies

The ever increasing demand for high bandwidth links to the office, the home, as well as on the road has spawn a fierce industrial competition in the market arena of broadband data services. Combined with recent advances in technology, the communications industry is offering a wide range of options for connecting to the Internet, both wireline and wireless. Because of technological as well as economical reasons, none of these is expected to dominate over the rest. According to [Blu97], "the search for the Holy Grail of information infrastructure does not lead anywhere, but everywhere." As result the Internet is evolving towards a hybrid information superhighway that combines many technologies, especially at its "last mile" (i.e., the link between the backbone and the office or the home) [BG96, KB96, Kha97].

Some of the technologies that the industry is trying to capitalize on are:

- Variations of digital subscriber lines (xDSL) over existing copper twisted pair wires [Kha97, Law98].
- Cable television (CATV) using hybrid fiber coax (HFC) networks [BMSS96, Law98].
- "Wireless cable" offered by local and multichannel multipoint distribution service (LMDS and MMDS) [Kha97].
- Cellular and personal communication networks [PGH95, Pan95].
- Direct broadcast satellites (DBS), mobile satellite systems (MSS), and hybrid satellite/terrestrial networks [ABF^{+}95, Gol98].

There are at least three interesting observations to be made about these communication alternatives. First, the conception and design of some of these were influenced by the asymmetric nature of information consuming. Thus, they are asymmetric in the sense that they offer different transmission rates in each direction. For example, HFC offers a maximum rate of 38 Mbps to the subscribers, while the rate from the subscribers does not exceed 4 Mbps [Kha97]. Second, most wireless technologies enable user mobility, and generally, the ability to access information from any place at any time. Last, but not least, some of these techniques are broadcast-based (e.g., CATV, DBS) which means that they inherently support efficient information broadcast to many users, possibly spread in a large geographic area, over a globally shared link and without any intermediate switching. In the face of the scalability problems experienced on the Web today, this capability opens the door for the deployment of new scalable broadcast-based data services.

1.1.3 Mobile Computing

We are in the midst of an important change in the way people access information systems. Empowered by wireless communications and powerful portable computers, more and more users carry their work "on the road", away from their base computing environment. The trend towards mobile computing is reflected in the fast growing market of laptop computers. But, the mobility of users, the wireless connectivity, and the portability of computing devices are generating several new research challenges, which do not occur, and thus were never addressed, in the context of stationary (fixed) distributed computing systems [AK93, FZ94, IB94, Sat95]. Some of the problems that rise in this new computing environment are the tracking of mobile users, dynamic system configuration, variable connectivity, battery powered devices, and different communication tariffs.

From the data management perspective, a host of interesting problems stem from **intermittent connectivity** and **disconnected operation** [KS91]. Mobile users often stop using their communication devices and prohibit all network accesses, either to minimize cost and energy consumption, or simply because they happen to be located in an area where there is no network coverage. The effects of disconnection are typically masked by hoarding, caching, or replicating data from central repositories into the portable units. Occasionally, users reconnect and reintegrate their data. This involves uploading to repository data updates made in the portable unit as well as downloading to the unit relevant updates registered at the repository. While both parts exhibit a set of difficulties (e.g., consistency checking, conflict resolution), the propagation of updates to mobile units can be a big data distribution problem, especially in the presence of large user populations.

1.2 Contributions

Driven by the ever increasing demand for on-line data services, the asymmetric nature of information consumption, and the capabilities of emerging networks, this thesis broadly addresses the problem of large scale data dissemination. It capitalizes on the scalability potential of broadcast communication for efficient delivery of data from an information server to thousands of clients.

Broadcast-based data delivery has been investigated by several researchers, along with a number of related issues.[1] This thesis focuses on push/broadcast data delivery, i.e., data broadcast regulated by the server and not by explicit client requests. So far, all the techniques that have been proposed for this type of data delivery follow a common thread: they assume that the server is fully aware of the global access pattern of the client population, which is used to produce static data broadcast schemes. The premise of these approaches is that users register with the system and provide profiles of their interests, which are compiled to derive the global data access pattern. While this may be suitable in some applications, it is not applicable when the interests of clients change continuously and even abruptly as in the case of "panic situations".

Instead, in this thesis we expand and complement broadcast delivery with traditional interactive data services to create highly scalable data dissemination

[1]For a survey refer to Chapter 2.

technique, adaptive to dynamic and unpredictable user demands. This technique is generally called **adaptive hybrid data delivery** as it dynamically integrates two different data delivery mechanisms.

In the following, we outline the key contributions of this thesis:

- We compare and contrast two popular data delivery techniques: the traditional request/response (pull/unicast) and the rather novel push/broadcast. After discussing their advantages and disadvantages, we make a case for adaptive hybrid data delivery, i.e., the combination of the two techniques in a dynamic and complementary manner.

- We introduce the concept of **air-caching**, a convenient abstraction that essentially disguises the problem of adaptive hybrid data delivery to a cache management problem [SRB96]. Air-caching is the temporary storage of popular data in a broadcast channel through repetitive transmissions. We identify the special properties of this type of data caching, discuss its performance goals, and set its basic management principles. The general goal of a hybrid system is to air-cache popular data to satisfy the bulk of the clients' demands, and leave only a small number of requests for unpopular data to be serviced by the server itself. A unique characteristic of the air-cache is that, contrary to typical caches, it must be managed relying exclusively on cache misses because the server does not have at its disposal any information about cache hits.

- We consider the problem of servicing very large numbers of user requests over a given database [SRB97a]. Assuming that these requests exhibit high degrees of skewness towards parts of the database, we propose a set of algorithms that dynamically detect and air-cache the hot-spots, even when they are rapidly changing.

- We propose to use the air-cache mechanism for propagating data updates from a central repository to mobile disconnecting clients. We describe a hierarchical version of the air-cache that adds the flexibility of multiple access latencies. In this case, we describe techniques that detect the (dis)connection pattern of the clients, and establish their needs for updates. Based on that, the server air-caches recent updates in a way that matches this pattern.

- In the context of mobile computing, we also consider the deployment of publish/subscribe services. Mobile clients subscribe to multiple, semantically different, data services. Every time they reconnect they need to retrieve any newly published data that matches their subscription. The server employs air-caching for the dissemination of the published information. For this application, multiple air-caches are realized in a single broadcast channel, one for each of the provided services. To this end, we propose a composite air-cache structure that multiplexes several simple air-caches in a single broadcast channel, and develop techniques for managing these air-caches independently from each other.

- All the proposed algorithms are validated with experimental results drawn from a detailed simulation model of the systems. In all cases, these results demonstrate the scalability, adaptiveness, and efficiency of air-caching, and establish its potential as a versatile and practical method for large scale data dissemination.

1.3 Organization

The rest of this thesis is organized as follows: Chapter 2 presents a literature survey of research work closely related to this thesis. In Chapter 3, we first make a case for hybrid data delivery, and then introduce the air-cache mechanism and discuss its properties, tradeoffs, and performance goals. Chapters 4, 5, and 6 constitute the main body of this thesis. They present three different sets of algorithms and techniques used to manage the air-cache in three different data intensive applications. Last, Chapter 7 summarizes the contributions and results of our work, and indicates some possible future research directions.

Chapter 2

Literature Survey

Broadly, our work falls in the general research area of wireless and mobile computing. For the interested reader, [IK96] presents a number of research projects that deal with a broad range of issues in mobile computing, including networking, operating systems, power management, and information services. Data management issues, which are of more interest to us, are discussed in [PS97]. Problems and challenges with respect to data management are also listed in [AK93] and [IB94]. Furthermore, different aspects of data dissemination are covered in [Fra96a].

In the following, we present a more focused survey of the literature in research areas, which are closely related to our work.

2.1 Data Broadcasting

The idea of broadcasting data from some information source to a large number of receivers has been explored for more than a decade. Several researchers have recognized its scalability potential, and tried to apply it in varied contexts. Teletext and videotex systems were the first to be considered for broadcast data delivery [WA85, AW85, Won88]. The focus of this work was on optimized data transmission schedules for interactive, broadcast, and hybrid systems. The Datacycle project at Bellcore developed an alternative architecture for a database machine based on the same idea [HGLW87, BGH$^+$92]. Data was periodically broadcast through a high bandwidth communication channel from the **storage pump** to an arbitrary large number of **access managers**. The managers relied on custom made VLSI data filters to perform associative searches over the broadcast data.

A pioneering project that used wireless data broadcasting for large scale information delivery was the Boston Community Information System (BCIS) [GBBL85, Gif90]. Gifford et al. proposed, built, and field-tested a flexible and cost-effective **polychannel system**, i.e., a system that combines **simplex** (broadcast) and **duplex** (interactive) communication channels, to provide up-to-the-minute access to information sources (e.g., New York Times, Associated Press) to an entire metropolitan area. The system used an FM channel to periodically broadcast updates (news articles) to locally maintained user databases. User queries were answered by the local database whenever possible. If the requested information were not locally available, the system would query the server through a regular modem connection.

More recently, due to the ever increasing popularity of mobile computing, this area has gained much more attention by the research community. The most notable examples are the **Dataman** [Imi96] and the **Broadcast Disks** [AFZ95] projects. The Dataman group at Rutgers University has focused on channel indexing for wireless data publishing [Vis94]. The broadcast disks project is based on the idea of hierarchical data broadcast. Their work addresses a wide range of problems including broadcast scheduling, client cache management and prefetching, and update dissemination.

2.1.1 Broadcast Scheduling

Typically, in a broadcast-based information system, the quality of the service provided to the users depends mainly on the data access latency, i.e., the time a user has to wait before the needed information appears on the channel. Assuming the available amount of bandwidth is fixed, this latency depends on the scheduling policy adopted by the server. In case the demand is uniformly distributed over all the items in a database, a periodic broadcast of all these items yields the minimum expected latency. However, in real life situations demand patterns are usually skewed, making broadcast scheduling a harder problem to solve.

Generally, broadcast scheduling algorithms can be classified into two groups. The first group of algorithms addresses the problem in the case of "data push", i.e., when clients do not send any requests to the server. Typically, the server relies on some a priori knowledge of the data access patterns to schedule the broadcast. The second group deals with scheduling in the presence of explicit client requests (on-demand or pull scheduling).

For push scheduling, researchers have proposed two types of algorithms: **probabilistic** and **periodic**. The probabilistic algorithms select data to broadcast one at a time, based mainly on their respective probabilities of being requested [IV94]. The main disadvantage of this approach is that latency can be arbitrary large for some items (starvation problem). There have been proposed techniques that remedy this by considering, along with the access probabilities, the time elapsed since the last item was broadcast [ST97, HV97a]. The advantage of these approaches is that they can very easily accommodate changing workloads. Periodic schemes, on the other hand, pre-compute an optimal schedule, which is continuously repeated [AW85, AW87, Chi94, AAFZ95]. Such optimal schedules are expensive to compute, and therefore less resilient to change. But periodic schemes exhibit some nice properties such as guaranteed maximum latency and minimum variance in inter-arrival times for the different items. This second property is very important for a number of reasons: First, high variance tends to increase the average latency [JW95]. Second, from the clients' point of view, minimum variance means that the appearances of data in the channel are more predictable. This can have a significant impact to client cache management and power savings [AAFZ95]. Examples of cache management techniques that rely on such predictability are described in [AFZ96b, TS97a].

Scheduling the broadcast based on client requests is a different problem. This problem was first studied in [DAW86] where they showed that, under skewed access patterns, the standard FCFS scheduling policy performs poorly. Instead, they proposed an algorithm, called **Longest Wait First (LWF)**, that schedules broadcast items in decreasing order of aggregate waiting time of pending requests. This algorithm yields the minimum average response time, but it is very expensive. In [ST97], the authors propose a more efficient algorithm that, given the data access probabilities, performs similarly to LWF. More recently, [AF98] defined a set of performance criteria for scheduling algorithms, including overhead and robustness to workload changes. With these in mind, they also proposed a parameterized algorithm that, depending on the values of the parameter, can perform as well as or close to LWF without, however, requiring the knowledge of data access distributions.

2.1.2 Hybrid Approaches

Since the main theme of this thesis is hybrid delivery, in this section we review other hybrid approaches. For clarity, we will classify each approach based on the taxonomy of data delivery options presented in [FZ96].[1] According to this taxonomy, data delivery mechanisms can be distinguished along three dimensions: (1) client initiated pull vs. server initiated push, (2) periodic vs. aperiodic, and (3) unicast vs. broadcast (or multicast) communication. Obviously, hybrid approaches combine two or more of these options.

The Boston Community Information System [GBBL85] was the first such hybrid system. It combined push-periodic-broadcast for news updates and pull-aperiodic-unicast for querying parts of the database not found in the client cache. According to [Gif90], the major conclusions of their experiments were that users valued both components of the hybrid architecture, and that this approach is indeed a very economic way to building large scale information systems.

A hybrid teletext-videotex architecture was proposed in [WD88]. Their approach involved only broadcast delivery, but for both periodically pushed with aperiodically pulled data. They evaluated the performance of the system based on some ad hoc partition of the data into two groups (one for each delivery option). Their results emphasize the need for an algorithm that adaptively selects the appropriate delivery modes. The same combination of delivery options was considered in [AFZ97] for the broadcast disks environment. The main difference of their work is that data are not partitioned into push and pull groups. Instead, data pull was adopted for expeditious delivery, as a performance improvement over the regular push mode of broadcast disks. In that work, they explore the efficacy of a

[1]This taxonomy is better explained in Section 3.2.

back-channel in a broadcast-only environment and discuss the involved tradeoffs.

In [IV94], Imielinski and Viswanathan propose an adaptive algorithm for hybrid delivery that combines push-aperiodic-broadcast and pull-aperiodic-unicast. Their goal is to optimally assign data and bandwidth to the two delivery modes in a way that expected response time remains below some threshold. However, the proposed algorithm is computationally expensive and relies on fairly static workloads.

Last in [DCK$^+$97], they consider mobility of users between cells of a cellular network, and propose two variations of an adaptive algorithm that statistically selects data to be broadcast based on user profiles and registrations in each cell.

2.1.3 Indexing

Under a data broadcasting scheme, clients are expected to get the needed information by filtering the broadcast channel. This means that they have to be monitoring the channel until they find what they need. For self-powered mobile computers, this yields a considerable energy waste since the receiver (and probably the CPU as well) has to be in **active mode** for long periods. Thus, in terms of power consumption, the ideal scenario would be for the clients to operate most of the time in **doze mode**, and switch into the power demanding active mode only when data they actually need are being broadcast. That would minimize the clients' **tuning time** [IVB94b, IVB94a].

Obviously, this is possible only if the clients have enough advance information about the broadcast schedule. This is easy under fixed periodic schemes; clients can, for example, get the schedule upon registering with the system use it thereafter. Under non-fixed schedules the problem is not so trivial. A natural solution is to publish some sort of index information along with the actual data. But, such index information takes up some of the broadcast bandwidth, penalizing the **access time** to the data. Therefore, when power consumption is critical, a tradeoff between access time and tuning time comes into the picture.

There have been several proposals about how to efficiently address this issue. The main idea is to structure the index information and distribute it within the broadcast schedule in a way that optimally balances the involved tradeoff. For example, in [IVB94a], they present ways of interleaving the nodes of a B-Tree with the broadcast data in order to support power-efficient filtering based on some key attribute. In [IVB94b] they discuss how the same can be achieved through hashing. Their work has been extended in [TY96, YT97] for non-uniform broadcast schedules. Finally, some alternative techniques based on the idea of Huffman encoding have also appeared in the literature [SV96, CYW97].

Although in this thesis we are not directly addressing power management issues, in Section 3.3.5 we explain the impact of indexing to our work.

2.2 Client Caching and Data Updates

Client data caching was introduced as a performance improvement for client-server systems (e.g., [HKM$^+$88, FC94]). But with the advent of mobile computing, it became indispensable as it enables **disconnected operation** for clients not able or willing to remain connected to main data repositories [KS91, Sat95].

A primary concern for client caching is the dissemination of updates to clients holding relevant data in their local caches. Basically, there are two basic techniques for dealing with updates: **propagation** and **invalidation**. Propagation means that the server sends the actual updated data to the clients. Invalidation, on the other hand, means that the server sends to the clients only notifications about the updates. The clients invalidate the stale data in their caches and, if necessary, request the new values from the server. For database systems, several algorithms that use one or both of these techniques have been proposed and evaluated, mainly as part of a cache consistency protocol [Fra96b].

Update dissemination has been considered from several other perspectives. For example, in [DR94] the authors compare five update propagation methods (on-demand and the combinations of immediate/periodic broadcast/multicast) for incrementally updating cached query results. The use of minimum spanning trees for pushing updates to a large number of networked nodes is discussed in [Ng90] and [WM91]. In another interesting study, the idea of **quasi-caching**, which allows controllable divergence in the values of cached data, is presented as a means of reducing the update propagation overhead [ABGM90].

The problem has also been studied in a broadcast environment. The dissemination of updated items in the broadcast disks architecture is investigated in [AFZ96a]. There, the authors explore several alternatives for propagating updated data (pages) within the broadcast disk program, and study their effects on client caching and prefetching under different consistency models.

In a mobile context, a key problem is the efficient (in)validation of data cached in reconnecting clients. This is usually addressed by periodically broadcasting some form of invalidation report [BI94, WYC96, JEHA97]. By examining these reports, clients can check the correctness of their cached data with respect to the main data repository. From the server's point of view, the problem is creating invalidation reports of small size that can be useful to a wide range of clients.

Finally, in [CTO97], the authors investigate the tradeoffs involved in incrementally updating views cached in mobile clients, for different types of periodically broadcast update reports. Issues concerning view maintenance for the mobile environment are also discussed in [WSD$^+$95].

2.3 Publish/Subscribe Services

In the literature, the term “information dissemination” has been extensively used to refer to publish/subscribe services. These services allow users (information consumers) to have a passive role in the delivery of data. They do not ask for information; instead, they define a set of interests, i.e., a profile, and submit it once[2] to a server (information producer or broker) expecting to automatically receive any newly published information that matches that profile. The server, or more generally the supporting infrastructure, is responsible to make sure that this actually happens.

This kind of interaction has been used in several different applications, including news delivery, mailing lists, web publishing, messaging systems, software distribution, and decision support systems. Depending on the nature of the application and the operating environment, different aspects of this model may be defined and implemented in different ways [Loe92]; often it even appears under different names (e.g., information filtering, document dissemination, event-driven delivery).

The filtering aspect of such services, i.e., the matching of subscriber profiles to published data is discussed in [LT92]. Along these lines, [TGNO92] describes a system that defines profiles as **continuous queries** over a database using a relational model. Users are notified whenever data in the database matches these

[2]The profile may be occasionally updated or refined

queries. The Stanford Information Filtering Tool (SIFT) [YGM96] proposes a set of indexing techniques to efficiently match a large number of documents against a large number of profiles. They also propose a distributed architecture for selectively disseminating documents over a wide area network [YGM94].

An early publish/subscribe system was the BCIS, described earlier in Section 2.1. Under this system, users defined keyword-based profiles, which were used by the client software to filter the information that was being broadcast by the server over a wireless network.

Furthermore, under the umbrella of event-driven delivery, publish/subscribe services have also been used commercially by financial institutions, manufacturing companies, and so forth [OPSS93, Cha98]. Finally, this model has recently gained popularity on the WWW with the—occasionally misleading—name "Internet push" [DeJ97, FZ98]. For example, several products offer personalized news delivery (e.g., [Poi98, Air98]), and automatic software distribution (e.g., [Mar98]).

Chapter 3

Hybrid Data Delivery

Electronic access to information sources is fast becoming an indispensable part of our lives. More and more people connect to the Internet every day for a wide array of reasons, both professional and personal. At the same time, many recent technologies, notably light-weight portable computing devices and wireless communications, enable users to carry out their on-line activities practically from anywhere and at anytime. In this emerging world of computing, one of the most important burdens for information providers is to deliver data in very large scale, possibly over different communication media.

Undoubtedly, the prevailing distributed computing model for the last two decades is the client/server model; and within this model, the traditional data delivery mechanism has been **request/response**. A client sends a message to a server requesting some information. The server processes the request and responds by sending back to the client the requested information (assuming the client made a valid request). However, the host of new applications and technologies are often exposing the limitations of this mechanism. To overcome these limitations, researchers and practitioners are now exploring alternative data delivery mechanisms [FZ96]. Among these, data push and data broadcast are receiving a lot of attention, mainly because of their scalability potential.

In this chapter, we introduce **hybrid data delivery**, the combination of the traditional request/response mechanism with data broadcasting. To put the technical issues in perspective, we start with a description of the presumed computing environment and the underlying assumptions. Then, we make a case for hybrid data delivery, and delineate the goals we want to achieve. In Section 3.3, we propose **air-caching** as an abstract vehicle for hybrid delivery adaptive to user demands. Last, we present the algorithms used to actually realize an air-cache over a broadcast medium.

3.1 Hybrid Networks

In this section, we present the necessary setting for the discussion that follows, as well as for the rest of this thesis. We give some basic definitions and identify the major underlying assumptions.

The key element for the techniques presented herein is a hybrid communications environment. Generally, this suggests a networking infrastructure where computing devices may or have to communicate over more than one communication media. Examples include hybrid fiber coax (HFC), fiber to the curb (FTTC), and hybrid satellite/terrestrial networks [KB96, BG96, Kha97, Gol98]. For our purposes, a hybrid network has a more specific definition:

Definition 1 *A* **hybrid network** *is any communications infrastructure that allows a server to both establish point-to-point conversations with any individual client as well as broadcast information to all (active) clients.*

In a hybrid network we identify two communication channels: the **unicast**

channel and the **broadcast channel**. As the names imply, the unicast channel is used by the clients and the server to exchange data on a point-to-point basis, while the broadcast channel is the medium through which the server broadcasts messages to the clients.

These definitions refer to the logical view of the network and do not necessarily imply networks that are physically hybrid. In fact, there may be many different configurations of the underlying communications media. These include:

Single medium (non-hybrid) networks All communication is done over the same medium that supports both unicasting and broadcasting (e.g., Ethernet, two-way cable networks)

Separate unicast/broadcast networks The medium used for broadcasting is different from that used for point-to-point connections between the server and the clients. An example would be the case where the server uses a satellite channel to broadcast data, while point-to-point conversations are carried over some tethered network.

Separate upstream/downstream networks One medium is used for information flow from the server to the clients (both broadcast and unicast) and another is used for the opposite direction. For example, a cable network may be used for downstream and regular telephone lines for upstream data flow.

A hybrid network that falls under the above definition is an essential requirement for the techniques and results presented in this thesis to be applicable. In addition, the scope of the thesis is also limited by three basic assumptions:

- The broadcast and unicast channel are logically independent and there is a predetermined amount of bandwidth allocated to each one. This assumption obviously holds for networks with separate broadcast and unicast media. If however, the two channels share the same medium, they are properly multiplexed using some bandwidth allocation policy (e.g., TDMA, FDMA [Tan96]).

- Upstream traffic is negligible. The premise of this assumption is that, for information consuming applications, clients send small request messages to the server (e.g., get a data page or a URL), and in return, they may get much larger amounts of data. As a result, management of the downstream traffic becomes a primary concern; upstream traffic is often much less of a burden. Moreover, as a central performance goal of this thesis is to control the number client requests for any workload, this observation is expected to hold even in very large scale. The essence of this assumption is two fold: First, we can ignore the ramifications of congested upstream channels. Second, for cases where the upstream and downstream flows share the same unicast channel, we can conjecture that, practically, all of the unicast bandwidth is used downstream.

- We assume reliable communications and ignore the effects of transmission errors. In practice, data transmissions are error-prone (especially with wireless media) and necessitate some mechanism for re-transmitting corrupted data. However, there are techniques that help combat errors at the receiver, obviating re-transmissions. They are called **forward error correction** techniques, and use redundancy to allow receivers to reconstruct any damaged block of data [Tan96]. They are particularly important in one way communication environments (e.g., satellite networks) and can be implemented either at the network or the application layer (e.g., [BB97]).

Relaxing any or all of these assumptions generates many interesting questions. For example, could we get better even better performance with dynamic (instead of static) bandwidth allocation? Or, how should the algorithms be modified to account for possible communication errors? These questions and more are challenging directions for future work.

3.2 Data Delivery Alternatives

In a client/server information system, a server is the central data repository that makes information available to interested clients. In other words, the server is the **information provider** and the clients are the **information consumers**. Between the two, some **data delivery model** is engaged that regulates the information flow as necessary. For many years, "one-to-one request/response" has been the predominant model for this purpose. When a client needs some piece of information, it sends a request to server and the server replies with the information to the requesting client. Recently, alternatives to this model are emerging, mainly as an effort to accommodate the size explosion of the Internet, information overload, as well as special requirements imposed by mobile computing.

In [FZ96, FZ97], the authors propose a taxonomy of data delivery mechanisms. They differentiate the possible mechanisms along three dimensions:

Pull vs. Push Pull-based delivery refers to the cases where the client initiates the data transfer by sending a request to the server, i.e., the client pulls information from the server. In contrast, with push-based delivery the transfer is initiated by the server without any specific client request. In other words, the server pushes information to the client(s).

Aperiodic vs. Periodic Aperiodic delivery refers to unscheduled data transfers caused by (random) events like a client request or the generation of new data. On the other hand, periodic delivery is repetitive transfer performed in a orderly manner according to some schedule.

Unicast vs. 1-to-N This distinction pertains the communication method used to actually transmit the data, and reflects the number of potential recipients. With unicast, only one client can receive data the server transmits. 1-to-N delivery suggests multicast or broadcast communication and the ability for multiple clients to receive the same server message.

According to this taxonomy, request/response is classified as a pull-aperiodic-unicast model. Other combinations correspond to less popular delivery mechanisms, like polling (pull-periodic-unicast), triggers (push-aperiodic-unicast), etc.

In our study, we concentrate on two models: **pull-aperiodic-unicast** and **push-periodic-broadcast**. These two models are on the antipodes of the taxonomy; they stand apart in all three dimensions. In principle, they are designed to serve different purposes as they exhibit different properties and performance characteristics. With this in mind, the theme of this thesis is to combine them in a hybrid method, where each one complements the other to yield an effective and very scalable data delivery mechanism. The merit of the other models, as well as the possibility of more types of hybrid approaches are indeed worth investigating, but beyond our current scope.

In the following sections, we first review the two models and discuss their advantages and disadvantages. Based on that, we then make a case for hybrid delivery.

Before proceeding with the discussion, we must make an important note on the terms used throughout this thesis. Given the focus of our work on the two particular models, we use a looser terminology, hopefully, without causing any confusion for the reader. Specifically, the term **push/broadcast** as well as the terms **push** and **broadcast** individually, are used to refer to push-periodic-broadcast. Similarly, the terms **pull/unicast**, **pull**, and **unicast** refer to pull-aperiodic-unicast. Occasionally, the latter is also called **on-demand** delivery.

3.2.1 Pull/Unicast Delivery

The client/server computing model has been founded on the very simple idea of a client requesting some kind of service from a server, and the server providing the requested service. When the service includes delivery of data to the client, this naturally translates to the request/response delivery model. Typically, the client establishes a connection with the server and sends a request message. The server processes the request, and replies with the requested data, assuming a valid client request. The client receives the data and drops the connection. This simple, straight-forward mechanism has been used almost exclusively since the earliest information systems.

This simplicity is a strong enough reason to believe that it will remain the most popular mechanism in the years to come. However, with the ever increasing popularity of the web, these systems are now being put under hard stress tests. In several occasions servers are called to handle huge waves of user requests. Often, they fail to perform; clients cannot get the requested data or, at best, they get it with very long delays. When that happens, we suspect that either the server cannot process requests as fast, or (probably more often) the available network bandwidth is not enough.

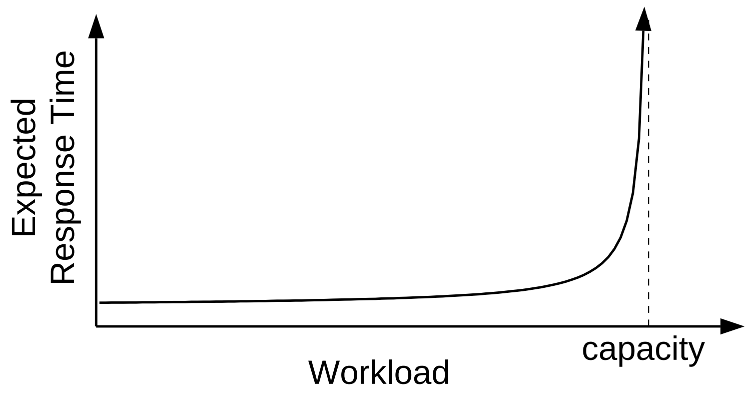


Figure 3.1: Typical performance of pull/unicast systems

These are clear demonstrations of the fact that the pull/unicast model suffers from scalability problems. The root of these problems is that the performance of such systems depends on the workload imposed on the server. The workload is usually expressed in terms of the **request rate**, i.e., the rate at which client requests arrive at the server. Figure 3.1 plots a typical performance curve for a pull/unicast server. The expected response time is a hyperbolic function of the workload. More importantly, every server has an upper limit as to how fast it can process client requests. If the request rate is sustained at higher levels for prolonged periods of time, the server cannot keep up with the demand. As a result, response times grow arbitrary large.[1] This limit defines the **server capacity** and reflects the maximum rate at which the server can service client requests.

The capacity of a server is determined by the hardware configuration (i.e., number and type of processors, main or secondary memory) as well as the network connections. For a system to perform acceptably at all times, it has to be designed

[1]Practically, because of limited input queue sizes, a lot of requests are not accepted by the server, and therefore they are never serviced.

with a capacity higher than the maximum anticipated workload. However, this is not practical for two reasons: First, it is not always easy—if possible at all—to predict what the maximum workload may be. Second and more important, even if a good prediction can be made, the maximum workload (as observed for example during "rush" hours, special events, or emergencies) can climb up to hundreds times the average. Therefore, designing a system to handle peak demand is very uneconomical, as it requires huge investments in equipment that would remain underutilized for most of the time. Naturally, the broader the difference between the maximum and the average workload, the more wasteful such investments are.

A related phenomenon is often the cause for wasteful usage of resources by pull/unicast data delivery. For many applications, a significant part of the information consumers are requesting exactly the same information. As a result, some pieces of information tend to be extremely popular, especially during peak periods. Examples are the last day's top news stories, stock market information, traffic on major highways, etc. What happens in such cases is that the server keeps processing the same request, and keeps sending the same data through the network over and over again. All this redundancy squanders a lot of processor cycles and, more importantly, high amounts of valuable network bandwidth.

But, what could we do to address these issues? How can we build scalable systems and avoid unnecessary expenditures in infrastructure? A proposal that has received a lot of attention lately is push/broadcast data delivery. In the next section we explain why this proposal is so attractive, but also discuss its vulnerabilities.

3.2.2 Push/Broadcast Delivery

Push-periodic-broadcast is the inverse approach for data delivery. To a large extent, it has been motivated by the fact that modern broadcast networks, both wireless and wireline, can be used to efficiently distribute information in very large and/or geographically wide scale. In principal, it is the same idea that has been very successfully used for distribution of radio or television programs.

Under this approach, clients do not request data from the server; instead, the server pro-actively sends data "towards" the clients. Broadly, this works as follows: the server repeatedly transmits a set of data items over a broadcast medium (e.g., satellite channel, cable network). When a client needs any of these items, it starts monitoring the broadcast medium until the item of interest gets transmitted. Using the terminology of [Vis94], the client **listens to** (or **tunes in**) the broadcast channel and **filters** the data it needs.[2]

Through repetition, the server keeps the data circulating in the broadcast channel. Data are continuously passing by the clients which just "grab" whatever they need, without making any requests. In some sense, this allows the broadcast medium to be perceived as a memory space for storing data. It can be thought of as a special type of a storage device, which any client can read from at any time. In its simplest form (i.e., "flat" broadcast), it functions like a one-cylinder disk or a closed-end tape. More complex forms can be achieved by employing proper broadcast scheduling techniques that result in more sophisticated storage organizations.

