Improving the Multi-Channel Hybrid Data Dissemination System

by

Jonathan L. Beaver

Bachelor of Science, University of Pittsburgh, 2001

Bachelor of Business Administration, University of Pittsburgh, 2001

Master of Science, University of Pittsburgh, 2004

Submitted to the Graduate Faculty of

Department of Computer Science in partial fulfillment

of the requirements for the degree of

Doctor of Computer Science

University of Pittsburgh

2006

UNIVERSITY OF PITTSBURGH

Department of Computer Science

This dissertation was presented

By

Jonathan L. Beaver

It was defended on

April 17, 2006

and approved by

Dissertation Advisor: Kirk Pruhs, Professor, University of Pittsburgh

Dissertation Advisor: Panos Chrysanthis, Professor, University of Pittsburgh

Alexandros Labrinidis, Assistant Professor, University of Pittsburgh

Vincenzo Liberatore, Assistant Professor, Case Western Reserve Univeristy

DEDICATION

I would like to dedicate this work to my parents and to Kristin. Without their constant support and motivation I would never have finished. They have been there for me at all times, unwavering in supporting every decision I make. I would be lost without them.

Improving the Multi-Channel Hybrid Data Dissemination System

Jonathan L. Beaver, M.S.

University of Pittsburgh, 2006

A major problem with the Internet and web-based applications is the *scalable* delivery of data. Lack of scalability can hinder performance and decrease the ability of a system to perform as originally designed. One of the most promising solutions to this scalability problem is to use a *multiple channel hybrid data dissemination server* to deliver requested information to users. This solution provides the high scalability found in *multicast*, with the low latency found in *unicast*. A multiple channel hybrid server works by using a *push-based* multicast channel to deliver the most popular data to users, and reserves the *pull-based* unicast channel for user requests and delivery of less popular data.

The adoption of a multiple channel hybrid data dissemination server, however, introduces a variety of data management problems. In this dissertation, we propose an improved multiple channel hybrid data dissemination model, and propose solutions to three fundamental data management problems that arise in any multiple channel hybrid scheme. In particular, we address the *push popularity problem*, *the document classification problem*, and the *bandwidth division problem*. We also propose a *multicast pull* channel to the common two-channel hybrid scheme. Our hypothesis that this new channel both improves scalability, and decreases variances in response times, is confirmed by our extensive experimental results. We develop a fully functioning architecture for our three-channel hybrid scheme. In a real world environment, our middleware is shown to provide high scalability for overloaded web servers, while keeping the response times experienced by clients at a minimum. Further, we demonstrate that the practical impact of this work extends to other broadcast-based environments, such as a wireless network.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

The work within this dissertation has been extensive and trying, but at the same time exciting, instructive, and fun. However, without the help, support, and encouragement from many people, I would have never been able to finish this work.

First, I would like to thank my advisors and committee co-chairs Panos K. Chrysanthis and Kirk Pruhs. Without their guidance, support, and aide I would not have been able to complete my work. They have been with me since before my graduate career began, and have motivated me to work hard and complete my research work.

Second, I would like to thank my committee members, Alexandros Labrinidis and Vincenzo Liberatore, for both being on my committee and for their helpful comments and questions which I used to complete my work.

Third, many thanks go to all those individuals that have worked on the Multicast-Based Data Dissemination project, on which this work is based. These people include my advisors, committee members, Vincent Penkrot, Joshua Gould, Shashank Vinchurkar, and Ganesh Santhanakrishnan. Special thanks go to Wenhui Zhang from Case Western Reserve University for helping in developing the release version of the software.

Finally, I would like to thank my entire family for their support during this long process.

1   INTRODUCTION

1.1   MOTIVATION BEHIND HYBRID DATA DISSEMINATION

The advent of the Internet has revolutionized the way in which people communicate, gather information, and discover truths about the world. It has also allowed any individual with a computer and an internet connection to host their own website, allowing them to communicate their thoughts, likes, and dislikes, with anyone that is interested. Paradoxically, the distribution of data generally becomes more difficult as the data becomes more useful and interesting. The increased popularity of the data may cause an influx of client requests, stressing a major weakness found in many systems, *scalability*.

Scalability can be defined as how well a system handles increases in its work load. Scalability is a key issue in any server. If a server's performance dramatically degrades the number of users increases, then the system can not be used for the mass distribution of data. A server with mediocre performance on low loads, but with good scalability is likely to be a more successful system than one with good performance on low loads but with poor scalability.

Examples scalability problems can be found across the Internet. One example is the terrorist attacks during 9/11/2001, when *msnbc.com* became overloaded with requests, and clients were forced to wait long periods of time just to access the main index page. Other examples are the virus patch from *mcafee.com* during the Slammer virus, and weather reports from the Federal Emergency Management Agency (*fema.gov*) during Hurricane Katrina on August 29, 2005. In these cases, the servers became overloaded with so many requests during a short period of time that the sites were not accessible by many users. A final example of scalability causing problems was during the 2004 US vice presidential debate, when Dick Cheney mistakenly mentioned the non-commercial site *factcheck.com,* when he meant to mention *factcheck.org*.[30] The owners of *factcheck.com* were unable to handle the incoming

request load, so they were forced to redirect traffic to another site run by billionaire George Soros (who happened to oppose George Bush and Dick Chaney running for president).

In addition to the previous examples, imagine the following hypothetical situation. A site administrator has a computer on which he wishes to host his own website. This website is just a collection of information the administrator is interested in, such as his favorite video games, books, and sports. In particular, he is a huge fan of his city's football team. He loves to find interesting facts out about the team, and post them on his website. Since football fans can never get enough of reading about their team, this site is moderately popular. On average, the site gets several thousand hits spread evenly through the day. This is not a very heavy load, and any standard server software (such as an *Apache Server*[50]) would suffice in serving this site,.

On game days, which happen only once a week, the site sees a continually increasing traffic load as the game time approaches. The reason for this increase is that during the games, this website provides play by play analysis and commentary. Because the team is very popular, there are a lot of people who want information about the game. These fans have all chosen to go to the web site to get the unique analysis provided by the author.

The point when this increase in requests occurs is when problems occur for the web site. Normally the load is light, and any standard server would suffice in servicing all requests. However, the load during the game is well beyond what the simple server can handle. This makes the web site inaccessible at times, frustrating both the web site administrator and the users. Imagine if the web site was getting small advertising fees. Losing users could be disastrous because game time is when the most users are available to click on the ads, which generates revenue for the web site. The web site administrator would like to have the server remain active and servicing all clients during both the times of low activity and especially during the times of high activity. However, the web administrator does not have a lot of resources available to purchase new hardware, or pay a service to host his site.

An ideal situation for the web site administrator would be to find a system that is easy to install and use, that is cheap to obtain and implement, that offers little hassle to the web administrator, and that provides the necessary scalability to the server. Notice that the scalability is needed only for a brief period of time. Therefore, an ideal system would not only provide scalability, but would do so only when it is needed without hindering the performance at other times. Based on this type of description of an ideal system, we are motivated to create an

architecture that will meet the aforementioned goals. Thus this work focuses on achieving these goals: *providing scalability beyond its current capacity to a web server with minimal cost required from the web administrator or clients*.

1.2 EXISTING APPROACHES

Providing scalability to a web server begins with the type of data dissemination that is used. There are generally two distinct approaches to data dissemination, which are *pull-based dissemination* and *push-based dissemination*. These approaches differ in both the way they distribute data and the philosophy behind the approach. Both approaches are also unique in the benefits provided and problems encountered. We will discuss each approach in detail, including solutions for data dissemination in the web that can be categorized under each approach.

Figure 1 shows a classification of solutions under these two approaches to data dissemination. We begin with the discussion of the pull-based solutions, which are those solutions in which clients make explicit requests for data. The most commonly employed pull-based solution is *Unicast* In a Unicast solution, the server returns the requested data back to only the client that issued the request.

Unicast performs well when the load at the server is below the server's *saturation point*, or maximum allowable load. As long as the number of incoming requests to the server is less than the number of requests that can be serviced over a similar time period, Unicast will distribute the data quickly to clients. When the number of clients exceeds the saturation point of the server, the response times experienced by clients degrades. This degradation is not gradual; rather, the response times increase rapidly after the saturation point[32].

Although Unicast, when used in a single server environment, has poor scalability, there are pull-based solutions that have better scalability. In general, these solutions can be broken into two categories, *heavy-weight* and *light weight* solutions. We define Heavy-weight solutions as those solutions that rely on new hardware or external services. One type of heavy-weight solution would be hardware solutions. We define Heavy-weight solutions as those solutions that rely on new hardware or external services. One type of heavy-weight solution would be hardware solutions. Hardware solutions are based on increasing the physical hardware used at

Data Dissemination
Solutions

Pull-Based Solutions

Push-Based (Broadcast)
Solutions

Heavy-Weight
Solutions

Light-Weight
Solutions

Hardware

Services

Access Path
Caching
Solutions

Peer-to-Peer
Solutions
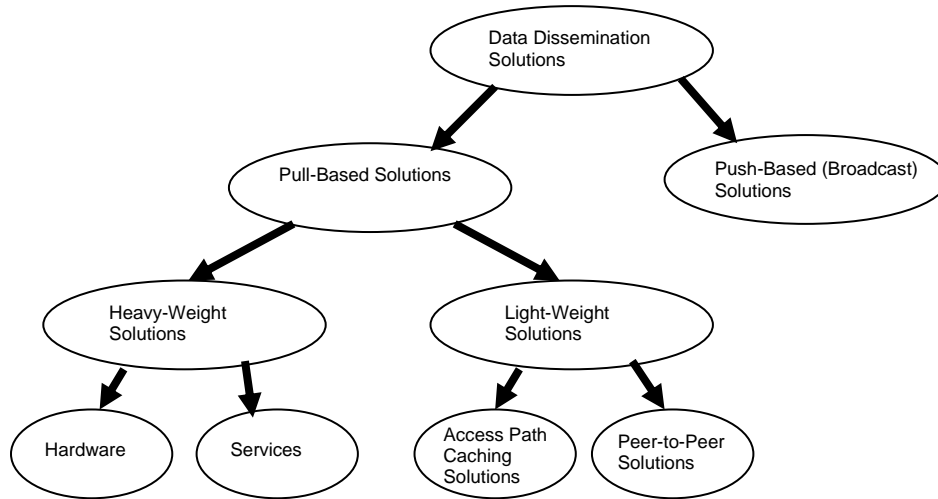
Figure 1 - Data Dissemination Solutions Hierarchy

the web site. For the most part, when we refer to hardware, we refer to adding additional machines, as would be done in a *server-farm*[18] like the ones found within Google[31]. In server farms, a large number of servers are joined together to increase aggregate processing power, and the ability of the site to respond to user requests. The benefit of this approach is that it is far less likely a single server ever reaches its saturation point as the work is spread amongst all the servers. The connection between client and server is still Unicast, however between which server and the client is transparent to the client, to the client it is still Unicast connection to a single server.

Another type of heavy-weight solution is a pay service like Akamai[3]. The web administrator can pay to have his site replicated at various physical locations across the globe. By having the site replicated at multiple locations, several objectives are accomplished. First, the request load is spread out amongst many servers, as was the case for server farms, and hence servers rarely reach their saturation point. Second, as the high load times are likely differ for different subscribers to the pay service, the pay service can potentially maintain higher average utilization than a server farm, resulting is lower cost per user. Third, by having the servers across the globe, there is a chance they are closer to clients than the single web server in our example. This leads to lower client latency, which is good to have in addition to high scalability.

Heavy-weight solutions can be very effective, and provide the scalability required by the web server in our example. However, heavy-weight solutions violate the second half of our goal, which was to provide the scalability with minimal cost. In this case, the cost is a monetary one.

For the server farm, many physical server boxes need to be purchased and maintained, which can be very expensive. For the pay services, a periodic fee is required to have the site hosted at various locations. Recall that our hypothetical web administrator's goal was scalability with minimal cost.

Another sub-category of pull-based solutions is light-weight solutions. Light-weight solutions are more software oriented and require less investment from the web site administrator. Unicast could be considered a light-weight solution; however we have shown that just Unicast alone will not provide the necessary scalability. Another type of light-weight solution, which we want to avoid, is *load shedding*[1,9]. Load shedding works by dropping (or shedding) requests that exceed the load threshold at the server. While load shedding can effective keep the load below the server's saturation point, we do not want to drop any user requests. When we refer to scalability, we mean serving all incoming requests, not a portion of the requests. There are many other types of light-weight solutions as well. However, to limit the discussion on light-weight solutions, we chose two particular light-weight solutions that can provide scalability, service all requests, and are being pull-based. These two solutions are *access path caching solutions* and *peer-to-peer solutions*.

Access path caching solutions are light-weight solutions based on *caching*[11,19,51] data along the path used by the client to request data from the server. Access path caching works by storing the data at locations along the paths traversed by client to server requests. Typically, this occurs at *client proxies*, which are gateways the client uses to access the Internet. When a request is made by a client, the proxy traps the request and determines if the item being requested is stored locally. If so, the data is returned directly to the client, and the request is not passed along to the server. If the data is not stored locally, the request is propagated to the next location, whether that is a routing node, another proxy, or the web server. If it is another proxy, that proxies caches can be checked for the item as well. This continues until the web server is reached, or the data has been found and returned to the client.