The merit of this approach has been under research investigation for more than a decade. Probably, the most important examples or research along this line

[2][Vis94] refers also to this delivery mode as publishing

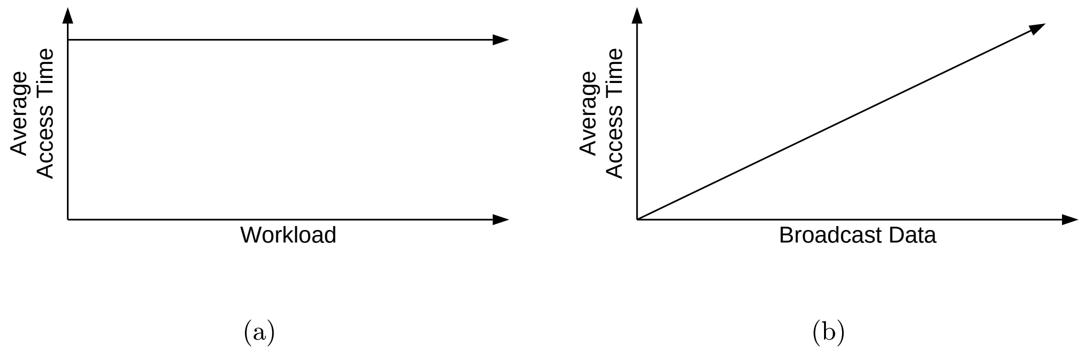


Figure 3.2: Typical performance of push/broadcast systems

are the Datacycle database machine [BGH+92], the work carried by the Dataman group [Imi96], and the Broadcast Disks project [AFZ95]. In the context of the latter, [Ach98] presents a thorough examination of push/broadcast delivery. It builds a case for it with several supportive arguments and studies a number of related issues and tradeoffs (e.g., broadcast organization, client caching and prefetching)

Let us now sort out some of the important advantages of push/broadcast delivery:

Scalability Broadcasting can be used to efficiently reach huge, and possibly widely dispersed, client populations. This is exactly what prescribes it as an attractive approach for large scale data dissemination. The number of recipients of broadcast messages can grow arbitrary large without any additional impact on the network. Given that, storing data in a broadcast channel through repetition creates a memory space with a unique and important property: it exhibits no access contention. Practically, this means that it can be accessed by any number of clients concurrently without any performance degradation. As shown in Figure 3.2(a), the size of the workload, i.e., the client demands for data, has no effect on the average data access time. If we contrast this to the performance of a pull/unicast system (Figure 3.1), we can easily see that push/broadcast has a significant potential as a scalable data delivery mechanism.

Bandwidth savings Compared to unicasting, broadcasting yields significant savings in terms of network bandwidth when a single message is intended for or expected by multiple clients. In fact, the more the recipients of the message the greater the savings are. In that respect, broadcasting is the most "bandwidth conscious" method of distributing popular content.

Asymmetry The information consumption model followed by many applications generates a unidirectional information flow from the server to the clients. Server-initiated (i.e., push) delivery is a natural match for this purpose. It eliminates the need for client requests, avoiding the associated overheads and costs. On one hand, the server avoids the computing overhead of servicing the requests as well as the communications overhead of connection-oriented protocols that are typically used for making the requests. One the other hand, at least in the mobile environment, clients save money and battery life since wireless connections are typically expensive and message transmissions consume significant amounts of power. In any case, push delivery actually eliminates the requirement for an uplink altogether.

Support for disconnection Repetitive broadcast also serves as a convenient way to disseminate data to clients that are not reachable at all times. These are clients that choose or have to stop using their communication devices, and operate in **disconnected** or **sleep mode** [KS91]. For example, this is the typical operation mode for mobile users. In such case, if data are kept rotating in the broadcast channel, a client can just pick up whatever it needs at any time, just by reactivating the communication device and filtering the incoming data stream.

In order to take advantage of these benefits, push/broadcast delivery needs to be very carefully designed and deployed. The reason is that the benefits may be offset by its two major weaknesses: sequential data access and lack of usage feedback. Let us examine each one separately.

Sequential access A client that wants to retrieve one of the data items that the server broadcasts, starts listening to the channel and waits until the specific item is actually broadcast. The client's **response time** or **access time**, i.e., the time it has to wait, depends on the amount of data that are being broadcast. Using the storage device analogy, data in the broadcast channel are accessed sequentially. As a direct consequence, the more data are stored the higher the data access latency. This is shown in Figure 3.2(b) where the average access time grows proportionally with the volume of data being broadcast.[3] On average, the access time for any data item is equal to half its repetition period in the broadcast. Thus, we must be really frugal in selecting data to broadcast so that the repetition periods, and therefore the access times, do not get prohibitively long.

No usage feedback The absence of clients requests has, however, a negative consequence: the server cannot have explicit information about what data the clients want exactly ("the burden of push" [FZ97]). This may lead to either or both of the following two unfortunate scenarios: First, the server may never deliver data that clients actually want. Unless clients have an alternative way to get that data, their needs will never be satisfied. Second, the server may be broadcasting

[3]The slope of this performance line depends on the bandwidth of the broadcast channel.

data that no client needs. Sending irrelevant data has an adverse effect not only because it is a waste of network bandwidth, but also because it unnecessarily increases the access time for useful data. On top of that, the server cannot know whether it is actually making either mistake, as the clients never acknowledge the usefulness of the broadcast content. In contrast, with pull-based delivery, the server makes use of the requests to build a "fully-informed" model[4] of the clients' needs and, obviously, transmits only useful data. i.e., data that a client requested. For a push-based model, the server relies on implicit information to predict what the clients want. For example, users may subscribe with the server and provide profiles of interests. Based on this information, the server can decide what data to push (e.g., [OPSS93, AAFZ95, DCK$^+$97]). In any case, the success of push/broadcast delivery depends on the server to create a clear picture of the clients' needs.

Besides these two weaknesses, the organization of the broadcast itself is a crucial factor for the success of such a system. In its simplest form, the broadcast storage is "flat". This means that data items are broadcast one after the other in a round-robin fashion. The repetition period, and thus the average access latency, is the same for all items. However, it has been recognized that this is not always the best approach. Instead, in many cases it is advantageous to be broadcasting some items more often than other (e.g., [Won88, IV94, AAFZ95, ST97, HV97b]). Compared to a flat broadcast, this decreases the latency for frequently broadcast items, but increases the latency of the rest. This "unfair treatment" is justified by the fact that, typically, not all data are equally popular. In fact, data access patterns often exhibit high degrees of skewness [DYC95, ABCdO96]. Therefore, by decreasing the latency for items in high demand, we decrease the overall ex-

[4]Assuming no client requests are dropped from the server's input buffer

pected latency. The optimal broadcast organization depends on the data access distribution [AW87]. Obviously, the performance gain from non-uniform schemes becomes more important as the volume and/or the popularity skewness of data increases.

3.2.3 Hybrid Delivery

As the name implies, hybrid data delivery is the combined use of two (or more) different mechanisms for delivering data from a server to a set of clients. In our case, the two mechanisms are pull/unicast and push/broadcast.[5] The goal is to create an integrated delivery mechanism that can efficiently support large scale information consumption in a modern networking environment.

According to the preceding discussion, each of these two delivery methods has limitations that may render it inappropriate for achieving this goal. Most notably, pull/unicast suffers from scalability problems as the server and/or the (adjacent) network links often become a bottleneck, while push/broadcast suffers from long response times when too much data have to be broadcast. Hybrid delivery aspires to overcome these limitations by combining them in a **synergistic** and **complementary** manner. The idea is to use each one for what it is best at, and the same time, exploit the advantages of the one to help overcome the limitations of the other. Push/broadcast is an effective way of distributing popular content, but it is wasteful to use it for data that very few clients need. Pull/unicast, on the other hand, can handle any type of data as long as the server is not saturated with requests. Therefore, in order to make the best use of both of them, we should be broadcasting only popular data, and unicasting the rest only when they get

[5]For the interested reader, some other hybrid approaches are briefly discussed in Section 2.1.2.

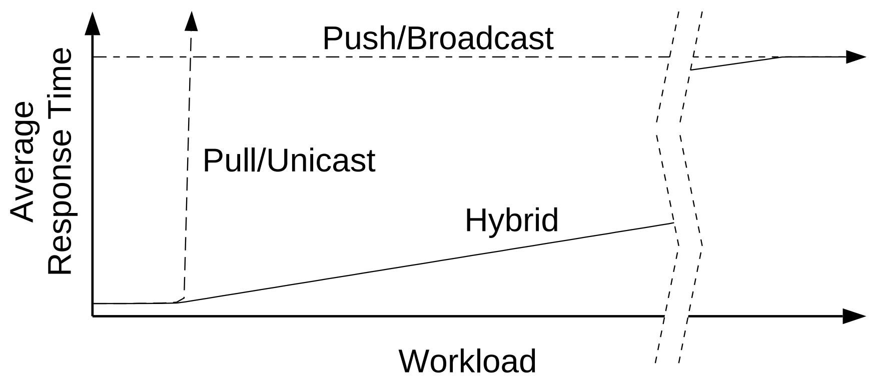


Figure 3.3: Target performance for hybrid delivery systems

pulled by clients. In other words, push/broadcast should be used to disseminate information satisfying the bulk of clients' needs, while pull/unicast should take over the (significantly reduced) request load for unpopular information.

Figure 3.3 attempts to provide some insight on the benefits of the hybrid approach. In the figure, we show the kind of performance we hope to achieve with a hybrid system, and compare it to the performance of both the individual methods (Figures 3.1 and 3.2). Our intention here is to make a qualitative comparison; the relative sizes of the curves may be quite different, depending on the application and the underlying infrastructure. What we see in the figure is that hybrid delivery aspires to improve the performance of either basic method. On one hand, by pushing popular data, it relieves the server from a heavy load of requests (those for popular data). As a result, access contention at the server is reduced dramatically, allowing the system to operate on a scale much larger than its (pull) capacity (the "Hybrid" line in Figure 3.3 extends well to the right of the "Pull/Unicast" line). On the other hand, when data can also be delivered on demand, the server does not have to broadcast any data item that any one client might need at any time. Instead, it can concentrate on broadcasting only popular data, reducing the response time of the push/broadcast delivery (the “Hybrid” line runs below the “Push/Broadcast” line). This response time is determined by the amount of data that are considered popular and need to be broadcast, and depends on the workload parameters (volume of requests and access distribution). At extreme cases of high workloads where all data receive a considerable amount of requests, the system has to broadcast all the data, performing like a push/broadcast only system.

Beyond this performance advantage, hybrid delivery also makes a better use of the available resources. This argument has two sides: First, the broadcast bandwidth is used for popular content only. This means that every broadcast message is expected to have a substantial number of recipients. Ideally, none of these messages will turn out to be superfluous. Second, as it handles requests only for less frequently requested information, the server does not have to process the same request and transmit the same data over and over again. Thus, redundancy is extenuated to a big extent, saving both processing cycles and unicast bandwidth.

Nonetheless, in order to successfully employ a hybrid delivery approach, we need to address a number of crucial issues: How we can classify data as popular or unpopular? How can we measure this popularity? How exactly do we decide what to broadcast and what to service on demand? Can we get good performance for all workloads? The answers to these questions depend on the applications and the underlying infrastructure (e.g., network bandwidth). For every situation, we need to develop techniques that take into account the involved parameters and the desired performance characteristics. In many situations, however, we need to answer one more question: Can we get this performance even under dynamic workloads, i.e., request loads with varying intensity and patterns? This question is becoming ever more critical for the success of information dissemination systems as user demands can be very dynamic and often unpredictable. Possible causes for this behavior include:

- Client interests may shift over time. For example, in the morning users usually need information about traffic delays and the weather, while in the evening they may want to know about movie showing times or table availability in local restaurants.

- Real life events may generate bursty requests for relevant information. These can be either scheduled (e.g., elections, sports) or unscheduled (e.g., breaking news, emergencies). The latter are typically more difficult to respond; often they are referred to as "panic situations".

- In mobile settings, clients unpredictably connect to and disconnect from the information sources. Also, in cellular networks, they arbitrarily join and leave coverage areas.

Therefore, we need not only techniques that can implement the advantages of hybrid delivery, but also techniques that can adapt to clients' demands. With this in mind, the main proposal of this thesis is **adaptive hybrid data delivery**. Broadly, the goal is

> Fast and scalable data dissemination, responsive to dynamic workloads
> with prudent usage of (server and network) resources

In the next section we introduce air-caching, the central concept of our work, as a general framework for achieving the above goal. We present its goals and basic principles, as well as define the general operating guidelines. These will serve as the basis for the applications described in the following chapters.

3.3 Air-Caching

Generally, the goals set for adaptive hybrid delivery in the previous section can be rephrased as follows:

- Increase system throughput in terms of requests satisfied or serviced per unit of time.
- Reduce contention at the shared resource, i.e., the server.
- Adapt according to the users data access patterns.

Stated this way, these goals are very familiar; they are the main goals set by **data caching** and **cache management** techniques employed in several (distributed) computing systems. Caches are memory layers placed between data providers and data consumers to decrease access latency and increase throughput. Examples include memory hierarchies of computer systems [HP96], main memory buffering in operating systems [SG94] and database management systems [EH84], client caching for client/server database systems [Del93, Fra96b], and more recently, proxy caching for web content [CDN$^+$96, GRC97]. Following this pattern, our approach to hybrid data delivery is based on the notion of data caching. Specifically, we use the available broadcast channel to create a global cache lying between the server and the clients. Driven by the fact that on most occasions the carrier of a broadcast channel is a wireless medium, we call this cache an **air-cache**.[6]

As we have seen earlier, repetitive broadcasting creates the effect of storing data in the broadcast channel. The air-cache builds exactly on this idea. Generally, it works as follows: The server selects a set of data that deems as popular which it broadcasts repeatedly. That appears as caching (i.e., storing) the most popular data on the air (i.e., broadcast channel). When a client needs a data item, it starts listening to the broadcast to find out whether the item of interest is air-cached (i.e., being broadcast). This is facilitated by some indexing information that is being broadcast along with the actual data. If the item is indeed in the air-cache, the client retrieves it from there as soon as it gets broadcast. This constitutes an **air-cache hit**. If however, the client determines that the item is not air-cached, then it sends a request to the server asking for it. Correspondingly, this is an **air-cache miss**.[7] The server processes the request and unicasts the reply back to the client.

The air-cache is a very convenient abstraction for describing and implementing hybrid data delivery. The main reason is that it allows the problem of adaptive hybrid delivery to be expressed—and addressed—as a cache management problem. Generally, the goal is to maintain an air-cache able to satisfy most of clients requests, leaving the server to handle a small volume of cache misses. The questions raised in the previous section can now take the following form: Which of the data should be air-cached? When should a data item be air-cached? When should it be removed form the air-cache?

[6]Obviously, this name does not preclude such a cache from being realized over a wired broadcast medium

[7]In the rest of the thesis, air-cache hits and air-cache misses will be referred just as hits and misses respectively.

Despite its similarities to more traditional caches, none of the existing management techniques can be applied in the case of air-caching. The reason is that the air-cache has a number of unique properties that require a completely different approach. Thus, management techniques have to be tailored to these special properties. Chapters 4, 5, and 6 are devoted to the presentation of such techniques used for this purpose. Below, we list the special characteristics of air-caching, as well as the performance tradeoffs that are involved. All these stem from the fact that air-caching is based on storage of data in a broadcast channel, and therefore apply to all similar architectures. For the interested reader, most of these are discussed extensively also in [Ach98].

3.3.1 Properties

First of all, the air-cache is not a fixed latency memory. Because of the sequential nature of the broadcast, the actual latency for retrieving a data item depends on the next transmission of the item relatively to the moment a client starts monitoring the channel. This is unlike main memory caches, where every access bears the same delay. It resembles more disk storage where the actual latency depends on the location of the disk heads and the rotation of the platters at the moment of the request. Therefore, similarly to disks, in the case of air-caching we use the **average access latency** as a performance metric. As we have mentioned earlier, the average access time for any item in the cache is equal to half the repetition period of that item. But, what makes this characteristic unique is that for air-caching even the average access latency is not fixed; instead, it can be dynamically adjusted. In fact, it is determined by the size and the structure of the air-cache as described below.

Contrary to traditional caches, the air-cache can be optimized in more than one way. Generally, an air-cache manager has the flexibility of modifying three decision variables:

Contents This variable is common to all types of caches. Actually, in most cases this is the only variable. Data are moved in to and out of the cache to better accommodate the consumers' demands. Naturally, this is a primary function of the air-cache as well.

Size Caches implemented in real (physical) memory (main memory, disk, etc.) are limited by the size of the memory space allocated to them. The cache management algorithm does not have the luxury of taking up more space if necessary.[8] The air-cache, however, does not have this limitation. Being a software construct, the cache manager can dynamically change its size. In other words, it can cache (i.e., broadcast) more or less data as it deems appropriate. This flexibility comes with a price though. Caching more data results in increased average access latency as it actually translates to having to broadcast more data.

Structure As it was discussed in Section 3.2.2, push/broadcast systems can adjust the data repetition frequencies to better match the clients demands. Similarly, the air-cache has many options for organizing the broadcast data. These range from a flat to a very fine-tuned scheme. In the former, all items are broadcast with the same frequency. In the latter, a separate, properly

[8]A system administrator can increase the size of memory used for caching by, for example, installing more memory to a system or adding another disk. However, this is an off-line process, beyond the control of the cache management algorithm.

selected, frequency is used for each individual item. In between, items are partitioned into groups, with each group having its own broadcast frequency, following the "multi-disk" model introduced in [ZFAA94]. In any case, the structure of the air-cache directly affects the average time to access the cached data.

Moreover, the air-cache mechanism differs in the way misses are handled. Typically, caches are intermediate levels in a memory hierarchy. Within such hierarchies, the consumer (e.g., processor, client, web browser) looks for data in the closest (in terms of access latency) cache level. Should this result in a miss, the cache manager takes over the task of retrieving the data from lower levels. The retrieved data are brought into the cache and then delivered to the consumer. If the cache happens to be full, some previously cached data have to be evicted first, to make space for the new data. The victim data are selected by the cache replacement policy. Except for some increased latency, this whole process is transparent to the consumer. For air-cache misses, however, the scenario is different. The client is responsible to detect a miss as well as request the missed data from the server. In addition, the requested data do not have to be delivered to the client through the air-cache; and even if they do they do not have to replace other cached data, as the size of the cache can easily change. In this thesis, we examine only the case where missed data are unicast to the clients. The implications of delivering requests through the air-cache (i.e., through the broadcast) are beyond our scope. To some extent, the involved tradeoffs for that case are studied in [AFZ97] in the context of Broadcast Disks.

The last, but very important, distinctive property has to do with the fact that the server does not get any feedback about the air-cached data. Clients do not acknowledge the retrieval any of the broadcast data. In other words, cache hits are not reported to the server. The only information available to the server is cache misses, i.e., requests for data not in the air-cache. Obviously, this unique property renders all the traditional (hits-based) cache management techniques (e.g., LRU, MRU) inapplicable in this case. Instead, we need to develop effective algorithms that rely solely on misses. But, as we explain below, this forces us to reconsider the problem of cache management along a different train of thought.

3.3.2 Management

Broadly, cache management techniques aim at solving a basic optimization problem: "Given the size of the cache, select (to cache) the set of data that minimizes the number of cache misses." The intuition behind this objective is that misses are expensive; they incur higher latencies by accessing lower level memories, and, where applicable, engage valuable shared resources (e.g., system bus, server). Therefore, a smaller number of misses yields lower average access time, less contention for shared resources, and therefore, higher throughput.

In practice, it is not possible to solve this problem optimally, as this would require perfect knowledge of future data accesses. Instead, the conventional approach is to employ a cache replacement policy that tries to assess the importance of data items based on the recent history of data accesses.[9] Therefore, such policies tend to keep in the cache data that have been accessed recently and/or frequently, and remove data that have not, on the premise that this behavior is likely to continue, at least for the near future. In other words, given the recent hits and misses

[9]In some cases, application specific "hints" can help the cache manager take more insightful decisions.

they assess the current workload, i.e., the consumers demands, and maintain the part of the cache expected to be more useful.

However, air-cache management is a slightly different problem. A first indication attesting to the peculiarity of the problem is that with air-caching the above objective, i.e., minimize the misses, is trivial to accomplish. As the air-cache does not have a size constraint, misses can be completely eliminated simply by caching all the available data. Nonetheless, this is not at all advisable since it may have disastrous effects on the cache performance. Except for small size databases, the access latency may increase beyond the point of any practical use. This makes obvious that we need to set an alternative optimization goal for air-caching.

This new goal is basically dictated by three observations. The first has to do with the role of the misses. A deficiency of air-caching is that the server does not receive information about cache hits. This impedes cache management considerably. The server can neither appraise the usefulness of the air-cache nor draw a clear picture about the workload. The only indication of any client activity is the misses sent to the server. But, these make up only for part of all the requests, offering an incomplete picture of the system's workload. Still, the server has to resort to them for all the related decisions. Therefore, misses now play a vital role for the system. In fact, the more the misses the better the workload statistics. This leads to the counterintuitive conclusion that misses are welcome.

The second observation enforces this conclusion even more, at least to some extent. Unlike conventional misses, the air-cache misses are not necessarily more expensive than the hits. In many cases, getting data from the server might be almost as fast as (if not faster than) getting it from the air-cache. However, this happens only if the server is not heavily loaded and can respond quickly to client requests. If the volume of misses is too big for the server to handle, then it becomes a serious bottleneck and fails to respond promptly. Miss service times grow arbitrarily high at a very fast rate (hyperbolically). Consequently, misses are indeed welcome as long as they do not swamp the server.

The final observation is that the latency of the air-cache is an increasing function of its size. This clearly suggests an optimization direction towards small size caches. In an ironic manner, the fastest cache is the almost empty cache. Albeit, an almost empty cache is of almost no use. Most clients cannot find what they need in the cache and are forced to generate misses, i.e., send requests to the server. Again, if too many, these misses can overload the server. Thus, the server must make sure that the air-cache holds enough data to limit the volume of misses at a tolerable level. At the flip side of this, the server must make sure that it is not caching any data that hardly any client needs. Such data obviously do not save the server from any serious load of requests. Instead, they unnecessarily increase the size and the latency of the cache. Hence, the server must be caching not only enough data, but also the right set of data. It turns out that this is the trickiest part of air-cache management, as the server cannot explicitly know whether cached data are actually used or not.

In the light of the above observations, we can now phrase the basic principle of air-caching:

> **Cache the minimal data set that results in the maximum load of misses the server can handle**

The maximum tolerable load is determined by the server's processing capabilities and its network connections. In other words, it is determined by the rate the server can process requests, and the rate it can transmit data to the clients. Depending on the application, the workload, and the operating environment, either one can be the limiting factor that ultimately defines the system's pull capacity.

For a server to follow the above principle, it needs algorithms that can estimate the popularity of the available data, as well as predict the effects of caching or not caching them. The way this can be achieved depends on the application under consideration and the data being managed. In the chapters that follow, we present a set of such techniques we have developed for three different applications. Through these techniques, and the experimental results that accompany them, our primary intention is to establish the viability and versatility of air-caching as an efficient and scalable data dissemination technique. At the same time, we want to highlight some of the subtle points and the related design tradeoffs. Hopefully, these can serve as the guidelines for an actual deployment of the proposed ideas.

3.3.3 Structure

In the previous section, we overlooked one of the decision variables of the air-cache, its structure. The reason was that the discussion was concentrated on the tradeoffs of hybrid data delivery. The structure was not intertwined as it affects only broadcast delivery. It is, however, an important variable that can have a significant impact on the overall system performance. As such, it should be appropriately selected in the system design or set by the cache manager.

In this thesis, we define two general structures for the air-cache, a simple and a composite:

Definition 2 *A* **simple air-cache** AC *follows a flat broadcast scheme where data get broadcast successively one after the other.*

Definition 3 *A* **composite air-cache** *is a combination of C simple air-caches* $\mathrm{AC}_1, \mathrm{AC}_2, \ldots, \mathrm{AC}_C$ *in a single broadcast channel. For a given set of non-negative numbers* $\phi_1, \phi_2, \ldots, \phi_C$ *and* $\Phi = \sum_{i=1}^{C} \phi_i$*, data broadcasts are multiplexed so that*

1. *data cached in* AC_k *take up at least* ϕ_k/Φ *of the broadcast bandwidth, and*

2. *the interval between successive broadcasts of the same item is fixed*

A flat air-cache is used in cases where a finer approach is not expected to yield any significant performance benefit. That may happen, for example, if the relative importance of the cached data does not vary significantly, or if the size and the latency of the cache are small with slim margins for improvement. A composite air-cache, on the other hand, is a more flexible structure that can be used in more intricate situations. It enables the partitioning of data into groups according to some application specific criteria, and the creation of a separate simple air-cache for each such group. In effect, it is a generalization of the "multi-disk" broadcast model [ZFAA94]. These simple air-caches can be managed either collectively or independently. In other words, there can be either a single composite air-cache manager, or multiple (independent) simple air-cache managers. The former method is used in Chapter 5 with popularity-based grouping of the data. The latter is used in Chapter 6 where data are semantically partitioned and managed independently.

In the above definition of the composite air-cache, we require fixed intervals between successive broadcasts of the same items.[10] This is a well known property for repetitive broadcasting [Chi94, AAFZ95, VH96] that originates from the "residual life paradox" [Kle75]. In particular, it has been shown that, for a given

[10]This trivially holds for flat air-caches.

mean repetition period, the average access time for any item in the broadcast is minimized when the variance of this period is zero, i.e., when the period is fixed [JW95]. Furthermore, this repetition regularity can be also exploited by the clients to make better use their resources. For example, if they know the exact arrival time of any item, clients can manage their local cache more effectively, and can save battery by turning off their receivers until an item of interest is scheduled to be broadcast [AAFZ95].

The parameters $\phi_1, \phi_2, \ldots, \phi_C$ determine the relative importance of the individual air-caches. The higher the value of ϕ_k the more bandwidth is allocated to AC_k and, thus, the more frequently data cached in it get broadcast. Note that ϕ_k/Φ is not the exact share of bandwidth allocated to AC_k; it the minimum share guaranteed to it. The exact share may be higher if one or more of the other air-caches are empty and not using their shares. If we define u_k to be 1 when AC_k is actually used (i.e., it is not empty) and 0 when it is not used (i.e., it is empty), then the actual share for AC_k is $b_k = \frac{u_k\phi_k}{\sum_{i=1}^{C} u_i\phi_i}$.

In the next section, we present how both simple and composite air-caches with the abovementioned characteristics can be realized over a broadcast channel.

3.3.4 Implementation

The effect of caching data in a communication channel is created by repetitive broadcasting. With that in mind, a flat air-cache is straightforward to realize. Our implementation uses a queue Q to keep cached data, and works as follows: We append all the cached items to Q. Each time the broadcaster (the process responsible for actually broadcasting the data) selects to transmit the item at the head of the queue. After it does, the item is removed from the head of the queue and gets re-queued back to the tail. This process repeats forever resulting in a flat periodic broadcast of all the items in Q. Occasionally, while this is happening, we need to either add data to the cache or remove from it. But, this is again quite simple. To cache more data, we just append the new data to the tail of the queue. To remove a data item, we wait until the item gets broadcast. When it does, we remove it from Q as usual, but we do not re-queue it, leaving it out of the loop.

A composite air-cache is realized by extending the above method to include multiple queues $(Q_1, Q_2, \ldots, Q_C)$. Each queue operates exactly as described above. However, only one item can be broadcast at a time. Therefore, the broadcaster needs an algorithm to decide each time the proper queue to broadcast from. The task of the broadcaster is to produce a schedule that adheres to the definition of the composite air-cache. It turned out that a perfect match for this purpose is a **worst-case fair weighted fair queueing** algorithm [BZ96]. This algorithm was designed to provide quality of service guarantees in packet switched networks. It propagates over a network link packets from several competing streams (input queues) according the bandwidth share allocated to each one of them. This obviously is in accord with the first requirement of composite air-caches. Furthermore, from the worst case analysis presented in [BZ96], it follows that it approximates the second requirement very well, yielding repetition intervals with minimum—but not always zero—variance. The complexity of the algorithm is $O(\log(C))$ which means that it is very efficient for reasonable number of air-caches. Last but not least, it can accommodate on-line changes in the structure of the air-cache, only with a simple adjustment of the parameters ϕ_k. The actual algorithm we used is a slightly modified version of that presented in [HV97b].

3.3.5 Indexing

An important question that we have not addressed so far is how the clients can find out what is being broadcast. For a hybrid delivery system, this is a crucial question since clients have to decide whether, when, and what for they should contact the server. The answer to this question lies on channel indexing. In the context of data broadcasting, an **index** is some sort of directory information about upcoming data. Typically, it lists the contents of the broadcast program, and it is repeatedly broadcast—either in parts or as a whole—interleaved with the actual data [IVB94a]. Often, it also contains detailed timing information that indicates the exact time each item is scheduled to be broadcast. This information usually comes in the form of offset units, i.e., the number of broadcast units (e.g., pages, buckets) following the index [IVB94a].

In the air-caching parlance, the index provides the necessary information to discriminate hits from misses. When accessing the air-cache through an index, a client tunes to the channel and waits for the next instance of the index to be broadcast. The time it spends waiting for the index is called **probe wait** or **probe time** [Vis94]. When it does receive the index, the client checks whether what it needs is listed or not. If it is, we have a cache hit, and the client keeps on monitoring the channel until the needed data get broadcast. Optionally, if the index contains timing information, the client can save energy by turning off the receiver until just before the data is actually transmitted. On the other hand, when (some of) the needed data are not listed in the index we have a cache miss, which means that the client has to send a request to the server.

This use of an index bears a performance tradeoff. According to the above description, the probe time is part of the client's response time. It affects mostly the misses because it precedes, and hence delays, the request/reply phase. The time clients have to wait for the index depends on how frequently it is repeated in the broadcast; the higher the repetition frequency the smaller the average probe time. But, in order to broadcast the index, we are "stealing" bandwidth from the actual data. Therefore, if we transmit the index very often we may reduce the effective bandwidth considerably. As a result, air-cached data take much longer to arrive, yielding worse access times. This means that very frequent transmission of the index works in favor of misses and against hits; it reduces probe time (i.e., the time to detect a miss and make a request) but increases the air-cache latency. Naturally, this counteraction is affected by the size of the index, as compared to that of the actual data. If it is very small, even very frequent index broadcasts have negligible effect on the cache latency. If, on the other hand, the index is long the system will be very sensitive to its broadcast frequency.

Notice that this tradeoff is similar to one discussed in Section 2.1.3. There, the concern was to save battery life for mobile units. The difference is that access time was traded for tuning time, i.e., the time a client has to be listening to the broadcast channel in order to locate and download the item of interest. Nevertheless, the nature of the tuning time, which consists of multiple small non-contiguous intervals, is different from the probe time discussed here.

In our work, we touch upon on indexing from the air-caching perspective. Specifically, we focus on what kind of information should be included in the index so that the clients can promptly decide whether they have to contact the server to request any data. In all cases we examine, the size of the index is not big enough to raise any serious concern about its broadcast frequency. Thus, we arbitrary choose rather high frequencies that yield small probe times. Last, we do not directly address the issue of energy conservation. However, the proposed indexing schemes can be extended with detailed timing information, enabling the clients to tune selectively.

3.3.6 Definitions

In the last section of the chapter, we present a set of definitions that will be used throughout this thesis to describe the structure and performance characteristics of an air-cache. The definitions are given in the context of composite air-caches. However, they are also applicable to flat air-caches as those can be considered degenerate cases of composite air-caches.

In our work, we follow the established convention for (simulation-based) broadcast systems of measuring time in broadcast units [Vis94, Ach98]. Generally, a **broadcast unit** is defined as the time required to broadcast a data item of some reference size (e.g., disk page, fixed size packet). The benefit of this abstract measurement method is that it factors out environment and application specific details (i.e., bandwidth and data sizes). The obtained results can easily be projected later to any actual setting.

The following definitions assume a composite air-cache consisting of C simple air-caches $AC_1, AC_2, \ldots, AC_C$. We define:

Definition 4 *The* **bandwidth factor** $\phi_k \geq 0$ *of* AC_k *determines the minimum share of broadcast bandwidth allocated to* AC_k.

Definition 5 *The* **bandwidth share** b_k *of* AC_k *is the exact share of broadcast bandwidth allocated to* AC_k*. As we saw earlier* $b_k = \dfrac{u_k \phi_k}{\sum_{i=1}^{C} u_i \phi_i}$.

Definition 6 *The* **cardinality** n_k *of* AC_k *is the number of data items cached in* AC_k*. The cardinality of all the air-cache is* $N = \sum_{k=1}^{C} n_k$.

Definition 7 *The* **size** s_k *of* AC_k *is the total size of the* n_k *items in* AC_k*. The size of all the air-cache is* $S = \sum_{k=1}^{C} s_k$.

Definition 8 *The* **period** T_k *is the minimum time interval during which all items cached in* AC_k *are guaranteed to be broadcast at least once. Similarly, the period* T *of all the air-cache is the minimum time interval during which all cached items (in all* C *simple air-caches) are guaranteed to be broadcast at least once. It follows that* $T = \max_{1 \le k \le C} \{T_k\}$.