Caching is effective at lowering the number of requests users would make by providing users results from a local source instead of the web server. This allows the web server to support more users and be more scalable. There are a lot of caching schemes available, and many have been shown effective in decreasing response times. The reason we did not just implement caching as our solution is that caching we see as an enhancement to a solution, instead of the sole

solution. Caching is most effective when the data is static and when the document popularity distribution is light tailed, meaning that most requests are to a relatively few data items. It is common that web servers host dynamic content, such as the game commentary in hypothetical example. Further, it is well known that almost all document popularity distributions on web servers are heavily tailed. Thus caching can in general only answer something like half of the requests[36]. However, because caching is effective at enhancing solutions, we do implement caching as part of our final solution.

Another light-weight solution is peer-to-peer solutions[22,35,48,52]. Peer-to-peer gets its name from the fact that a collection of clients, or peers, work together towards a common goal. In our case, that common goal is retrieving data from the same web server. Each node in a peer-to-peer network can be assigned data to store, routing information to where data is located, or simply just use the network to search for data. Peer-to-Peer systems can be classified as either unstructured or structured.

In unstructured peer-to-peer solutions, data is permitted to be stored anywhere (or nowhere) in the peer-to-peer network. Common examples of unstructured peer-to-peer systems are Napster[40] or Gnutella[45]. Napster uses a centralized server for search, but a distributed approach for storage and distribution. When a user wants a file, it asks the central server if the data can be found, and the central server responds with either a negative response if the data does not exist anywhere or the location of the data in the network. The client then contacts that node directly for the actual data.

Gnutella implements search in a more distributed fashion than Napster. In Gnutella the network is broken up into sub networks or clusters[22], where a cluster head is chosen and that node is responsible for gathering and responding the requests from regular nodes. All the cluster heads form their own network in which requests are flooded amongst themselves and replies to requests are found. The cluster head then distributes the results of the search to the client that initiated the search, and that client can then directly connect to the node within the network that had the data the client needed.

There are also structured peer-to-peer solutions, such as include CAN[36] or Chord[49], which are based on a Distributed Hash Table (DHT). The way these systems work is that the DHT is used to determine where a data item is stored in the network. Basically, the DHT hashes the request into a location in a multi-dimensional space. That space is broken into a set of zones,

and each node in the network is responsible for one or more of the zones that exist. When a local node wishes to make a request, it hashes the request determine the zone that item should be stored in. The node then directs that request towards that zone through one of its neighbors. If the neighbor node owns the zone, then the neighbor will respond, otherwise the neighbor will pass the request on to another node that is closer to desired zone. This continues until the request is answered. At this time the node with the data directly communicates this data to the node that requested the data.

Peer-to-peer solutions can provide the scalability distributing the work across all the nodes. For the most part, requests will be spread out amongst the nodes. Since there is no server, there is no single place that all the information will be requested from. This relieves the stress that a single server would experience. Another benefit is that peer-to-peer solutions allow clients dynamically join and quit the network   This allows it to be dynamic in the number of clients that join, so that it can work with a little number of clients or with a large number of clients.

There are some disadvantages to peer-to-peer networks as a scalability solution. One issue is reliance on nodes in the network. Peer to peer systems are noticeably unreliable, in that nodes can come and go as they please, which does not create a very steady network. If nodes are constantly moving and joining (or leaving), then there is a lot of work that needs to be done to maintain document ownership. Even by using location based routing, the node near a location needs to have the document or the search is useless. In the centralized approach, is a node which had the data leaves, the data is no longer available, whereas using a server which always has the data does not cause this problem.

Another issue is if a single item is very popular, these peer to peer schemes will experience network traffic and overload on the node with the data in a similar way that a regular unicast server would, except these nodes are not designed to be pure servers and would therefore perform in a much poorer manner. This would lead back to the original problem of server overload and dropped requests which we tried to solve with these scalability solutions. Finally, peer to peer systems face issues such as greedy clients, unsecured clients, and a host of other security issues which can ruin the effectiveness and usability of a peer to peer system[21]. While this may not directly apply to the workload we present in this work, it is still another aspect of the system and we preferred to have the client required to do as little work as possible to use our scalability system.

7

Another approach to distributing data is to use a push based solution. in a push based solutions there are no explicit requests for data. . For example, television is essentially a push based system. The server pushes all the documents (or data) it has available to all clients. A client can join or register to receive the data from the server, and the client will continuously receive that data in a stream.

The benefit of this approach is that, regardless of the number of clients that want data from the server, the average response times for those clients will not be affected. The reason for this is that there are no requests being made to the server, thus is does not suffer the scalability problem that the Unicast server faced. A push based distribution system is completely scalable regardless of the number of clients that exist. Because of its high scalability, push based dissemination is ideal when client load is extremely high, or when scalability is the primary objective.

The way in which push based distribution is done can depend on the medium and on the availability. One method is through *broadcast*, where all clients within range of the broadcast receive it. Broadcast usually is done primarily in a wireless environment, because the basic underlying transport medium is broadcast. In a wired environment, a similar distribution medium would be multicast, where a tree is formed to distribute the data[25,33,41]. With multicast, the server only has to send out the data once, and then intermediate nodes are used to further distribute the data throughout the group of interested clients. While this functionality can in principle be built into a wired network, it has generally not been implemented in the current Internet.

The primary problem that exists with a push-based model is that clients can experience long response, since much of the data sent over the channel will not be relevant to that client. In addition, using a push-based model means that all items in the server must be properly scheduled to both service all requests and keep response times at a minimum. If the broadcast is not *flat*, meaning each item sent with equal frequency, this scheduling problem is even more complicated. Push-based data dissemination has no mechanism for feedback from clients. Thus, a misestimation of the popularity of the documents might cause in increase in the latency perceived by the clients.

For a server like the one in our motivational example, using only a push based distribution system would not be optimal. Most of the time, there are too few clients to warrant

using a push based system. Simple Unicast would provide much better performance. The only time that using the push based system would be of benefit is when the game is going on, and the load is high. At that time, push based distribution would allow all the users to get their data without getting requests dropped by the server, or having to experience extremely long response times. At all other times, users are going to be frustrated with having to wait for such a long time to get the data that they want. This may cause the users to never want to use the server. Fewer users may hurt the web administrator just as much as dropping the requests during the game.

What would be ideal is a system that provides fast responses, on the order of Unicast, when the server load is low, yet is able to provide large scalability, similar to the push based system, when the client load is very high.

1.3   HYBRID DATA DISSEMINATION

The solutions shown in Figure 1, which we previously discussed, have been based on a single mode of data dissemination. In general, using pull-based solutions provide low latency when the load is below a particular saturation point, but provide very poor scalability for loads above the saturation point. Push-based solutions gave almost infinite scalability, but provide poor latency at low loads. Thus, neither solution uniformly provides both scalability, and low latency at low loads.

To achieve both of these goals, a *multiple channel* distribution system has been proposed. Referred to as a *Hybrid* system, this type of distribution system combines certain features of pull-based solutions with features found in push-based solutions. The multiple channels improve both performance and scalability over the previously defined models. While there are several different kinds of proposed hybrid systems, we focus on two particular types for this discussion. One type of system is a *pure multicast pull* based system, where requests are made over a unicast channel and all the resulting data is distributed over multicast to all clients. The second type of system is one that uses two distinct physical channels to distribute data to clients. One channel is pull-based and the other channel is push-based. Unlike the pull based multicast, both channels are used to distribute data instead of only the push channel.

The pure multicast pull system is a type of hybrid system where a single unicast channel is used to make requests to the server, and a multicast channel is used to push results out to clients.

Work that is closer in design to ours are projects such as the DBIS-toolkit[7], the Air-Cache[46,47], and other hybrid middleware systems[16,20,36]. In these systems, the focus is on using two channels to distribute data, a push based channel similar to our multicast push channel and a pull based channel similar to our unicast channel. In such a system there is a set of popular, or *hot*, documents that are multicasted out to clients. The clients make no explicit requests for the hot documents. The non-hot documents are multicasted out after receiving requests from clients.

In a single channel hybrid system the two logical channels are collapses into one channel. The DBIS-toolkit[7] is an example of a single channel hybrid system. In such a system the non-hot documents at which point they are scheduled to be sent out are intermixed with the hot documents. There are several problems that arise in such an implementation. The most important problem is that clients have to wait for the requests of other clients to appear on the multicast channel before their own appear. Just as important, clients for unpopular, or *cold*, documents will have very long response times because they may have to wait for popular data to be multicasted multiple times before the cold data is sent out. Another major problem with this approach is how to schedule the different items onto this single push channel. The scheduling in these systems is not a trivial problem, and has been the subject of much prior research[2, 4, 5, 28].

Air-Cache[46,47] is an example of two channel hybrid system. In these systems, the focus is on using two channels to distribute data, a multicast push channel and a unicast pull channel. There some potential limitations in the AirCache system. One limitation is that they only use two channels. Another limitation the way they estimate document popularity (through a drop and check method, which we will discuss in detail in the next Chapter). Because AirCache is the closest design to our proposed system, we did perform comparisons against this method both in terms of performance of the overall architecture (shown as hybrid method in our experiments) and performance of the popularity estimation method (shown through direct comparison with our own method). Additionally, Air-Cache does not provide a near optimal way to divide the documents, instead relying on a threshold, which in our experiments we showed to be less than efficient. Other middleware use different ways to distribute the data; however none provide the three channels of distribution we propose, or the robustness we provide with our architecture.
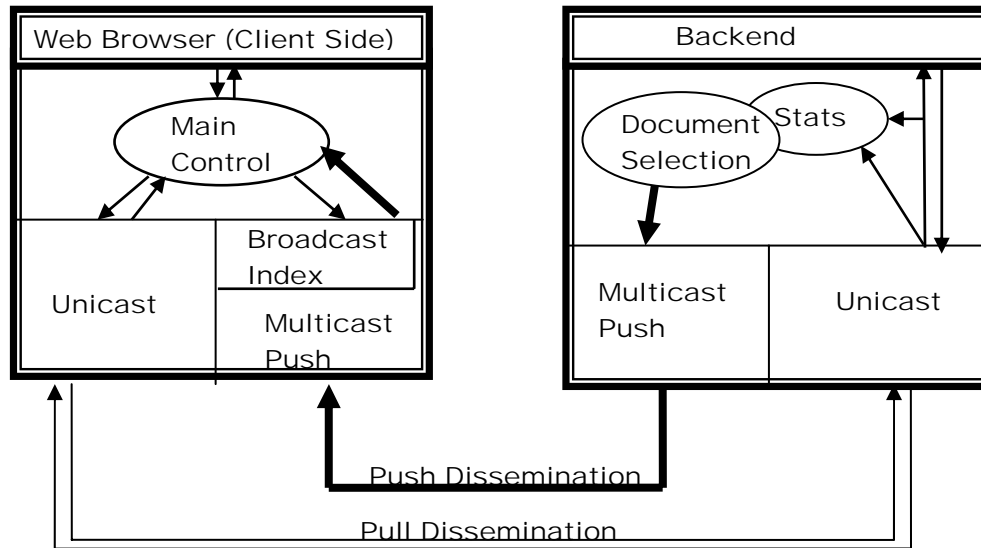
Figure 2 - Basic Hybrid System Architecture

An alternative type of hybrid system is to again use a single unicast channel to make requests to the server but have both channels used to distribute data back to clients. The way this is done is on the multicast push channel; hot documents are placed and pushed out to all clients in multicast *cycles*, or iterations of sending out all the data on the multicast channel. These documents do not require explicit requests from clients and are designed to increase scalability. The increase in scalability comes from not requiring an influx of requests to appear for the most popular documents on the server. The unicast pull channel is used to respond to requests like a normal unicast server would. The requests made over this channel are for the cold documents in the system. The clients are replied to directly, in contrast to the multicast pull model where the requests would be scheduled to be sent out over multicast. This leads to lower response times for unicast clients because they are responded to right away. It leads to lower response times for push clients because those clients only have to wait for a small set of popular data to be sent out and not have cold documents intermingled within channel. Example architecture for this type of system is shown in Figure 2.

The architecture shown in Figure 2 introduces several key questions that must be addressed for the system shown to operate. Some of the questions are:

1. How does the document selection portion of the architecture work?
2. How and what statistics are gathered by the system, and what statistics are necessary?
3. How does the operation at the client flow through the main control?

11

4. How is the bandwidth divided among the different channels?

In this dissertation, we provide answers to these questions by proposing an improved hybrid data dissemination architecture which is based on the general idea displayed in Figure 2.

1.4   CONTRIBUTIONS OF THIS THESIS

In this dissertation, we aim to solve several data management problems associated with the multiple channel hybrid data dissemination model such as that shown in Figure 2. We propose a new architecture for the multiple channel hybrid system. We summarize our contributions as follows:

1. We provide an algorithm that will close to optimally solve the *document selection problem*. The document selection problem involves deciding which items should be placed on the multicast push channel, and which items should be requested over the unicast channel.

2. We provide a method to solve the *bandwidth division problem*. The bandwidth division problem involves deciding how much bandwidth to give the push channel and how much bandwidth to give the pull channels. This is important because if there is not enough bandwidth for the pull channel, it will get easily overloaded. If there is not enough bandwidth for the push channel, the latency for the popular documents will be high; causing overall system latency will be high. Hence, an appropriate balance must be maintained. We accomplish this through an integrated algorithm, called SELDIV, which solves both the document selection and the bandwidth division problems.

3. We propose a new multiple channel hybrid data dissemination architecture in which a new, third channel is added. This third channel takes a portion of the documents chosen to be pull based and distributes them over an additional multicast channel separate from the multicast push channel. This third channel, which we refer to as the multicast pull channel, is used to both enhance scalability and performance, while keeping the variance of experienced client latencies lower. We also describe several methods for using this channel; empirically determine the most effective method.