The fact that air-cache AC_k uses b_k of the broadcast bandwidth means that a item of unit length cached in AC_k takes, on average, $1/b_k$ broadcast units to be broadcast. Given that the total size of items cached in AC_k is s_k, we can derive that, on average, the period T_k is $\frac{s_k}{b_k}$ broadcast units. The value of T_k reflects the average latency of AC_k. This means that the latency of a simple air-cache is not completely determined by the bandwidth allocated to it; it also depends on the amount of data cached in it. A related performance indicator is the broadcast frequency, which we define next.

Definition 9 *The* **broadcast frequency** f_k *of* AC_k *is the average number of times any item in* AC_k *gets broadcast within a period* T *of the air-cache. It easily follows that* $f_k = \frac{T}{T_k} \ge 1$.

The broadcast frequencies are not an absolute performance metric of the air-cache. Instead, they are only used in a relative sense to compare different components of a composite air-cache. For example, we can infer that AC_k broadcasts its data f_k/f_m times more (or less) frequently than AC_m. This also means that AC_k is f_k/f_m times faster (or slower) than AC_m. Finally, note that $\frac{T_k}{T_m} = \frac{f_m}{f_k}$. Table 3.1 contains a summary of the symbols defined thus far and can be used as a reference for the rest of the thesis.

	For each AC_k	For entire air-cache
Bandwidth factor	ϕ_k	$\Phi = \sum_{k=1}^{C} \phi_k$
Bandwidth share	b_k	$1 = \sum_{k=1}^{C} b_k$
Cardinality	n_k	$N = \sum_{k=1}^{C} n_k$
Size	s_k	$S = \sum_{k=1}^{C} s_k$
Period	T_k	$T = \max_{1 \leq k \leq C} \{T_k\}$
Broadcast frequency	f_k	-

Table 3.1: Summary of symbols

Chapter 4

Demand Driven Air-Caching

Over the last few years the number of people accessing information electronically over the Internet has been growing very fast, and it expected to continue to grow at a similar pace. This trend creates unprecedented requirements on information services in terms both of the networking infrastructure and server capacity. This phenomenon is aggravated during special events, such as the Olympics, national elections, and so forth. For example, [Las98] reports that during the 1998 Winter Olympics the official web site received about 450 million requests, with a peak rate exceeding 100 thousands requests per minute.

In this chapter, we show how hybrid data delivery can be used to address such huge workloads of user requests. Several studies of web access traces have identified that, typically, within popular information sources there is a relatively small number of data items that receive most of the user requests [ABCdO96, DMF97, AW97]. In other words, there are few items that are extremely popular and attract a massive number of requests. The set of these items is usually called the **hot-spot** of the database. The majority of data outside the hot-spots get accessed only occasionally, if at all. This high degree of skewness in the data access pattern makes hybrid delivery a suitable alternative for efficiently handling very high request rates. If the database hot-spot is kept in an air-cache, most of the user requests can be serviced without any interaction with the server. This leaves the server with the significantly easier task of servicing requests only for unpopular data not found in the air-cache.

Obviously, the key issue for this scheme is the identification of the database hot-spots. There are at least two major obstacles that we have to overcome: First, user needs can be neither characterized nor predicted a-priori because of the dynamic nature of the demand. In the extreme cases, for example, emergency or weather related situations may cause abrupt shifts in demand. Second, as we discussed in the previous chapter, with air-caching the server gives up a lot of invaluable information about data accesses, and cannot directly assess the actual user needs.

In this chapter, we propose a technique that, driven by the partially observed user demands, adjusts the air-cache contents to match the hot-spot of the database. We show how this hot-spot can be accurately obtained by monitoring the limited number of air-cache misses. We develop an adaptive algorithm that relies on marginal gains and probing to identify the popular data. With this technique, the overall performance of this hybrid system can surpass the capacity of a traditional unicast-only server by multiple orders of magnitude. The advantage of hybrid delivery is that its performance is not directly affected by volume of the workload; instead, it is determined by the size of the database hot-spot, i.e., the amount of frequently requested data. Thus, it exhibits significant scalability margins, even for rapidly changing access patterns.

4.1 Performance Analysis

In this section, we develop a simplified analytical model for hybrid data delivery, which provides some intuition behind the algorithms presented later in the chapter, and illustrates the involved tradeoffs. Based on this model, we discuss how broadcast and unicast can work synergistically to yield high data service rates. In this study, we consider only simple (flat) air-caching. The main premise of this decision is that hot-spots are relatively small, and therefore, require small size air-caches. Because small size air-caches have also small average access latency, a more elaborate air-cache structures is not expected to bring any significant performance improvement. However, as part of our future work we intend to investigate the implications of large and more sophisticated air-caches to the results presented herein.

4.1.1 The Hybrid Model

In a hybrid scheme, we can exploit the characteristics of each of the data delivery modes and integrate them in a way that better matches the clients' demands. The objective is to deliver the needed data with minimum delay to very large numbers of clients. Striving for that goal, we can look for solutions that range between pure push/broadcast and pure pull/unicast.

Consider a database containing M data items of equal size. Assume that the demand for each item i forms a Poisson process of rate λ_i with the items numbered such that $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_M$. A server, modeled as an M/M/1 system, services requests for these items with mean service time $1/\mu$. In addition, this server can broadcast data over a channel, at a rate of one item per time unit. Also assume that, for some reason, the server decides to broadcast (i.e., air-cache) the N first

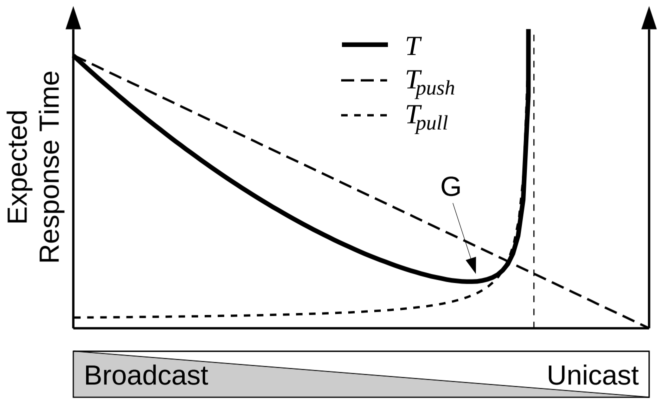


Figure 4.1: Balancing data delivery methods

items, and offer the rest on-demand. If we define $\Lambda_k = \sum_{i=1}^{k} \lambda_i$, then the expected response time for requests serviced by the server is $T_{pull} = \frac{1}{\mu - (\Lambda_M - \Lambda_N)}$, while for those satisfied by the broadcast it is $T_{push} = \frac{N}{2}$, half the time required to broadcast all N items. The expected response time T of the hybrid system is the weighted average of T_{pull} and T_{push}.

Figure 4.1 plots a representative example of how T, T_{pull} and T_{push} vary with respect to the number of items being broadcast. To the left, all items are repeatedly broadcast; to the right, all are unicast on demand. We have assumed that the total workload is greater than μ, which is a safe assumption for large scale systems with huge client populations. Henceforth, we refer to μ as the system's **pull capacity**. The first thing to note in this figure is that the performance of the pull service T_{pull} is a hyperbolic function of the imposed load. It is evident that with too little broadcasting, the volume of requests at the server may increase beyond its capacity, making service practically impossible (right side of the graph). On the other hand, the response time for pushed data is a straight line, growing proportionally to the volume of the broadcast data. Hence, too much broadcasting is not desirable either. Obviously, for best performance, we must look for solutions in the area around point G, where we can maintain the best balance between data push and pull.

4.1.2 Practical Considerations about Workloads

The discussion of the previous section suggests that it is possible to balance data delivery modes in order to obtain optimal response time. However, this optimal solution depends on the shape and size of the imposed workload. In what follows, we explore hybrid delivery from a practical perspective and give a qualitative answer to how a combination of broadcasting and unicasting can be advantageous.

Intuitively, data broadcasting is helpful when its content is useful to multiple receivers. The benefit is twofold: first, with each broadcast message the server saves several unicast messages that otherwise would have to be sent individually, and second, the satisfied receivers avoid sending requests that might end up clogging the server. On the other hand, broadcast data that are useful to hardly any receivers do not yield any benefit,[1] but instead harm overall performance by occupying valuable bandwidth. This implies that broadcasting is effective when there is significant commonality of reference among the client population. Ideally, we would like to detect and exploit that commonality.

Consider, for example, a data set of M items and assume that they get requested according to the skewed access pattern of Figure 4.2. For clarity, we assume that items are sorted according to their respective request rates. From the discussion so far, it becomes clear that we are looking for the optimal point G to

[1]Assuming there is another way to satisfy those very few receivers

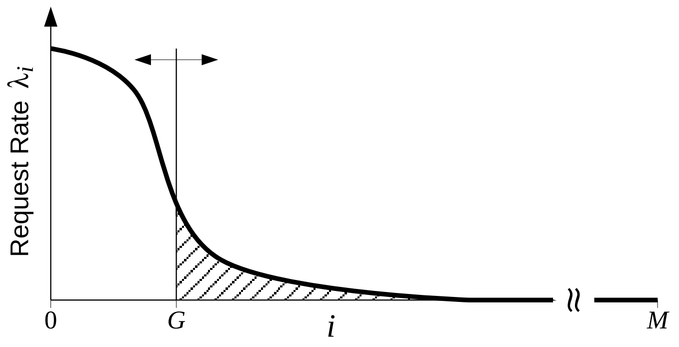


Figure 4.2: Example of a skewed data access pattern

draw the line between data that should be pushed and data that are left to be pulled. The area to the left of G (the head of the distribution) represents the volume of requests satisfied by the broadcast. The shaded area to the right of G (the tail of the distribution) represents the volume of the explicit requests directed to the server. According to the model presented in the previous section, the response time depends on the area of the tail and the width of the head (i.e., the number of broadcast items). The height of the head reflects the savings of broadcasting. Generally, the selection of G should satisfy two constraints:

1. The tail should be maintained below the pull capacity.
2. The head should be wide enough to accommodate all popular data but should not include rarely requested data.

While the first constraint is intuitive, the second deserves some clarification, as it is critical to the practicality of a hybrid solution. Consider a case where the tail is a very long area of very small, but not zero, height. That represents a large number of items that get requested very infrequently. If this area is larger than the pull capacity, we need to move the point G even more to the right. But, since each item contributes very little to the total area, the optimal G would be found deep into this tail. This means that the quality of the broadcast content would substantially deteriorate by including lots of rarely requested items, yielding unacceptably high response time, which nonetheless would be optimal according to our model. Consequently, under such workloads, slightly increased pull capacity is a more favorable solution than inordinate broadcasting.

Bearing this in mind, we consider cases where the optimal solution does not require broadcasting rarely requested data. It is assumed that the pull capacity is at least such that it can handle the aggregate load imposed by requests for such data. Under this assumption, we propose an air-caching mechanism that, in a near optimal way, exploits broadcasting to take the load of hot data off the server which is left with a tolerable load imposed by infrequently requested data.

4.2 Methodology

In this section we elaborate on the proposed methodology for managing the air-cache according to the client demands. First, we propose a dynamic classification of the available data into three groups depending on the rate they get requested. Then, we present some details on the implementation of the air-cache through repetitive broadcasting. Section 4.2.3 describes the algorithm used to actually modify the contents of the air-cache, while Section 4.2.4 presents a technique that helps the system avoid disastrous effects of erroneous decisions. Finally, in Section 4.2.5 we discuss how the server can reduce the overhead of the required bookkeeping.

4.2.1 Vapor, Liquid and Frigid Data

For each item in the database, we define a **temperature** that corresponds to its request rate λ_i. In addition, each item can be in one of three possible states:[2]

Vapor: Items deemed as heavily requested which are therefore broadcast, i.e., put in the air-cache.

Liquid: Items currently not broadcast for which the server has recently received a moderate or small number of requests, but not enough to justify broadcasting.

Frigid: Items that have not been requested for a while and their temperature λ_i has practically dropped to zero.

In the proposed adaptive scheme, the server dynamically determines the state of the database items, relying on air-cache misses. These can be considered as the "sparks" that regulate the temperature and state of the data. Specifically:

- Vapor data are retrieved from the air-cache, and the server does not get any feedback about their actual temperature. As they are not heated by requests, they gradually cool down and eventually turn into liquid. The duration of the cooling process depends on the temperature that initially turned them into vapor.

- Liquid data items that continue being requested either turn into vapor or remain liquid, depending on the intensity of the requests. If they stop being requested they eventually freeze.

[2]For a more intuitive presentation, we borrow terminology from the analogy to the physical states of water

- Frigid data items that start being requested turn into liquid or even vapor, again depending on the intensity of the requests. Obviously, as long as they get no requests they remain frigid.

The hardest part of this process is distinguishing vapor from liquid data, and this is the focus of this chapter. The distinction between liquid and frigid data items is the same to that achieved by a buffer manager of a database system using a frequency-based replacement policy [RD90, OOW93]. Likewise, the server should maintain liquid items in main memory anticipating new requests in the near future, and can retrieve frigid items from secondary memory only when necessary. In practice, the distinction of frigid data plays an important role in terms of overhead, especially in the case where frigid data make up the largest part of the database. With a default zero temperature, the server is off-loaded from tracking their demand statistics, and can also safely ignore them when looking for candidate vapor items.

4.2.2 Air-Cache Implementation

As we have already discussed, the effect of caching on the air is realized through repetitive broadcasting. In Section 3.3.4 we described the technique used to implement a simple (flat) air-cache. In this section, we discuss how this basic technique is actually used in this case. Basically, there is a queue Q that stores all air-cached data, i.e., all vapor data. The server picks the next item to broadcast from the head of Q. After an item gets broadcast, it is removed from the head and gets appended back to tail of Q. At the same time, in order to reflect the cooling process of vapor data, its temperature is multiplied by a predetermined $\mathit{CoolingFactor} \in (0, 1)$.

The contents of Q are modified once every cycle, the end of which is identified by a vapor item specially assigned as a placeholder. Once this placeholder is broadcast, the server re-evaluates the state of data and updates the queue accordingly. In this adaptation process, described in detail in the next section, it pinpoints vapor items that should be demoted to liquid, and liquid items that need to be promoted to vapor. Vapor items selected for demotion are marked, so that after their next broadcast they will be removed from the queue. New vapor items are placed on the tail of queue. Finally, the (new) item on the tail of Q is assigned as the next placeholder. The result is a repetitive broadcast scheme with evolving size and content.

An integral part of the hybrid delivery scheme is the indexing of the air-cache. Since clients are expected to select between the two data delivery paths, the server needs to make them aware of items forthcoming in the broadcast channel. Here, we have adopted a simple technique that uses the signature of Q (i.e., the list of data identifiers in the queue) as an index that is broadcast interleaved with the data. The clients examine the index and decide whether to wait for the required item to arrive or to make an explicit request for it. The broadcast frequency of the index can be adjusted to trade overhead for the maximum time clients are willing to wait before making the decision. Note that, depending on the size and the number of vapor items, it is possible that this simple indexing scheme will yield considerable overhead. For such cases, more elaborate indexing schemes could be used, such as bit-vectors or a variation of those proposed in [IVB94a] and [IVB94b].

4.2.3 Adaptation Based on Marginal Gains

In this section, we present the algorithm that adapts the contents of the broadcast. As we already mentioned, in the adaptation phase, the server needs to make two kinds of decisions: which of the vapor data have cooled down enough to be demoted to liquid, and which of the liquid data have become hot enough to be promoted to vapor. A straightforward approach of establishing absolute temperature thresholds cannot be applied because the state of an item depends also on the aggregate workload, i.e., the relative temperature of the other items. To account for that, we have developed an algorithm that makes these decisions based on the expected marginal gain of each possible action.

Let us first present how the expected marginal gain is computed when considering an item i for promotion to vapor state or demotion to liquid. Note that this is computed similarly in both cases, except for the sign of the involved quantities. Therefore, to avoid duplication in the presentation, we use the variable A which takes the value -1 if the item i is vapor and considered for demotion to liquid, and $+1$ if it is liquid and considered for promotion to vapor. The computations are based on the model described in Section 4.1.1. The only difference is that now we also take into account the overhead of broadcasting the index. The additional variables used here are the aggregate request rate for liquid data Λ_L, the aggregate request rate for vapor data Λ_V, the number of vapor items N, and the size of each index entry s_{I}. The expected overall marginal gain dT is given by the weighted average of the marginal gains dT_{push} and dT_{pull}. If we define $d\Lambda_V = A\,\lambda_i$ these are:

$$dT_{push} = A\,\frac{1 + (2N + A)\,s_{\mathrm{I}}}{2}$$

$$dT_{pull} = T_{pull}\frac{d\Lambda_V}{\mu - \Lambda_L + d\Lambda_V}$$

Figure 4.3 depicts these computations graphically. Ideally, the system should try to reach and operate at the minimum point of the curve T. However, it turns

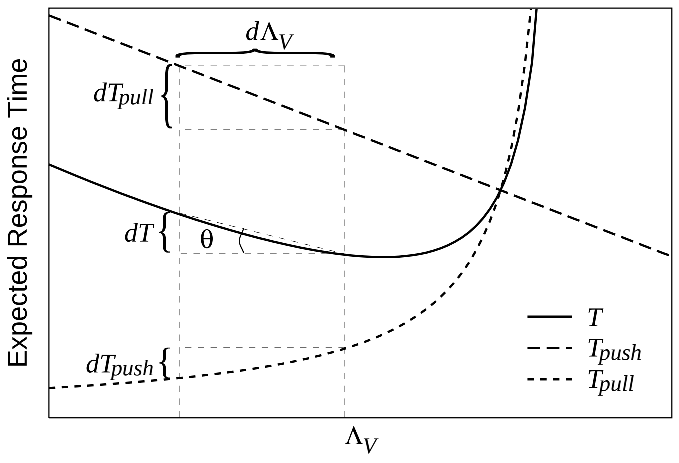


Figure 4.3: Graphical representation of marginal gains

out that in practice this is not the best thing to do. This is explained by the fact that to the right of this minimum point the response time grows very fast. As a result, under a dynamic workload it is very probable that even a small change can have a very bad effect on the system. Therefore, operating at or too close to the minimum can make the system very unstable. This was indeed verified by our experiments. Instead, we have to force the system to operate in a suboptimal area to the left of the minimum, safely avoiding instability. We achieve this by establishing some small (but not zero) threshold θ_0 for the angle $\theta = \tan^{-1} \dfrac{dT}{d\lambda_V}$.

The actual algorithm that updates the contents of the air-cache consists of three simple steps: First, it demotes to liquid all vapor data with temperature lower than the hottest liquid item. Then, using the respective marginal gains, it continues demoting vapor items in increasing order of temperatures while $\theta > \theta_0$. Last, it takes the opposite direction, and as long as $\theta < \theta_0$, it promotes liquid data to vapor in decreasing order of temperature. Note that if at least one vapor item is demoted in the second step, then no liquid item will be promoted in the third

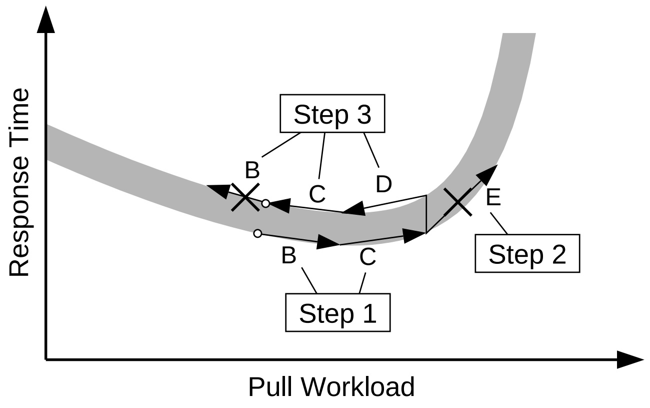


Figure 4.4: Example execution of adaptation algorithm

step. Also, it is possible that vapor items that get demoted in the first step will be re-promoted in the third. If data items are sorted by their temperatures, the complexity of this algorithm is in the order of the number of items that change state.

Figure 4.4 presents an example of how the algorithm works. We assume that initially items B, C, and E are vapor, item D is liquid, and that $\lambda_B \leq \lambda_C \leq \lambda_D \leq \lambda_E$. In this case, the algorithm firsts demotes items B and C since their temperature is lower than that of the liquid item D. Then, it checks whether item E should be also demoted. It computes the effects of this demotion, and decides not to demote it as that would hurt performance. So, it skips the second step. At the third step it promotes three items D and C (C was demoted in the first step), and stops before promoting B, having decided that no more promotions are necessary.

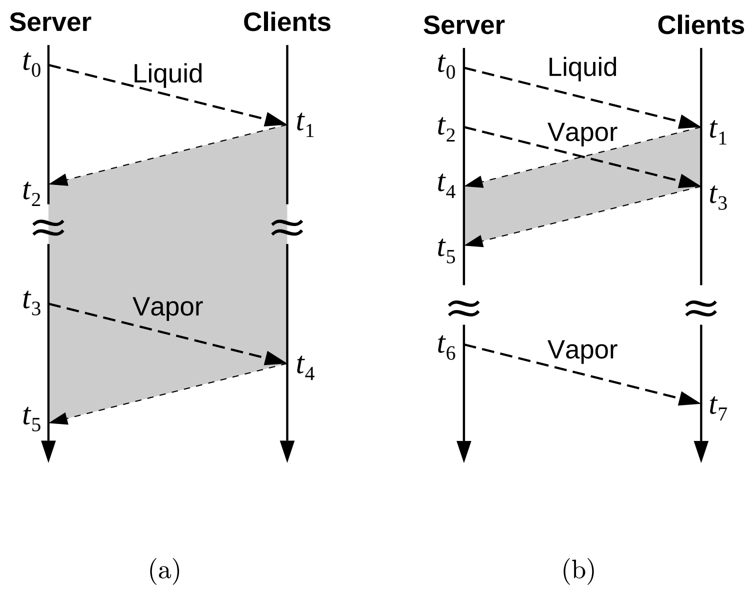


Figure 4.5: Examples of demotions without and with probing

4.2.4 Temperature Probing

A potential weakness in what has been described so far is the artificial cooling of vapor data. It was introduced for the sole purpose of giving the server a chance to re-evaluate the temperature of vapor data regularly, Thus, it is not expected to reflect the actual evolution of data demand, and may very well result in a situation where a very hot item is demoted to liquid. Should that happen, the server would be swamped with hundreds or thousands of requests for that item. Although the adaptive algorithm will eventually correct this by re-promoting the item, the reaction time lag may be big enough to cause serious performance degradation.

This is better explained in Figure 4.5(a) where we present the time line of events after a decision to demote a hot vapor item at time t_0. This decision is reflected in the next broadcast of the index that reaches the clients at t_1. From that point on, all the requests for that item are directed to the server. If the item is still hot, the server decides to re-promote it to vapor at t_3, and includes it at the next index broadcast, received by the clients at t_4. But, considering data transmission and server inertia delays (i.e., the time to re-promote the item), the interval between t_1 and t_4 could be substantial. The shaded area in the figure represents the total request load that this wrong decision may generate. The cumulative penalty of consecutive improper demotions can be heavy enough to make the system practically unusable.

This section introduces **temperature probing** as a way of preventing any disastrous effects by premature demotions of vapor data. The algorithm that we propose remedies potential errors by a "double clutch" approach, which is illustrated in Figure 4.5(b). Soon after the decision to convert an item from vapor to liquid at t_0, and before it is actually heated up by misses, the item is re-promoted at time t_2. This creates a controllably small time window (from t_1 to t_3) that limits the expected number of client requests for the demoted item, but still can provide the server with concrete information about the actual demand. In effect, through a small number of misses, we give the server the opportunity to probe for the actual temperature of the data, before committing to its decision. After the re-promotion of the item at t_2, the server waits for requests generated during the period $[t_1, t_3]$ in order to re-evaluate the item's actual temperature. Considering the time required for client messages to reach the server, we delay this re-evaluation at least until t_5. Finally, depending on the result of the probing, the item is either demoted or reinstated to the broadcast queue with corrected temperature at t_6.

A critical factor for this double-clutch approach is the probing interval $[t_0, t_2]$. If it is too short, hardly any requests will be generated to help the server in the re-evaluation. If it is too long, it essentially defeats its purpose. Therefore, it should be selected very carefully, and should preferably be dynamically adjusted to the intensity of the workload. For these reasons, we found that a very good selection can be based on the average request rate of vapor data. More specifically, we set the probing time to be $\mathit{ProbingFactor} \times \dfrac{N}{\Lambda_V}$, where $\dfrac{N}{\Lambda_V}$ is inverse of the average temperature of vapor data. Essentially, with this demand-adjusted probing window, the *ProbingFactor* determines the expected number misses generated per probe, and allows the system to explicitly control the total probing overhead.

4.2.5 Monitoring Overhead

The implementation of our hybrid scheme requires some considerable bookkeeping, which may impose by itself a heavy computational load on the server. The server needs to monitor the temperature of liquid items, keep them sorted so that each time the next candidate for promotion can be identified instantly, as well as detect those that have not received any attention for a period long enough to freeze.

To keep this overhead down to a minimum, we chose to organize bookkeeping around the idea of **slotted time**, which considers time as divided into slots $D_0, D_1, D_2, \ldots$, each taking time t_s. During each slot, we record the total number of requests for every item that gets requested. We then compute the request rate of each item using a moving average over time slots exponentially weighted by a factor α. Formally, the request rate of item i at the end of slot D_j is $\lambda_{i,j} = \alpha \dfrac{r_{i,j}}{t_s} + (1-\alpha)\,\lambda_{i,j-1}$, where $r_{i,j}$ is the number of requests for item i made during D_j.

The computational benefits of this approach are two-fold. First, note that for the items that were not requested during the last slot we have $r_{i,j} = 0$, and therefore $\lambda_{i,j} = (1-\alpha)\,\lambda_{i,j-1}$. In practice, the server does not even need to update these values, since for an item last requested during D_{j-k} it holds that $\lambda_{i,j} = (1-\alpha)^k \lambda_{i,j-k}$. This way we avoid many computations that are performed only when (if ever) needed. The second benefit is that the relative order of temperatures for items not requested during the last slot does not change. This is exploited to significantly reduce the overhead of keeping a list of liquid items in decreasing order of temperatures. Only the items requested in the last slot need to be sorted according to their new computed temperatures and then quickly merged with the rest.

Last, with the time slots it is straightforward to identify when liquid items become frigid. Assume that an item freezes if it is not requested for k slots, i.e., for the last $k\,t_s$ time units. Then, at the end of slot D_j, the items that were last requested during D_{j-k} turn into frigid. As a result, each time we need to keep information only about the last k time slots.

4.3 Experiments and Results

4.3.1 Simulation Model

In order to establish the potential of hybrid data delivery and investigate the involved tradeoffs, we have built a simulation model of the proposed system. We assume that the provided information is a collection of self-identifying data items of equal size. For the results presented herein, we set the size of this collection to 10000 items. Clients generate requests for data that are satisfied either by the broadcast or by the server upon explicit request. Under this assumption, we have modeled all the client population as a single module that generates the total workload, a stream of independent requests for data items. The exact number

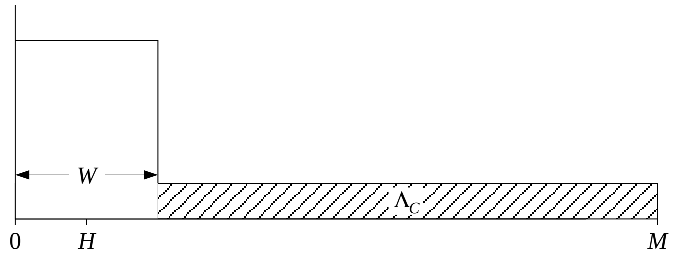


Figure 4.6: HotColdUniform distribution

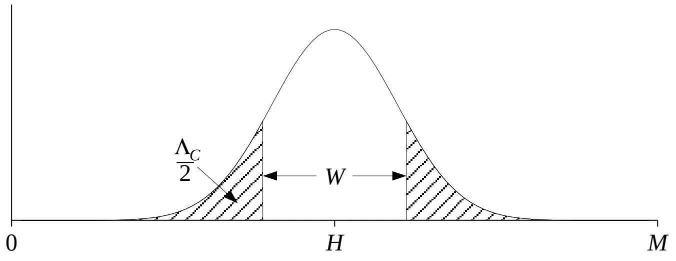


Figure 4.7: Gaussian distribution

of clients is not specified but instead it is implicitly suggested by the aggregate request rate. For the data access pattern we used two different distributions: **HotColdUniform** and **Gaussian** (Figures 4.6 and 4.7). The first one is only used as an ideal case where there is a clearly defined hot-spot in the database. The second is more realistic, but at the same time it allows explicit customization through the same four parameters: the aggregate request rate Λ, the aggregate request rate for cold data Λ_C, the width of the hot-spot in terms of data items W, and the center of the hot-spot H. In order to create the effect of dynamic workloads, the value of these parameters can vary in the course of an experiment. For example, by changing the value of H we can simulate workloads with moving hot-spots.

For the server we have used a simple data server model, enhanced with a transmitter capable of broadcasting, and the functionality required to implement our adaptive algorithm. Even though it is modeled in detail through several parameters (e.g., cache size, I/O characteristics, etc.), the presentation and interpretation of our results is based only on one parameter, the system's pull capacity μ, which corresponds to the maximum rate at which requests can be serviced. Depending on the experimental setup, this is determined by (a combination of) the processing power of the server, and the available bandwidth. For the network, since we want to capture hybrid environments, we need to specify the characteristics of three communication paths: (1) the broadcast channel, (2) the downlink from the server to the clients, and (3) the uplink from the clients to the server. For simplicity, we assume that all clients use similar but independent paths for establishing point-to-point connections with the server. Also, because of the small size and limited number of requests we do not consider the possibility of congestion in the uplink. The downlink, on the other hand, is a shared resource that is used for all server replies. Similarly to the broadcast channel, the downlink can transmit one item per time unit. Finally, assuming enough computing power at the server, this bandwidth also determines the system's pull capacity. In other words, we assume that $\mu = 1$.

4.3.2 Static Workloads

For the first set of experiments we used static workloads, even though they cannot demonstrate the system's adaptiveness. The reason is that they can provide a solid base for comparison, since for those we can easily determine the optimal behavior of a hybrid delivery system. Actually, the graphs in this section include two baselines for comparison. The first, marked "Optimal", represents the theoretically optimal performance, based on the model of Section 4.1.1. For the second, marked “PerfectServer”, we used a stripped version of our server that does not adapt, but instead, broadcasts periodically the optimal set of data, obtained through exhaustive search. For static workloads, the line “PerfectServer” is the ultimate performance goal of our system.

In order to test the behavior of the system in different scales, we vary the volume of the total workload from light ($\Lambda < \mu$) to very heavy ($\Lambda = 100\,\mu$). We intend to demonstrate that, under the assumptions discussed in Section 4.1.2, our approach performs close to the optimal, and exhibits very high scalability. The main performance property of this system is that, contrary to pull/unicast systems, the expected response time does not directly depend on the intensity of the workload. Instead, it is determined by the size of the hot-spot, i.e., the amount of frequently requested data. In other words, it depends on the workload only as a function of the data access distribution. This important property can yield a significant performance advantage, especially under highly skewed distributions exhibiting small hot-spots.