4. We provide a fully functioning architecture for multi-channel hybrid data dissemination system. We explain the rational for the various design choices that we made in the process of creating our system. This middleware has been finalized in a release version that is available for download from our website.

5. We perform both empirical and analytic analysis of our architecture and algorithms in a simulation environment, where we can isolate the individual pieces of our improved architecture. These experiments will validate the near optimality of our algorithms, the optimal settings for various parameters within our architecture. This set of experiments show that the multicast pull channel does in fact lower the variance in user perceived performance (average latency), and does not adversely affect system performance.

6. We perform experiments in a real world environment of Planet Lab. These experimental results show that the architecture does hold up under the *Planet Lab* environment, which provides the network traffic and delays that would be experienced in an Internet deployment. These experiments again show using the third channel provides lower variance in response times and lower average response times overall.

7. We provide an implementation of our hybrid architecture that can be used in a wireless environment.

Our architecture is designed to be easy to use and easy to implement. We have written it as both a client and server middleware that can be placed in front of any server back-end and any client front-end. All our experiments were run on either a simulated or real version of our middleware to ensure that results we found will closely match those discovered by end users.

1.5 READING THE DISSERTATION

To present our contributions, the remaining chapters in this dissertation are organized as follows:

- Chapter 2 looks more in depth at the general hybrid system architecture before examining our improved hybrid system architecture. In addition to simply describing the architecture, we explain in detail every facet of the architecture, including caching, scheduling, and our new way to provide feedback for items on the push channels. We then look at an actual implementation of the architecture including additions or changes

that were needed to create the release version. Finally, we look at a wireless implementation of our architecture.

- Chapter 3 presents the algorithm selection-division, or SELDIV, that we have created for simultaneously solving the document selection problem and the bandwidth division problem.

- Chapter 4 contains a large set of experiments to validate using our architecture. We provide two distinct realms of experiments: one in a simulated and isolated environment, and one in a real world environment using Planet Lab.

- Chapter 5 contains our conclusion and future work.

2 AN IMPROVED HYBRID SYSTEM ARCHITECTURE

In this chapter, we examine the architecture that we have developed to enable a hybrid, multicast based data dissemination system. The architecture is based on the general multiple channel hybrid architecture found in previous works[7,47,48]. We will first introduce the details of the general hybrid architecture. The purpose of this introduction is to both provide an understanding of basic operation and provide a baseline to compare our improved architecture against. We will go through the improvements that we made, while addressing issues such as document popularity for pushed items and real world implementation information that was gathered from implementing the system as a fully functioning middleware. While examining these different features, we will present solutions that we developed to problems identified, and provide discussion of alternate solutions.

2.1 BASIC HYBRID SYSTEM ARCHITECTURE

The general hybrid architecture is shown in Figure 2. There are three main parts to this architecture: *the server side*, *the client side*, and *the data dissemination layer*. The client side consists of several components that allow it to operate within a multiple channel distribution environment:

- The *unicast module* handles end to end communication with the server. This is performed over regular means of communication, where the client initiates an HTTP request to the server, and then awaits responses from the server in the form of the requested data. This channel is responsible for all communication from the client to server, and thus will be responsible for requests made for the cold documents in the system.

- The *push module* handles the push connection between the server and client. In particular, the distribution from the server to the client on this channel is done through a push based dissemination method. The push module continuously listens to and monitors the push channel for data from the server. This data is used to generate the necessary information for the broadcast index module on the client. This module contains a listing of all information that is on the push channel, in the form of a broadcast index that can then be used by the *main control* module to determine which channel to request and receive the data from.
- The main control module handles the cache, which can be used as determined by the system setup, as well as the basic operation of the client side component.

The main control operates by following the steps outlined below:

1. A request for document D_i is made by the client on any standard browser which is then passed to the main control of the client side proxy.
2. The main control first checks whether D_i is listed on the stored version of the most recent index of the documents on the multicast push channel.
   a. If D_i is in the index, the client waits for D_i to appear on the push channel, and when D_i arrives it is provided to the application client.
   b. If D_i is not in the latest stored version of the push index, the client makes a direct request for D_i from the server using the unicast module.
      i. After making this request, the client monitors the unicast channel. When the client receives D_i, it passes the document back to the application client.

The server side component resides in front of the web server and acts as a distribution engine for documents that are requested from the web server. The server side has several features that allow it to both distribute the data in various ways, and communicate with client proxies of the same distribution system:

- The unicast module, similar to that of the client proxy, is used to receive and respond to requests made for documents by the client.

16

- The push module is responsible for distributing the hot documents to the clients by continuously pushing the data out over the push channel. The push module also contains the broadcast index, which defines what will be coming over the push channel.
- The *push schedule module* determines when and which documents will be pushed out and in what order.
- The cache is used to hold any information from the web server that the system deems necessary to speed up the delivery of data to clients.
- The *statistics module* handles the gathering of statistics based on the method defined for the system.
- The *document selection module* is used to divide the documents between the multicast push channel and the unicast channel. When the document selection module is called, it will use the statistics as part of the information set used to determine the division of documents. Within the document selection module resides the document selection algorithm. When run, the documents that are deemed as hot documents are then sent to the push module to be pushed to the clients, while the cold documents are kept at the web server until requested by clients
- The main control module, similar to the client, handles the interaction between the channels and the overall operation in general at the server side component.

In general, operation in the main control of the server side component behaves in the following manner:

1. When the server receives a request for a document D_i from the client over a unicast connection, one of two actions can be taken.
    a. If the requested document is currently on the push queue, the server drops connection with the client since the document will be served by pushing it to the client. No further action is needed.
    b. If D_i is not on the push channel
        i. The count of recent access to D_i is incremented by 1.
        ii. D_i is en-queued on the pull queue and serviced when scheduled to be so by the unicast module

2. The document selection is run as determined by the system's internal schedule and replaces any necessary documents on the push channel

While this model provides good scalability and low response times, it still has several problems which need to be addressed. The major problems are:

- How to decide which items to place on the hot channel and which to place on the cold channel?

- How to determine the popularity of items on the push channel?

- What to do in cases where the wrong set of items is on the push channel? This means occurs when popularity of items has been mispredicted, or the popularity of items has shifted over time.

These are important issues because they can lead to poor response times at clients and possible server overload when the misprediction is occurring on all documents. Therefore, these issues should be addressed in any developed hybrid system.

2.2 OUR PROPOSED HYBRID SYSTEM ARCHITECTURE

In order to improve upon the general hybrid system architecture, we have developed our own hybrid data dissemination architecture, which is build directly onto the general architecture. This improved architecture is shown in Figure 3. As this figure shows, there are several additions which were made to the general architecture. Most notably an additional channel was added. This channel combines the distribution technology used on the multicast push channel with the pull based request approach found on the unicast channel. This channel, as will be seen shortly, acts as both an enhancement to the distribution ability of the system overall and as a safety net for the system, in cases where predictions are off, or popularity changes are taking place. The channel can be implemented in several fashions, as we explain in Chapter 3.2.

Other system improvements include:

- How to use the cache in combination with the three channels we have specified

- How to gather statistics for items on the push channel
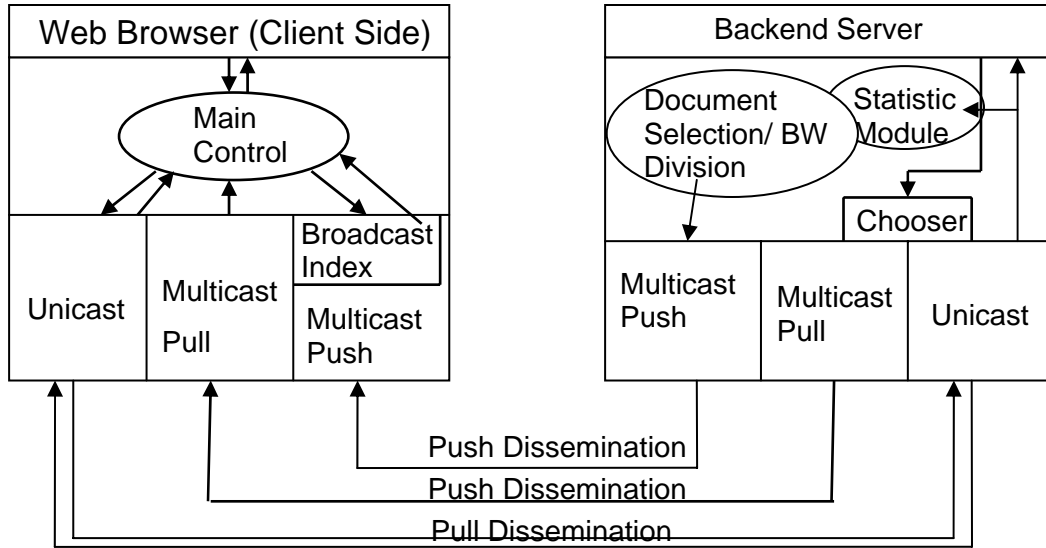
Figure 3 - Our Improved Hybrid System Architecture

- Creating a completely modularized architecture, where everything from caching schemes to scheduling can be plugged in and out of the system with no ill effects towards operation.

These changes also cause the client and server proxy main controls to behave differently than in the general architecture. In particular, the client side proxy behaves in the following manner:

1. A request for document D_i comes from the web browser front end into the client side proxy

2. The cache is checked for D_i, if D_i is found, return result to client

3. The client side proxy checks the index of the push channel for D_i

    a. If D_i is in the push index, the proxy listens to the push channel for document D_i

        i. If a new index appears on the push channel, the proxy checks the new index for D_i and if D_i is in the index, the client waits for D_i

        ii. If the document appears

            1. Generate a random percentage r_i and check it against the associated percentage $p(Di)$ for Di

                a. If $r_i \geq p(Di)$, then make a statistic request to the server

                b. Return results to client

4. Send a request over the unicast channel to the server proxy for D_i, and begin to monitor all three channels (Multicast Push, Multicast Pull and Unicast) for the document.

19

5. When the document arrives on any channel, close the connection to the server if it is not already closed

6. The contents of document D_i are compiled together and returned to the client side front end.

By using the above steps, the client uses all three channels to receive data from the server. The unicast channel is used for making direct requests to the server, and data can arrive over any channel when it is requested. The reason all three channels must be monitored is to avoid a race condition from occurring. A race condition is an undesirable situation that occurs when a system attempts to perform two or more operations at the same time, but because of the nature of the system, the operations must be done in the proper sequence in order to be done correctly. In our system, the race condition could occur if the user made a request for an item which was, at the same time, being moved to the push channel. In this case, the client already checked for the item on the push channel. Since the item was not on the push channel, the user makes a request over unicast and expects the results over the pull channels. In our example, the document was actually moved to the push channel. Thus, the user will never receive the document, since the user is waiting on the pull channels for the document, when it is actually on the push channel. For this reason, the user not being able to get the requested document, we have the client monitor all three channels when a pull request is made.

On the server side proxy, there is an additional module called the *pull decision* module. This module is used to determine onto which distribution channel the pulled document will be placed. In particular, this module can run one of the several methods explained in Chapter 3. For our specific implementation, we chose to use the *runtime, dynamic threshold* implementation of the pull decision module, where the channel used to distribute a document is decided as the document is being served. The number of pending requests is compared against a user defined threshold when making the decision of whether to send the document over the multicast pull channel. Using this implementation, the server operates in the following manner:

1. The server receives a request for document D_i over the unicast connection.

2. If D_i is scheduled to appear on the push channel

    a. The statistics module count for document D_i is incremented by $\frac{1}{r_i}$ (popularity modifier) and the connection with the client is closed.

20

3. If D_i is not scheduled to appear on the push channel, the multicast pull scheduler is checked for D_i.

4. If D_i is on the multicast pull schedule, the count is incremented in the multicast pull scheduler.

5. The pull document queue is checked for D_i

6. If D_i is in the pull document queue, the count for D_i is incremented by 1. If D_i is not on the pull document queue, D_i is added into the queue with a count of 1

7. Documents are removed from the pull queue by a separate service thread one at a time, in a first come first server manner

8. When D_i is taken off the pull queue, the count for D_i is checked against the multicast pull threshold MpT

9. If $Count(D_i) \geq MpT$, the document is placed into the Mulitcast Pull scheduler to be send out over the multicast pull channel

10. If $Count(D_i) < MpT$, the document is sent over the unicast channel to each client that has an open connection for Di

11. The statistics for Di are incremented by $Count(D_i)$.

By using the steps above, the server accomplishes several key tasks:

- It is able to utilize three separate channels for distributing the document based on the popularity of the document.

- It is able to keep track of the request statistics of documents regardless of the distribution method used for the document.

- It provides opportunities for parts of the server to be plugged in and out as needed. These can be used to either fine tune the server, or to use methods that are improvements over the existing methods.

We now examine several plug and play modules in our architecture, with a discussion of options for each.

2.2.1   Gathering Push Channel Statistics

Being able to gather statistics for the push channel is a vital part of using the hybrid data dissemination method. Without an accurate account of popularity for particular documents, the document selection module may incorrectly label documents as being push documents or pull documents. This can lead to the server being forced to handle too many pull requests, rendering the server inaccessible for brief periods of time. In order to solve this problem, several solutions have been suggested.

One way that the popularity could be estimated is to use multicast population estimation techniques[6,27,41]. The idea behind these techniques is to estimate the number of clients that are actually connected to a multicast channel at any given time. This will tell the popularity of the channel, which could then be implied to mean the popularity of the items on that channel. This would work very well if there were only one item on each channel and clients selectively chose which channels to connect to. If each document were on its own channel, being able to estimate the number of clients on that channel would be good estimation of the popularity if the channel and therefore the item on that channel. The issue we run into is two fold in attempting to apply this kind of popularity estimation. First, we force our clients to connect to both multicast channels which we use. This would cause all the items on both channels to seem popular, thus we would not be able to properly determine which items belong on which channel, and that hurts our architecture tremendously.

The second, and bigger issue we find with this approach, is that it gives popularity at a channel level, whereas we require it at a document level. In essence, combining the fact that users need to be connected to the multicast push channel to receive any popular documents with the type of popularity these methods provide, the best we could hope for would be determining if at least one of the items on the multicast push channel is popular enough and a possible max popularity for the items on the channel. This does not, however, tell us the popularity of an individual item, which both our algorithm and architecture rely on to distribute data to clients efficiently while providing high scalability. Therefore, while this is an effective means to determine some form of popularity, it is not at the level we require, and hence could not use it.

In order to determine the popularity of the documents on the push channel we examine two methods which focus on the document itself, and not on the number of users. These two

methods are the *drop and test method*, proposed in the Air Cache work[48], and the *probabilistic testing method* we have developed and use in our system.

2.2.1.1   Drop and Test Method

The drop and test method operates by producing three different layers of documents: hot documents, warm documents and cold documents. The hot documents are those that are placed on the multicast push channel, while the warm and cold documents are served over the pull channel. Warm documents are those that have become popular but not popular enough to be placed on the push channel. Documents can move between the three channels, based on the number of incoming requests that are being received. The requests for warm and cold documents can be calculated using the number of requests that are received for those documents over the unicast channel.

The hot document's popularity is generated by periodically dropping the document off of the push channel for a couple of broadcast cycles. The document is then placed back on the push channel, and the statistics are calculated and the hot documents are recalculated. While the document is on the warm channel (from being dropped down), requests will appear for that document as would any pulled document. The popularity can then be calculated based on the number of requests that were received over that brief period it was on the warm channel. The document is not permanently dropped right away because if the document is still extremely popular, too many requests would come in, forcing the server into an overload state before it has a chance to correct the hot document set.

The benefit of this method is that it allows for a document's popularity to be tested with a simple change in channel distribution. Given a constant popularity stream the popularity of the document can be estimated with relative correctness. It also allows for documents to be gradually moved between channels, as they must go from cold to warm to hot in order to be placed on the push channel. This ensures that brief popularity spikes do not cause constant changes in the push channel contents, which could cause incorrect documents to appear on the push channel. Popularity spikes can adversely affect the performance of a hybrid system, as it may take a while to get a document off the push channel if past requests are included in calculating its popularity. However, making a document go through several stages of popularity leveling can make sure that a brief spike only places it from cold to warm, and thus keeping the

correct documents on the hot channel for the duration of the server. The main disadvantages of this approach are:

- The request spikes it can cause

- The miss popularity estimates that can occur

- The difficulty in deciding when and how often to drop a document off the push channel.

The request spikes are those that occur when the document is actually dropped. As documents are dropped, the clients will make requests for the now pull based document. If this document is the most popular document, it may lead to a large number of requests being made for this document. This could cause a brief request spike at the server, which may take a while to overcome. Using the multicast pull channel, as we have proposed above, may help to speed up the recovery; however the spikes will still have an effect on the overall latency in the system.

What can further exasperate this issue is the scheduling of when, and how often, to drop certain items from the push to pull channel. If the dropping of items is not appropriately done, several documents could be dropped at the same time, which would further increase the request spike and further hinder the server. Also, deciding how often to drop the items can be problematic if popularity changes, or an incorrect decision could cause the request spikes to remain somewhat constant. For example, if the dropping of an item causes a request spike that lasts ten seconds, and that item is dropped every ten seconds, the server will experience a constant increase in response times. Likewise, if an item is only dropped every minute, and document popularities shift, the server will take a long time to properly adjust the push channel contents, which will adversely affect the latency in the system.

Concurrent with the scheduling issue is the issue of dropping the item at the wrong time. If item A is dropped during cycle t, but the requests for A appear at time $t-1$ and $t+1$, the requests for document A will appear as 0 even though it should be much larger. Thus, A is determined to be of very low popularity, and permanently moved to the warm channel at time $t+2$. If requests then start appearing at time $t+3$, and come in a large amount, it will take the system several cycles to get A back on the hot channel. If this pattern continues, the popularity of A will always be miscalculated. A will keep moving between the hot and warm channels, causing the server to receive too many requests. This hurts the overall performance of the server

2.2.1.2   Probabilistic Testing Method

The goal of our approach is to prevent the major problems of miscalculated popularity and request spikes. The approach we take to gather the popularity of items on the push channel is to calculate the values by estimating the popularity based on random sampling. The server publishes a report probability s_i for each pushed document i. Then, if a client wishes to access document i, it submits an explicit request for that document with probability s_i. In principle, clients would not need to submit any request for push documents, but if they do send requests with probability s_i, the server can use those requests to estimate p_i. At the same time, the report probability s_i should be small enough that server is almost surely not going to be overwhelmed with requests for pushed documents.

In particular, we consider the objective of minimizing the maximum relative inaccuracy observed in the estimated popularities of the pushed documents. In this case, we show analytically that each report probability should be set inversely proportional to the predicted access probability for that document. First, the server calculates the rate B of incoming requests it can tolerate. Presumably, B is approximately equal to the rate that the server can accept TCP connections minus the rate of connection arrivals for pulled documents. Therefore, the value of B can be estimated from the access probabilities and the current request rate, all scaled down by a safety factor to give the server a little leeway for error. Then, the s_i's have to be set such that $\sum_{i=1}^{k} \lambda p_i s_i \leq B$, where documents $1...k$ are on the push channel. The expected number of reports u_i that the server can expect to see for i over a unit time period is $\lambda p_i s_i$.

Using standard Chernoff bounds, the probability that number of reports is more than *(1-D)u_i* is roughly $e^{\frac{-u_i D^2}{4}}$ and that the probability that number of reports is less than *(1-D)u_i* is roughly $e^{\frac{-u_i D^2}{2}}$. If the goal is to minimize the expected maximum relative inaccuracy of the reports, all of the upper tail bounds should be equal and all of the lower tail bounds should be equal. That is, all u_i should be equal, or equivalently it should be the case that for all i, where *1 < i ≤ k*, $s_i = \dfrac{B}{\lambda p_i k}$. Hence, each document should have a report percentage inversely proportional to its access probability.

The use of this probability can be seen in step 3 for the client proxy and step 2 of the server proxy. When the client goes to make a request, and finds the item on the push channel, it

will use s_i to decide whether to send a request to the server. If the request is to be made, it will make the request as a statistics only request. When the server receives the request, that connection will be dropped. Both the client and server expect this connection to end quickly, to make sure that the server does not have too many open connections. At the server, the number of requests this single request counts for is $\frac{1}{s_i}$. Thus the popularity for the push documents is created. In Chapter 4.1.6 we provide experiments testing our push popularity scheme against the alternative above. We show that both keep the latency low (meaning the division of documents was fairly accurate), but that the former scheme suffers from brief latency spikes that our scheme does not suffer from.

2.2.2 Caching in a Hybrid Data Dissemination System

Another feature of our system is the cache that exists for the client and server. Unlike normal systems, where the caching policy can be the same for all requests that appear, the caching in a hybrid system must take into account the fact that items appear over multiple channels and will be received in various manners. There are several ways this can be used to the advantage of the system. First, we examine caching on the client side of the system, and then examine the server side caching.

At the client, one option is to cache only pulled documents and relies on the fact that the push channel will be continuously providing the client with documents. The documents that are coming over the push channel will be appearing in each broadcast cycle until they are moved off the push channel. If a client makes a request for a pushed document, the server can wait on the push channel for the document to arrive. Because these documents are continuously arriving, the client will not have to make a direct request to the server and sit waiting for a response. Instead, the document will eventually be pushed to the client and the client can get it in that manner. This will also leave the cache open for less popular documents, which will lower the response time for clients that wish to get those files.

The issue with this option is that it relies on timing on the part of requests for items on the push channel. This approach would work well if the number of items on the push channel is smaller in size and the client makes the request before the request appears on the push channel.

If the request is made after the item appears on the push channel, the client will have to wait until the next broadcast cycle begins. If the item is no longer going to appear on the push channel, the client will be forced to make a request for that item over the unicast channel, because it was not being cached. This creates a much longer response time than if the requests had just been made directly. Considering that the majority of the requests will be for items on the push channel, the chance of this increased response time is very high.

Another option, which we chose to follow in our system, is to divide the cache between the push channel and the pull channels. In particular, the amount of cache space given to the push channel is large enough to contain all the items on the push channel. Thus, it is a variable sized cache, with the remaining cache available to store the pulled documents. The reason for doing this is that if a client needs an item on the push channel, it will be able to immediately get that file. Since these are the most popular files, this will lead hopefully to lower overall system latency. If the cache is not large enough to handle all items coming over the push channel, we follow a most requests first approach to removing items from the cache, where the larger popularity documents (which can be determined by the item's popularity estimation) are kept in the cache. The way we divide the cache is to set the pin byte of the document to true if it is from the push channel, and to false otherwise.

We also implemented the Gray Algorithm[38] as a possible caching scheme to be used by both the server and client, as it has been shown to be extremely effective when dealing with caching and broadcast channels. The Gray algorithm works by using access history and retrieval delay to make decisions on which items to keep in cache and which to evict. By using three colors, black, white and grey, the algorithm can choose which documents to keep in cache. Documents migrate from black to gray, and then gray to white. The black pages are those which are most recently accessed and caused page faults, gray those less frequently accessed, and white for documents not recently accessed at all. The cache works by keeping all black documents and those grey documents which will take longest to fetch if a page fault occurs. This has been shown to work extremely well in a broadcast push environment and would be especially effective in the cases where the cache could not hold all of the items from the different multicast channels[38].

The remaining cache is divided between all pulled documents and can use any standard replacement algorithm. However, because there are actually two channels returning the data, we

27

can use that to our advantage. If an item was received over the multicast pull channel, this means it had several requests pending for it at the time it was serviced. This implies the popularity of this item is most likely greater than that of a similar item retrieved from the Unicast channel. This should signal to the cache that the multicast pulled item is of higher priority than a Unicast item, all other factors being equal. Therefore, we provide these multicast pulled items with a popularity larger than the Unicasted items. In cases where the cache replacement policy relies on popularity, this will provide it with more information when making ejection decisions. Regardless of the caching scheme used, it is important to utilize the additional channels and use them to the advantage of the system when making decisions on the cache.

On the server side, caching can once again be done in any fashion as determined by the scheme used. However, in a manner similar to the client side cache, the fact that there are multiple channels should be taken into account when deciding what items to keep in the cache and which items to eject. Unlike the client, the server can not count on the multicast pushed items being continuously pushed into it, allowing it to use those as a cache. On the contrary, the server is responsible from pushing those items out continuously, and should therefore have the data from those documents readily available as it goes to broadcast each new item. On the server, as many of the items as possible from the push channel should be placed in the server cache. This is done so that the multicast push channel can broadcast without having to go to the web server backend to get each item. Fetching each item can cause delays in the multicast push data delivery. This can then adversely affect client latency.

In the cache we implemented, we follow this approach of giving multicast push documents priority in the cache, providing as much of the cache as possible to those items. To accomplish this, we attach a pin byte onto each file as it is put in the cache. The pin byte is set for each item on the multicast push channel. Regardless of which scheme is used to eject items, if the pin byte is set to one, the item is not to be ejected. Once the multicast push channel is updated from a document selection run, any items which are no longer on the multicast push channel have their pin byte unset, and can be safely removed. If the cache is not big enough for all the items on the multicast push channel, then the cache will fill with the most popular items first, and will not remove any items from the server cache unless the document selection is rerun.

The remainder of the cache is left open to be used in the manner dictated by the caching policy. Similar to our client cache, we chose to implement this cache by taking the last access

time of a document divided by the popularity of a document, which is based on the Lowest Total Stretch First (LTSF) scheme[2]. Those with highest values are ejected first. This allows us to make sure that either items, which have the most requests or are being requested the most often, are kept in the cache, while low popularity items (either from lack of requests or not being requested very recently) are removed. This will tend to make sure that those items which had to be sent over the multicast pull channel, when distribution time occurred, remain in the cache, which is good because these are the items that are most likely becoming hot. In general, the cache on the server side needs to make sure that the documents which will be distributed most often are in the cache. This will lower the latency that clients experience, and lower the work the server must do to get those documents, which can leave it open to handle more client requests.