First, we present the results we obtained under the ideal HotColdUniform workload distribution. In order to highlight the above mentioned property, the size of the hot-spot W is constant (100 items) for all values of Λ. Practically, this means that the popularity of items outside the hot-spot of the 100 items remains low ($\Lambda_C < \mu$) for any scale. In Figure 4.8, we show the average response time as a function of the request rate Λ. In this graph, we include the performance of the pure pull system, which, as expected, cannot accommodate workloads higher than its capacity (1 request per time unit). Contrast this to the performance of the hybrid system (curve marked “Adaptive”). Under very light loads ($\Lambda < \mu = 1$)),

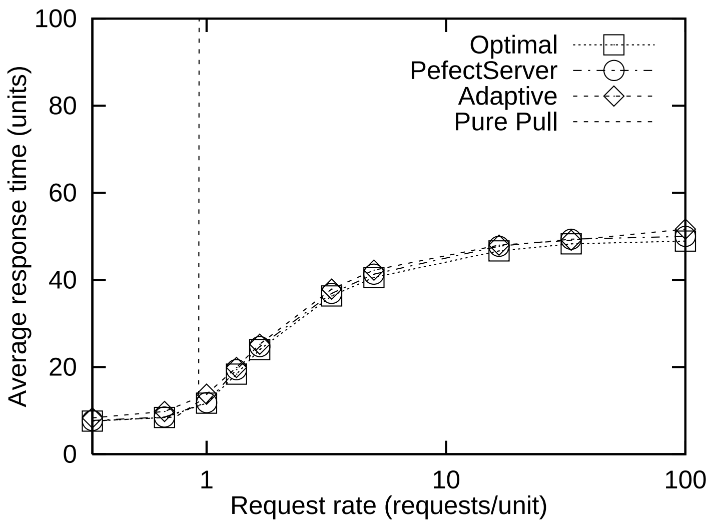


Figure 4.8: Experiment 1: HotColdUniform distribution

the system does not broadcast any item and performs like a pull only system since it can efficiently handle all the client requests. But, as the request rate increases beyond the pull capacity ($\Lambda > 1$), the server starts air-caching some of the popular data to accommodate the additional demand. At the same time, more and more of the requests become air-cache hits, and the average response time is dominated by the performance of the hits. As a result, the overall performance of the system is determined by the number of air-cached items, i.e., by the number of frequently requested items. The load incurred by air-cache misses is maintained below the pull capacity, consistently yielding fast responses. Therefore, the response time increases with the average number of air-cached items. This increase is noticeable until the workload reaches about 10 requests per unit. At this point almost all 100 items of the hot-spot are air-cached, and the average response time is 50 units, i.e., half the time it takes to broadcast 100 items. For even heavier workloads, the response time practically remains constant. Note that the horizontal axis is

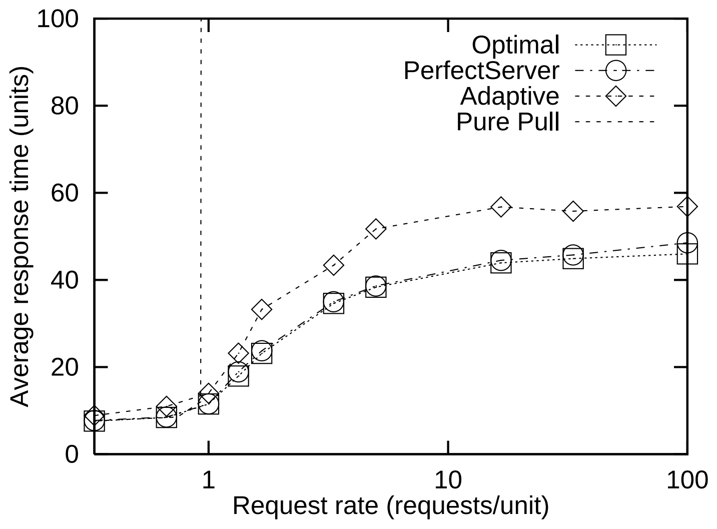


Figure 4.9: Experiment 2: Gauss distribution, fixed size hot-spot

in logarithmic scale and extends up to 100 times the pull capacity of the system. This shows that, under skewed workloads, the hybrid delivery approach can efficiently scale to workloads many times its nominal capacity. Finally, observe that under this ideal separation of hot and cold data, our approach performs optimally, matching both the theoretically minimum response time and that of the perfect server.

Next, in order to test our system under more realistic workloads, where the boundaries of the hot-spot are not clearly defined, we performed a set of experiments using the Gaussian distribution. All the system parameters are the same as in the previous case. However, here we present the results obtained for two variations of the workload. In the first variation, the number of popular items remains constant throughout the scale of the experiment so that the results are comparable to the previous experiment. This means that as the workload increases so does the skewness of the distribution. This is easily achieved with a proper decrease of

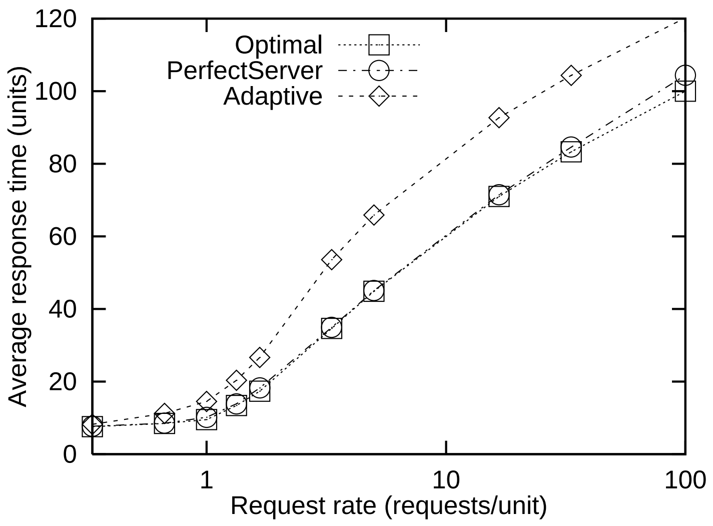


Figure 4.10: Experiment 3: Gauss distribution, fixed standard deviation

the distribution's standard deviation. In the second variation, the standard deviation does not change. Therefore, when the intensity of the workload increases the popularity of all the data increases with it.

Figure 4.9 shows the performance results for the first variation (fixed number of popular items). The results and conclusions are similar to the previous experiment. Again, the hybrid system can efficiently accommodate workloads at least 100 times the pull capacity of the system. However, compared to the HotColdUniform distribution, there is one small difference. This time there is a discrepancy between the performance of our system ("Adaptive" curve) and the optimal. The reason is that our system selected to air-cache, on the average, a few more items over what both the theoretical model and the "PerfectServer" suggest as optimal. This is an artifact of the threshold θ_0 (see Section 4.2.3) which urges the adaptive algorithm to slightly favor broadcasting. Contrary to the previous case and because of the continuous distribution, the algorithm now detects outside the optimal vapor set items hot enough to be considered for promotion.

Figure 4.10 presents the results for the second variation. In this case, the skewness of the distribution does not change; when the workload increases, the popularity of all items increases. This means that more and more items need to be air-cached so that the miss workload is kept below the pull capacity. This is evident from the two baselines in the graph ("Optimal" and "PerfectServer"), where we can see that the optimal vapor set—and consequently the optimal response time—is indeed growing with Λ. Nevertheless, even in this case our system scales very well, in the sense that it manages to follow the optimal performance very closely.

4.3.3 Tuning Parameters

In Section 4.2, we introduced three tuning parameters, namely θ_0, *CoolingFactor*, and *ProbingFactor*. While the first is used just to keep the system at a safe distance away from instability, the other two are essentially the knobs that control its adaptiveness and overhead. Here, we concentrate on the effects of the latter two parameters. For θ_0, we have established from previous experiments that a good a selection is such that $\frac{dT}{d\Lambda_V} = \tan\theta_0 \geq 0.1$ [SRB97b].

Temperature probing was introduced to prevent the detrimental consequences of early demotions of vapor items. But, the probing window needs to carefully selected; if it is either too small or too big, it is essentially the same as no probing at all. In Section 4.2.4, we defined the probing window to be dynamically adjusted by the *ProbingFactor* and the average request rate for vapor items. This way, we directly control the number of expected misses per probe, i.e., for a *ProbingFactor*=4 we get an average of 4 requests per probe. The *CoolingFactor* (Section 4.2.1) is also closely related to probing, and must be carefully selected as well. A small value

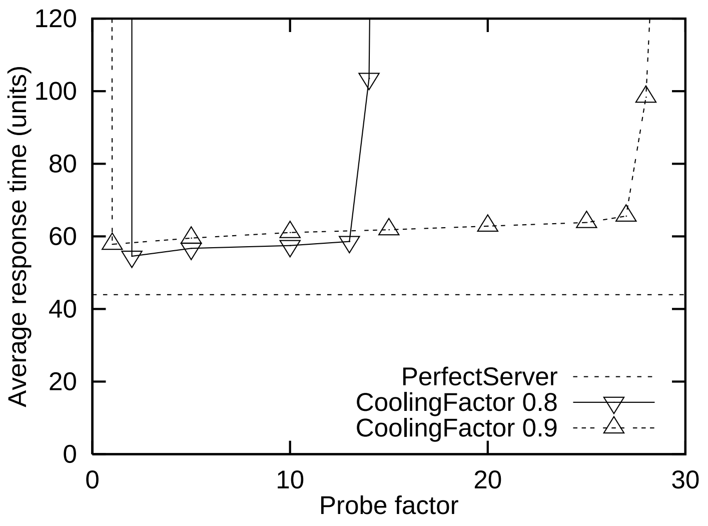


Figure 4.11: Experiment 4: Effects of probing parameters

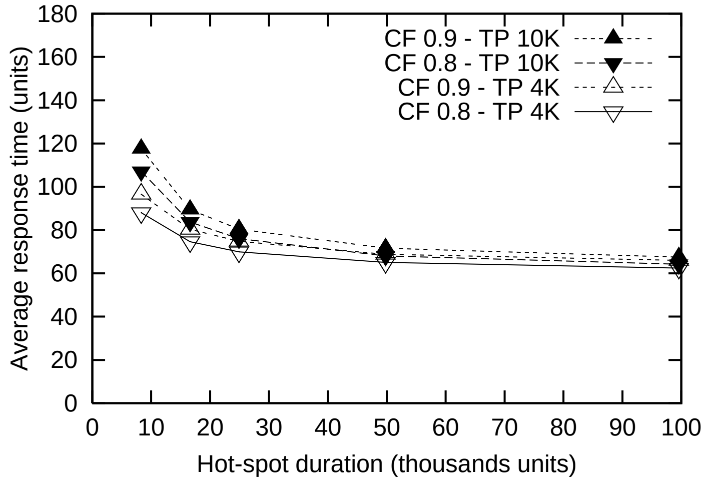
causes the temperature of vapor data to drop quickly, yielding frequent probing, and thus, high overhead in terms of probed misses. But, on the positive side, a small value also allows the system to adapt faster to changes in the demand. Large values have the opposite effect; they cause less probing but hinder the adaptiveness of the system.

Figure 4.11 shows how the *ProbingFactor* affects the system's performance, for two different values of the *CoolingFactor*. For this experiment, and the rest of the experiments presented hereafter, we used the Gaussian access pattern with Λ=20 requests per unit, and W=100 items. The first thing we note is that, without probing (*ProbingFactor*=0), the system cannot recover from the incorrect demotions, and the response time grows arbitrarily large. But, even a very small number of probed misses (≥ 2) are sufficient to correct the temperatures of vapor data, thus allowing the system to operate close to the optimal. As the *ProbingFactor* increases further, so does the volume of the probed misses. The rate at which this happens depends on the frequency of the probing (i.e., the *CoolingFactor*) and the number of items being probed (i.e., the number of vapor items). Beyond some point, the overhead of probed misses becomes too big for the server to handle, leading again to very slow responses. In other words, with a very large *ProbingFactor*, probing causes the problem that it was supposed to solve in the first place. Naturally, this happens earlier when probing is more frequent (*CoolingFactor*=0.8).

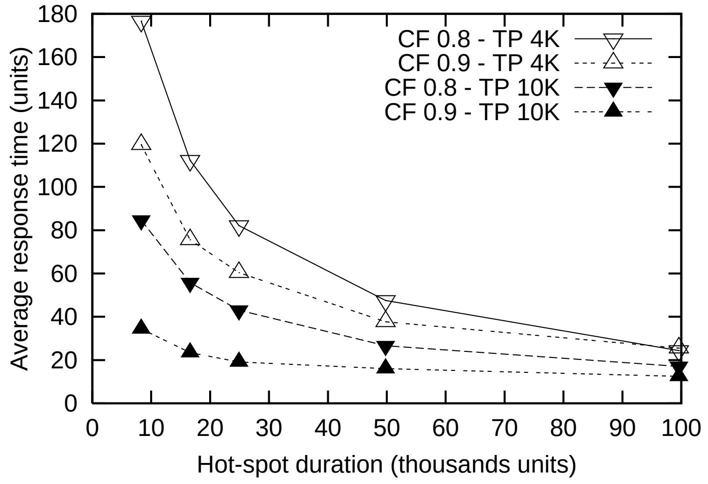
4.3.4 Dynamic Workloads

For the last set of experiments, we used dynamic workloads in order to evaluate the adaptiveness of our system in cases when the focus of the clients changes. Such a change was modeled as an elimination of a hot-spot and a generation of a new one in another (randomly selected) part of the database. This process was not instant, but instead it was taking a transient period of *TP* units to complete. Every new hot-spot persisted for *Duration* units. For easier interpretation of the results, all the hot-spots were similar, and the total workload remained constant (Gaussian access pattern with Λ=20 requests per unit, and W=100 items).

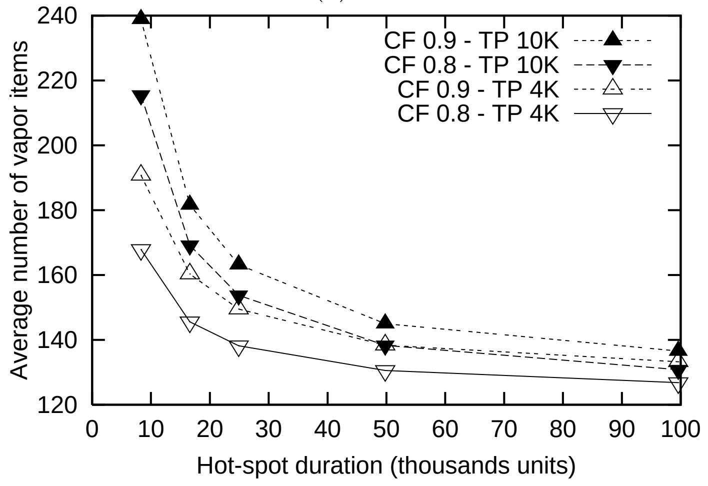
In Figure 4.12, we present the obtained results as a function of *Duration*. The workload in these graphs is more dynamic on the left side, since with smaller *Duration* changes occur more often. We used two different values of *TP* for comparing fast (white marks, *TP*=4000 units) and more gradual (black marks, *TP*=10000 units) changes. In addition, we give results for two values of the *CoolingFactor* (*CF*=0.9 and *CF*=0.8) which determines the adaptation speed of the system. For better comprehension of the results, we plot the total average response time (Figure 4.12(a)), the average response time for pulled data (Figure 4.12(b)), and the average number of vapor items (Figure 4.12(c)). For all these experiments, we used



(a)



(b)



(c)

Figure 4.12: Experiment 5: Dynamic workloads

ProbingFactor=5 and $\tan\theta_0 = 0.1$.

The most significant observation is that the system adapts very well to changing access patterns (Figure 4.12(a)). Even on the left side where changes occur very frequently, the response time remains small. In most cases, performance lies within 30 units of that achieved under the static workload (Figure 4.9). This means that the server is very effective in detecting shifts in the clients demand, and thus can react promptly. As expected, the system adapts and performs better with a smaller *CoolingFactor*. But, an unexpected result shown in Figure 4.12(a) is that the system appears to perform better under more abrupt changes (*TP*=4000). However, this will be justified in the following where we discuss how the system is affected by dynamic workloads.

Changing hot-spots impact the performance of both the pull (Figure 4.12(b)) and the push (Figure 4.12(c)) part of the system. First, an item that suddenly becomes hot can generate a large number of requests before the server is able to react and append it to the air-cache. The cumulative effect of these requests may cause significant build-up in the server's input queue, and therefore increase the average response time for pulled data. This build-up is worse when the changes are faster and more frequent. Indeed, in Figure 4.12(b) we see that the average pull response time increases when the changes occur more often (left side) and when new hot-spots are heating up faster (white marks). Second, in transient periods the server actually perceives two hot-spots, the old and the new. Thus, in order to meet the demand during those periods, it has to expand the vapor set to include them both. This explains why in Figure 4.12(c) the average number of vapor items increases as the *Duration* decreases. With decreasing *Duration*, the transient periods make up more and more of the total time. Consequently, the server appears to be broadcasting, on the average, more data. Note that for *Duration*=*TP*=10000 the workload is continuously in transient state and the server almost always detects two hot-spots. Therefore, the size of vapor set is close to double that of the static case. We also observe that this phenomenon is worse with longer transient periods (black marks) as the server spends more time broadcasting both hot-spots. Since, in these experiments, the average response time is dominated ($\approx$90%) by broadcast accesses, this also explains why the system appears to perform better under more abrupt changes (*TP*=4000).

Finally, here we can also notice the effects of the *CoolingFactor* to the adaptiveness of the system. On one hand, the smaller value (*CF*=0.8) harms the pull response time since it causes more frequent probing and, thus, more misses (Figure 4.12(b)). But, on the other hand, it limits unnecessary broadcasting and reduces the "double hot-spot" phenomenon since it allows the server to detect faster loss of interest for vapor data (Figure 4.12(c)). Consequently, the *CoolingFactor* should be selected as small as it causes tolerable probing overhead. Note that the probing overhead can be estimated (and controlled) by the *CoolingFactor*, the *ProbingFactor*, and the number of vapor items. Also, it is even possible to employ a self-tuning strategy for the system. In other words, the system can monitor the workload behavior and use the outcome of its previous actions to learn how it should be operating more efficiently. As an example, if after a series of probings the outcome is always the same, it may be good idea to increase the *CoolingFactor* and sample less frequently. Overall, one of the strongest features of this approach is that, with a proper combination of two parameters, we can explicitly control fairly accurately the adaptiveness of the system, the effectiveness of the probing, and the incurred overhead.

4.4 Conclusions

In this chapter, we described how adaptive hybrid data delivery can be address the ever increasing demands for on-line information access. We proposed to use the air-cache mechanism to disseminate data from heavily accessed data sources. This mechanism takes advantage of the skewness of user requests towards a small (but possibly changing) subset of the available information, and combines data broadcast for massive dissemination of the popular data with upon-request individual delivery of the rest.

Initially, we analyzed the performance of a hybrid delivery scheme under skewed access patterns, and laid out the goals and tradeoffs of our approach. Then, the main problem we addressed was the identification of the data items to be air-cached, i.e., the detection of the database hot-spot. We presented an algorithm that, based on expected performance marginal gains and data temperature probing, recognizes heavily requested data and continuously adapts the air-cache contents accordingly. We showed that the database hot-spot can be accurately obtained by monitoring the air-cache misses and therefore no other implicit knowledge on the actual usage of the broadcast data is necessary. This is one of the major distinctions between the work presented here and other push/broadcast schemes, which are dependent on accurate, comprehensive, but not readily available statistics on workload access patterns.

Our simulation experiments have demonstrated both the scalability and versatility of the proposed technique. Under the assumption that the server's capacity is sufficient for servicing the demand for cold data, the proposed technique performs very close to the optimal, even under dynamic, rapidly changing workloads. An important result is that the performance of this hybrid system is not directly affected by the volume of the workload, but instead it depends on the amount of frequently requested data as defined by the data access distribution. Because of this characteristic, we believe that hybrid data delivery can be the basis of very scalable data dissemination systems.

Chapter 5

Disseminating Updates to Mobile Clients

The remarkable sales increase of portable computers and the proliferation of wireless communication technologies are strong evidence that mobile computing is becoming ever more important. Today, rapid technological advances offer laptop computers comparable to desktop workstations with significant processing power and large disk capacity. Such machines enable a wide range of applications to be carried away from the office desk. As a result, many organizations today—and many more in the future—have a portion of their workforce accessing corporate information on the road, from their home, or from other remote locations. Most of the time they operate off-line, i.e., not connected to the corporate network, relying on local data replicated from a central repository. While replication masks the effects of disconnection, it also brings about the problem of staleness and the need to refresh data regularly and efficiently. In other words, any data updates occurring at the repository must sooner or later be propagated to the mobile clients.

This model of operation is important for any corporation in which business transactions may occur outside the office. For instance, sales agents visiting potential customers need information about new products or services, new pricing policies, special offers, product availability, etc. Money managers need the latest stock and bond indexes. Realtors accompanying potential buyers need new house listings, possibly together with photographs, directions, and other related information. The growing market for these mobile applications has already been recognized by the database industry. Several products are emerging that support off-line operation, offering data replication and update propagation between a central database and "lite" DBMSs running on mobile computers (e.g., Oracle Lite, Sybase SQL Anywhere Studio).

Update propagation techniques typically rely on the transaction log, where the server records changes committed in the database [RK86, GWD94, BDD$^+$98]. Log entries are sent to the clients, where they are "replayed" to refresh the local copies. For such a refresh in the mobile environment, a client needs to reconnect to the network and receive all the updates that were committed while it was off-line. This requires reviewing the part of the log that was appended since its previous refresh, finding all relevant updates, and applying them to the local data.

In this chapter, we address the problem of propagating logged updates to large, widely deployed mobile workforces. Generally, updates exhibit very high locality of reference. All clients want the updates since they went off-line, making the recent part of the log an extremely hot spot [DR92]. This makes broadcast-based dissemination of the log very appealing in this case. Driven by this, we propose using the air-cache mechanism to disseminate the log to the clients. The server acts merely as a "pump" of updates. Each client individually takes over the task of filtering out the updates that affect its data. Client-side filtering of the updates has been shown to be preferable in large scale systems since it avoids contention at the server [DR98]. When combined with broadcasting, the benefits are multiplied since data transmission cost is amortized over many clients.

The key issue with the proposed idea is identifying what part of the log is popular enough to be air-cached. This cannot be a one time decision as the popularity of the updates is time-dependent and continuously changing. They start very hot, but as they age their popularity drops. Thus, we need an adaptive algorithm that manages the air-cache according to the clients' needs and the age of the data. The main complication we have to address is the unique property of the air-cache that it must be managed exclusively based on cache misses.

What makes the problem even more intricate in this case is that not all clients have the same habits, in terms of connecting to and disconnecting from network. Borrowing the terminology from [BI94], clients can range from **workaholics**, who stay connected most of the time, to **sleepers**, who only connect sporadically. Therefore, upon reconnection, workaholics usually need a small part of the log, while sleepers tend to require much longer parts. In order to accommodate this diversity in the clients' needs, we employ a hierarchical air-cache which provides multiple levels of data caching, each with different performance characteristics. This gives us the flexibility to air-cache the log in a way that suits different client groups.

In the rest of the chapter, we first describe the hierarchical version of the air-cache. Next, in Section 5.2, we develop a performance model for broadcasting the log through the hierarchical air-cache. This model serves as the foundation for the proposed hybrid system presented in Section 5.3. Finally, Section 5.4 contains several experimental results obtained from a detailed simulation.

5.1 Hierarchical Air-Cache

A **hierarchical air-cache** is a flexible cache structure that supports different broadcast frequencies, and therefore, different access latencies. It is a special case of a composite air-cache that adopts the "multi-disk" model of the **broadcast disks** architecture [AAFZ95] to create a memory hierarchy on the air. Generally, a hierarchical air-cache consists of C cache levels, named $AC_1, AC_2, \ldots, AC_C$. The average latency of each level AC_k is determined by the frequency f_k at which data cached in AC_k are repeated in the broadcast. More popular data are cached in the faster (lower latency) levels and are broadcast more often; less popular data are cached in the slower (higher latency) levels. Note that the repetition frequencies have only a relative meaning. This means that the value of f_k only suggests that an item cached in AC_k is being broadcast f_k/f_j times more (or less) often than an item in AC_j. Put another way, between two consecutive broadcasts of an item in AC_j there are on average f_k/f_j broadcasts of an item in AC_k. As a convention, we select AC_C to be the fastest cache and AC_1 to be the slowest, i.e., $f_C > f_{C-1} > \ldots > f_1$. These frequencies are not restricted to integer values, and since they are important only in a relative sense, we can always set them so that $f_1 = 1$.

For our purposes, we assume that data are organized into equal size pages. The size of such a page is the reference data size, and thus, a broadcast unit in this case is the time required to broadcast a page. Figure 5.1 presents an example of a 3-level hierarchical air-cache and a portion of the broadcast stream it generates. Each level AC_k is characterized by its frequency f_k and the number of data pages n_k it contains. In this example, AC_1, AC_2, and AC_3 contain 4, 6, 2 pages respectively. Furthermore, their frequencies are 1, 2, and 4. This means that pages in AC_3 get broadcast twice as often as pages in AC_2, which in turn get broadcast twice as

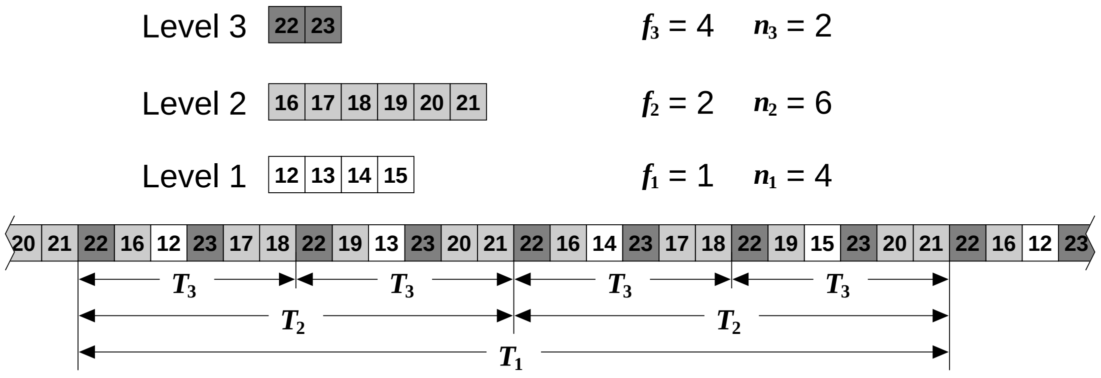


Figure 5.1: Hierarchical air-caching

often as those in AC_1.

The actual latency of AC_k is determined not only by its own size and frequency but also by the sizes and frequencies of all other levels. The period T of the air-cache is the minimum time interval during which all cached pages are broadcast at least once. Because AC_1 is the slowest level, this period is equal to the interval between two consecutive broadcasts of any one page in AC_1. On average, within each period there must be $f_1 = 1$ broadcast of each page in AC_1, f_2 broadcasts of each page in AC_2, and so forth. Thus, the period of the air-cache lasts $T = \sum_{k=1}^{C} n_k f_k$ page broadcasts, or broadcast units. Similarly, the period T_k of level AC_k is the average interval between two consecutive broadcasts of any one page in it. Since within every T there must be f_k broadcasts of the pages in AC_k, we can infer that $T_k = T/f_k$. This also means that $T_C < T_{C-1} < \ldots < T_1 = T$.

In terms of implementation, the hierarchical air-cache can be realized with the general algorithm for the generation of composite air-caches (see Section 3.3.4), given the right selection of broadcast factors ϕ_k's. Remember that if b_k is the bandwidth share for a level AC_k, and s_k is its size, then the period of that level is

$T_k = \frac{s_k}{b_k}$. Also, because each level contains n_k pages of size 1, its size is $s_k = n_k$. So, we have

$$\frac{f_k}{f_j} = \frac{T_j}{f_k} = \frac{s_j b_k}{s_k b_j} = \frac{n_j b_k}{n_k b_j} \Rightarrow \frac{b_k}{b_j} = \frac{n_k f_k}{n_j f_j}$$

which means that in order to achieve the desired effect of hierarchical air-caching, it suffices to set the bandwidth factors so that $\phi_k = f_k n_k$. Notice that these factors are not fixed; they vary with the size of each level. This, however, is not a problem for the air-cache multiplexing algorithm since changes in the broadcast factors can be accommodated on-line with no overhead.

5.2 Hierarchical Log Air-Caching

In this section, we present an analytical performance model for a system that uses a hierarchical air-cache to disseminate logged updates to mobile clients. We are restricting the model to a broadcast-only case in order to show how the structure of the air-cache affects the refresh time of the clients. Our ultimate goal is to define an optimization problem that relates the structure of the air-cache to a given log access pattern. This will be the base for the adaptive technique, presented later in the chapter.

5.2.1 Definitions

Let us consider a set of mobile clients that operate on data replicated from some data server. This server is the central site that records all updates and enforces data consistency. All committed updates are recorded in a log. This log consists of equal size pages $\ell_1, \ell_2, \ldots,$, where ℓ_1 is the oldest page, ℓ_2 the second oldest, and so forth. The subscript corresponds to the page's **log sequence number** (LSN).

Suppose that at some point the server keeps in a hierarchical air-cache log pages $\ell_c, \ell_{c+1}, \ldots, \ell_z$, with ℓ_c being the oldest page in the cache, and ℓ_z is currently the most recent page. Because the popularity of log pages decreases with their age, more recent pages are cached in higher levels, and older pages in lower levels. We put the n_C most recent pages in the highest air-cache level AC_C, the next n_{C-1} pages in AC_{C-1}, and so forth. Within each level, log pages are broadcast in decreasing order of age (i.e., older pages first). Figure 5.1 is an example of log air-caching, where pages ℓ_{12} through ℓ_{23} are cached in three levels.

A mobile client is said to be in **sleep mode** (off-line) when it is neither listening to the broadcast channel nor connected to any network. At times, it "wakes up" and comes on-line, i.e., starts monitoring to the broadcast stream and possibly connects to the network, in order to refresh its data. This requires that it retrieves all the updates that occurred while it was sleeping. In other words, it has to download all log pages created after its previous refresh.

Definition 10 *A client requires an* **(m)-refresh** *if, in order to get up to date, it needs to download all the recent log pages starting with ℓ_m. If z is the LSN of the currently most recent log page, an (m)-refresh requires retrieving pages $\ell_m, \ell_{m+1}, \ldots, \ell_z$.*

For the sake of the analysis, let us assume that all the pages required for an (m)-refresh can be found in the air-cache. If page ℓ_m is cached in level AC_k, the client needs to download some pages from level AC_k (at least ℓ_m) plus all the pages cached in the higher levels $AC_{k+1}, \ldots, AC_C$. None of the pages cached in the lower levels $AC_1, \ldots, AC_{k-1}$ is of interest since they are older than ℓ_m.

Definition 11 *For a given air-cache structure, a client refresh results in a* **(k, u)-hit** *if it requires downloading the u most recent log pages cached in* AC_k*, as well* *as all the pages cached in the higher levels* $\mathrm{AC}_{k+1}, \ldots, \mathrm{AC}_C$*. The* u *pages of* AC_k *required by a* (k,u)*-hit are called the* **useful segment** *of the hit.*

Going back to the example in Figure 5.1, a (18)-refresh results in a $(2,4)$-hit as it requires 4 pages from AC_2, and all the pages from AC_1. Pages ℓ_{18}, ℓ_{19}, ℓ_{20}, and ℓ_{21} are the hit's useful segment.

From the description of the air-cache, we know that all the pages cached in AC_k are broadcast exactly once every T_k units, in decreasing order of age. According to its definition, the useful segment of a (k,u)-hit consists of the u most recent pages in AC_k. This means that the client discerns two separate parts within any interval of T_k units: the part during which the pages of the useful segment are broadcast, and the part during which the rest pages from AC_k are broadcast (both interleaved with pages from other levels). Because the client needs u contiguous pages out of the n_k pages cached in AC_k, we can infer that the first part lasts $T_u = \dfrac{u}{n_k} T_k$ units. The second obviously lasts $T_k - T_u$ units. As we will see in the next section, this observation is critical for the performance of a (k,u)-hit.

5.2.2 Performance Model

Here we compute the time required for an (m)-refresh to be satisfied by the air-cache. Let **refresh time** R_m be the time elapsed from the moment the client wakes up and starts monitoring the air-cache until it retrieves all the pages it needs. Let x be the number of pages required for the (m)-refresh. In terms of broadcast units, R_m is the total number of pages the client scans from the broadcast until every one of the x pages it needs is broadcast at least once. Usually R_m is greater than x for two reasons: First, older pages that the client does not need are broadcast, and second some of the pages it does need may be broadcast more often. From the client's perspective, the optimal performance, i.e., minimum refresh time, is $R_m = x$.

In the following, given the structure of the air-cache, we compute the expected refresh time $\mathrm{E}[R_m]$ for an (m)-refresh. This refresh will be satisfied by means of a (k, u)-hit for the proper values of k and u. Therefore, in order to compute the desired result, we generally examine the performance of a (k, u)-hit.

The first thing to note is that we can identify lower and upper bounds for the refresh time. On one hand, since the client needs all the pages from AC_{k+1}, it will take at least T_{k+1} units, which is the minimum time to download all the pages of that level. At the other extreme, it cannot take more than T_k units since in this time all the pages it needs (and probably more) must be broadcast at least once. The actual time the client will take to download the x pages depends on the broadcast time of the useful segment relatively to the arrival time of the client, i.e., the moment it starts monitoring the broadcast. Let X be a random variable that represents the time the first broadcast of the useful segment ends after the client wakes up. With the help of X, we can compute the expected refresh time $\mathrm{E}[R_m]$. Since the useful segment is broadcast once every T_k units and a client can wake up at any moment, the variable X is uniformly distributed over the interval $[0, T_k)$. It turns out that we need to consider three cases. These are depicted in Figure 5.2 where the thick vertical line corresponds to the moment the client wakes up, and the grey box represents the useful segment.