2.2.3 Multicast Push and Multicast Pull Scheduling Schemes

Another feature of our system that is modularized is the scheduling schemes used on the multicast pull and multicast push channels. The multicast push channel, in particular, needs a schedule because of how our system works. The client needs to check the index of the multicast push channel to find out if the requested document is going to appear on the push channel or if an explicit request needs to be made to the server. This means that the server needs to place an index on the multicast push channel which relays this information to the client. There are many scheduling algorithms that can be used for the multicast push channel[4,10,16,38]; we chose to go with a *flat broadcast*, with items ordered by popularity. We chose this schedule for multiple reasons:

- It is relatively simple to implement and to use with our client access model.
- When ordered by popularity, it fits in nicely with our document selection algorithm. The algorithm contains a list of documents ordered by popularity which it uses to speed up the search process it performs. Because the algorithm returns the item number of the document at which the split should occur, the multicast push channel simple makes its index based on the array of items starting at the beginning up to the split location.
- It allowed us to associate an order that the client can use to determine whether the item it is searching for on the push channel has already passed by or is still going to come. Take a request for document D4, which is the 4th most popular document in the system. If the

current item on the broadcast is item D6, the client instantly knows that it will have to wait for the next broadcast cycle to get the item it wants. Likewise, if it is on item D3, the client knows the next item over the multicast push channel with be its request and can wait for it accordingly. This will provide more benefits when we talk about using our architecture in a wireless environment, for this type of schedule allows for easy selective tuning.

Scheduling on the multicast pull channel can also be done in a variety of ways. In our system, we focused on two main scheduling schemes for items on the multicast pull channel. The first was to send out items in a first come, first server manner. Because the multicast pull channel is an on-demand channel, using a scheduling algorithm which is also on-demand (as items appear, send them out) fits with the channel nicely. However, it does not take into account the number of requests pending for an object, which directly affects the overall latency of the system (which is measured as average latency for all clients). To account for total pending requests, we also chose to implement LTSF, as we previously explained for the client. While our modular implementation allows for any scheduling algorithm to be used, we implemented the LTSF algorithm in the prototype.

2.3   REAL WORLD SYSTEM IMPLEMENTATION NOTES

In this section, we describe an actual implementation of our architecture into a middleware system that has been fully developed. The system is currently available to be downloaded at *http://www.cs.pitt.edu/~beaver/mbdd/* and can be downloaded as either a server side component, client side component, or an entire system. We will describe how our system was implemented, what decisions were made and what enhancements had to be made in order to make the system fully operational in a real world environment.

The first decision we had to make was in what programming language to use to develop our system. We chose to go with Java, both for the reasons of ease of programming, ability to run under any operating system, and the availability of multicast layer components. We wrote the system in such a way that all components are completely modularized. This includes the following:

- The client side caching scheme

- The server side caching scheme

- The server side multicast push scheduling scheme

- The server side multicast pull scheduling scheme

- The multicast distribution layer

- The connection between the client side proxy and the client front end

- The connection between the server side proxy and the server back end

In the previous section we explained the decisions we made for many of these components. However, we did not explain what multicast system we used, or how the proxies communicate with their front/back ends.

We implemented two different java based multicast layers into our system, Java Reliable Multicast System (JRMS)[46] and Hypercast[39]. JRMS is a reliable multicast layer which works in the following manner. The server chooses a "multicast" address and port combination, which will be used for the multicast layer (in our case, two different address and port combinations). When a client wishes to join a multicast channel, it sets up a listener on that channel. When the server wants to send data out, it simply sends the data to the address it specified. Any clients which were set up to listen on that address will set up an internal tree which will be used to distribute the data to all nodes in the multicast tree. The multicast layer is built over TCP, so the communication is reliable within the network. This multicast layer was used for all the experiments we performed in the lab environment, as it was easy to implement and maintain. The issue this middleware layer had at the time we used it is that it does not work outside of a local intranet.

In order to solve the problem of the multicast layer not working outside a local intranet, we chose to implement the Hypercast multicast layer for our full system implementation. This multicast layer works by providing a central multicast server at a given server and port location. When the middleware server wishes to initiate a new multicast channel, it contacts the multicast server as a document sender and begins to push documents over a TCP connection to the multicast server. When a client wishes to join the multicast channel, it contacts the multicast server as a document receiver, which then organizes the new client into the existing multicast

tree. All connections between clients are TCP connections, which makes this multicast layer a reliable multicast layer.

Operation on Hypercast proceeds in the following manner: The server pushes the documents to the central organization server node, which then distributes those documents to the nodes that are at the top of the multicast tree. Those nodes then distribute the data to those clients below them in the tree, and communication continues until all clients receive the documents. In essence, the server publishes the documents to the central server, on which clients are subscribed and receive the documents as needed. This multicast layer, unlike the previous one, works over the internet and therefore was chosen to be used in our large network tests and in our final distribution package of our middleware.

Based on the discussion of the multicast channels above, one question is how does the client know where and how to access the multicast channels. Based on the type of multicast channel being used, the way the client will be connecting is different. For example, in JRMS the connection is to an address, which in hypercast it is to a central server which then connects the client into the multicast tree. Additionally, as with any program, there are different pieces of code that need to be loaded to access the different multicast layers.

We made our system to allow any type of multicast layer to be plugged in, which means that if there are three separate servers, each running the middleware, a client may have three different multicast channel types to connect into. Each channel has its own address and port combination that must be used to correctly access and join the channel. Because of the unknown address and type of multicast channel a server may be using, we have created an initial way for the server and client to interact and share this data.   The client side middleware component contains within it a table containing all the web sites it has visited and whether those sides are middleware enabled or not (meaning whether the web server is using our middle distribution system as a proxy). On the first visit to a new server, the client proxy sends out a discovery packet on the predefined proxy port (which we set as 8080). If the client does not receive an expected response from that request, it marks the server as non-multicast enabled and any further requests to that server are handled as regular server requests, where communication is pull based over a TCP connection with the server.
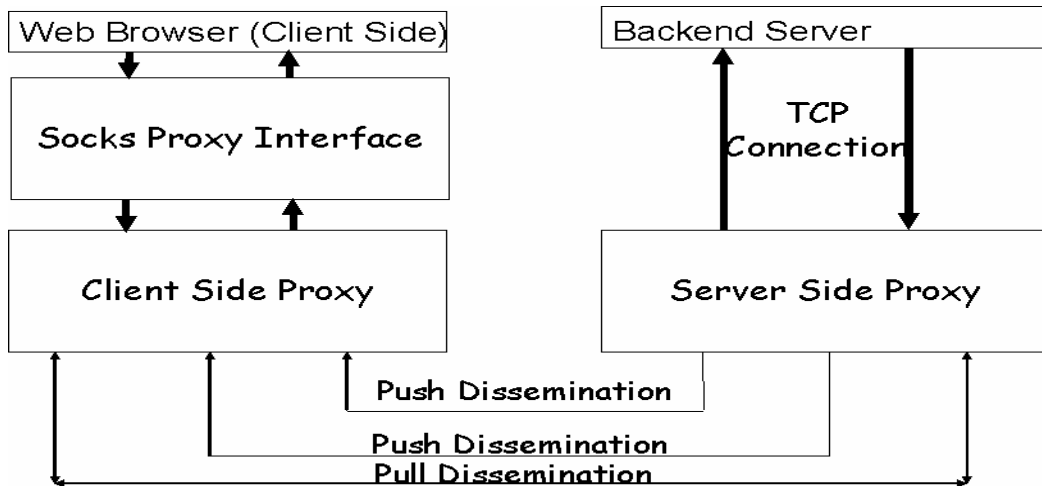
Figure 4 - Interaction between client front end, client proxy, server back end, and server

If the server is a middleware enabled server, it will be listening on the port 8080 in addition to the normal port 80. When it receives a request over port 8080, it generates an initialization packet for the client. This packet contains the address and ports for the two multicast channels along with the class type for the two multicast channels. When the client receives this information, it will create new instances of the two multicast channel classes. It will then use the address and port combination for that channel to join the two multicast channel and start to receive information.

If the server receives any requests for hot items from non multicast enabled clients, it will use the item to update statistics and respond with an information page. This page lets the client know the item is on the multicast channels and that the middleware should be downloaded and installed to properly use the web site. In this way, the clients that are multicast enabled can properly get the documents off the appropriate channels. Non-middleware enabled clients can still request and receive cold documents, but for popular documents will have to get the middleware themselves. This helps improve overall latency and server scalability in the long run.

Another aspect of our middleware implementation is the interaction between the client front end and the client proxy, and the server back end and the server proxy, which is shown in Figure 4. As the figure shows, the server proxy intercepts all requests that come from clients. Any request that the server proxy does not contain locally will cause the server to make a TCP connection to the back end and requests the data for the client. It will then pass on the data to the

33

client. This allows for any backend existing behind the server side proxy as long as it can handle any request from the proxy.

On the client side, the connection between the client and the front end web browser was done by updating a socks proxy to act as our client side proxy. A socks proxy is a client proxy which accepts requests from web browsers and forwards them to our client side middleware. It uses a standard protocol interface recognized by web browsers so requests can be easily intercepted and handled by the method we define. The socks proxy is responsible for trapping requests from the server, and forwarding them to the middleware client proxy. The middleware client proxy contains the list of servers (whether they are middleware enabled or not) and the logic to check and work with the multicast channels. The proxy requires very little work from clients, other than a simple setting for their web browser to use the socks proxy.

2.4   WIRELESS IMPLEMENTATION

With the advent of mobile devices and wireless communication, there has been much interest in developing systems to work in this type of environment. This type of system environment has been becoming more popular as the technology advances, the devices are able to remain small yet perform advanced operations, and people in general are forced to be on the go more than ever before. As more people go wireless, the systems that are built to deliver data must be able to handle the new workload that appears.

Unlike the wired environment, however, there are several differences in both the client's abilities and the method of communication that must be accounted for during development in a wireless environment. One of these major differences is the limited energy which the client has available to use. A mobile client may be running on batteries or other power sources that must be replenished. Actions such as communication and processing take away the energy at the client. When the client is out of energy, it will no longer be able to operate.

In terms of energy usage, communication is one of the major power drains that exist for mobile clients. In order to receive data, clients must power up their antenna, listen to the data till they receive the requested data, then power down the antenna. Whenever the antenna is in use, it is using energy, which aids in decreasing the available power of the mobile device. This makes
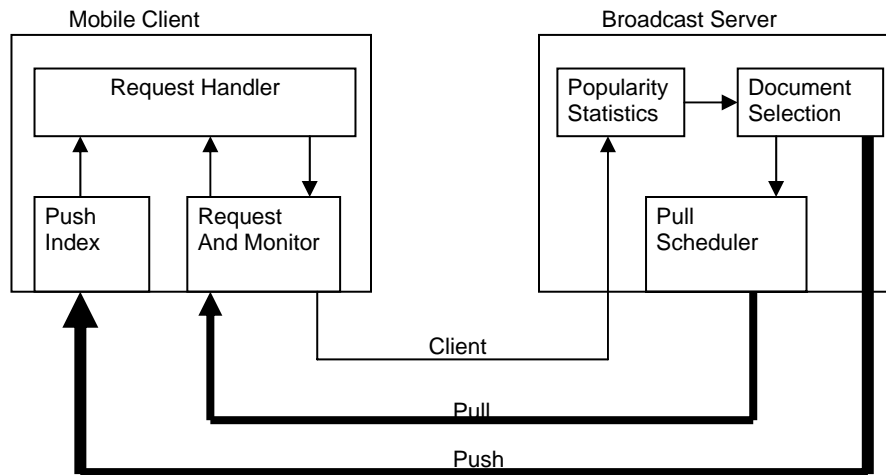
Figure 5 - Wireless System Architecture

limiting the amount of communication one of the constraints that should be accounted for against in a wireless based system. Additionally, communication is not the most reliable and clients can get easily disconnected, especially when trying to maintain a connection with a remote server. Thus, having clients communicating for the least amount of time is also a major goal of any wireless system.

Another difference that must be accounted for is the general mode of distribution in the wireless network. For the most part, wireless servers tend to broadcast data out to clients as opposed to maintaining a one to one connection with the client and using that connection to reply to requests from the clients. This means that having all clients connect to the server, make a request, and then maintain the connection, while the server queues the request and eventually serves, it will not work. Instead, connection from clients to servers should be brief connections during which the request is made to the server, and then the client waits to get the results over some other form of communication. We once again follow a three channel approach in our architecture for wireless environments.

Figure 5 shows our updated architecture for a wireless server version of our system. There are several differences to note between this architecture and the one we previously described. The first difference is that all the communication channels are only unidirectional in nature. There are once again three channels total, which are the *Broadcast Push channel*, the *Broadcast Pull Channel*, and the *Unicast backchannel*. The Unicast backchannel provides the means of communication from the clients to the server. Clients will connect to the server, submit

their request, and then immediately disconnect from the server. Because this is a low bandwidth channel, the server is not able to respond with the actual data for the client. Instead, this channel is used only for gathering request statistics and the actual requests which should be responded to. This can be seen as the same operation that requests for multicast push documents followed, where the client would send a "statistic" request only and then disconnect.

The other channels, the Broadcast Push and Broadcast Pull documents, are broadcast channels which are constantly pushing data out to the clients. The difference between the channels is in the type of data which they push out. The Broadcast Push channel pushes out the documents determined as hot by our document selection algorithm. Similar to the way it is done in the wired architecture, these items will be broadcasted out with an index which lets the clients know which documents are coming, which order those documents will appear in, and the request probability value for that item. Recall that the request probability value is the percentage of the requests for hot documents which will make a direct "statistic" request to the server.

The Broadcast Pull channel is another broadcast channel which operates in a similar manner to the Multicast pull channel in our previous architecture. On this channel, the non popular documents will be pushed out to all clients based on the client requests that come in over the client requests backchannel. The difference between this architecture and our previous architecture is that no data is distributed to clients over the unicast channel; otherwise the system operates as it normally does. The algorithm used to divide the documents is still the same as we used previously. One interesting note is on the use of the costs for sending an item over the pull channel versus over the push channel, which is used as the crux of our algorithm.

Although it may initially seem that using the same costs is not correct, it actually does work because the turnaround time at the server is the same regardless of how the pull channel set is implemented. Since we were only dividing the documents into push and pull, whether the pull channel is a push based pull channel or a unicast channel the amount of bandwidth needed to place the item on the channel is the same, and thus the algorithm is still relevant. If the costs had been the client response times, the algorithm would have to be altered. Since the costs used are the amount of bandwidth needed to meet the requested turnaround response time at the server, and given the same amount of bandwidth available at a wired or wireless server, the bandwidth costs to get out the data is the same. This allows us to use our algorithm to divide the documents, and still provide a close to optimal division even though the system has changed.

One change that must be made to our architecture is how the clients handle requests for pull based documents. In the wired architecture, all requests for pull based documents are made directly to the server. In the wireless architecture, because all clients get all the data being requested, it is possible for a client to check the index on the pull channel and get the requested document without having to make a direct request. This also allows for selective tuning and further energy savings from the client, by being able to get the data without having to make a request. The issue is that this will skew the popularity of items, which are almost popular enough to be on the broadcast push channel, but are not on it yet. Since these will be the items that appear the most on the broadcast pull channel, they would also be the items the users could get without making a direct request. By not making a request, the items perceived popularity would fall off, when it should have been increasing and eventually placed on the broadcast push channel. Thus, our solution is that even though the client did not have to directly request this item, the client will make a request to the server for it similar to the way the requests are made for items on the broadcast push channel. The difference is unlike the broadcast push channel, where requests were made only x% of the time, requests will be made for every item gotten off the broadcast pull channel if a direct request was not made first.

Another change that must be made is how we handle the race conditions between the channels. The race condition that can occur is if the item being requested has just been moved to the broadcast push channel when it was previously on the broadcast pull channel. When the client requests the document from the server, the server may have already have it scheduled to be placed on the broadcast push channel. The client, having already checked the push channel index, will expect the document to appear on the pull channel. Thus, the client will continue to monitor the pull channel while the server is placing it on the push channel, causing the client to never receive the document it requested. This is an unacceptable condition in our system which focuses on lowering response times and increasing overall system scalability.

Our solution to this race condition is to have the server continue to broadcast items even after they have been scheduled and moved to the broadcast push channel. These items will only be sent over the broadcast pull channel, however, when a non statistic request (recall the requests for items on the push channel are statistic requests) appears for an item on the push channel, the item is still scheduled and placed on the pull channel as well. The client will then eventually receive the requested item over the pull channel, although the delay may be more than if the

client had been monitoring the push channel. Thus, we prevent the starvation that the race condition can result in, and ensure all clients receive the data they have requested.

The final change we made to convert our architecture from wired to wireless is found in how the broadcast pull channel distributes data. In the wired version, the multicast pull channel simply pushed items out on it as necessary, without telling the clients what was coming through the use of an index. In the wired architecture, where the client may selectively tune to help with saving energy, an index on the broadcast pull channel is necessary. This will not only allow clients to know what is coming over the broadcast pull channel, but also when it is coming. This index is dynamic in nature, though we limit the size to 10 documents, so that clients are not waiting too long for items that were requested. This index will be created and sent out while the previous broadcast cycle is occurring, thus simulating a combination of the multicast pull and unicast channel in our original system design.

3 DOCUMENT SELECTION AND BANDWIDTH DIVISON

In this chapter, we examine the division of documents and bandwidth among the different types of channels. We first focus on the division of documents and bandwidth between the Multicast Push channel and the collective set of pull channels. We present an algorithm to achieve this division of documents and bandwidth in a near-optimal fashion. Then, we present an algorithm for dividing the documents and bandwidth among the two pull channels.

3.1 DOCUMENT AND BANDWIDTH DIVISION BETWEEN PUSH AND PULL CHANNELS

As it was mentioned in the Introduction, the division of documents and bandwidth is vital in a hybrid data dissemination scheme. The combination of using a push channel for one set of documents and the pull channels for the remaining documents gives the system high scalability with low overall latency. In the hybrid scheme, the server must dynamically assign each document either to be pulled by the clients, by placing it on the unicast pull channel, or to be pushed out to clients, by placing it on the multicast push channel, in a process called *document classification*[47].

In addition to dividing the documents, the server must also partition dynamically its bandwidth between the pull channels and multicast push channel, in a process referred to as *bandwidth division*. Document classification and bandwidth division are actually inter-related issues because a given bandwidth division determines the performance of a document classification choice and, conversely, a given document classification determines a bandwidth split that optimizes performance. In turn, both document classification and bandwidth division depend on the popularity of data items because download latency is smaller when hot items are

assigned to multicast push, cold items to unicast pull, and the bandwidth is divided appropriately between the two channels.

Because of this relationship, the server must estimate the popularity of the documents it serves in what is referred to as the *push popularity* problem. The estimation of document popularity is complicated by the fact that no requests are made by clients for multicast push documents. In particular, if the popularity wanes for a specific document and that document was on the multicast push, the shift in client interests is not reflected in request logs at the server. In turn, the server would not know that it is time to demote a document to the pull channels. What is required is the ability to determine the popularity both of items for which requests are made (the pulled documents) and also those for which requests are not made (the pushed documents). A solution to this problem was presented in Section 2.2.1. In it, we provided a way to estimate the popularity of documents both on the push and pull channels, as will be required in our selection-division (SELDIV) algorithm.

SELDIV, whose details are presented next, is an integrated algorithm for solving simultaneously and to near-optimality the bandwidth division and document classification problems. Our algorithm is evaluated through emulations on a comprehensive middleware platform for scalable data dissemination. The algorithm exhibited lower average latency than previous schemes. The underlying reason for this performance is that if document selection is addressed separately from bandwidth division, a certain bandwidth split can be fixed to a level that is suboptimal for a certain assignment of documents to channels. More generally, the performance trade-offs differ quantitatively and qualitatively under the combined scheme. For example, the assignment led to multicast push latency that is significantly faster than pull latency due to the higher relative popularity of multicast items over unicast documents.

3.1.1 SELDIV Algorithm

To better understand the SELDIV algorithm, it is first necessary to understand the differences in average latency for the multicast push and for the unicast pull channels. The average latency for documents on the multicast push channel is roughly linear in the number of documents on this channel. Specifically, the average latency for a document on the multicast push channel is half of the period of the broadcast cycle, since we assume that documents are sent once, not

Parameter	Description
n	Number of Documents
λ	Observed Request Rate λ
α	Pull Over-provisioning factor
L	Current Required Latency
B	Total Available System Banndwidth
S	Array of document sizes S_i
p	Array of document probabilities p_i
ε	Tolerance factor

Table 1 - Algorithm Parameters for document selection and bandwidth division algorithms

fragmented, and broadcast sequentially. In particular, the delay expected on the push channel is equal to half the total time it takes to broadcast all documents placed on the channel, which given that document *i* is of size S_i, is $\sum \frac{S_i}{2}$ for all *i* on the push channel. This makes one of the goals to keep the amount of data on the push channel as small as possible while maintaining the pull channel below a full load level, as that will enable very fast response times for items on that channel, which helps to lower system latency.

The delays for pulled documents, however, are radically different from those of pushed documents. If document *j* is assigned to unicast pull, a client request for *j* is queued at the server for transmission. Let S_j be the size of document *j*. Basic queuing theory tells us that the corresponding queuing delay is either $O(S_j)$ or unbounded, depending on whether the server load is less than 1 or not. Thus, to minimize average latency, the server should require as many documents as possible be pulled, as long as the load for the pulled documents is bounded by a constant less than 1.

Our solution to document classification and bandwidth division is to use an integrated algorithm that minimizes average latency. The steps of our algorithm are shown in Algorithm 1. This solution works in conjunction with a subroutine shown as Algorithm 2. Algorithm 1 uses a tolerance factor ε > 0, which is an arbitrarily small positive number, and finds a solution that has latency within ε of the optimum for the given bandwidth and popularities. The algorithm also assumes that the list of documents passed in is ordered by decreasing popularity (meaning item 1

Algorithm 1: SELDIV - Bandwidth Division and Document Classification

---

Require n, λ, α, B, S, p, and ε as defined in Table 1, and $p_i \geq p_{i+1}$ ($1 \leq i < n$)
Ensure k is the optimal number of documents on the push channel, pullBW is the optimal pull bandwidth, pushBW is the optimal push bandwidth
1. **for** i =1….n do
2. $rspt_i = rspt_{i-1} + p_i S_i \lambda$
3. $sizeTotal_i = sizeTotal_{i-1} + S_i$
4. **end for**
5. $lMax = sizeTotal_n / B$
6. $lMin = 0$
7. **while** ($lMax - lMin$) > ε **do**
8. $L = (lMax + lMin)/2$
9. $k = tryLatency(L, p, \lambda, n)$
10. $pullBW = \alpha(rspt_n - rspt_k)$
11. $pushBW = B - pullBW$
12. **if**($pushBW \geq (sizeTotal_k/(2L))$)
13. $lMax = L$
14. **else**
15. $lMin = L$
16. **end if**
17. **end while**

---

is the most popular, item *n* the least popular) and that the list includes the popularity of the items on the push channel. The parameters for the algorithms are summarized in Table 1.

Algorithm 1 proceeds in the following manner. It first pre-computes the sums $\sum_{i}^{k} \lambda p_i S_i$ (which is a running total of the bandwidth requirements for the first k items) and $\sum_{i}^{k} \lambda S_i$ (which is a running total of the size of the first k items), placing the totals in the arrays *rspt* and *sizeTotal* respectively (Lines 1-4). This will help to optimize the run time because these values would otherwise have had to be computed during every loop. The algorithm then sets the initial minimum latency to be 0 and maximum latency to be the amount of time required to send all the documents out over individual connections with each client (Lines 5-6). A binary search is then performed, for which each loop consists of taking the average latency between the current minimum and maximum latencies and passing that average latency to the subroutine in Algorithm 2. Algorithm 2 will use that average latency to calculate the number of items *k* that

---

Algorithm 2: tryLatency subroutine

Require: n, λ, L, p as defined in Table 1
Ensure: returns the number k of items pushed given that average latency of L is required
1. **while**{Max-min > 1} **do**
2. k = (max+min)/2
3. **if**(($p_k λL$) > ½) **then**
4.   min = k
5. **else**
6.   max = k
7. **end if**
8. **end while**
9. Return k

---

should be placed on the broadcast push channel (recall that the list of items is ordered by popularity, so this *k* refers to the *k* most popular documents)(Line 8-9).

Using the return value of *k* most popular documents, Algorithm 1 calculates the amount of bandwidth that should be given to the pull channel based on the choice of *k* (Line 10). Notice that in this calculation, a value α > 1 is used that measures the target level of over-provisioning for the pull channel. More precisely, the actual bandwidth we reserve for pull is α times what an idealized estimate predicts. Queuing theory asserts that α > 1 guarantees bounded queuing delays, whereas α ≤ 1 leads to infinite queuing delays. As such, the parameter α can also be thought of as a safety margin for the pull channel.

After calculating the pull bandwidth, the remaining bandwidth is partitioned to the push channel and it is determined whether the amount of push bandwidth provided is actually enough to sustain the amount needed for the push channel (Lines 11-16). If there is enough bandwidth, then the latency could be lowered, and the max latency is decreased and the search performed again. Likewise, if there is not enough bandwidth, the latency is increased and the search performed again. This continues until the ε value is met, at which point the number of items for the push channel (and therefore which items), the push channel bandwidth and the pull channel bandwidth are all returned to be used to divide the bandwidth and documents for the system.

Let us now examine the details of Algorithm 2. Algorithm 2 requires as input latency along with the request rate (λ), number of documents (*n*) and the list items with popularities *p*. It

then calculates and returns *k,* the number of items that should be placed on the broadcast push channel. The starting point for Algorithm 2 is a method suggested[8] that minimizes the bandwidth *B* to achieve a target latency *L.* The known method is not directly applicable to document classification and bandwidth division because our goal, on the contrary, is to minimize the latency *L* given a fixed amount of available server bandwidth *B.*

Algorithm 2 operates by using two bandwidth costs, one if the item is kept on the pull channel and one if the items are placed on the push channel. If document *i* is assigned to the pull channel, it will use bandwidth $\lambda p_i S_i$. If document *i* is assigned to the push channel, it will use bandwidth $\dfrac{Si}{L}$, which is also the rate at which the document must be broadcast to give *worst-case* response time *L.* As it was stated[30] a document should be pushed if $\lambda p_i S_i > \dfrac{Si}{L}$. Because the items are passed in with an order of most popular (first item) to least popular (last item), a binary search can be performed on the items to find the item *k* at which the division should occur.

As the algorithms above show, we are able to determine, based on document popularity, request rates and available bandwidth, the correct division of documents for the push channel and the amount of bandwidth to provide to that push channel. This gives us a solution to the *document selection* and *bandwidth division* problems which exist for hybrid systems, allowing us to develop a fully integrated hybrid system for highly scalable data dissemination.