Case a: $[0 \leq X \leq T_u]$ The client starts monitoring within a broadcast of the useful segment (Figure 5.2(a)). This means that it just missed a portion of the useful segment and it has to wait for its next broadcast. The probability of that

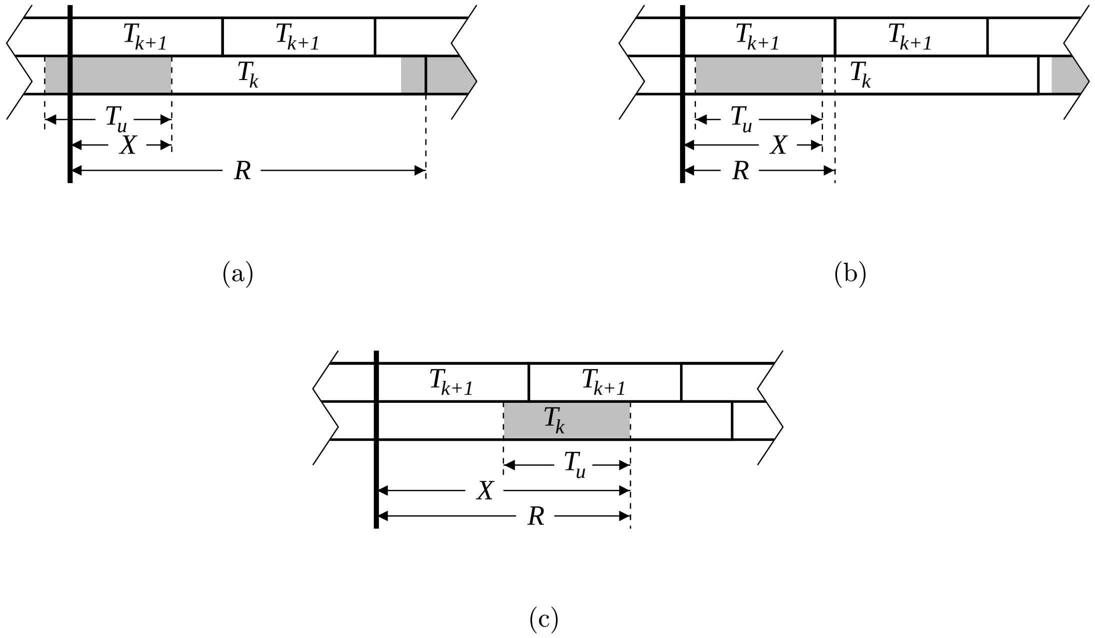


Figure 5.2: Effect of the useful segment on performance

happening is $p_1 = \Pr[0 \leq X \leq T_u] = \dfrac{T_u}{T_k}$. Because the client has to wait until the next broadcast of the useful segment, the refresh time will be equal to a full period of AC_k, i.e., $R_m = T_k$. $\square$

Case b: $[X > T_u$ **and** $X \leq T_{k+1}]$ The client starts listening outside the useful segment which, however, completes in time less than T_{k+1} (Figure 5.2(b)). Note, this case is possible only if $T_u < T_{k+1}$. Also, this case is not possible either if $k = C$, simply because there is no level AC_{C+1} (by convention $T_{C+1} = 0$ and $f_{C+1} = \infty$). With this in mind, the probability of the second case occurring is $p_2 = \Pr[T_u < X \leq T_{k+1}] = \max\left(0, \dfrac{T_{k+1} - T_u}{T_k}\right)$. Here, the pages cached in AC_{k+1} delay more than the useful segment of AC_k, and therefore the refresh time is equal to the period of level $k+1$, i.e., $R_m = T_{k+1}$. $\square$

Case c: $[X > T_u$ **and** $T_{k+1} < X < T_k]$ For the last case, the client wakes up outside the useful segment, which completes after level $k+1$ (Figure 5.2(c)). The probability of this third case occurring depends on the relative sizes of T_u and T_{k+1}. More specifically, $p_3 = \Pr[\max\{T_u, T_{k+1}\} < X < T_k] = \dfrac{T_k - \max\{T_u, T_{k+1}\}}{T_k}$. Now, the refresh time is determined by the end of the useful segment. Therefore $R_m = X$, where X is uniformly distributed over $(\max\{T_u, T_{k+1}\}, T_k)$. □

From the above model, we can compute that the expected refresh time of an (m)-refresh is $\mathrm{E}[R_m] = T_k\,\Phi(k,u)$, where

$$\Phi(k,u) = \begin{cases} \dfrac{1}{2} + \dfrac{u}{n_k} - \dfrac{2u}{n_k}\dfrac{f_k}{f_{k+1}} + \left(\dfrac{f_k}{f_{k+1}}\right)^2 & \text{if} \quad u < \dfrac{n_k f_k}{f_{k+1}} \\ \dfrac{1}{2} + \dfrac{u}{n_k} - \left(\dfrac{u}{n_k}\right)^2 & \text{if} \quad u \geq \dfrac{n_k f_k}{f_{k+1}} \end{cases} \tag{5.1}$$

5.2.3 Cache Optimization

In the previous section we computed the expected time for an (m)-refresh to be satisfied, given the structure of the air-cache. The air-cache, however, is created to serve a large number of clients with different needs in terms of number of log pages. Hence, in order to optimize it for the whole client population we need a performance metric that normalizes over the size of clients' demands. A natural choice for such a metric is the **refresh factor** $F_m = \dfrac{R_m}{x}$. This intuitive metric gives the number of pages a client examines for every page it actually needs. More important, it gives a better indication as to how good the air-cache is for any client irrespectively of its disconnection time and the volume of updates it needs. Obviously, in the best case $F_m = 1$, which means that a client examines only the pages it needs, no more than once each.

Using this metric we can now formulate the air-cache optimization problem.

Our goal is to structure the air-cache in a way that minimizes the expected refresh factor over all clients. The inputs to the problem are the range of pages to be cached (ℓ_c to ℓ_z), the maximum number of cache levels to be created C, and the log access pattern. The last is expressed in terms of the a **probability vector** $\boldsymbol{P} = (p_c, \ldots, p_z)$, where p_m is the probability that a reconnecting client needs a (m)-refresh. Formally, we have to solve the following optimization problem:

$$\begin{array}{rc}
\text{Given} & c, z, C, \boldsymbol{P} \\
\text{find} & f_2, \ldots, f_C, \text{ and } n_1, n_2, \ldots, n_C \\
\text{that minimize} & \sum_{i=c}^{z} p_i \, \mathrm{E}\left[F_i\right] \\
\text{under the constraints} & f_C > f_{C-1} > \ldots > f_1 = 1, \\
 & n_1 \geq 1, \\
 & n_k \geq 0 \text{ for } 2 \leq k \leq C, \\
\text{and} & \sum_{k=1}^{C} n_k = z - c + 1
\end{array} \tag{5.2}$$

The expected refresh factor $\mathrm{E}\left[F_m\right]$ used in the objective function can be computed from Equation 5.1 since $\mathrm{E}\left[F_m\right] = \dfrac{\mathrm{E}\left[R_m\right]}{z - m + 1}$. Notice that in the formulation of the problem, we do not allow AC_1 to be empty ($n_1 \geq 1$). The reason behind this constraint is that for any optimal solution that AC_1 is empty we can get an equivalent solution where it is not empty, by removing all lower levels that are empty, and properly adjusting the relative frequencies. Finally, in practice, we also need to decide what range of the log should be air-cached, i.e., determine the parameter c. However, this is an orthogonal problem that we address in the next section.

5.3 Hybrid Log Dissemination

In the previous section, developed a performance model for hierarchical air-caching of logged updates. Here, we propose a hybrid system that builds on this model to efficiently disseminate updates to large populations of mobile/disconnecting clients. The term "hybrid" reflects the fact that we use a broadcast channel to air-cache some recent part of the log, but also allow clients to directly connect to the server, and pull data in case of air-cache misses.

Basically, the proposed system has three major objectives: efficiency, scalability, and adaptiveness. In our context, efficiency translates to small refresh factors for reconnecting clients. When serving many clients with different needs, this calls for a solution to the abovementioned optimization problem. Scalability requires that the system performs equally well for a very large number of clients. As it was demonstrated in our previous work, such a hybrid system achieves scalability with a careful balance of broadcast and unicast data delivery. On one hand, the goal is to air-cache enough data to serve the bulk of clients needs, and let the server handle only a tolerable volume of cache misses. This prevents the server from becoming a performance bottleneck. On the other hand, we do not want to cache more than we have to, since that would unnecessarily increase the broadcast size. Last, adaptiveness requires that the system is efficient and scalable under any (possibly changing) workload. This requirement emphasizes the pivotal role of air-cache misses. As they are the only indication of the clients' activity, the server relies on them to assess the system's workload, and adapt accordingly.

The overall system architecture is shown in Figure 5.3. Clients can connect and disconnect at any time. A reconnecting client first tunes into the broadcast channel to determine how many log pages have been created since its last connection, and

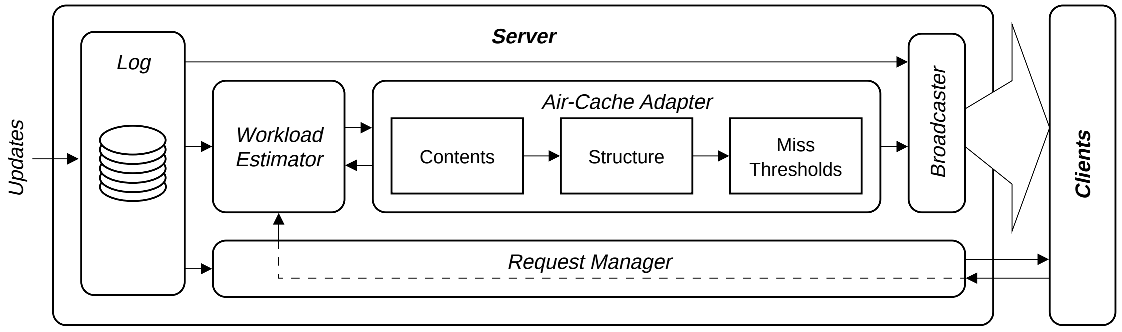


Figure 5.3: System overview

whether it should get them all from the air-cache or not. If yes, it does not contact the server; otherwise it sends a request for one or more log pages.

The server consists of five modules:

1. The **Log** records all the data updates, grouped in equal size pages. When new log pages are created, it notifies the other modules as necessary.

2. The **Broadcaster** creates the air-cache by broadcasting log pages in the proper sequence.

3. The **Request Manager** handles client requests, and collects the necessary statistics on the misses.

4. The **Workload Estimator** uses the miss statistics to assess the clients' activity and log access pattern.

5. The **Air-cache Adapter** controls the contents and structure of the air-cache, based on the output of the workload estimator.

Next, we present the key components of this hybrid system. First, we introduce a new twist to the idea of air-cache misses, then we show how the estimator can estimate the workload from those misses, and finally, describe how the adapter dynamically modifies the air-cache.

5.3.1 Soft Cache Misses

Naturally, a client is expected to generate a cache miss when (some of) the updates it requires are not air-cached, i.e., when it needs more log pages than it can get from the air-cache. In this case, it will get all the pages it can from the broadcast, and request the remaining from the server.[1] Note that the **size** of the miss, i.e., the number of pages the client requests from the server, is variable.

We can, however, relax the notion of a cache miss and, sometimes, allow clients to generate misses even for air-cached data. The rationale behind this idea is that such misses may yield significant savings in terms of the refresh time. Consider, for example, the scenario in presented in Figure 5.4(a). This particular refresh translates into a (k, u)-hit with a small useful segment T_u (grey box) that will start being broadcast S units later. As we have seen, the refresh time R for the client is determined by the end of the useful segment. For the first T_{k+1} units after it wakes up, the client downloads pages from $AC_{k+1}, \ldots, AC_C$. After that, it has to wait for another D units until it can download the useful segment; no other page that it needs or it does not already have is broadcast in that period. Therefore, within the refresh time R there is a lengthy "dead interval" D which the client spends just waiting. It is not hard to see that if it did not have to wait for the delayed useful segment, the refresh time R' would be only T_{k+1}. But of course,

[1]Alternatively, for big requests, the client could scrap its local replica and rebuild it from scratch. Although not considered in this study, such an option could be easily supported by the system.

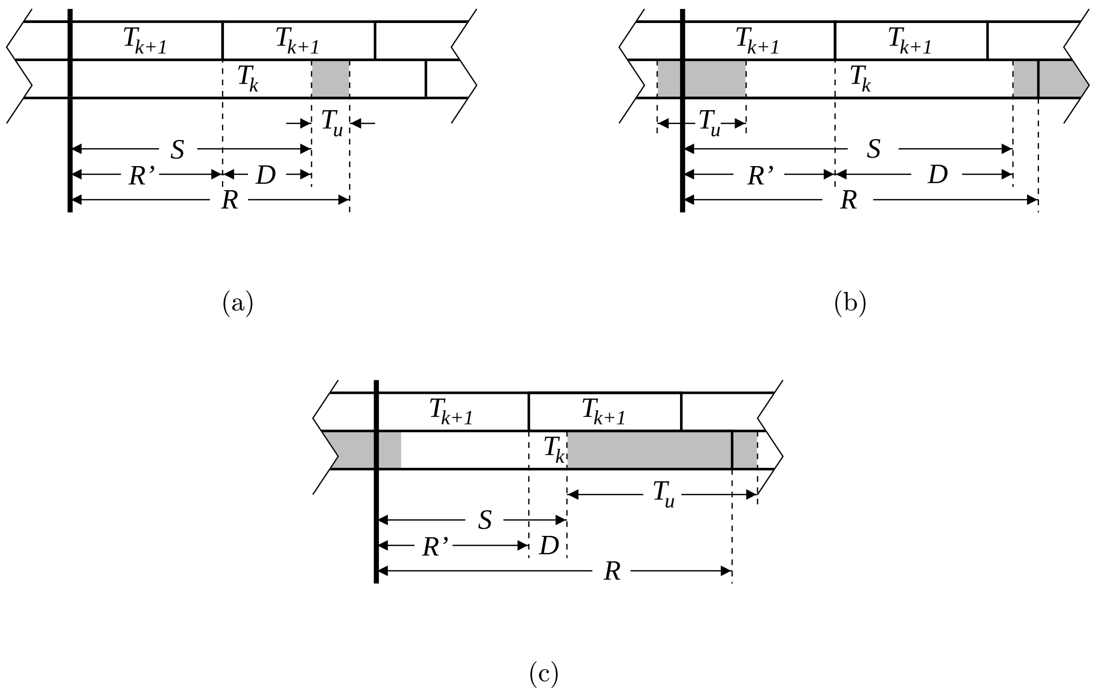


Figure 5.4: Examples of possible soft cache misses

this would be possible only if the client could get the useful segment in another way, i.e., directly from the server.

In cases like the above, and from the client's perspective, it pays off to actually generate a miss even for pages that can be found in the air-cache. Since such a miss is only a performance enhancement and not a functional requirement we call it a **soft miss**. In this example, the soft miss turns the (k, u)-hit into a faster $(k + 1, n_{k+1})$-hit. Another similar scenario where a soft miss can make significant difference is shown in Figure 5.4(b). This time, the client wakes up within a broadcast of the useful segment, which means that it will wait for $R = T_k$ to get the pages that were just missed. That includes a long dead interval D. If it could get those few pages by means of a soft miss, the refresh time would be cut down to $R' = T_{k+1}$.

But there is also a downside to soft misses; they add up to the server's load. However, exactly because they are "soft", it can be left to the server's discretion which of them (if any) to serve. For example, consider the example of Figure 5.4(c). Again, the client wakes up within a broadcast of the useful segment, but towards the end of it. A soft miss in this case would indeed reduce the refresh time substantially. However, its size would be quite big, and, depending on its load, the server might be reluctant to serve it.

For the server to be able to decide which soft misses it can accommodate and inform clients when to send one or not, we need to quantify the "importance" of a miss. The above examples suggest that we must give preference to small size misses with big potential to reduce the clients' refresh time. Based on that, we define the **merit** of a (k, u)-hit to create a soft miss to be $M = \frac{S - T_{k+1}}{T_u}$, where S is the time when the useful segment starts being broadcast after the client tunes in the channel. Intuitively, this metric favors situations with a small useful segment that starts being broadcast long after the client wakes up, causing lengthy dead intervals. Note that if the useful segment starts before T_{k+1}, there is no dead interval and M is negative. Also note that this definition applies even for the top level AC_C where $T_{C+1} = 0$.

The server establishes and broadcasts a **merit threshold** θ_k to instruct clients to send soft misses only if their merits exceed it. This way it can explicitly control the volume of such misses it receives. When it is fairly busy it should select a high threshold in order to limit the "performance misses" to a minimum. On the contrary, when not loaded, it can lower the threshold and offer more performance improvement chances to clients. In Section 5.3.3 we explain how exactly the server regulates that. However, the less obvious but more important advantage of this technique is that, based on the analysis of Section 5.2.2, we can compute both the probability $b(k,u)$ that a (k,u)-hit will cause a soft miss as well as the expected size $g(k,u)$ of that miss. As we will see later, this provides the grounds for accurate estimation of the workload. If we define

$$h(k,u) = 1 - \frac{\theta_k u}{n_k} - \frac{f_k}{f_{k+1}}$$

then the probability that a (k,u)-hit will create a soft miss, i.e., its merit will be higher than a threshold θ_k, is

$$b(k,u) = \begin{cases} h(k,u) & \text{if} \quad u < \dfrac{n_k}{\theta_k}\left(1 - \dfrac{f_k}{f_{k+1}}\right) \\ 0 & \text{if} \quad u \geq \dfrac{n_k}{\theta_k}\left(1 - \dfrac{f_k}{f_{k+1}}\right) \end{cases} \tag{5.3}$$

The expected size of such a miss is

$$g(k,u) = u - \frac{u^2}{2n_k h(k,u)} \tag{5.4}$$

The last piece of the picture is a suitable indexing scheme for the broadcast channel. In other words, along with the log pages, we need to broadcast information about the cache contents and structure so the clients can figure out how many pages they need, estimate how long it will take to download them, and compute their merit to send a miss to the server. In Table 5.1, we present the information that the clients need to do that. Assuming a small number of levels, the volume of this data is quite small. Thus, we choose to broadcast this index along with every log page.

An enhancement over this simple scheme would be to extend index entries with more detailed information about the updates being broadcast. For example, bit-vectors could be used to indicate the data that were updated by the log entries

For the Air-Cache
C : Number of air-cache levels
T : Period of air-cache
old : The oldest page in the air-cache

For every level AC_k
f_k : Frequency (if not fixed)
θ_k : merit threshold
new_k : most recent page
$next_k$: page to be broadcast next

Table 5.1: Index information for log air-caching

in each page [JEHA97]. These would allow clients to detect which of the log pages affect their data, and possibly, save time and power by downloading pages selectively. Depending on the size of the additional information, such enhancement would require a more sophisticated indexing technique [IVB94a].

5.3.2 Workload Estimation

In this section we describe how we can assess the actual workload of the system from the relatively small number of misses that reach the server. For our purposes, the workload is expressed as the rate $\boldsymbol{L} = (\lambda_1, \ldots, \lambda_z)$, where λ_m is the rate at which clients require (m)-refreshes. This is the output of the workload estimator that is passed on to the air-cache adapter. Note that the size of $\boldsymbol{L}$ grows as new log pages are created. But, at the same time more and more of the early log pages stop being accessed, zeroing the respective elements of $\boldsymbol{L}$. Therefore, in practice we only need to keep a reduced version of $\boldsymbol{L}$ starting with the oldest log page that got accessed over the last adaptation period.

The way λ_m is estimated depends on the whether page ℓ_m is air-cached or not. If it is not, then all (m)-refreshes yield hard misses since at least one of the required pages (ℓ_m) cannot be found in the air-cache. Therefore, these hard misses are the actual number of clients that required an (m)-refresh.

If, however, ℓ_m is air-cached the problem is a little more complicated. In this case, a client request for an (m)-refresh will result in some (k, u)-hit with the proper values of k and u. The server does not get any information about this hit, unless a soft miss is created. This means that it receives feedback for only a fraction of the actual (m)-refreshes. But, here is where the knowledge of the soft miss probabilities can be of service. If (m)-refreshes, resulting in a (k, u)-hit, create soft misses with probability $b(k, u)$, then the actual number of (m)-refreshes can be computed by dividing the number of soft misses by this probability.

Still, not all elements of $\boldsymbol{L}$ can be computed in this way. A small difficulty arises for computing the value of an λ_m when the corresponding miss probability $b(k, u)$ is zero or close to zero. In such a case, clients do not send any soft misses, and thus, the server has absolutely no information about those (m)-refreshes. To overcome this problem, we estimate such missing values by interpolating on the ones that are available. Obviously, there is no way of knowing whether these estimates reflect the real workload. However, our experiments confirm that this procedure yields a quite accurate estimation of the workload.

5.3.3 Air-cache Adapter

The air-cache adapter is the core of the system which makes the critical operating decisions. It has to provide answers to three questions: how many log pages should be air-cached, what is the best way to structure the air-cache for the given log access pattern, and when clients are allowed to send soft misses. Naturally, the answers to these questions are based on the output of the estimator. As it is shown in Figure 5.3, the adapter consists of three separate modules that operate in sequence, each deciding on one of abovementioned issues.

The adapter is invoked periodically at predefined intervals. In between adaptations, new log pages may be created. These are placed in a separate level of their own. In other words, if the last adaptation phase created an air-cache of C levels, all new pages created before the next adaptation phase will be placed in AC_{C+1}. This way, new pages do not affect the relative structure of the other levels, as this was last determined by the adapter.

Next, we elaborate on the three modules of the air-cache adapter.

Air-Cache Contents

The first decision the system has to take is the extent of the log that needs to be air-cached. The issue here is that we need to satisfy two contradicting goals. On one hand, we would prefer to select as few log pages as possible so that we end up with a small broadcast period and, therefore, small air-cache latency. But, on the other hand, the less pages we select the more refreshes will span beyond the air-cache, and cause hard misses to be sent to the server. So, in order to prevent the server from overflowing, we have to make sure that it does not receive more misses than it can handle.

We define the capacity (or throughput) μ of the server to be the maximum rate at which it can unicast log pages. This is determined by the server's processing power and/or the available network bandwidth. Our goal is to limit the workload imposed by the misses below that capacity. But, an important aspect of the system is that server must handle two types of misses. For this reason, we divide the server's capacity into two parts, μ_h and μ_s, and allocate them to hard and soft misses respectively ($\mu_h + \mu_s = \mu$).

The number of log pages in the air-cache affects only hard misses. Therefore, the goal of the first module of the adapter is to find what is the minimum number of log pages that should be air-cached so that the workload of hard misses does not exceed the allocated capacity μ_h. Bear in mind that misses can be of different sizes, i.e., different misses request different number of log pages. As a consequence, besides the estimated rate of the misses, we need to take into account the load that each miss generates in terms of the number of pages it requires. Formally, the problem is to find the maximum c for which

$$\sum_{i<c} (c-i)\lambda_i < \mu_h$$

This is rather easy problem; the proper value of c can be found with a single scan of the vector $\boldsymbol{L}$. This value determines the oldest page ℓ_c to be put in the air-cache.

Air-Cache Structure

Having selected the range of log pages to broadcast, we need to decide how to structure the air-cache so that we minimize the refresh time for the clients. As we discussed earlier, this calls for a solution of the optimization problem presented in Section 5.2.3. The problem was formulated in its most general form. However, the adapter is required to make fast on-line decisions for the structure of the air-cache. Thus, for practical solutions to the problem, we limit the number of decision variables. Specifically, we preselect the broadcast frequencies to be $f_k = 2^{k-1}$. This means that each cache level is twice as fast as the next lower level, i.e., $f_{k+1} = 2f_k$ and $T_{k+1} = \frac{T_k}{2}$. Under these assumptions, our problem is reduced to finding the n_k's to distribute the cached pages in the C levels so that the objective function is minimized.

But even this is a hard combinatorial problem that we cannot afford to solve optimally online. Instead, we have developed a greedy algorithm that finds a good solution by minimizing an approximation of the objective function. The inputs to the algorithm are the rate vector $\boldsymbol{L}$ provided by the workload estimator, and the range of log pages to be cached as determined by the first module of the adapter.

Formally, given that pages ℓ_c through ℓ_z should be cached, we have find the n_k's that minimize the objective function

$$\sum_{i=c}^{z} p_i \, \mathrm{E}\left[F_i\right]$$

where p_i is the probability that a client that wakes up requires a (i)-refresh. Note that this probability can be computed from the rate vector $\boldsymbol{L}$ since $p_i = \lambda_i / \sum_{j=c}^{z} \lambda_j$. Our algorithm is based on the following approximation of the objective function:

$$\begin{array}{rrcl}
 & \displaystyle\sum_{i=c}^{z} p_i \mathrm{E}\left[F_i\right] & = & \displaystyle\sum_{i=c}^{z} p_i \frac{\mathrm{E}\left[R_i\right]}{z+1-i} \\
\left(i = c + \displaystyle\sum_{q=1}^{k-1} n_q + j - 1\right) & & = & \displaystyle\sum_{k=1}^{C} \sum_{j=1}^{n_k} p_i \frac{T_k \Phi(k,j)}{z+1-i} \\
\left(\dfrac{\Phi(k,j)}{z+1-1} \approx \dfrac{1}{z+1-i}\right) & & \approx & \displaystyle\sum_{k=1}^{C} \sum_{j=1}^{n_k} p_i \frac{T_k}{z+1-i} \\
 & & = & \displaystyle\sum_{k=1}^{C} T_k \sum_{j=1}^{n_k} \frac{p_i}{z+1-i} \\
\left(\pi_i \equiv \dfrac{p_i}{z+1-i}\right) & & = & \displaystyle\sum_{k=1}^{C} T_k \sum_{j=1}^{n_k} \pi_i \\
\left(\Pi_k \equiv \displaystyle\sum_{j=1}^{n_k} \pi_i\right) & & = & \displaystyle\sum_{k=1}^{C} T_k \Pi_k \; = \; T \sum_{k=1}^{C} \frac{\Pi_k}{f_k} \\
\left(f_k = 2^{k-1}\right) & & = & 2T \displaystyle\sum_{k=1}^{C} \frac{\Pi_k}{2^k}
\end{array}$$

The algorithm starts by assigning all the pages to the lower level, and computes an initial value for the objective function. Then, it examines if the pages can be split into two parts so that if the most recent part is moved to the higher level, the value of the function is decreased. If there is not such a split then it stops. If there is, it splits the pages in the way that yields the minimum value for the objective function, and assigns the most recent part to the higher level. Then, it recursively applies the same check for the next level, i.e., it checks whether some of the pages that were moved in that level can be raised to even higher levels. The recursion stops when there is no split that can further reduce the value of the objective function.

Defining Soft Miss Thresholds

The last part of the adaptation process is the selection of merit thresholds for soft misses. The tradeoff here is similar to that of hard misses. We want to allow as many soft misses as possible without, however, swamping the server. The limiting factor here is μ_s, the server capacity allocated to soft misses.

The approach we adopt is to control the volume of soft misses on a per-level basis. We establish a merit threshold θ_k for each level AC_k, limiting the number of soft misses for pages in this particular level. Also, AC_k is allocated a capacity μ_k, a portion of μ_s proportional to the number of pages cached in it.

Given the structure of the air-cache, we can compute the probability $b(k,u)$ that a (k,u)-hit will generate a soft miss, for any value of the merit threshold θ_k. In addition, we can compute the expected size $g(k,u)$ of such a miss. Therefore, as we also have an estimate for the rate of (k,u)-hits, we can compute the expected soft misses load for level AC_k. This is our basis for selecting the merit thresholds. Specifically, if q is the LSN of the most recent log page cached in AC_k, then θ_k is assigned the minimum value for which

$$\sum_{u=1}^{n_k} g(k,u)b(k,u)\lambda_{q-u+1} < \mu_k$$

5.4 Experiments

In this section we present the most important results that we obtained from a detailed simulation of the proposed system. In the presentation of the experiments, time measurements as well as simulation parameters are expressed in terms of broadcast units.

The simulation model consists of a server, a variable number of mobile clients, and the network interconnecting them. This network is hybrid in the sense that there are three separate communication paths:

1. The broadcast channel with a fixed bandwidth capable of delivering 1 page per time unit.

2. The unicast downlink from the server to the clients which is a shared resource used for all server replies. We have assume that this link has similar characteristics with the broadcast channel, i.e., it too can transfer 1 page per unit.

3. The uplink(s) from the clients to the server. Because of the small size of requests from the clients, we do not consider the possibility of congestion in the uplink channels.

The server model implements the architecture shown in Figure 5.3. We assume that the processing power of the server is sufficient to utilize the full bandwidth of the downlink. This means that the server can unicast log pages at a maximum rate of $\mu = 1$ page per unit. The generation of updates is simulated through a separate module running at the server. Its function is to create new log pages, and notify the air-cache adapter every time it does. This process is governed by the "inter-arrival" time distribution of log pages, i.e., the distribution of the interval between the generation of successive pages. For all the experiments presented herein, we used exponential inter-arrival time with mean 1000 time units. The adaptation period of the server was also set to 1000 units.

The client model we used is quite simple. Basically, the only characteristic of the clients is the distribution of their sleep time. At the end of a sleep period, a client wakes up, retrieves all log pages created during this period, and then goes back to sleep. We assume that clients do not remain awake after they receive the updates they need. For this kind of operation, the only state information requires for each client is the most recent log page it received the last time it woke up.

5.4.1 Fixed Size Air-Cache, No Hard Misses

For the first set of experiments we consider a (probably unrealistic) scenario where the server maintains a fixed number of log pages in the air-cache. Essentially, we relieve the server from the first part of the adaptation procedure, i.e., from having to decide how many pages to air-cache. In addition, clients that wake up require log pages only within the range of these pages (following some given distribution), regardless the last log page they received before they went to sleep. This means that no hard misses are sent to the server because clients never require refreshes beyond what is air-cached. Consequently, all the server capacity is allocated to soft misses. The reason we used this hypothetical scenario is that it constitutes some sort of a static case for which we can compute the theoretically optimal air-cache performance. This provides a solid baseline to compare our system against.

For the results presented here, the server always air-caches the last 200 pages. Clients always require refreshes that start with any of these pages. We tested the system for three different distributions for the range of the refreshes, which are

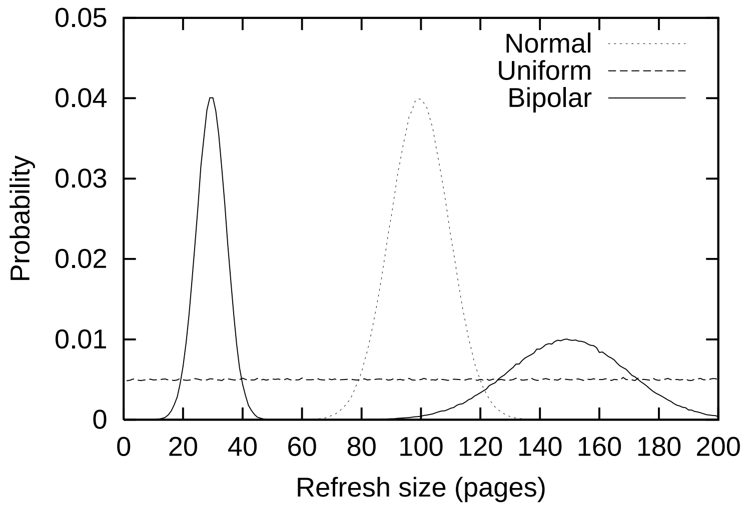


Figure 5.5: Distributions of client refreshes

depicted in Figure 5.5:

Normal The clients' refresh size follows a normal distribution with mean 100 and standard deviation 10. This is close to a best case situation where all clients need approximately the same number of pages.

Uniform The sizes of refreshes are uniformly distributed over all 200 pages. For the server, this is a worst case scenario, since it has to equally satisfy a very wide range of client needs.

Bipolar This reflects the situation where clients are partitioned into two equal size groups: workaholics who sleep only a little and usually need few log pages (normally distributed with mean 30 and standard deviation 5), and sleepers who tend to sleep more and require many more updates (normally distributed with mean 150 and standard deviation 20).

Our goal is to show that the system can efficiently disseminate updates even in very large scale, adapting to the (dis)connection habits of the client population. Efficiency is measured in terms of the clients' refresh factors. In the rest of this section, we present the results we obtained for the above three types of workload at a variable scale, i.e., a variable number of clients.

For these experiments, we plot up to four different curves to emphasize different aspects of the system's performance. In particular, we want to demonstrate the effectiveness of the cache optimization algorithm, the accuracy of the workload estimation procedure, and the performance benefit of soft misses:

Optimal Broadcast Only The first curve corresponds to the theoretically optimal performance of the air-cache, when only broadcast delivery is used (i.e., there are no misses). Basically, it is the solution to the optimization problem of Section 5.2.3 for the given frequencies, and serves as our comparison baseline. These results were obtained with exhaustive search over all the possible air-cache configurations. Keep in mind that the performance for all "Broadcast Only" scenarios depends only on the log access pattern and not on the number of clients. Therefore, it is the same at any scale.

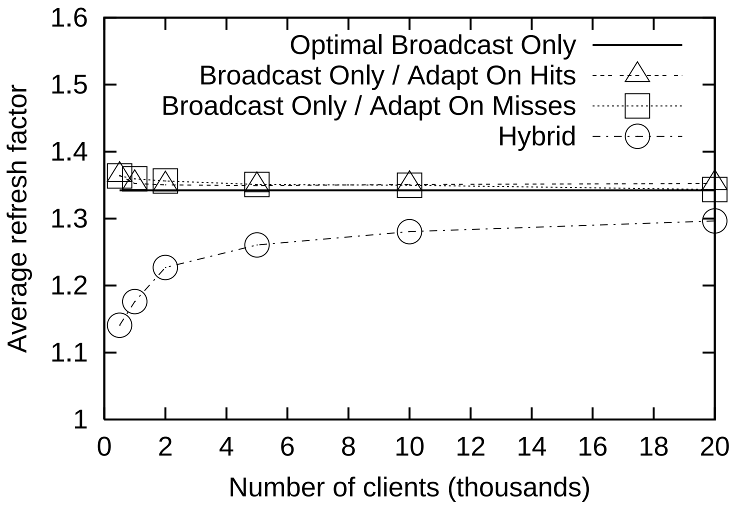
Broadcast Only / Adapt On Hits This and the next two curves were obtained from the simulation, each under a different setup. In this one, log pages are delivered only through the air-cache, and clients do not generate any misses. Instead, we (magically) provide the server with all the information about the clients air-cache hits. This way, the server has a complete picture about the activity and the needs of the clients. Its only task is to structure the air-cache to according this picture. Compared to the "Optimal Broadcast Only" curve, this case demonstrates the effectiveness of the air-cache optimization algorithm, without any possible side effects of the miss-based workload estimation.

Broadcast Only / Adapt On Misses For this curve, we allow clients to send soft misses to the server. The server uses the misses to estimate the workload, but it does not reply to the clients; clients still get the data from the broadcast. Compared to the previous curve, this time the server has the additional task to compose a picture of the workload just from the misses. Therefore, this curve shows the ability of the server to estimate the workload.

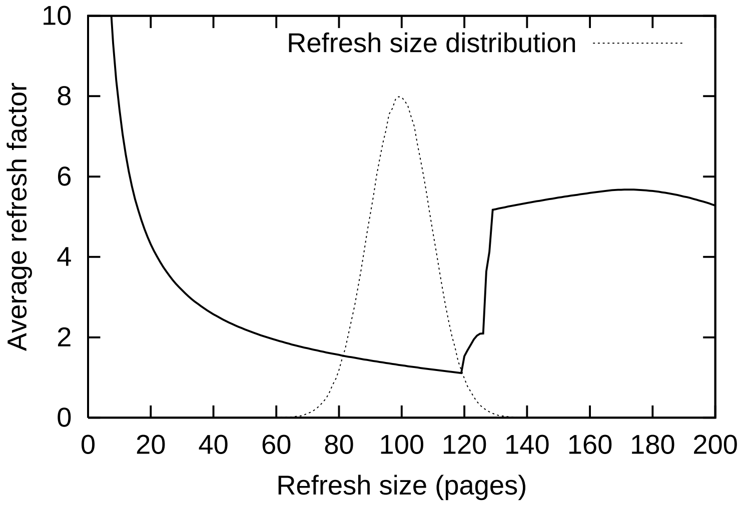
Hybrid The last curve corresponds to the performance of the system under normal operation. Clients send soft misses which the server does service, helping them download the required log pages faster. This result shows the performance improvement from soft misses over the broadcast-only cases.

Normal Distribution Figure 5.6 presents the results for the normal distribution of client refreshes. First, in Figure 5.6(a) we show the refresh factor for different sizes of the client population which vary from 500 clients to 20000 clients. The clients sleep time is uniformly distributed with mean 15000 units, and the mean size of each refresh is 100 pages. With simple arithmetic, we can compute that at the low end of the scale the clients request, on average, about 3 pages per time unit, while at the high end about 130 pages per unit. Notice that, considering both the unicast and the broadcast channel, the server can transmit only two pages per unit. This means that, for these experiments, the rate at which clients request data is 65 times larger than the available network bandwidth.

The first thing to note from this figure is that, for the most part, the three "Broadcast Only" curves are virtually indistinguishable. They all yield the same average refresh factor. This result suggests two things: first, the proposed greedy



(a)



(b)

Figure 5.6: Fixed size air-cache - Normal distribution

air-cache optimization algorithm generates a near optimal air-cache structure, and second, the workload estimator can accurately assess the clients needs relying only on the misses. An interesting observation is that there seems to be a slight performance degradation at the left end of the graph, i.e., under a light workload. The reason behind this surprising behavior is that under very small request rates, it is harder for the server to detect a pattern in the clients' demands. Note that this happens even when the server adapts on the hits, which means that it should not be attributed to the workload estimation.

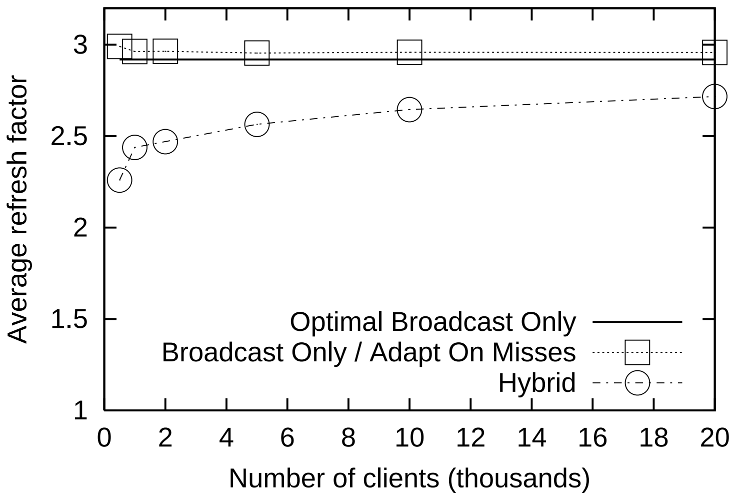
Probably more interesting is the fact that the performance of the full fledged hybrid system is better than the broadcast-only setups. Obviously, this is a reflection of the performance advantage soft misses bring to the system. We defer the discussion on this issue for the next set of experiments where this performance difference is more meaningful.

As we mentioned earlier, this normal distribution is almost a best case scenario for disseminating the log. The reason is that all clients need approximately the same number of pages. Thus, the system can structure the air-cache to match these types of requests really well, yielding an average refresh factor (1.34) close to 1. To demonstrate how this is actually achieved, in Figure 5.6(b) we present a break down of the average refresh factor for different refresh sizes. The solid line gives the average refresh factors. To put things into perspective, we have also superimposed the log access pattern, at no particular vertical scale. Clearly, the air-cache is optimized to yield minimum factors where the bulk of refreshes are (for sizes between 80 and 120) at the expense of very rare refresh sizes outside this range.

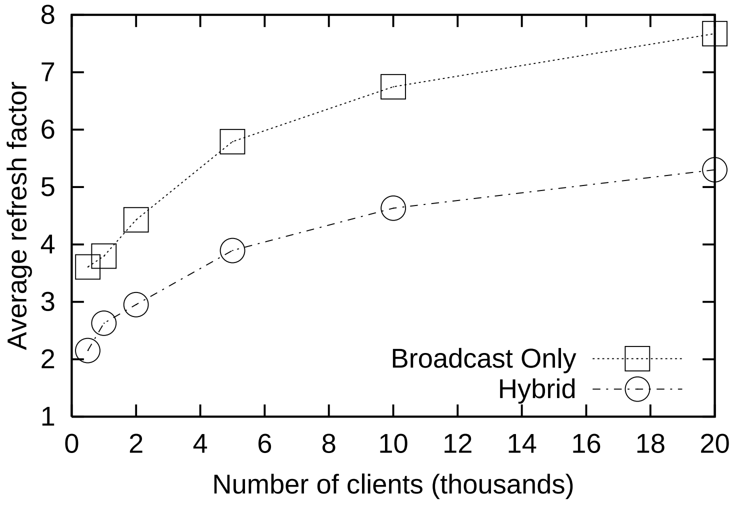
Uniform Distribution Under a uniform distribution of refresh sizes the system has the hardest possible task: structuring the air-cache to serve a very wide range of client demands without giving preference to any one group in particular. This means that it cannot undermine the performance for one type of refreshes in order to improve performance for another, as it did in the previous case. Instead, it tries to level off the performance of everyone as much as possible. The result is a higher average refresh factor. This shown in Figure 5.7(a) where we plot the average refresh factors for this type of workload. The other parameters of the experiment are the same as in the previous one. This time, the optimal performance for a broadcast-only delivery is 2.92, more than double the factor obtained under the normal distribution. Nonetheless, even in this case, the broadcast-only version of our system performs very close to the optimal.

Here, because of the higher refresh factors, there is also a higher margin for improvement with soft misses. Indeed, in Figure 5.7(a), the gap between the broadcast-only curve and the hybrid is wider. The reason is that soft misses alleviate the delays of lengthy broadcast refreshes. The extent of the performance improvement depends on the volume and size of soft misses the server can accommodate. The lighter the workload the more and bigger soft misses it can service, and thus, the bigger the refresh time savings.

The performance savings are better illustrated Figure 5.7(b), where we present the average refresh factors for only those refreshes that actually generated a soft miss. The line labeled "Broadcast Only" reflects their performance when the misses were not serviced by the server, and the clients eventually received the data from the broadcast. The other line shows the performance of the same clients when the soft misses were served by the server. The improvement from soft misses is clear;



(a)



(b)

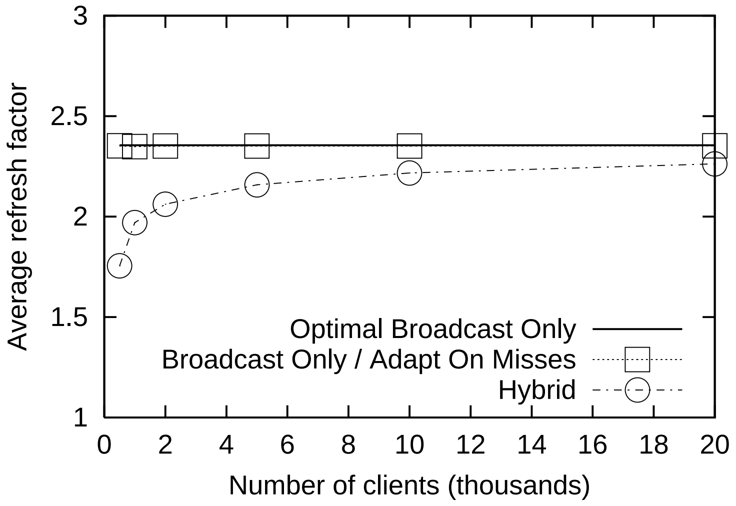
Figure 5.7: Fixed size air-cache - Uniform distribution

the average refresh factor drops between 30% and 40%. This big difference also implies that the generated soft misses indeed exhibited high merits.

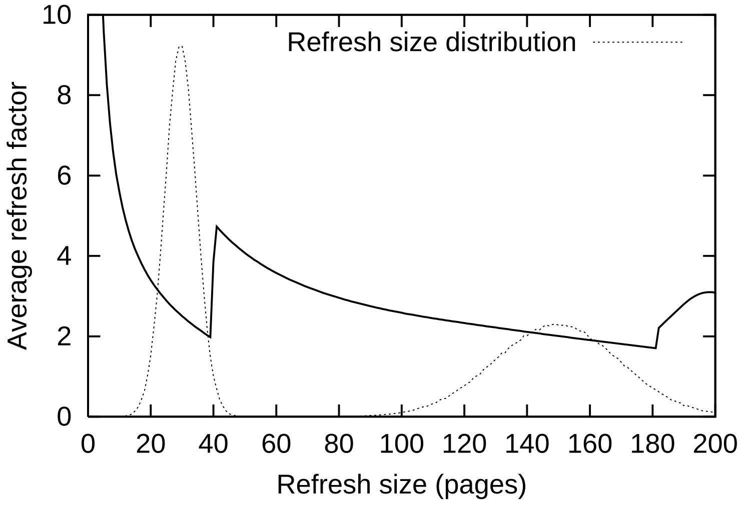
It is also interesting to observe the behavior of these two lines with respect to the scale of the experiment. As the workload to the system increases, soft misses affect higher refresh factors. This is an artifact of the dynamic selection of merit thresholds. When the workload increases, the server has to be more selective as to what soft misses it is willing to serve. Thus, it raises the merit thresholds, limiting the ability to create soft misses to refreshes with high refresh factors.

Bipolar This distribution reflects an in-between scenario where the system opts to satisfy two groups of clients with very different needs for updates. As it was expected, in this case the air-cache can do a better job delivering the log pages than under the uniform case, but not as good as the normal case. This time the optimal average refresh factor for broadcast-only delivery is 2.35 (Figure 5.8(a)). As far as our system is concerned, once again we achieve optimal performance for the broadcast-only versions, and similar improvements with the hybrid version. Figure 5.8(b) plots the performance of the system for the different refresh sizes. Again, with the help of the superimposed log access pattern, we can see that the algorithm allocated the log pages into cache levels so that the smallest refresh factors fall under the two bells of this bipolar distribution. In other words, it structured the air-cache to satisfy both sleepers and workaholics alike.

Finally, this experiment revealed another significant aspect of the system. Even though the broadcast-only version of our system matches the theoretically optimal performance, it does so using an apparently different air-cache structure. The exhaustive search indicated that the optimal structure is to allocate the 200 pages in four levels so that $(n_1, n_2, n_3, n_4) = (19, 141, 1, 39)$. On the other



(a)



(b)

Figure 5.8: Fixed size air-cache - Bipolar distribution

hand, with the adaptive algorithm the average number of pages in each level were $(159.6, 2.5, 37.9, 0)$; yet they both yield the same performance. In order to ensure that this was not an error in our algorithm or the simulation model, for this particular workload, we computed and compared the theoretical expected refresh factors for all the possible allotments of the 200 pages in up to four cache levels. What we found was that there is a considerable number of combinations that yield performance very close to the optimal, including one similar to that produced by our algorithm. The explanation for that is that even though they may appear quite different, in practice they produce very similar broadcast sequences. In this case, for example, the optimal structure puts the 39 most recent pages in AC_4 where they get broadcast 4 times more often than the bulk of the rest pages (141) cached in AC_2. Our system created a similar effect in a different way: it placed almost the same number (37.9) of the most recent pages in AC_3 where again they get broadcast 4 times more often than the bulk of the rest pages (159.6) placed, in this case, in AC_1.

5.4.2 Variable Size Air-Cache

For the second set of experiments we used a more realistic scenario, where the size of the air-cache is not fixed. This time the decision of how many log pages to air-cache rests with the server, as it would in a actual deployment of the system.

Clients also operate in a more natural way. In other words, every time they wake up, they ask for all the log pages generated while they were sleeping. After they get all these pages, they go back to sleep for a random period. Sleep times are chosen to create a mixed sleepers/workaholics client population. Half of the clients are characterized as workaholics with sleep time normally distributed with mean

30000 units and standard deviation 5000 units. The other half are sleepers with sleep time normally distributed with mean 150000 units and standard deviation 20000 units. These parameters were chosen so that we obtain a workload similar to the bipolar distribution of the previous section; on average a workaholic requires 30 pages, and a sleeper 150.

Contrary to the previous case, now clients may also generate hard misses, if they happen to need old pages that have been dropped out of the air-cache. As it was described in Section 5.3.3, in this case the capacity μ of the server must be split to μ_h for hard misses, and μ_s for soft misses. The actual split is specified by the parameter *SoftMissesShare* which corresponds to the portion of μ allocated to soft misses. For example, a value of 0.2 for this parameter means that $\mu_s = 0.2\,\mu$ and $\mu_h = 0.8\,\mu$.

The results of this experiment are shown in Figure 5.9. Again, the number of client ranges from 500 to 20000. In order to show the effects of the capacities allocated to each part of the system, we plot the results for two different values of the parameter *SoftMissesShare*. For *SoftMissesShare*=0.2, most of the capacity of the server is allocated to hard misses; the opposite holds for *SoftMissesShare*=0.8. For each value of this parameter, we also include the results of a test where the system uses the hits to adapt on. These contrast the performance of the system to the ideal, but unrealistic, scenario where the server has perfect knowledge about the workload.

The first conclusion from this graph is that the system exhibits the same significant scalability. Under all configurations, it can efficiently service at least up to 20000 clients, yielding small refresh factors. In fact, the resulting factors are even smaller than those in the previous experiment (Figure 5.8(a)). This is because in

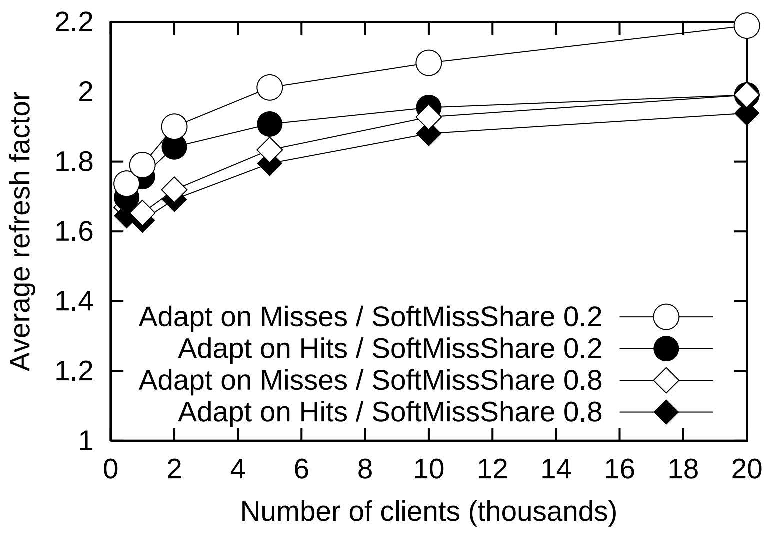


Figure 5.9: Variable size air-cache

the previous experiment the server was forced to always air-cache 200 pages, even though it was not necessary. This time the server could decide for itself, and indeed chose to keep a smaller air-cache. It is important to point out that the system not only performs and scales very well, but also exceeds its nominal throughput by many times. At the highest scale, it delivers data at a rate 65 times the available network capacity. In other words, by exploiting the commonality between multiple clients, we achieve a manyfold increase of the effective bandwidth.

Furthermore, by comparing the four curves in the figure, we see that the system performs better for the large value of the parameter *SoftMissesShare*, i.e., when it allocates more resources to soft misses. There are two reasons for that: First, the server can accommodate more soft misses, and thus, help more clients improve their refresh times. Second, more misses provide a better picture of the clients' needs and help the server make more informed decisions. This is evident by the fact that, for *SoftMissesShare*=0.8, the performance curve of the system when adapting on the misses follows very closely the ideal curve of hits-based adaptation. On the contrary, the gap between hits-based and misses-based adaptation is wider for *SoftMissesShare*=0.2, especially in large scale when the server cannot afford to serve many misses.

These results suggest that most of the server capacity should be allocated to soft misses. In order to see whether there is more benefit allocating even more than 0.8 of the server's capacity, we also ran experiments with $\mu_s = 0.9\,\mu$, and $\mu_s = 1\,\mu$. In the first case the results almost matched those for $\mu_s = 0.8\,\mu$. However, when all the server was allocated to soft misses, the refresh times of the clients more than doubled. But, this was an expected result. When we allocate all capacity to soft misses, we effectively prohibit hard misses. This means that the server has to make sure that it receives no hard misses. The only way this can happen is by air-caching all the log, or at least a very big part of it. Naturally, the result is high broadcast periods and, consequently, high refresh factors. Given these observations, a value of 0.8 appears to be a good and safe choice for the *SoftMissesShare* parameter.

5.5 Conclusions

In this chapter, we addressed the problem of propagating updates from a data server to a large number of mobile clients. Such clients typically operate off-line on data replicated from the central database, but occasionally they need to refresh their data with changes committed at the server. We proposed a system that employs adaptive hybrid data delivery, i.e., air-caching, to disseminate the log of updates to the clients.

First, we described a hierarchical form of air-caching that supports multiple cache levels. each with different average access latency. We analyzed the performance for broadcasting a log of updates using the hierarchical air-cache, and formulated the optimization problem of structuring the air-cache according to the clients' access pattern. Then, we described the proposed adaptive hybrid system, and elaborated on its key components. We also introduced the notion of soft air-cache misses, i.e., misses for cached data, that allow clients to improve performance over broadcast delivery.

The experimental results confirmed our performance expectations. The system can detect the clients request patterns, and adapt the structure of the air-cache almost optimally to match the sleeping habits of the clients. Also, the dissemination of updates was very efficient. The refresh times for clients was almost constant across a wide scale (up to 20000 clients). The system exploits the commonality of among clients needs and uses the broadcast capability very efficiently, yielding a throughput many times higher than its nominal capacity. Moreover, the results demonstrated the important double role of soft misses for the system: they provide information on the clients sleep time habits, and in some cases help improve performance over broadcast only delivery, especially under not very heavy workloads.

Chapter 6

Publish/Subscribe Services

Publish/subscribe services are been used as alternative data distribution mechanisms for a number of applications including mailing lists, Netnews [YGM96], document dissemination [LT92], and financial systems [OPSS93, Gla96]. Typically, in such systems, there is a server (**publisher**)[1] that generates and/or collects information of potential interest to a client population. Each client (**subscriber**) specifies a set of interests, and expects to receive any pertinent information generated thereafter. This set of interests is often referred to as the client's **profile**. The goal of the system is to match the generated information with client profiles, and deliver it accordingly.

This kind of interaction between publishers and subscribers raises (at least) the following question: Where in the system should data be matched to profiles? In other words, which side should take over the task of filtering the information? There are three possible answers to this question: the publisher, the subscriber,

[1]In the general case there may be several publishers. Here, we limit the discussion to cases of a single publisher. Depending on the application and the setting, the publisher can either be the original source of the information, or a broker that collects and relays information from multiple sources

or both [Loe92]. In the first case, the publisher filters the data and sends it only to interested clients. Ideally, clients receive exactly what they want, which means that neither client resources nor network bandwidth is wasted on processing and transmitting irrelevant information. On the downside, the server carries all the filtering load, which is proportional to the number of clients as well as the rate at which new data are generated. Hence, in large scale systems the server can easily become a bottleneck. In the second case, the server merely acts as a pump of unclassified data to the clients which are responsible to distill any useful content. In some sense, the dissemination of updates, as it was explored in the previous chapter, corresponds to this extreme case of a publish/subscribe service. The main advantage is that the server avoids the filtering load which is now distributed among the clients. Furthermore, each client has complete control over what is useful and what is not. Obviously, a requirement is that the clients are capable of performing this task. The pitfalls of this approach are that clients may have to process an overwhelming amount of information, and that too much network bandwidth may be wasted for data destined to be rejected.

A compromise between the two approaches is to split the filtering task between the server and each client. Initially, the server makes some coarse classification of the generated information, and a similar classification of the client profiles. Thereafter, it propagates data items only to those clients whose profiles intersect semantically with the items' class. The clients again have to filter the incoming data. However, this time the probability of a match can be significantly higher.

In this chapter we turn our attention to this last type of publish/subscribe data services. Similarly to the previous chapter, we place the problem in the context of mobile computing and intermittently connected clients. We are proposing to use the air-cache mechanism to disseminate information to mobile subscribers.

Generally, there are a number of applications that can fall under this general model. Below we describe two motivating examples:

Decision Support Systems In the corporate world, the number of mobile decision-makers is increasing fast. One of their main weapons is on-the-road access to enterprise data warehouses, such as datamarts and desktop OLAP (DOLAP) tools. To alleviate problems of mobility and disconnection, they need to store in their portable computers materialized views which are pertinent to their work. These, however, have to be kept up-to-date with respect to the main warehouse. A possible approach is for mobile users to register their views with the server, expecting to receive any relevant updates. To reduce processing overhead, the server can collect the subscriptions, aggregate them, and select to materialize and maintain a set of basic views that cover all clients subscriptions. In the literature, there are a number of algorithms that can be used for this purpose (e.g., [Rou82, RCK^{+}95, RSS96, Gup97, TS97b]). It also provides each client with a set of rewrite rules that can be used to derive the specific client views. Thereafter, the server disseminates changes to the basic views, which are used by the clients to refresh their own views.

Selective Information Dissemination Systems that selective disseminate information are becoming an attractive answer for the information overload of the WWW. With such systems, instead of distressingly searching for information, users effortlessly receive information (mainly documents) relevant to their interests, according to their pre-specified profile. As we mentioned earlier, very fine matching between the profiles and the generated documents limits the scalability of the system [YGM94]. Instead, the server can classify the user profiles according to some crude similarity criteria, and disseminate a document to all clients in a matching profile class. Each one of these clients completes the filtering process in order to actually establish the relevance of the document. This kind of information dissemination is being used in commercial systems offering news services, like the PointCast Network [Poi98], or AirMediaLive [Air98]. Typically, these products organize information into **channels** based on the originating source (e.g., CNN, the Sports Channel) and a general subject (e.g., world news, basketball). Users subscribe to one or more of these channels and selectively examine the information they receive.

In the rest of the chapter we explain how the concept of air-caching can be used for disseminating data to mobile subscribers. We consider cases where clients subscribe to rather general data services, and thus are responsible for filtering or personalizing the received information. For clarity, the description is based on the example of decision support systems and materialized views. However, the methods and the results can be easily projected to any other application by drawing simple analogies (e.g., a news channel corresponds to an append-only table, and a new story is a new tuple).

6.1 Problem Definition

In this section we define the problem more formally. We consider a server that maintains and publishes a data warehouse consisting of a specific set of C materialized views $\mathbf{V} = \{V_1, V_2, \ldots, V_C\}$. The server collects updates in batches from the original data sources and refreshes these views. Each refresh produces a new **version** (or **snapshot** [AL80]) of the view either by fully recomputing the view or by generating a proper **view increment**. Generally, the refresh method can be selected dynamically based on server policies, type of view, volume of updates, and so forth [Vis97]. Here we restrict our study to cases where the refresh method is decided a priori. We must note, however, that the proposed methodology can be extended to incorporate a more flexible scheme at the expense of implementation complexity.

Let $V_{k,i}$ be the i-th version of view V_k generated at time $t_{k,i}$. For recomputed views, once a new version $V_{k,i}$ is generated, the previous version $V_{k,i-1}$ is dropped from the server and replaced by the new one. On the other hand, for incrementally maintained views, an increment $\delta_{k,i}$ is generated which is then applied to the previous version $V_{k,i-1}$ to yield the new version. Again, the old version is replaced by the new one. But in this case, the server stores the increments as well. This way, the current version of the view can be computed from any older version by successively applying all younger increments, i.e., $V_{k,i} = V_{k,j} + \delta_{k,j+1} + \ldots + \delta_{k,i}$, for any $j < i$. Note that the size of a view is not fixed and can vary from version to version. We define $L_{k,i}$ to be the size of view $V_{k,i}$, and $l_{k,i}$ the size of increment $\delta_{k,i}$.

Each mobile client selects to maintain in a portable computer its own set of views. Initially, it goes through a registration process to provide the server with a set of definitions for the views it wants to store. The server checks these definitions and determines how these views can be derived from those in the set $\mathbf{V}$. It generates a set of rewrite rules and a subscription vector $\mathbf{s} = (s_1, s_2, \ldots, s_C)$, where $s_k = 1$ if V_k is involved in at least one of these rules, and $s_k = 0$ if not. The rewrite rules and the subscription vector are sent back to the client, along with the current version of every V_k for which $s_k = 1$. The client uses this information to derive a first version for its own views. It also stores the set of rewrite rules and the subscription vector. Given this vector, the client is considered to be subscribed to all the views V_k for which $s_k = 1$. Here, we assume that client subscriptions do not change very frequently, and therefore, do not address reorganization issues dealing with such changes.

Most of the time, a mobile client operates in sleep mode, i.e., with its communication device(s) turned off. During those periods it relies on whatever data are found in its local storage. Occasionally, it wakes up to refresh that data. This requires to refresh the copies of the views it has subscribed to. Let us assume that $V_{k,j}$ is the most recent version of V_k stored in the client, while $V_{k,i}$ is the latest version at the server $(i \geq j)$. If V_k is incrementally maintained, the client needs to download all the increments that were generated while it was sleeping, i.e., increments $\delta_{k,j+1}, \ldots \delta_{k,i}$. If, on the other hand, V_k is recomputed, the client needs to download the latest version $V_{k,i}$ of the view. In both cases, if $i = j$ then V_k has not changed, and therefore, no data need to be downloaded.

The goal of this chapter, is to show how air-caching can be applied for large scale publish/subscribe services that fit the above description. For such applications, it is expected that the fresh increments or recently recomputed versions will be in very high demand, specially for the most popular views. As we have shown so far, the air-cache is an excellent technique to efficiently disseminate such data. But, in order to make it work also in this case, we need to decide which data are worth air-caching, and how should those be structured in the air-cache. In the next section, we describe a methodology that addresses these issues. The effectiveness of the proposed techniques is demonstrated by the experimental results that follow.

6.2 Methodology

The proposed methodology is based on the idea of autonomous view managers. A **view manager** M is a process running at the server, responsible for disseminating one of the views. For a server that maintains C views, we define C of those managers $M_1, M_2, \ldots, M_C$. Manager M_k gets a share of the system resources, and takes over the task of propagating updates of view V_k. Practically, this means servicing client requests for (increments of) it, and air-caching it as necessary. Figure 6.1 depicts an overview of a server with C view managers. One of these, manager M_k, is shown in more detail to better illustrate their functionality.

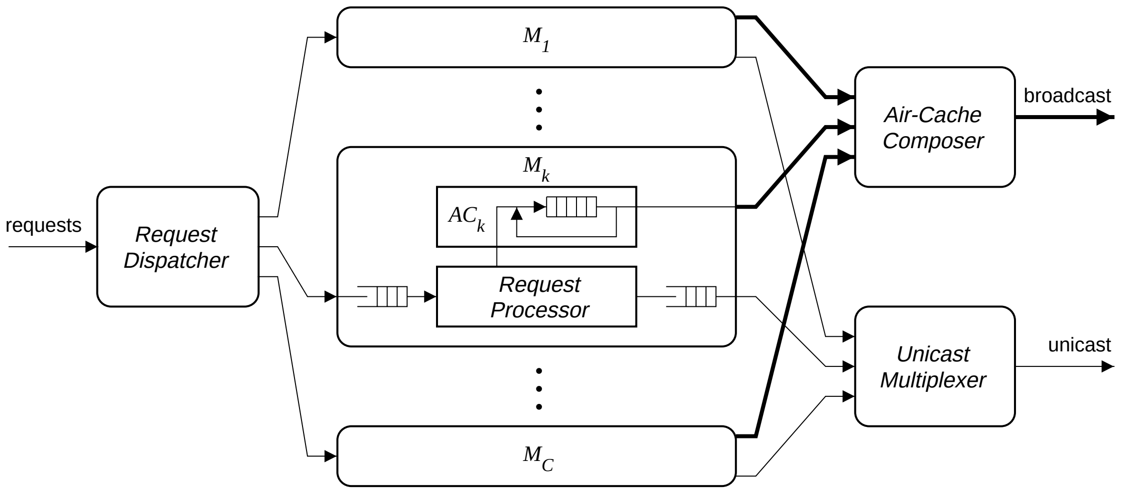


Figure 6.1: Server architecture

Each manager M_k creates and manages its own air-cache AC_k where it can store (increments of) view V_k. It also services clients requests (i.e., air-cache misses) related to this view. The actual use of this air-cache depends on the refresh method of the view:

- If V_k is recomputed, the manager may cache the latest version of the view for as long it remains popular. When its popularity drops enough, it is removed from AC_k leaving it empty, typically, until a new view version is created. While the view is cached, subscribed clients always retrieve it from the air-cache, and the manager receives no requests for it. On the contrary, when it is not cached all subscribed clients have to ask the server for the latest version of it.