The running time of our solutions consists of several parts. First, we assume that the list of documents is ordered by popularity. The first time this list is created, it will take *O(nlog(n))* to create the list, and *O(log(n))* to maintain the list thereafter. The actual run time of algorithm 2 is *O(log(n))* for each run through with a different latency, and for algorithm 1 the runtime is $O(max(n,log(\dfrac{\sum_{i=1}^{n} S_i}{B\varepsilon})))$ for the number of loops through that binary search. Thus, the overall runtime is $O(max(n,log(\dfrac{\sum_{i=1}^{n} S_i}{B\varepsilon}))log(n)).$

3.2   DIVIDING DOCUMENTS BETWEEN MULTICAST PULL AND UNICAST

In this section we examine the division of both the documents and the bandwidth between the Multicast Pull channel and the Unicast channel. The documents and bandwidth available to the pull channel are the results of having already run the document selection and bandwidth division algorithm from the previous section. In the previous section, the multicast push channel was used to handle very popular documents and the pull channel was left for all other documents. We now focus on how, and when, to take documents and place those on the multicast pull channel in order to minimize system latency while maximizing both system scalability and system adaptability to changing request environments.

We will present three different ways that the division of the pull documents can be done between Unicast and Multicast Pull. It is important to understand that the goal of this division is to provide the system with a way to not only manage a new influx of requests during times of document popularity shifts, but also to still try to meet the goal of minimal response times. This goal immediately rules out the idea of having all documents being multicast pulled and having no unicast responses at all, when operating in the wired environment. The reason for this is that, as is shown in Figure 6, using only Multicast Push and Multicast Pull, without using unicast to directly respond to requests, performs significantly poorer than using Multicast Push and Unicast together without Multicast Pull.

The reason using only Multicast Pull and Multicast Push performs poorly is that the overhead required to put an item onto the Multicast Pull channel, send it out, and have it received by all clients is much higher than a single Unicast connection between the client and server. The bandwidth used by the server is the same, so the overhead is that to distribute the data to all clients connected on the multicast channel, which can include a large number of nodes. Because the data must be distributed to each node, this can cause significant delay to the nodes that are awaiting the answer, especially when only one node is awaiting the response. Using only the Multicast Pull channel can also cause clients who are awaiting documents to have to wait while requests not related to their own go across the channel before their own requests appear. This leads to long response times for the clients, which makes using Multicast Pull alone a very unwanted solution to data dissemination. Multicast pull should be used only when it will be
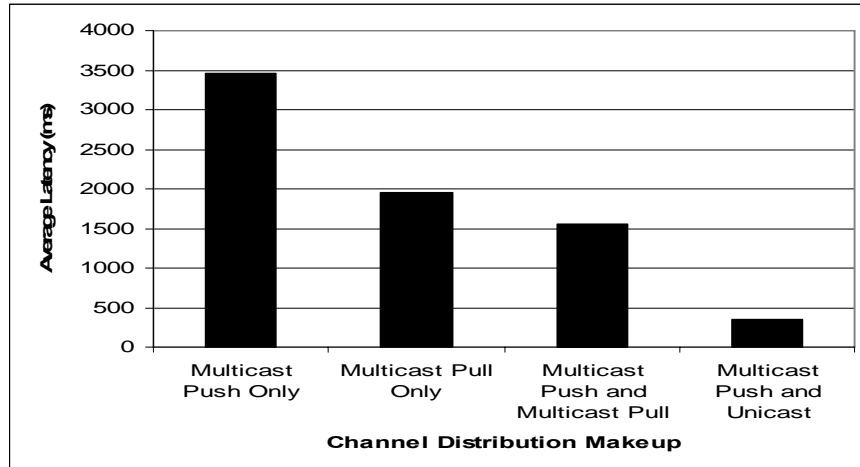
Figure 6 - Comparing different distribution methods

beneficial to the overall system, either to lower client response times or to alleviate request pressure at the server.

As mentioned earlier, we have developed several solutions to how to divide the documents between the Unicast channel and the Multicast Pull channel. This division is assumed to occur after the documents have already been split into push and pull. The solutions are:

- Use a threshold
- Rerun the document selection we have developed, but with updated costs for the multicast pull channel
- Use the channel as an intermediate channel to stage documents on both being upgraded to and dropped from the multicast push channel.

We will discuss all three of these solutions in details along with the pros and cons of each approach.

3.2.1 Using a Threshold

One of the ways that the multicast pull channel can be utilized is to serve as a runtime channel for the dissemination of data when the number of pending requests exceeds a given threshold. Requests are served one request at a time as usual by the request handler thread within the

system. When the time comes to disseminate out a given document, the number of pending requests is checked. If that value exceeds the system defined threshold amount, the item is sent over the multicast pull channel instead of being sent over unicast to each individual client. Clients will listen to the pull channel and unicast channels when making requests. When they receive the document, from either channel, they will return it to the client. There is no index of what is coming over the multicast pull channel in this case, as it is dynamic at the time the document is being served.

There are multiple advantages to using a threshold to determine if documents should be placed on the multicast pull channel or the unicast channel. The first advantage is that the threshold is relatively easy to implement and quick to use when running the system. The server in general will bring in requests and serve those requests in order as they arrive. When using the threshold approach, requests are queued at the server as would usually be done. When a new request arrives, the queue is checked for that request. If the request is found, instead of adding a new request to the queue, the count is updated. When the request is finally served, if the count exceeds the threshold, it is sent over multicast pull. Otherwise, it is sent over unicast to each client individually.

Another advantage is that the multicast pull threshold can be tuned to improve system performance as needed. Because the threshold is a simple number, it can either be seen as a passed in system parameter. This can be tuned based on simulations to create the best split between channels. It can also be an adaptive threshold that changes based on the current conditions within the system. Using the multicast pull threshold becomes very easy, and makes the entire system simple to implement. Adding in the ability to adapt the threshold to the system environment would require slightly more work. It requires that a tolerance still be specified for how much of an increase in load or response times is allowable before the system should switch to multicast pull. This channel would then be used for most documents until the request rate can be stabilized by running document selection and recreating the multicast push channel.

One final advantage of using a threshold based mechanism for document division is that it can allow the system to completely change the data distribution makeup of the system. A threshold of 0 would create a data dissemination system where hot items appear on the multicast push channel and all other items appear on the multicast pull channel via requests. This would allow a low ability server to still handle large work loads by not having to maintain connections

with clients. As requests come in, the server would determine which push dissemination channel to place the data on without continuing connection with the client. In the same manner, if the server was very powerful or part of a larger server farm, the multicast pull channel threshold could be set extremely high, allowing all non push items to be requested and served over unicast. The threshold method is very adaptive to overall server setup while allowing users a degree of control over how the server proxy is set up and used.

Using a threshold has a major disadvantage as well. As a benefit above was deciding how to tune the threshold amount, deciding on that threshold amount is a disadvantage. The amount that a threshold is set at, tested at, and works at for a given workload, may cause different results if the workload changes. This is seen in many applications where the reported best experimental threshold does not hold up when the system is used for a different purpose than tested. In order to properly use the static threshold, several trial runs may be necessary, which makes the setup time for the system very high. If the threshold is just arbitrarily set to limit set up time, it may hinder system performance in a way unforeseen.

This problem also exists for dynamic threshold, where the method for increasing and decreasing the threshold needs to be determined and explored. A dynamic threshold must be developed to handle any situation that may appear, something that is no small task. Creating a method that does not adapt quickly enough (or that adapts too quickly) to the changing environment could cause the division to be useless. This means it will not provide the benefits the multicast pull channel is meant to create. Having a method that generates the correct division most of the time, but does not in critical situations, may be just as bad as having no division at all. Having a threshold in general can provide many benefits but the overhead and planning to develop the right threshold can be a very hard and daunting task.

The bandwidth division for using such a document division system is another issue that must be resolved when using a threshold. One way the bandwidth can be divided is on an as needed basis. This means that the multicast pull channel gets as much bandwidth as it needs to send out all documents placed on the channel, while the rest of the bandwidth is given to the unicast channel at all times. This gives a minimal amount of wasted bandwidth, because the multicast pull channel is never using more than it needs. However, this provides no mechanism to stop the multicast pull channel from starving the unicast channel, which could cause problems when requests are still coming over the unicast channel into the server. In this case, a method

must be created which divides the bandwidth but does not leave any channel starved, which would in turn hurt overall system response time.

3.2.2   Using SELDIV

Another solution we have developed for dividing the pull documents between multicast pull and Unicast is to rerun the SELDIV algorithm we have previously described but this time only on the pull set of documents, instead of the entire document request set. This will give us a set of documents that should be placed on the multicast pull channels and should therefore be pushed out to clients in a similar manner as the multicast push channel. When clients are making a request, they check the multicast push channel, and if the document is not to appear on it, send a direct request to the server. Unlike previously, when the client starts to simultaneously listen to the multicast pull and Unicast channels, the client will have a multicast pull index to check for the requested document. If the document is coming over the multicast pull channel, the client can listen and receive it from the multicast pull channel, instead of having take a wait and see approach.

One advantage to this approach is that when the client makes a request over the pull channel, it will immediately know which channel the result is coming over based on the index for the multicast pull channel, which is more defined than in the previous approach. This allows clients to drop connections sooner than before, which at times when the documents on the push channel need to be reconfigured, can provide even more leeway for the server to handle the changed document load. This is especially important when the multicast pull channel is needed the most, when the document selection was inaccurate or changing popularities are causing the server to be experiencing massive overloads in the number of requests appearing. Instead of the server having to maintain the connections with clients until the item is serviced and determined to appear on the multicast pull channel, both the server and client will know immediately what channel the document will appear on, and can adapt appropriately.

Another advantage of having a multicast pull channel with a defined index is that it can be used to pre-filter requests coming into the server. Before, we mentioned that connections could be dropped immediately after being made if the server and client know what channel the document is appearing on. With a defined index, these initial request connections can be filtered

in a similar way that the multicast push channel behaves. Either only a certain percentage of clients could send their requests or if needed, no clients could send their requests. This would allow the system to handle even more Unicast connections if needed, in cases where the request patterns are too random to properly handle with the multicast channels. This would provide even more scalability than previous methods.

A final advantage we note is in the decreased response time having a continuously pushed multicast pull channel could provide. Similar to the way the multicast push channel behaves, the multicast pull channel would have a set of documents on it because those documents are near popular, meaning they are receiving a fair number of requests. This means that there will be a lot of clients whose pull based requests are for items currently being pushed out on the multicast pull channel. Based on the size of the cache provided for the multicast pull channel, this can provide immediate responses in the manner the multicast push channel does. This in turn provides lower latency for clients than making the Unicast request to the server.

Using our previously defined document selection algorithm also provides an inherent advantage, that of giving the bandwidth division to use with the given document division. Because the division of documents with our algorithm assumes constant sending out of documents on the pull channel, it will need a set amount of bandwidth to use for the sending of those documents. The algorithm gives that division, which will properly change itself based on the documents that are chosen to be placed on the multicast pull channel.

One of the major problems with using our algorithm to divide the documents is it gives the set of documents to place on the multicast pull channel, assuming those items will be continuously pushed out to clients. This creates another area where a misjudgment of which documents should be placed on the channel can cause problems in cases of ever changing document popularity, an issue that did not exist when the decision of which channel to use was based on a runtime threshold check. This means that if the multicast push channel currently has documents that are no longer hot, and the multicast pull also has the wrong set of documents, the unicast is forced to handle a load beyond its current abilities.

Another disadvantage this method of document division has is that it is taking away a set amount of bandwidth from the unicast channel, when means the unicast channel does not have as much bandwidth available to service requests. Because the system is designed for low latency and high scalability, the unicast is relied on to make much of the low latency occur. By giving

the unicast channel less bandwidth, it will not be able to service documents at as high of a rate as before, which could lower performance. As mentioned above, less bandwidth available also leads to a lower ability to handle times of changing or misjudged document popularity.

Using our algorithm also brings into question when to run the document selection algorithm on the documents on the pull channel. Towards these questions there are several solutions that can be examined. One idea is to run the document selection when the push document selection is run. This will allow for the greatest sharing of information between the two algorithms and ensure that both are using the same statistics. The issue there is that if the push selection is not run often enough, the multicast pull channel may actually be wasting a lot of bandwidth because the documents on it were only warm for a brief period of time and no longer belongs on the multicast pull channel.

Another method for determining when to run the document selection is to once again check the current rate of requests coming into the system. If the rate of requests coming in has increased beyond a certain tolerance, the document selection should be run. This may be further pushed forward by having the document selection on the multicast pull channel run every time it finishes its cycle. This would provide the greatest accuracy in correctly placing items on the multicast pull channel, but could put extra strain on the server to calculate and schedule the items on the channel.

3.2.3 Use Multicast Pull Channel as an Intermediate Channel

One final way we have determined the multicast pull channel can be set up with documents is as a intermediate channel for items that are not quite popular enough to warrant being put on the multicast push channel, but are popular enough that serving each request individually is not the best approach. In essence, this approach combines the ideas of the two approaches above. It takes the constant broadcasting from the approach of using our algorithm with a threshold for popularity that determines if the item should be placed on the multicast pull channel. The idea behind this approach is that as a document becomes more popular, it will move from the unicast channel, to the multicast pull channel, then to the multicast push channel. Likewise, items that go from popular to unpopular must go first to the multicast pull channel before moving to the

unicast channel. This is very similar to the channel system presented in Air Cache[47], except we physically have a "warm" channel and theirs is a logical one.

The advantage of using an approach like this is that it provides a nice way to move documents gracefully from the push channel to the unicast channel (and vice versa) without getting caught by false positives. By false positives, we mean items that are determined to be hot when they are actually only getting a brief popularity spike. Additionally, it helps with false negatives, where popular items are briefly deemed not popular enough to be on the push channel perhaps because they are no receiving enough requests from the feedback mechanism. In these cases, having an intermediate channel will prevent the popular channel from having brief popular items on it and prevent hot items from going directly to unicast.