- If V_k is incrementally maintained, the manager M_k may cache a number of recent increments $\delta_{k,j}, \ldots, \delta_{k,i-1}, \delta_{k,i}$, where $\delta_{k,i}$ is the most recent increment. As the popularity of these increments decreases, they are removed from the air-cache in decreasing order of age. When clients subscribed to V_k wake up, they retrieve from the air-cache most (maybe all) of the increments they need. In case they do need older increments not found in the air-cache, they have to contact the server to get them.

Each AC_k follows a flat broadcast scheme which, as we have already discussed, is implemented through a cyclic queue Q_k. In total, there are C such air-caches sharing the same broadcast channel, multiplexed to form a composite air-cache structure. This multiplexing is carried by the **air-cache composer** who schedules the broadcast in a way that adheres to the properties of the composite air-cache (see Section 3.3.3).

Managers also share the server resources to service air-cache misses, i.e., client requests for versions or increments of views not found in the air-cache. Each manager has an input queue for requests, and an output queue for replies to those requests. When a request related to view V_k arrives to the server, it goes through the **request dispatcher** who directs it to the input queue of M_k. From there, it gets processed (in FIFO order) by M_k, and the result is placed in the output queue, where it waits to be transmitted to the requesting client. At this stage, results from different managers may be waiting as well. For this reason, there is a **unicast multiplexer** who coordinates the transmissions and regulates the use of the unicast link.

The advantage of this technique is decreased complexity by decomposing the problem of managing the air-cache. Instead of having to globally optimize the air-cache over all the views, each M_k takes a local decision for the view it is responsible for, given the evolution of the view, the air-cache misses generated by interested clients, and the system resources allocated to it. In this way, view managers operate independently from one another and, as we will see later, share very little information.

Nonetheless, in order to make this scheme work, we need to answer two basic questions: How do view managers share (or compete for) the system resources, and how do they use and adapt their air-caches? From our perspective, the system resources in question are the broadcast bandwidth, and the server capacity. In the following sections, we provide answers to these questions and discuss the details of our approach. We start by discussing the broadcast bandwidth distribution among the air-caches. Then, we explain how view managers share the server resources. Section 6.2.3 describes the adaptive part of the system, i.e., the management of the individual air-caches. Last, we present the broadcast indexing method adopted in this case.

6.2.1 Sharing the Air-cache

According to the above description, the broadcast bandwidth is shared by all view managers of the server through a composite air-cache structure. As we saw in Section 3.3.3, a composite air-cache is regulated by a set of factors $\phi_1, \phi_2, \ldots, \phi_C$.

These determine the minimum bandwidth allocated to the different simple air-caches, and therefore, their performance characteristics (i.e., average latency). In this section, we discuss the selection of these factors.

There are two parameters that can affect the performance of the air-cache, and thus, the selection of the bandwidth factors: the popularity of each view, and the size of each air-cache. A well known result in the context of repetitive broadcast systems is the **square root rule**. According to this rule, the clients' expected response time is minimized when the repetition frequency of any item in the broadcast is proportional to the square root of its access probability (i.e., its popularity) [AW85, AW87]. This rule applies to data items of equal size. Generalized to data of various sizes, it also prescribes that the optimal frequency for each item is inversely proportional to the square root of its size [VH96]. In our case, the popularity of a view is expressed by the number of clients that have subscribed to it. Let r_k be the number of subscribers to view V_k. The size s_k of air-cache AC_k is determined by the amount of data manager M_k has decided to cache. For recomputed views, if the latest version $V_{k,i}$ is cached then $s_k = L_{k,i}$; otherwise AC_k is empty ($s_k = 0$). For incremental views, the size s_k is the sum of the sizes of the cached increments, i.e., $s_k = l_{k,j} + \ldots + l_{k,i-1} + l_{k,i}$. Given that, the square root rule suggests that the relative repetition frequencies of data in any two caches AC_k and AC_m should be such that

$$\frac{f_k}{f_m} = \sqrt{\frac{r_k}{r_m}}\sqrt{\frac{s_m}{s_k}} \tag{6.1}$$

Nonetheless, this result is not directly applicable to our setting. The reason is that the square root rule aims at minimizing the response time of client requests for a single item. Here, however, a client may be subscribed to more than one view. This means that every time it wakes up it attempts to retrieve from the air-cache updates for all the views it has subscribed to.

Early in our experimentation, we made the observation that the average response time of requests for multiple items depends on the repetition period of the slowest of these items. The repetition frequencies of the other items have very little effect, since on an average case clients have to wait for the slowest item anyway. This bears an important consequence: **the expected response time of a request for multiple items is minimized when all these items are repeated with the same period**. The intuition behind this claim is that if we decrease the period for any one of the items, we necessarily increase the period of the rest, which translates to higher expected response time.

Driven by this observation, we characterize a client as **bound** by view V_k when V_k is the less popular view it has subscribed to. According to the above claim, the expected air-cache latency for clients bound by V_k depends on the repetition period of (the cached increments of) that view. In other words, it depends on the broadcast frequency of AC_k. Practically, this implies that the number of clients bound by each view is more crucial than the number of clients actually subscribed to it. Therefore, instead of the popularity, a new metric based on the number of clients bound by a view can better quantify the "importance" of the view with respect to the air-cache performance. We have defined such a new metric which we call the **weight** of a view. If we number the views in increasing order of popularity, i.e., $r_1 \leq r_2 \leq \ldots \leq r_C$, and let d_k be the number of clients bound by view V_k, then we define the weight w_k of view V_k to be

$$w_k = \max\{d_1, d_2, \ldots, d_k\}$$

Essentially, the weight of a view actually reflects the number of clients bound by it, with the additional constraint that no view should have weight smaller than any less popular view. Notice that if the d_k's are non decreasing then $w_k = d_k$ for all k. In the extreme case where every client subscribes to only one view, we have $w_k = d_k = r_k$ for all k. Also note that, in terms of implementation, computing the weights requires two scans of the clients subscription vectors to compute the r_k's and the d_k's, followed by one iteration over the d_k's.

These weights, being better "importance quantifiers" in this case, are used replace the popularities in the square root rule. This means that instead of Equation 6.1, the air-cache is governed by the following rule

$$\frac{f_k}{f_m} = \sqrt{\frac{w_k}{w_m}}\sqrt{\frac{s_m}{s_k}} \tag{6.2}$$

This last rule also dictates the selection of the bandwidth factors. It can be easily shown that the composite air-cache follows this desired behavior if, for every view V_k, we set the bandwidth factor to be $\phi_k = \sqrt{w_k s_k}$.

6.2.2 Sharing the Server

Besides the broadcast bandwidth, view managers also share the server resources. They compete for the total pull capacity of the system in order to accommodate air-cache misses generated by the clients. Each manager services misses related only to its own view.

The approach we take here is to apportion the pull capacity of the system according to number of clients each manager has to deal with. In other words, we allocate to each M_k a share proportional to the number of subscribers r_k to view V_k. This share is the minimum guaranteed to each manager. The actual share may be occasionally higher, since not all managers consume the full of their share at all times. For example, a manager will not be receiving any misses if it decides to keep latest version of its view air-cached.

As in previous chapters, the resources that define the pull capacity of the system are the processing time of the server and the unicast bandwidth. For both of these, we need management techniques that can actually yield the desired sharing effect. Processing time can be distributed to view manager processes by an appropriate CPU scheduling policy [SG94]. Alternatively, in a multi-threaded system, we can achieve the same effect by assigning a proper number of threads to each manager. On the other hand, the sharing of the unicast bandwidth is controlled by the unicast multiplexer. This module, similarly to the air-cache composer, it implements a fair queueing algorithm to propagate packets from the view manager to the clients. As a final remark, the experimental results showed that the system is not very sensitive to the allocation of the server resources. Thus, any technique that just grossly approximate the desired allotment could be employed for this purpose.

6.2.3 Adaptation

In this section we present the technique that each view manager employs to control its air-cache. Basically, we will describe how it decides to move view versions or increments in to and out of the cache. As usual, the performance goal is to be broadcasting the minimum amount of information, while making sure that the server does not get swamped by client requests. At the same time, we exploit cache misses to assess the workload, i.e., the clients needs.

The performance metric that each manager uses to regulate its air-cache is the probability of it being idle. The **idle probability** p_k of view manager M_k corresponds to the portion of time M_k does not have any request to process or any reply to transmit. Basically, this is an indication of how busy the manager is, and is a measure of the request arrival rate compared to the manager's service rate. If $p_k = 1$ then the manager is idle, meaning that it does not receive any requests. At the other end, if $p_k = 0$ then the manager is always busy because requests arrive at a rate it cannot keep up with. Any value in between, indicates that the manager does get some requests, but at a safe rate that does not exceed its capacity.

According to the general air-caching principle, a view manager should be neither too busy nor idle. Driven by this principle, each manager periodically executes an algorithm that attempts to avoid both situations by properly adapting the contents of its air-cache. Basically, the algorithm uses the current estimate of the idle probability p_k and decides whether data should be added to or removed from the air-cache. When $p_k = 1$ (i.e., when the manager is not getting any requests) it considers removing (increments of) the view from its air-cache, since it may be capable of handling the extra misses. On the other hand, when $p_k = 0$ (i.e., when the manager is very busy servicing requests) it air-caches more data in order to reduce the misses. If p_k lies somewhere in between, we conclude that the manager is busy enough, and the air-cache is left unmodified.

The exact algorithm depends on the refresh method of the view it manages. In Figure 6.2.3 we present two faces of the algorithm, one for recomputed views (Figure 6.2(a)) and one for incrementally maintained views (Figure 6.2(b)). Their differences stem from two facts: First, when it comes to air-caching decisions, a manager has different options for each type of view. For recomputed views, it can only choose either to cache or not to cache the entire view. With incremental views, it has to decide how many—if any—increments to cache. Second, when a recomputed view is air-cached, clients do not send any miss for it, and thus, the

```
if $V_{k,i} \in AC_k$ then                          /* Is $V_{k,i}$ air-cached? */
  if $\lambda_{k,i} < \mu_k$ then            /* Can $M_k$ handle misses for $V_{k,i}$? */
    Remove $V_{k,i}$ from $AC_k$
  end if
else if $p_k = 0$ then                    /* Is $M_k$ getting too many misses? */
  Air-cache $V_{k,i}$
end if
```

(a) Recomputed views

```
/* If $j \leq i$ then $AC_k$ contains increments $\delta_{k,j}, \dots, \delta_{k,i}$ */
/* If $j = i + 1$ then $AC_k$ is empty */
if $p_k = 1$ then                                        /* Is $M_k$ idle? */
  if $j \leq i$ then           /* Is there at least one increment in $AC_k$? */
    if $\lambda_{k,j} > \mu_k$ then            /* Can $M_k$ handle misses for $\delta_{k,j}$? */
      Remove $\delta_{k,j}$ from $AC_k$
    end if
  end if
else if $p_k = 0$ then                    /* Is $M_k$ getting too many misses? */
  repeat                                      /* Air-cache more increments */
    $j = j - 1$
    $p_k = p_k + \lambda_{k,j}/\mu_k$
    Air-cache $\delta_{k,j}$
  until $p_k > 1$
end if
```

(b) Incremental views

Figure 6.2: Adaptive algorithms for view air-caching

manager is idle. For incremental views, the manager may be getting misses for old increments even when several increments are in the air-cache.

The inputs to both versions of the algorithm are:

- The status of the view, i.e., the most recent version $V_{k,i}$ or increment $\delta_{k,i}$.
- The contents of the air-cache.
- The current idle probability p_k of manager M_k.
- The average service rate μ_k of manager M_k. This is dynamically computed as a moving average over the recently processed air-cache misses.
- The workload for view V_k. This is expressed as the estimate of the rate $\lambda_{k,i}$ at which each version $V_{k,i}$ or each increment $\delta_{k,i}$ gets requested by clients. Note that these are the total request rates and should account both air-cache hits and misses. In the following section, we discuss how a manager can get these estimates only from the misses.

The adaptive steps are based on the queuing theory result that the idle probability for M/M/1 systems is $1 - \lambda/\mu$, where λ/μ is the traffic intensity of the server [Jai91]. In our case, this means that requests arriving at a rate $\lambda_{k,i}$ add to the traffic intensity of manager M_k a factor of $\lambda_{k,i}/\mu_k$, and therefore, decrease its idle probability by $\lambda_{k,i}/\mu_i$. Using this result, both versions of the algorithm try to appraise the effect of an air-caching decision before committing to it.

Beyond these adaptive algorithms, there is one more issue related to the management of the air-cache in this case: What happens when new view versions are generated? Here, we follow the "safer" for the manager approach to directly air-cache new view versions or new increments, even if the corresponding air-cache is currently empty. For recomputed views, if the current version is air-cached it is immediately replaced by the new one in the air-cache. For incremental views, the new increment is just added to the air-cache. In this way, all new data start being disseminated through the broadcast. Thereafter, the manager will decide the proper dissemination method in the regular way we just described.

6.2.4 Workload Estimation

In the air-caching context, the problem of estimating the workload refers to drawing conclusions about the hits from the available misses. Here, a view manager gets requests for view versions or increments not in the air-cache, but has no information about accesses for those in the air-cache. Therefore, while it can accurately compute the request rates for the former, it needs to guess the rates for the latter.

This is where a view manager is not alone; it can turn for help to the other managers. Other managers may be able to help because misses they collect for their own views can convey useful information about hits of that manager. We make this possible through a simple trick. We assume that the request rate for view version $V_{k,i}$ or increment $\delta_{k,i}$ is $\lambda_{k,i} = r_k \bullet \mathrm{H}(t_{k,i})$, where r_k is the number of subscribers to view V_k, and H is a some global function that captures the sleeping habits of the clients with respect to the age of $V_{k,i}$ or $\delta_{k,i}$. With this assumption, if we know function H, we can compute the request rate for any view version based only on its creation time.

Obviously, our problem is that we do not know what the function H is. We do know, however, the value of H at some points. This is because miss rates should fall under this model as well. If, for example, the measured miss rate for a non cached increment $\delta_{k,i}$ is $\lambda_{k,i}$ then we can conclude that $\mathrm{H}(t_{k,i}) = \dfrac{\lambda_{k,i}}{r_k}$. Each manager may compute the value of H for one or more points from client misses. Furthermore, if we compile these values from all the managers, we can have enough sample points to derive a good model for the function H. In some sense, managers can cooperate to derive a model of the workload.

Practically, this means that we can get a rather accurate estimate of the function's value at any point. If t_{max} and t_{min} are the most recent and the oldest sample points respectively, we can compute the value of H(t) for any t as follows:

- If $t < t_{\text{min}}$ then $\text{H}(t) = 0$, meaning that old versions or increments do not get requested.
- If $t_{\text{min}} \leq t \leq t_{\text{max}}$ we compute the value of $\text{H}(t)$ with polynomial interpolation over the sample points [PTV93].
- If $t > t_{\text{max}}$ then $\text{H}(t) = t_{\text{max}}$. This selection is driven by the fact that the function H must be non decreasing, because generally $\lambda_{k,i} \geq \lambda_{k,i-1}$ (a client that needs increment $\delta_{k,i-1}$ definitely needs increment $\delta_{k,i}$ as well)

This way, a view manager can get an estimate of the request rate for any air-cached version or increment. We must note that such estimates are required only for air-cached data. For the rest, a manager knows exactly what the request rate is. Consequently, estimates are used only when a manager is considering to remove data from the air-cache. Hence, a manager does not have to blindly take data off the air-cache. Instead it uses these estimates as "hints" to take more informed and cautious decisions. Essentially, this technique eliminates the need for probing, as it was used in Chapter 4.

6.2.5 Air-cache indexing

To complete the picture of the proposed system, in this section we present the air-cache indexing technique. Basically, in this case the index is a list of C entries, one for each view. Each entry contains the view identifier, the current version number of the view, the refresh method (recomputed or incremental), and the number of versions or increments currently in the air-cache. Table 6.1 shows index entries for both incremental and recomputed views, along with their possible values.

	Recomputed Views	Incremental Views
View ID (k)	$1, 2, \ldots, C$	$1, 2, \ldots, C$
Current Version Number (i)	$1, 2, \ldots$	$1, 2, \ldots$
Air-Cached Versions/Increments	$0, 1$	$0, 1, \ldots, i$
Refresh Method	R	I

Table 6.1: Index entry for view V_k

It is possible that no data are air-cached for certain views. In this case the index is used just to inform the clients about the most recent version of the view in the server. If it is more recent than the one cached at the client, the client has to send a request to the server for the new version or increments of the view.

Besides the above information, the index could also include precise timing data about when each increment is scheduled to be broadcast so that clients can selectively tune in the channel, and thus conserve energy [IVB94a]. However, the details of such an implementation are beyond our current scope.

More important for our work is the repetition frequency of the index in the broadcast channel. As we explained in Section 3.3.5, there is a performance trade-off involved in the selection of this design variable. Broadcasting the index very often works in favor of misses (clients can detect them quickly) but against hits (broadcast data are delayed more). Here, to battle this tradeoff, we exploit the composite air-cache structure to implement a tunable scheme for broadcasting the index. Specifically, we regard the index as a special view that is permanently stored in its own simple air-cache. This air-cache, called $AC_{\rm I}$, always contains one item, the index, and gets multiplexed in the existing composite air-cache. In this way, the repetition frequency of the index is now regulated the broadcast bandwidth allocated to $AC_{\rm I}$.

In order to enforce a regular repetition frequency for the index, and avoid variations or side-effects from changes in the other air-caches, we assign a fixed portion β of the bandwidth to $AC_{\rm I}$. This portion is a basic design parameter of the system. It controls the average probe wait, i.e., the time clients have to wait for the first appearance of index, and the overhead of the index itself for the broadcast. With our technique, both these can be easily computed analytically and predicted at system design time. If $s_{\rm I}$ is the size of the index (and the size of $AC_{\rm I}$) then the average probe wait is $\frac{s_{\rm I}}{2\beta}$ broadcast units. The index overhead is $\frac{\beta}{1-\beta}$ which means that, for every byte of information in the broadcast program there are $\frac{\beta}{1-\beta}$ bytes of index.

6.3 Experiments

In this section we present a representative set of experimental results that validate the proposed methodology. Again, these results were drawn form a detailed simulation model of the system. This model consists of one server that maintains C views, and K mobile clients that subscribe to some of these views. Let us describe each part of the simulation in more detail. Bear in mind that data sizes are given in terms of some abstract **size unit**, and time is measured in **broadcast units**, i.e., the time it takes to broadcast an item of unit size.

Views: In order to model views with different characteristics, we partition them into **view groups** and create each view according to the specifications of the group it belongs to. A view group is specified in terms of the five parameters shown in Table 6.2. The first two parameters define the name of the group, and the number of views that belong to the group (out of a total of C views). The third parameter indicates the method used to refresh the view with new updates (recomputation or incremental). Recall that the refresh method for each view is fixed throughout every experiment. *RefreshDistr* describes the distribution of time intervals between refreshes of the view. It can be set to constant for views that are periodically updated, or some other distribution (e.g., exponential) if updates happen randomly. For each distribution the necessary parameters (e.g., mean, variance) need to be specified. The last parameter, *SizeDistr*, defines the size of the result of a refresh. For recomputed views, this corresponds to the size of new versions. For incrementally maintained views, *SizeDistr* determines the size of each new increment.

Clients: Similarly to views, clients are also partitioned in **client groups** to model a diverse population. Again, five parameters are used to describe each group (Table 6.3). *Name* is the name of the client group. *ClientsPortion* specifies the portion of clients that belong to the group. Here, we use the portion instead of the actual number of clients to facilitate the scaling of an experiment to different client population sizes. The third parameter, *SleepDistr*, expresses the "sleeping

Parameter :	Description
Name :	Name of the group
NumViews :	Number of views in the group
RefreshMethod :	Recomputation or Incremental
RefreshDistr :	Distribution of new version generation time
SizeDistr :	Distribution of view/increment size

Table 6.2: View group specification

habits" of clients, i.e., the distribution of their disconnection periods. The last two parameters specify the interests of the clients in the group. *SubscrSizeDistr* defines the number of views a client subscribes to. If set to constant, all clients in the group subscribe to the same number of views. The last one dictates which views a client subscribes to. For our experiments, we used this parameter to relate client groups to view groups. Specifically, *SubscriptionDistr* is defined as a list of pairs (X, Y), each indicating that a portion Y of a client's subscriptions are for views in group X. Given the specifications of the groups, each client generates a subscription vector according to the distributions of its group. This vector is generated once, and remains the same throughout each experiment.

Server: The simulation model for the server implements the architecture shown in Figure 6.1. Following the pattern of the last two chapters, we select the unicast bandwidth to be equal to the broadcast bandwidth. For implementation simplicity and comparison purposes, we assume that the server throughput is bound by the unicast bandwidth. Besides the bandwidth, there are two parameters that affect the operation of the server. The first is the portion β of bandwidth assigned to

Parameter :	Description
Name :	Name of the group
ClientsPortion :	Percent of clients belonging to the group
SleepDistr :	Distribution of sleep time
SubscrSizeDistr :	Distribution of number of subscriptions
SubscriptionDistr :	Distribution of subscriptions

Table 6.3: Client group specification

the broadcast index, as it was described in the previous section. For the results presented herein, we set $\beta = 0.02$, meaning that the index increases broadcast response times by 2%. The other is the adaptation period which determines how often each manager re-examines the performance of its air-cache. This period is the same for all managers. Nevertheless, we do not allow them to adapt all together at the same time. Instead, we choose to spread the adaptation phases of different managers over time. There are two reasons why we prefer this approach. First, the computation load is portioned out over time, and second, we avoid abrupt changes in the state of the system, and reduce the risk of unstable behavior by limiting the number of changes that can occur at any point in time.

6.3.1 Weight-based Bandwidth Allocation

The first experiment we present was designed to demonstrate the merit of weight-based broadcast bandwidth allocation to the individual air-caches. As it was discussed in Section 6.2.1, the bandwidth allocated to each air-cache controls the repetition frequency of data cached in it.

This experiment deals only with the structure and the performance of the air-cache, and has no effect on the hybrid part of the system. Thus, we created a static (non-adaptive) scenario where clients retrieve data only from the broadcast channel. In this scenario, there are 100 views which are partitioned into two groups. Views in these two groups are identical except for their popularity. They are all incrementally maintained, and all generated increments are of size 1. In addition, we force each view manager to always air-cache the last 5 increments of its view so that all air-caches have the same size. Every time a client wakes up, it retrieves all the cached increments of the views it has subscribed to. This, in conjunction with the fact that we consider only broadcast delivery, renders the actual number and sleeping habits of the clients irrelevant for this experiment.

In terms of popularity, we used a 80/20 rule to differentiate the two view groups. In other words, clients subscribe in a way that 80% of their subscriptions are for 20% of the views. The 20 popular views form the first group which is called PV; the other 80 form the second which is called UV. Within each group, views are equally popular.

Name	PO	PU
ClientsPortion	α	$1-\alpha$
SubscrSizeDistr	Constant(10)	Constant(25)
SubscriptionDistr	$[(\mathrm{PV}, 1)]$	$\left[\left(\mathrm{PV}, \frac{20-22\alpha}{25(1-\alpha)}\right), \left(\mathrm{UV}, \frac{5-3\alpha}{25(1-\alpha)}\right)\right]$

Table 6.4: PO and PU client groups

In order to demonstrate the differences between weight-based and popularity based bandwidth allocation, we keep the view popularities fixed throughout the experiment, and modify only the number of clients bound by each view. For this purpose, we define two client groups. The first group, called PO (Popular Only), consists of very selective clients that subscribe only to 10 of the popular views. The second group, called PU (Popular/Unpopular), consists of less selective clients that subscribe to 25 views, both popular and unpopular. The specification of these groups is shown in Table 6.4. The values of some parameters in this table are given in terms of variable α which determines the mixture of the client population. Out of total K clients, αK belong to group PO, and the rest belong to PU. By varying this variable, we essentially vary the number of clients bound by each view.

Note that, for maintaining the 80/20 ratio for all values of α, the subscription distribution of PU clients must also be a function of α. This is better explained in Figure 6.3. The area of the "L" shaped polygon represents all the clients subscriptions. The left (tall) part of it corresponds to subscriptions for views in PV. The right (flat) part corresponds to subscriptions for views in UV. Also, the whole area is divided in two parts: one for the subscriptions of PO clients, and one for the subscriptions of PU clients. The dividing line is anchored by the variable α, which ranges from 0 to 0.91. Note that, because PO clients do not subscribe to unpopular views, a 80/20 popularity ratio cannot be achieved unless at least $0.09K$ clients are in the PU group.

Figure 6.4 presents the performance of the air-cache with respect to the variable α. The performance metric we use is the average refresh time, i.e., the average time clients have to spend monitoring the broadcast in order to retrieve all the data they need. The three lines plotted in this figure correspond to three different policies for allocating bandwidth to the individual air-caches. The "Flat" policy assigns exactly the same amount of bandwidth to all air-caches. The "Popularity-based" uses the original square root rule, i.e., Equation 6.1. Last, the "Weight-based"

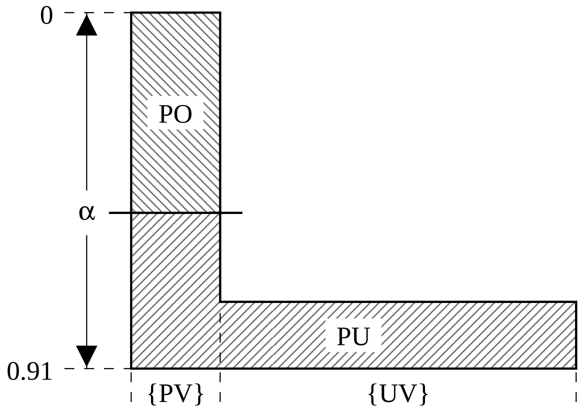


Figure 6.3: PO/PU client group mixture for 80/20 view popularities

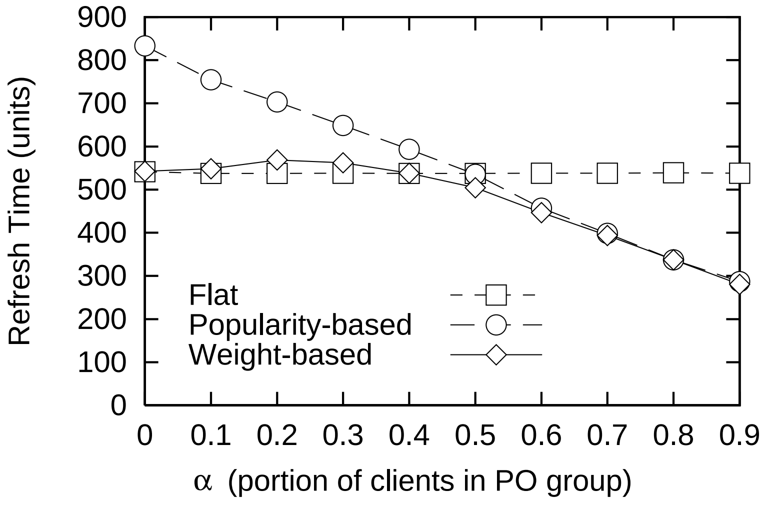


Figure 6.4: Performance of bandwidth allocation policies

policy uses the weights of the views according to Equation 6.2.

Let us first compare the first two policies. A first observation is that under the flat policy the air-cache performs the same for all values of α. This was expected since this policy is not considering client subscriptions at all. A rather surprising result is that, in some cases, the flat policy performs better than the popularity-based, even though there are significant popularity discrepancies among the views. This occurs on the left part of the graph (small values of α) where most clients belong to the PU group, and is explained by the fact that these clients are bound by unpopular views. As a result, no matter how often the popular views get broadcast, most client have to wait for some of the unpopular ones. In fact, broadcasting the popular views more often makes things worse by delaying the rest even more; thus the bad performance of the popularity-based policy. Therefore, when the performance of most clients depends on unpopular views the flat policy appears to be a better choice.

Things change, however, as the percentage of PO clients increases. As we move towards the right part of the graph, we notice that the popularity-based policy is winning over the flat policy. The reason is that for large values of α there are many clients bound only by popular views. These are the clients that can benefit from non-uniform repetition frequencies and favorable treatment of popular views. Hence, in such cases popularity-based bandwidth allocation is preferable.

Clearly, none of these two policies is an all-around winner. However, the third policy seems to be one. Figure 6.4 shows that weight-based allocation is, in all cases, at least as good as any of the other two. It captures the performance dependencies of client subscriptions and yields (almost) the best refresh times across the range of the graph. For small values of α, it recognizes that, no matter how popular, no view is particularly important with respect to the performance of the air-cache. On other hand, it also recognizes cases (large α) where view popularities can indeed have an impact on the performance, and allocates bandwidth accordingly. Note that for $0.1 < \alpha < 0.35$, it is not performing as good as the flat policy. This is because it does allocate some extra bandwidth to some of the views, even though a flat scheme is still a better choice. However, even there the difference is too small to raise any serious concern. This means that the weight-based policy is still the safest choice for allocating broadcast bandwidth to individual air-caches over any mixture of clients and subscription interests.

6.3.2 Scalability & Adaptation

In this section we test the scalability and the adaptiveness of the proposed system. Here, we create again 100 views that follow the 80/20 popularity ratio. This time, however, in order to test all the aspects of the system, not all the views are incrementally maintained. Half of them are completely recomputed every time a new batch of updates arrives at the server. In total, we define four view groups which correspond to the four combinations of popularity and refresh method, as shown in Table 6.5. There are 10 popular incremental views (PI), 10 popular recomputed (PR), 40 unpopular incremental (UI), and 40 unpopular recomputed (UR). For all views, the interval between refreshes are exponentially distributed with mean 5000 units.

Clients are split into two groups similar to those used in the previous section. The specifications of the groups are given in Table 6.6. This time half of the clients are in the PO group and the other half are in the PU group. The disconnection times of all clients follow a normal distribution with mean and standard deviation

Name	PI	PR	UI	UR
NumViews	10	40	10	40
RefreshMethod	I	R	I	R
RefreshDistr	Exp(5000)	Exp(5000)	Exp(5000)	Exp(5000)
SizeDistr	Normal(2, 0.4)	Normal(5,1)	Normal(2, 0.4)	Normal(5,1)

Table 6.5: View groups for adaptation & scalability experiment

10000 and 2000 units respectively. The distribution of subscriptions is selected properly (parameter *SubscriptionDistr*) to yield the desired 80/20 popularity ratio.

The main result of the experiment is presented in Figure 6.5. In this graph we show the performance of the system, i.e., the average clients refresh time, as a function of the number of clients which ranges from 500 to 20000. We show the results for two configurations. The first ("Adapt on Hits") corresponds to the ideal, but unrealistic, scenario where all the information about client requests is given to server. In this way, the server adapts based on a fully informed model of the workload. The second ("Adapt on Misses") corresponds to the normal operation of the system where the server relies solely on misses for managing the air-cache.

Name	PO	PU
ClientsPortion	50	50
SleepDistr	Normal(10000, 2000)	Normal(10000, 2000)
SubscrSizeDistr	Constant(10)	Constant(25)
SubscriptionDistr	$[(\mathrm{PI}, 0.5), (\mathrm{PR}, 0.5)]$	$[(\mathrm{PI}, 0.36), (\mathrm{PR}, 0.36),$ $(\mathrm{UI}, 0.14), (\mathrm{UR}, 0.14)]$

Table 6.6: Client groups for adaptation & scalability experiment

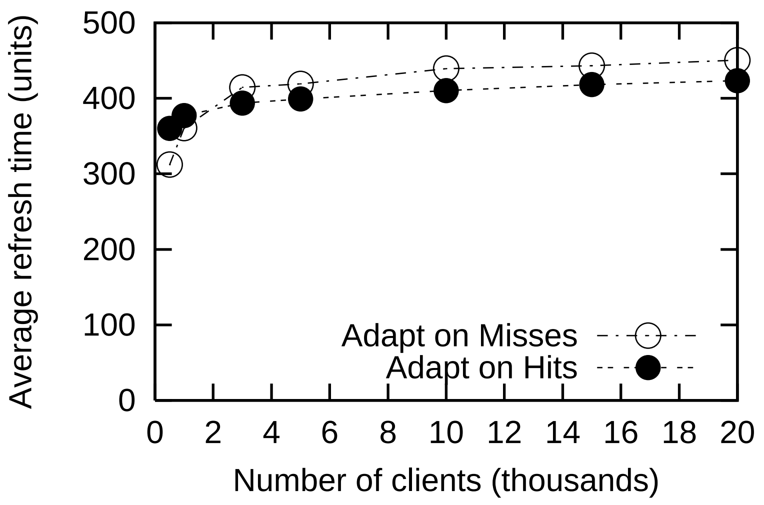


Figure 6.5: Scalability and adaptability

These results lead to two important conclusions, but also reveal an odd, at a first look, behavior. First of all, here again air-caching yields a very scalable data dissemination system, in the sense that average refresh time of the clients is affected very little by the size of the population. The server detects and exploits the commonality among the clients needs to service thousands of them with an effective combination of the two delivery methods. In fact, the larger the number of clients, the greater this commonality and the bigger the benefit of air-caching. Second, by comparing the two curves of the graph, we see that adaptation based on the misses is almost as good as the ideal case of adaptation based on the hits. The average refresh time for the former configuration is at most 7% worse than the latter. This means that the workload estimation procedure described in Section 6.2.4 is very effective. The difference in performance is attributed to the fact that this procedure actually slightly overestimates the workload. Thus, it forces the managers to air-cache data for a little more than necessary, increasing the average size of the air-cache (up to 7%), and consequently, the average refresh time of the clients.

The odd result is that, in small scale, the system appears to be performing better when it has limited information (misses) than when it has all the information (hits) about the clients' requests. This strange behavior is explained by the fact that, under small request loads, the refresh time of the clients is mostly affected by the responsiveness of the server, and not the latency of air-cache. Let us elaborate on this subtle issue. When there are not many clients in the system, view managers air-cache little data since the server can handle a significant portion of their requests. As a result, the air-cache is rather small, meaning that its latency is also small. At the same time, most of the client refreshes (almost up to 100% for 500 clients) are hybrid, in the sense that not all of the needed data are downloaded from the air-cache; at least some are retrieved from the server. As the air-cache latency is small, many of these hybrid refreshes may be delayed by the unicast delivery of the data not found in the air-cache. But, unicast delivery time depends on the load imposed on the server. This is exactly where miss-based adaptation "coincidentally" wins over hits-based adaptation. As we discussed above, when relying on the misses, the workload is overestimated. This makes view managers to air-cache more data in anticipation of the supposedly higher client demands. However, as the actual demand is lower, a smaller number of requests reach the server. Consequently, the server ends up handling a lighter load of requests which, naturally, are serviced faster.

Above, we characterized the system as very scalable based on the observation that it performs about the same for any number of clients. Nonetheless, the results presented in Figure 6.5 are not at all enlightening as to how good this performance is. The problem is this case is that we do not have a comparison measure like, for example, a theoretical optimal performance similar to those we used in the previous chapters. Hence, we cannot directly verify the efficiency of the proposed system. We can, however, resort to indirect indications to draw some insightful conclusions. These are some related performance metrics as observed in our experiments:

- Each broadcast message was, on average, useful to 1 out of every 7.25 awake clients. If, say, 1000 clients were awake at some point a broadcast message would have on average 138 recipients. This means that the broadcast bandwidth is used very effectively for data that are indeed popular.

- On average, only about 5% of a client's refresh time was wasted on scanning "old" data in the air-cache, i.e., data that the client already had retrieved in a previous refresh. This is an indication that view managers can successfully detect when the popularity of views or increments drops, and avoid keeping old data in the air-cache.

- The average utilization of the server ranged between 0.81 and 0.89. In other words, the server was getting almost as many misses as it could handle, which is in accord with the general air-caching principle. This also suggests that the managers were not air-caching more data than they had to.

- Last, but more importantly, at the highest scale the hybrid system is achieving an effective data throughput about 70 times its nominal capacity. This means that it would require at least 70 times the resources (bandwidth and/or processing power) to achieve similar levels of performance with a traditional pull-only system. Instead, once again, the air-cache mechanism detects and exploits the commonality of client demands, making a very effective use of the broadcast capability.

6.4 Conclusions

In this chapter, we proposed to use the air-cache mechanism to implement publish/subscribe services for mobile users. We considered cases where users provide coarse subscription profiles, specifying their general interests. For example, these can be sets of view definitions over a central data warehouse, or selections of information channels like, say, world news. The server is responsible for delivering to them any published data that match these general profiles, such as new view versions generated in the warehouse, or breaking news stories from around the world. Being mobile, the clients operate most of the time in sleep mode. From time to time, they come on-line and wish to retrieve pertinent new data. This requires getting all newly published information that matches their profiles, and further personalize it by filtering out exactly what is of interest to them.

Our approach was to define a set of independent managers, each managing a separate view or information channel. Each manager is allocated a part of the system resources and takes over the task of disseminating the data it is responsible for. For that, it creates its own simple air-cache to cache data as it deems necessary, as well as service misses for data kept outside this air-cache. All these air-caches are multiplexed in the same broadcast channel to form a composite air-cache. For that purpose, we proposed a bandwidth allocation scheme that regulates the use of the broadcast channel based on the client subscriptions. We also explained how the managers share the server resources for servicing cache misses. Under this approach, each manager decides how to best use its air-cache independently from the other managers. Then, we presented the algorithm that managers use to adapt their air-caches. The algorithm relies on a manager's idle probability which is a measure of the miss load it receives with respect to its service rate. Last, we proposed a method for managers to "share" information from misses and help each other derive good estimates about hits, i.e., estimate the request rates for cached data.

The experimental results showed that air-caching can serve as a very effective data dissemination mechanism in this case as well. First, we showed that the proposed bandwidth allocation policy can correctly evaluate the effects of clients subscriptions to the performance of the air-cache, and distribute the broadcast bandwidth appropriately, under any mixture of client interests. We also demonstrated the adaptiveness and scalability of this technique. The air-cache mechanism was able to detect the commonality among user needs, and efficiently deliver data in very large scale (in our experiments up to 20000 clients, with effective throughput 70 times the nominal capacity of the system).

In closing, we must note that, beyond these significant performance results, this chapter is an example of how multiple (almost independent) air-caches can be implemented in one broadcast channel. We believe that similar techniques can be used to combine different data services, possibly from different servers, in an integrated manner.

Chapter 7

Conclusions

Over the last few years there has been an astonishing increase of the demand for on-line data services. An exponentially growing number of people are using the Internet, craving for access to all kinds of information. This trend imposes a heavy data distribution load on the information infrastructure. Internet resources, i.e., network bandwidth and data servers, often fail to carry this load, exposing the scalability limitations of data services. A major source of the problem is that typically these services employ the request/response (pull/unicast) data delivery model which scales at best linearly with network bandwidth and server capacity.

To overcome these problems, this thesis capitalizes on the asymmetric nature of information-centered applications and the broadcast capabilities of emerging communication networks, and proposes adaptive hybrid data delivery as the basis of highly scalable data dissemination services, responsive to dynamic and unpredictable user demands. Adaptive hybrid data delivery refers to the dynamic integration of two data delivery mechanisms, namely the traditional request/response (or pull/unicast) and the rather novel push/broadcast.

In this thesis, we first contrasted the two basic mechanisms, and discussed their advantages and disadvantages. We argued that these can be combined in a synergistic manner, and made a case for adaptive hybrid data delivery by presenting its potential performance and scalability benefits. Then, we introduced the concept of air-caching, i.e., the temporary storage of popular data in a broadcast channel through repetitive transmissions. Requests for data in the air-cache (i.e., air-cache hits) are satisfied without contacting the server. The rest (i.e., air-cache misses) are serviced by the server. The air-cache serves as an abstract vehicle for pursuing and implementing adaptive hybrid data delivery because it disguises the problem as a cache management problem. We identified the special properties of this new type of caching, discussed its performance goals, and laid its basic management principles. The general goal of a hybrid system is to air-cache popular data expecting to satisfy the bulk of the clients' demands so that only a small number of requests (for unpopular data) are left to be serviced by the server itself. A unique characteristic of the air-cache is that, contrary to typical caches, it must be managed relying exclusively on cache misses because the server does have at its disposal any information about cache hits.

Based on that, we presented three sets of algorithms and techniques for using and managing the air-cache in three different applications. First, we considered the problem of servicing data requests over heavily accessed databases. Assuming that these requests exhibit high degrees of skewness towards parts of the database, we proposed a technique that dynamically detects and air-caches the hot-spots, even when these are rapidly changing. This approach uses the expected performance marginal gains and data temperature probing to effectively balance the two delivery modes.

The second application was the propagation of data updates from a central repository to several mobile, often disconnecting, clients. We described a hierarchical version of the air-cache, that adds the flexibility of multiple access latencies. In this case, we proposed techniques that detect the (dis)connection pattern of the clients, and establish their needs for updates. Based on this pattern, the server air-caches recent updates in a way that matches the clients' needs. We also introduced the notion of soft air-cache misses, i.e., misses for cached data, that allow clients to improve response time over broadcast delivery.

Last, we tackled the deployment of publish/subscribe services, again in the context of mobile computing. We considered a population of mobile clients that subscribe to multiple, semantically different, data services, and a server that employs air-caching to disseminate newly published information. In this case, our approach used multiple air-caches, one for each of the provided services. We proposed a composite air-cache structure that multiplexes all these simple air-caches in a single broadcast channel, and developed techniques for managing the air-caches independently from each other.

All the proposed algorithms were validated with experimental results drawn from a detailed simulation model. In all cases, the results demonstrate the scalability, adaptiveness, and efficiency of air-caching. We showed that the proposed algorithms can very effectively detect and air-cache the data hot-spots relying only the cache misses, even under fast changing access patterns. As a result, the performance of hybrid systems is not directly affected by the volume of the workload, but instead it depends on the amount of frequently requested data (i.e., the size of the hot-spot) which is a function of the data access distribution. This means that, under highly skewed workloads, such a system can exploit the commonality of among clients needs, and use the broadcast capability very efficiently to yield an effective data throughput many times higher than its nominal capacity.

7.1 Future Work

We believe that the results of the thesis have far reaching implications, as they suggest an effective way of deploying large scale wide area information systems. Therefore, there is a lot of interesting research work to be done in the future. In the following, we highlight some possible directions.

First of all, air-caching could be employed in more applications, with different data dissemination requirements. For instance, an interesting area to look into is "on-time" data delivery and, more generally, real time applications. Also, related to these are the problems imposed by the delivery of multimedia content, for applications like video-on-demand.

Several interesting problems are surfacing if we relax one or more of the basic assumptions made in this thesis. For example, in our work we assumed that data transmission is error-free. Often, this is not a valid assumption in practice, especially in wireless communications. Thus, the proposed algorithms need to be extended to account for possible communication errors. Then, we also assumed that the broadcast and the unicast channels are independent, with a fixed amount of bandwidth allocated to each one. This assumption limits the decision variables and the complexity of the system. However, in settings where the two channels share the same link, there may be an performance advantage in taking a more dynamic approach. In other words, the bandwidth allocated to each channel could be an additional optimization parameter left to the discretion of the adaptive algorithms.

Furthermore, in our work we considered only two specific data delivery methods. However, as it was discussed in Section 3.2, there are more data delivery alternatives. Generally, each one serves a different purpose, and has its own advantages and disadvantages. While individually these have more or less been studied, the merit of other hybrid approaches are indeed worth investigating, similarly to the approach taken in this thesis.

BIBLIOGRAPHY

[AAFZ95] Swarup Acharya, Rafael Alonso, Michael J. Franklin, and Stanley B. Zdonik. Broadcast Disks: Data Management for Asymmetric Communications Environment. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 199–210, San Jose, California, May 1995.

[ABCdO96] Virgílio Almeida, Azer Bestavros, Mark Crovella, and Adriana de Oliveira. Characterizing Reference Locality in the WWW. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*, Miami Beach, Florida, December 1996.

[ABF$^+$95] Vivek Arora, John S. Baras, A. Falk, Douglas Dillon, and Narin Suphasindhu. Hybrid Internet Access. In *Proceedings of the 12th Symposium on Space Nuclear Power and Propulsion/Commercialization*, pages 69–74, Albuquerque, NM, January 1995.

[ABGM90] Rafael Alonso, Daniel Barbará, and Hector Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM Transactions on Database Systems*, 15(3), September 1990.

[Ach98] Swarup Acharya. *Broadcast Disks: Dissemination-based Data Management for Asymmetric Communication Environments.* PhD thesis, Computer Science Department, Brown University, 1998.

[AF98] Demet Aksoy and Michael Franklin. Scheduling for Large-Scale On-Demand Data Broadcasting. In *Proceedings of IEEE INFOCOM Confenerence*, San Francisco, CA, March 1998.

[AFZ95] Swarup Acharya, Michael J. Franklin, and Stanley B. Zdonik. Dissemination-Based Data Delivery Using Broadcast Disks. *IEEE Personal Communications Magazine*, 2(6), December 1995.

[AFZ96a] Swarup Acharya, Michael J. Franklin, and Stanley B. Zdonik. Disseminating Updates on Broadcast Disks. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pages 354–365, Mumbai (Bombay), India, September 1996.

[AFZ96b] Swarup Acharya, Michael J. Franklin, and Stanley B. Zdonik. Prefetching from Broadcast Disks. In ICDE [ICD96], pages 276–285.

[AFZ97] Swarup Acharya, Michael J. Franklin, and Stanley B. Zdonik. Balancing Push and Pull for Data Broadcast. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 183–194, Tucson, Arizona, May 1997.

[Air98] AirMedia. AirMedia Live. http://www.airmedia.com, 1998.

[AK93] Rafael Alonso and Henry F. Korth. Database System Issues in Nomadic Computing. In SIGMOD [SIG93], pages 388–392.

[AL80] Michel E. Adiba and Bruce G. Lindsay. Database Snapshots. In *Proceedings of the 6th International Conference on Very Large Data Bases*, pages 86–91, Montreal, Quebec, Canada, October 1980.

[AW85] Mostafa H. Ammar and John W. Wong. The Design of Teletext Broadcast Cycles. *Perfomance Evaluation*, 5(4):235–242, December 1985.

[AW87] Mostafa H. Ammar and John W. Wong. On the Optimality of Cyclic Transmission in Teletext Systems. *IEEE Transactions on Communications*, 35(1):68–73, January 1987.

[AW97] Martin F. Arlitt and Carey L. Williamson. Internet Web Servers: Workload Characterization and Performance implications. *IEEE Transactions on Networking*, 5(5):631–645, October 1997.

[BB97] Sanjoy Baruah and Azer Bestavros. Pinwheel Scheduling for Fault-Tolerant Broadcast Disks in Real-time Database Systems. In ICDE [ICD97], pages 543–551.

[BDD$^+$98] Randall G. Bello, Karl Dias, Alan Downing, James Feenan, William D. Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. Materialized Views in Oracle. In *Proceedings of the 24th International Conference on Very Large Data Bases*, pages 659–664, New York, NY, USA, August 1998.

[BG96] Gordon Bell and Jim Gemmell. On-ramp Prospects for the Information Superhighway Dream. *Communications of the ACM*, 39(7):55–61, July 1996.

[BGH$^+$92] Thomas F. Bowen, Gita Gopal, Gary E. Herman, Takako M. Hickey, K. C. Lee, William H. Mansfield, John Raitz, and Abel Weinrib. The Datacycle Architecture. *Communications of the ACM*, 35(12):71–81, December 1992.

[BI94] Daniel Barbará and Tomasz Imielinski. Sleepers and Workaholics: Caching Strategies in Mobile Environments. In SIGMOD [SIG94], pages 1–12.

[Blu97] Marjory S. Blumenthal. Unpredictable Certainty: The Internet and the Information Infrastructure. *IEEE Computer*, 30(1):50–56, January 1997.

[BMSS96] C. Bisdikian, K. Maruyama, D.I. Seidman, and D.N. Serpanos. Cable Access Beyond the Hype: On Residential Broadband Data Services over HFC Networks. *IEEE Communications Magazine*, 34(11):128–135, 1996.

[BZ96] Jon C. R. Bennett and Hui Zhang. WF^{2}Q: Worst-case Fair Weighted Fair Queueing. In *Proceedings of INFOCOMM*, San Francisco, CA, March 1996.

[CDN$^+$96] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[Cha98] Arvola Chan. Transactional Publish / Subscribe: The Proactive Multicast of Database Changes. In SIGMOD [SIG98], page 521.

[Chi94] Tzi-cker Chiueh. Scheduling for Broadcast-based File Systems. In *MOBIDATA - NSF Workshop on Mobile and Wireless Information Systems*, Rutgers University, New Brunswick, NJ, November 1994.

[CTO97] Jun Cai, Kian-Lee Tan, and Beng Chin Ooi. On Incremental Cache Coherency Schemes in Mobile Computing Environments. In ICDE [ICD97], pages 114–123.

[CYW97] Ming-Syan Chen, Philip S. Yu, and Kun-Lung Wu. Indexed Sequential Data Broadcasting in Wireless Mobile Computing. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 124–131, Baltimore, Maryland, May 1997.

[DAW86] H. D. Dykeman, M. H. Ammar, and J. W. Wong. Scheduling Algorithms for Videotex Systems Under Broadcast Delivery. In *Proceedings of the 1986 International Conference on Communications*, pages 1847–1851, Toronto, Ontario, June 1986.

[DCK$^+$97] Anindya Datta, Aslihan Celik, Jeong G. Kim, Debra E. VanderMeer, and Vijay Kumar. Adaptive Broadcast Protocols to Support Efficient and Energy Conserving Retrieval from Databases in Mobile Computing Environments. In ICDE [ICD97], pages 124–134.

[DeJ97] Edmund X. DeJesus. The Pull of Push. *BYTE Magazine*, August 1997.

[Del93] Alexios Delis. *Client-Server Databases : Architectures and Performance Analysis.* PhD thesis, Department of Computer Science, University of Maryland, 1993.

[DMF97] Bradley M. Duska, David Marwood, and Michael J. Feeley. The Measured Access Characteristics of World-Wide-Web Client Proxy Caches. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 23–35, Monterey, CA, December 1997.

[DR92] Alexis Delis and Nick Roussopoulos. Performance and Scalability of Client-Server Database Architecture. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 610–623, Vancouver, Canada, August 1992.

[DR94] Alex Delis and Nick Roussopoulos. Management of Updates in the Enhanced Client-Server DBMS. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, Poznan, Poland, June 1994.

[DR98] Alex Delis and Nick Roussopoulos. Techniques for Update Handling in the Enhanced Client-Server DBMS. *IEEE Transactions on Knowledge and Data Engineering*, 10(3):458–476, May/June 1998.

[DYC95] Asit Dan, Philip S. Yu, and Jen-Yao Chung. Characterization of Database Access Pattern for Analytic Prediction of Buffer Hit Probability. *The VLDB Journal*, 4(1):127–154, January 1995.

[EH84] Wolfgang Effelsberg and Theo Härder. Principles of Database Buffer Management. *ACM Transactions on Database Systems*, 9(4):560–595, December 1984.

[FC94] Michael Franklin and Michael Carey. Client-Server Caching Revisited. In M. Tamer Özsu, Umeshwar Dayal, and Patrick Valduriez, editors, *Distributed Object Management.* Morgan Kaufmann, San Francisco, CA, 1994.

[Fra96a] Michael Franklin, editor. *Special Issue on Data Dissemination.* Bulletin of the Technical Committee on Data Engineering. IEEE Computer Society, September 1996.

[Fra96b] Michael J. Franklin. *Client Data Caching: A Foundation for High Performance Object Database Systems.* Kluwer Academic Publishers, Boston, MA, February 1996.

[FZ94] George H. Forman and John Zahorjan. The Challenges of Mobile Computing. *IEEE Computer*, 27(4):38–47, April 1994.

[FZ96] Michael J. Franklin and Stanley B. Zdonik. Dissemination-Based Information Systems. *IEEE Bulletin of the Technical Committee on Data Engineering*, 19(3):20–30, September 1996.

[FZ97] Michael J. Franklin and Stan Zdonik. A Framework for Scalable Dissemination-Based Systems. In *International Conference Object Oriented Programming Languages Systems (OOPSLA 97)*, Atlanta, GA, October 1997. (Invited Paper).

[FZ98] Michael Franklin and Stan Zdonik. "Data in Your Face": Push Technology in Perspective. In SIGMOD [SIG98]. (Invited Paper).

[GBBL85] David K. Gifford, Robert W. Baldwin, Stephen T. Berlin, and John M. Lucassen. An Architecture for Large Scale Information Systems. In *Proceedings of the Tenth ACM Symposium on Operating*

System Principles, pages 161–170, Orcas Island, Washington, December 1985.

[Gif90] David K. Gifford. Polychannel Systems for Mass Digital Communications. *Communications of the ACM*, 33(2):141–151, February 1990.

[Gla96] David Glance. Multicast Support for Data Dissemination in OrbixTalk. *IEEE Bulletin of the Technical Committee on Data Engineering*, 19(3):31–39, September 1996.

[Gol98] Leonard S. Golding. Satellite Communications Systems Move into the Twenty-first Century. *Wireless Networks*, 4(2):101–107, February 1998.

[GRC97] Syam Gadde, Michael Rabinovich, and Jeff Chase. Reduce, Reuse, Recycle: An Approach to Building Large Internet Caches. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 93–98, May 1997.

[Gup97] Himanshu Gupta. Selection of Views to Materialize in a Data Warehouse. In *6th International Conference on Database Theory*, pages 98–112, Delphi, Greece, January 1997.

[GWD94] Alex Gorelik, Yongdong Wang, and Mark Deppe. Sybase Replication Server. In SIGMOD [SIG94], page 469.

[HGLW87] Gary E. Herman, Gita Gopal, K. C. Lee, and Abel Weinrib. The Datacycle Architecture for Very High Throughput Database Systems. In *Proceedings of the 1987 ACM SIGMOD International Conference* *on Management of Data*, pages 97–103, San Francisco, California, May 1987.

[HKM$^+$88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture : A Quantitative Approach*. Morgan Kaufman Publishers, 2nd edition, 1996.

[HV97a] Sohail Hameed and Nitin H. Vaidya. Efficient Algorithms for Scheduling Single and Multiple Channel Data Broadcast. Technical Report 97-002, Department of Computer Science, Texas A&M University, February 1997.

[HV97b] Sohail Hameed and Nitin H. Vaidya. Log-time Algorithms for Scheduling Single and Multiple Channel Data Broadcast. In *The 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking*, Budapest, Hungary, September 1997.

[IB94] Tomasz Imielinski and B. R. Badrinath. Wireless Mobile Computing: Challenges in Data Management. *Communications of the ACM*, 37(10):18–28, October 1994.

[ICD96] *Proceedings of the 12th International Conference on Data Engineering*, New Orleans, Louisiana, February 1996.

[ICD97] *Proceedings of the 13th International Conference on Data Engineering*, Birmingham, U.K., April 1997.

[IK96] Tomasz Imielinski and Henry F. Korth, editors. *Mobile Computing.* Kluwer Academic Publishers, Boston, MA, 1996.

[Imi96] Tomasz Imielinski. Mobile Computing: Dataman Project Perspective. *Mobile Networks and Applications*, 1(4):359–369, 1996.

[IS98] Matrix Information and Directory Services. Matrix Maps Quarterly MMQ 501: Internet State. http://www.mids.org/mmq/501/pages.html, January 1998.

[IV94] Tomasz Imielinski and S. Vishwanathan. Adaptive Wireless Information Systems. In *Proceedings of SIGDBS (Special Interest Group in DataBase Systems) Conference*, Tokyo, Japan, October 1994.

[IVB94a] Tomasz Imielinski, S. Viswanathan, and B. R. Badrinath. Energy Efficient Indexing on Air. In SIGMOD [SIG94], pages 25–36.

[IVB94b] Tomasz Imielinski, S. Viswanathan, and B. R. Badrinath. Power Efficient Filtering of Data on Air. In *4th International Conference on Extending Database Technology*, pages 245–258, Cambridge, United Kingdom, March 1994.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* John Wiley & Sons, 1991.

[JEHA97] Jin Jing, Ahmed Elmagarmid, Abdelsalam (Sumi) Helal, and Rafael Alonso. Bit-Sequences: An adaptive cache invalidation method in mobile client/server environments. *Mobile Networks and Applications*, 2(2):115–127, October 1997.

[JW95] Ravi Jain and John Werth. Airdisks and AirRAID: Modeling and scheduling periodic wireless data broadcast. Technical Report 95-11, Center for Discrete Mathematics and Theoretical Computer Science (DIMACS), 1995.

[KB96] Randy H. Katz and Eric A. Brewer. The Case for Wireless Overlay Networks. In *SPIE Multimedia and Networking Conference*, San Jose, CA, January 1996.

[Kha97] Bhumip Khasnabish. Broadband To The Home (BTTH): Architectures, Access Methods and the Appetite for it. *IEEE Network*, 11(1):58–69, Jan./Feb. 1997.

[Kle75] Leonard Kleinrock. *Queueing Systems: Theory.* John Wiley & Sons, January 1975.

[KS91] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, pages 213–225, Pacific Grove, California, October 1991.

[Las98] Edwin R. Lassettre. Olympic Records for Data at the 1998 Nagano Games. In SIGMOD [SIG98], page 537.

[Law98] George Lawton. Paving the Information Superhighway's Last Mile. *IEEE Computer*, 31(4):10–14, 1998.

[Loe92] Shoshana Loeb. Architecting Personalized Delivery of Multimedia Information. *Communications of the ACM*, 35(12):39–47, December 1992.

[LT92] Shoshana Loeb and Douglas B. Terry. Guest Editors. Special Section on Information Filtering. *Communications of the ACM*, 35(12), December 1992.

[Mar98] Marimba. Castanet. www.marimba.com, 1998.

[Ng90] Tony P. Ng. Propagating Updates in a Highly Replicated Database. In *Proceedings of the 6th International Conference on Data Engineering*, pages 529–536, Los Angeles, California, February 1990.

[OOW93] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In SIGMOD [SIG93], pages 297–306.

[OPSS93] Brian M. Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus - An Architecture for Extensible Distributed Systems. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 58–68, Asheville, North Carolina, December 1993.

[Pan95] Raj Pandya. Emerging Mobile and Personal Communication Systems. *IEEE Communications Magazine*, 33(6):44–52, June 1995.

[PGH95] Jay E. Padgett, Christoph G. Gunther, and Takeshi Hattori. Overview of Wireless Personal Communications. *IEEE Communications Magazine*, 33(1), January 1995.

[Poi98] PointCast, Inc. The PointCast Network. http://www.pointcast.com, 1998.

[PS97] Evaggelia Pitoura and George Samaras. *Data Management for Mobile Computing*. Kluwer Academic Publishers, Boston, MA, 1997.

[PTV93] William H. Press, Saul A. Teukolsky, and William T. Vetterlin. *Numerical Recipes in C : The Art of Scientific Computing*. Cambridge University Press, 2 edition, January 1993.

[RCK$^+$95] Nick Roussopoulos, Chungmin Melvin Chen, Stephen Kelley, Alexis Delis, and Yannis Papakonstantinou. The ADMS Project: Views R Us. *IEEE Bulletin of the Technical Committee on Data Engineering*, 18(2):19–28, June 1995.

[RD90] John T. Robinson and Murthy V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 134–142, University of Colorado, Boulder, Colorado, May 1990.

[RK86] Nick Roussopoulos and Hyunchul Kang. Principles and techniques in the design of adms+/-. *IEEE Computer*, 19(12):19–25, 1986.

[Rou82] Nick Roussopoulos. View Indexing in Relational Databases. *ACM Transactions on Database Systems*, 7(2):258–290, 1982.

[RSS96] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Materialized View Maintenance and Integrity Constraint Checking: Trading

Space for Time. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 447–458, Montreal, Quebec, Canada, June 1996.

[Sat95] Mahadev Satyanarayanan. Fundamental Challenges in Mobile Computing. In *14th ACM Annual Symposium on Principles of Distributed Computing*, Ottawa, Ontario, Canada, August 1995.

[SFL96] Shashi Shekhar, Andrew Fetterer, and Duen-Ren Liu. Genesis: An Approach to Data Dissemination in Advanced Traveler Information Systems. *IEEE Bulletin of the Technical Committee on Data Engineering*, 19(3):40–47, September 1996.

[SG94] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, 4 edition, 1994.

[SIG93] *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington, DC, may 1993.

[SIG94] *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, Minnesota, May 1994.

[SIG98] *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, Seattle, WA, June 1998.

[SRB96] Konstantinos Stathatos, Nick Roussopoulos, and John S. Baras. Adaptive Data Broadcasting Using Air-Cache. In WOSBIS [WOS96], pages 30–37.

[SRB97a] Konstantinos Stathatos, Nick Roussopoulos, and John S. Baras. Adaptive Data Broadcast in Hybrid Networks. In VLDB [VLD97], pages 326–335.

[SRB97b] Konstantinos Stathatos, Nick Roussopoulos, and John S. Baras. Adaptive Data Broadcast in Hybrid Networks. Technical Report CSHCN 97-11 / ISR 97-40, Center for Satellite and Hybrid Communication Networks, Institute for Systems Research, University of Maryland, College Park, Maryland, April 1997.

[ST97] C.-J. Su and Leandros Tassiulas. Broadcast Scheduling for Information Distribution. In *Proceedings of IEEE INFOCOM'97*, Kobe, Japan, April 1997.

[SV96] Narayanan Shivakumar and Suresh Venkatasubramanian. Efficient Indexing for Broadcast Based Wireless Systems. *Mobile Networks and Applications*, 1(4):433–446, 1996.

[Tan96] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall Press, 1996.

[TGNO92] Douglas B. Terry, David Goldberg, David Nichols, and Brian M. Oki. Continuous Queries over Append-Only Databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 321–330, San Diego, California, June 1992.

[TS97a] Leandros Tassiulas and Chi-Jiun Su. Optimal Memory Management Strategies for a Mobile User in a Broadcast Data Delivery System.

IEEE Journal on Selected Areas in Communications, 15(7):1226–1238, 1997.

[TS97b] Dimitris Theodoratos and Timos K. Sellis. Data Warehouse Configuration. In VLDB [VLD97], pages 126–135.

[TY96] Kian-Lee Tan and Jeffrey Xu Yu. Energy Efficient Filtering of Nonuniform Broadcast. In *Proceedings of the 16th IEEE International Conference on Distributed Computing Systems*, pages 520–527, Kowloon, Hong Kong, May 1996.

[VH96] Nitin H. Vaidya and Sohail Hameed. Data Broadcast in Asymmetric Wireless Environments. In WOSBIS [WOS96], pages 38–52.

[Vis94] Subramaniyam R. Viswanathan. *Publishing in Wireless and Wireline Environments.* PhD thesis, Department of Computer Science, Rutgers University, November 1994.

[Vis97] Dimitra Vista. *Optimizing Incremental View Maintenance Expressions In Relational Databases.* PhD thesis, Department of Computer Science, University of Toronto, 1997.

[VLD97] *Proceedings of the 23rd International Conference on Very Large Data Bases*, Athens, Greece, August 1997.

[WA85] John W. Wong and Mostafa H. Ammar. Analysis of Broadcast Delivery in a Videotex System. *IEEE Transactions on Computers*, 34(9):863–866, September 1985.

[WD88] John W. Wong and H. D. Dykeman. Architecture and Performance of Large Scale Information Delivery Networks. In *12th International Teletraffic Congress*, Torino, Italy, 1988.

[WM91] Ouri Wolfson and Amir Milo. The Multicast Policy and Its Relationship to Replicated Data Placement. *ACM Transactions on Database Systems*, 16(1):181–205, March 1991.

[Won88] John W. Wong. Broadcast Delivery. *Proceedings of the IEEE*, 76(12):1566–1577, December 1988.

[WOS96] *1st International Workshop on Satellite-based Information Services*, Rye, New York, November 1996.

[WSD$^+$95] Ouri Wolfson, Prasad Sistla, Son Dao, Kailash Narayanan, and Ramya Raj. View Maintenance in Mobile Computing. *SIGMOD Record*, 24(4):22–27, December 1995.

[WYC96] Kun-Lung Wu, Philip S. Yu, and Ming-Syan Chen. Energy-Efficient Caching for Wireless Mobile Computing. In ICDE [ICD96], pages 336–343.

[YGM94] Tak W. Yan and Hector Garcia-Molina. Distributed Selective Dissemination of Information. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 89–98, Austin, Texas, September 1994.

[YGM96] Tak W. Yan and Hector Garcia-Molina. Efficient Dissemination of Information on the Internet. *IEEE Bulletin of the Technical Committee on Data Engineering*, 19(3):48–54, September 1996.

[YT97] Jeffrey Xu Yu and Kian-Lee Tan. An Analysis of Selective Tuning Schemes for Nonuniform Broadcast. *Data & Knowledge Engineering*, 22(3):319–344, May 1997.

[ZFAA94] Stanley Zdonik, Michael Franklin, Rafael Alonso, and Swarup Acharya. Are 'Disks in the Air' Just 'Pie in the Sky'? In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, December 1994.