The disadvantages of this approach are actually related to its advantages. While using this method allows for items to not be prematurely removed, it also can cause items to remain on the hot channel or prevent items from getting onto the hot channel soon enough. This leads to a very hard question of does the method of dividing the documents need to take into account the fact that they will still remain on certain channels and not instantly go one way or another. It is a very delicate balance that must be performed in order to make sure that the system maintains its low latency while providing the scalability we envision.

Another set of disadvantages that pertain to this approach are all of those that are listed for the approach of using our algorithm to divide the pull based documents into two groups. As a quick summary, these were wasted bandwidth, misjudged document popularity, less bandwidth available to Unicast to use, and the amount of time and power needed by the system to process which documents to divide into which channels. The bandwidth issue is very important as it must be appropriately set to match the channel allocation. This is a key aspect of our algorithm and would have to be developed is this new division method is to be used.

3.2.4   Overall Analysis

Based on the above analysis of the different methods for splitting up the pull documents into those to be placed and serviced from multicast pull and those for unicast, we come to the following conclusions, as shown in Table 2. As this table shows, we have identified five main areas in which these different methods can be measured. Those areas are whether the approach

Method	Lowers Latency	Increases Scalability	Handles Bursts	Affected by mispredictions	Ease of Implementation
Threshold	Low	Medium	High	Low	Medium
SELDIV algorithm	Medium	Medium	Low	High	Medium
Intermediate Level	Medium	Medium	Low	High	Low

Table 2 - Comparison of different distribution methods

lowers latency, whether it increases scalability, how well the approach handles request bursts, how the approach is affected my bad popularity predictions, and how easy the approach is to implement.

The table shows that both approaches which continuously use the extra channel instead of using it only on demand will provide better latency in general that the threshold method. This is because with the continuous pushing of items, there is a chance clients will be able to instantly get their request result without have to make requests and await the responses. This can lower the latency for the system overall. Additionally, all three approaches seem to be similar in how well they can increase the scalability of the system. While the threshold scalability is based on per request service basis, the others are based on putting the semi-popular items on a push based channel, and providing scalability in a way that the original push channel created scalability.

While scalability and latency are the focus of our system, the ability to handle bursts and mispredictions is also a very important. Average latency may be slightly lowered using certain approaches, but if those approaches can not handle bursts, they may end up hurting overall system performance. In these cases, it is actually better to use a method that is not depending on the items being correctly predicted and having to continuously push the items out. In this case, an adaptive, run time approach similar to the threshold may actually be a better fit, as it will allow the system to quickly adapt when it is needed.

One final aspect that is always of interest is how easy or hard it is to implement an approach. In this case, using a threshold or our algorithm we determine as being similarly easy, while using the intermediate level is a little more complicated, mainly because based on our current method for diving documents between push and pull, we already have the ability to run the algorithm again and get the final split. This gives an advantage in implementation over the

intermediate level approach, which needs to be developed and implemented form scratch. Overall, we find that any method used provides similar benefits. Because we feel that handling bursts and mispredictions is more important than a slight decrease in latency, we chose to use the threshold approach in our system.

4   EXPERIMENTS

In this chapter, we present our experimental methodology and result set for exploring the effectiveness of our architecture and associated hybrid scheme. We will present two distinct experimental sets, consisting of different experimental test beds and results. The first set of experiments was performed to test the various aspects of the architecture in a more controlled environment, where we could check the features of our system. We then set up and ran large scale experiments in a more widespread environment, noticeably the Planet Lab[43] simulation environment, to test overall scalability and to validate the results that we get from our simulated experiments. The overall reason for these experiments is to validate our claims on the effectiveness of the architecture aspects we have developed.

4.1   SIMULATION EXPERIMENTS ON ARCHITECTURE ASPECTS

In this set of experiments, we focus on evaluating the aspects of our architecture that are unique to our architecture. In particular, we want to look at the effectiveness of our algorithm along with the effectiveness of the additional channel we added, the multicast pull channel. We look at this additional channel both in the light of static popularity patterns (where what is popular stays popular) and dynamic popularity patterns (where what is popular either shifts or popularity spikes occur at random times). We also look at how the altering of when (how often) the document selection algorithm is run affects the performance of the system.

The main evaluation metric for this set of experiments is the client-perceived delays to download requested documents. This delay can be seen as the amount of time that lapsed from when the user enters the request into the web browser to the time that the completed requested document is delivered to the client. The purpose of this metric is that it meets the overall goal of

our system. Delay statistics incorporate the system's response times for requests and its resiliency to unexpected load peaks. If the server is not able to properly scale to given workloads, the response times experienced at the client will reflect this fact. This also allows us to directly compare the multiple enhancements we have, isolating individual parts or tests to see how our proposed improvements affect the system. We can also use the results to cross compare different experiments, since all the experiments focus on the same metric and its meaning does not differ amongst experiments.

All results shown are in milliseconds but should be interpreted as relative time values. Because of the isolated environment these experiments are run in, many of the response times are in the order of at most a few hundred milliseconds. This may cause the impact of the difference between result times to seem inconsequential, though this could be no further from the truth. Instead, we view the results as relative measurements. For example, if scheme A has a running time of 50, and scheme B has a running time of 100, scheme A should be seen as twice as fast and the numbers as relative units. We did measure them as milliseconds, and due to the environment we are using; a millisecond is a very long time. There is no network delay to speak of, so the measurements are how fast the system can turn documents around and produce results. Therefore, all results should be seen as relative differences, instead of actual differences.

In this set of experiments, the system used to run the experiments on was an emulation of the architecture described in the Chapter 2.2 as the Improved Hybrid Data Dissemination Architecture. The emulation environment consisted of a simulated application that uses the architecture as a middleware between the client and server side applications. The actual environment is a local area network where most of the networks lag and traffic has been filtered out. This allows us to avoid network-induced variability and to isolate the intrinsic properties of multicast pull

The way such an environment was generated was by running the server and the associated multicast servers on one physical machine and the client on another physical machine. Both machines were within the same internal network and located in the research lab. As mentioned above, this caused the resulting time units to be very small and in fact mainly focuses on the turn around time for documents within the server. Because of the minimal network lag and interference, all the response time is that of the server receiving the request, servicing the request, and returning the results to the client. The only other major factor

in the response times was the time it took for clients to check for items on the push based channels, which makes sure that the times include the overhead associated with our improved hybrid architecture.

The actual hardware used for this set of experiments was two 2.0Ghz dual processer machines with 1.2GB of RAM and running the Linux Redhat 8.0 operating system. Each machine was isolated from other activity during the times of operation so that the machines efforts could be focused on operating the server and the client middleware. All code used was written using Java 1.4 as the coding language. The multicast channels were created and maintained through the use of JRMS on the local network tests and Hypercast on the real world experiments. For Hypercast, both of the multicast channels ran on the same machine as the server side middleware.

In this set of experiments, we kept the documents at a fixed size: little deviations were found when documents have variable size and we include variable sized documents in our experiments found in section 4.2. Generally speaking, the correlation between document size and popularity is unclear[19] or weak and can be ignored[17] for the most part. In order to place the server in overload position, which is where we wanted the server to be in order to test the effectiveness of both hybrid schemes in general and our improved scheme, we created a simulator module to generate requests, which we refer to as the request filler. The request filler only added requests into the queue at the server if those items were not on the push channel, but it did not make explicit TCP connections with the server. The latency was measured at one client that actually made the connections with the server as a normal client would. On the whole, the client generated 10,000 requests during each experiment. The other relevant experimental parameters for this first set of experiments are found in Table 3. All parameters shown contain both the range of values that were available during the experiments and the default value for that parameter. All parameters are fixed to the default value unless otherwise stated in a given experiment.

Parts of these experiments were executed to evaluate non-stationary access patterns and the impact of multicast pull. In these experiments, the document popularity follows a Zipf distribution, which means that the ith most popular document is requested with probability proportional to $\frac{1}{i^\theta}$. However, the exact identity of the ith most popular document changes with

Parameter	Value	Default
Document Size	0.5KB	0.5KB
Zipf Parameter	1.1 - 2.0	1.5
Multicast Pull Threshold	2	2
System Bandwidth	100 KB/Second	100 KB/Second
Request Rate	250 requests/second	250 requests/second
Re-configuration Period	1 - 60 seconds	5 seconds
Total Available Items For Request	1000 documents	1000 documents
Total Requests Made	10000 requests	10000 requests
Hot Spot Movement Type	Off, Small, Big	Off
Alpha Parameter	2.0	2.0

Table 3 - Simulation Experiments Relevant Parameters

time. Changing popularity will be considered in the following two models:

- *Small move model:* The popularities change gradually over time to reflect a gradual client shift in interest over time. In this model, periodically each document would swap popularities with the next most popular document with probability $\frac{1}{2}$. For example, with probability $\frac{1}{2}$, the second most popular document would become the third most popular document. For these experiments the access probabilities change every 500 requests received from the monitored client.

- *Big move model:* The location of the most popular document changes suddenly to reflect a sudden change in client interest, perhaps in response to an important event. The shift is simulated by making the probability of requesting the ith document proportional to $\frac{1}{(1+(i-b)\bmod n)^{\theta}}$. The ordinary Zipf distribution corresponds to $b = 0$ and the document popularity can be quickly rearranged by changing the value of b. This means that by changing b to 1, documents will shift in popularity by one, so the most popular is no longer popular and the second most popular is now most popular, and so on.

We also experiment on the varying time between when the document selection was last run and when it will be run again. This is relevant because with over one thousand possible documents to request, using our algorithm may or may not cause the server to spend a lot of time performing the document selection. In addition, the time between running the document selection can cause the server either to be accurate in responding to request changes or take too long and therefore cause the server to become overloaded. Because our hybrid architecture requires document selection to be run at some point, we explore different times between runs to see how the system is affected.

4.1.1   Selecting the correct α value

One of the most important variables used in our algorithm is the α value. As explained in the description of our algorithm, the α value is an over provisioning factor used to give more bandwidth to the pull channel. The reason more bandwidth should be given to the pull channel is that if the document selection is off by even a little, there is a chance that the pull channel will have to handle a large load of documents, which take up a lot of bandwidth. Having additional bandwidth provided to the pull channel allows for miscalculations to be taken in stride instead of causing a complete system lockdown. The question that remains is how much additional bandwidth should be given to the pull channel?

Figure 7 shows the effects of various values of α on the average latency of the SELDIV algorithm from Chapter 3.1.1. The curve in Figure 7 is jagged because an infinitesimal change in α can have a discrete effect in the number of items pushed. Figure 7 shows that the value of α that minimizes average latency is between 2.0 to 3.0. Notice that the difference in latency is not very high as the alpha value is varied between 1.8 and 3.0. While at first this seems confusing, it can be explained in the following way. The push channel needs a certain amount of bandwidth to continuously push documents to clients while the remaining bandwidth can be given to the pull channel. If the SELDIV algorithm is examined, we actually determine the pull bandwidth first and give the remaining bandwidth to the push channel. The value of α is used in this equation, and is used to determine the amount of bandwidth that will be given to the pull channel.
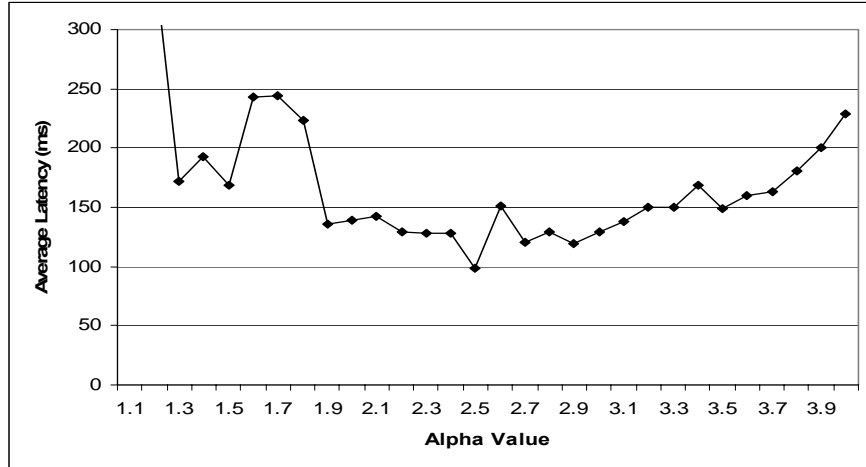
Figure 7 - Effects of various α values on average latency

Because of the way α is used, modifying the value between 2 and 3 does not dramatically effect the amount of bandwidth the push channel receives. In essence, the over provisioning of the pull channel is not under provisioning the push channel in any way. When α is too small, it is not giving the pull channel enough leeway, which is causing the higher response times. When α is too large, it begins to under provision the push channel, causing the overall latency to begin to increase and the latency of items on the push channel begins to increase. Thus, we have found that keeping alpha between 2 and 3 does not negatively effect either the push or pull channels in terms of bandwidth being provided to either channel.

We adopt α = 2.0 in the remainder of this work - although this is not the actual minimum, any value in the range described above produces similarly good results. Note that as α changes in Figure 7 our system adjusts the bandwidth division and document classification to maintain optimality. This also helps to explain in part why the average latency is near optimal for a relatively wide range of α.

4.1.2 Performance of the SELDIV Algorithm

Figure 8 can be interpreted as a brute force search for a good bandwidth split and document classification by trying several closely spaced values of *k* and *pushBW*. In the chart legend, the first number in the bandwidth split refers to the percentage of bandwidth given to the pull channel. In addition to the points plotted in the figure, we verified that if less than half of the
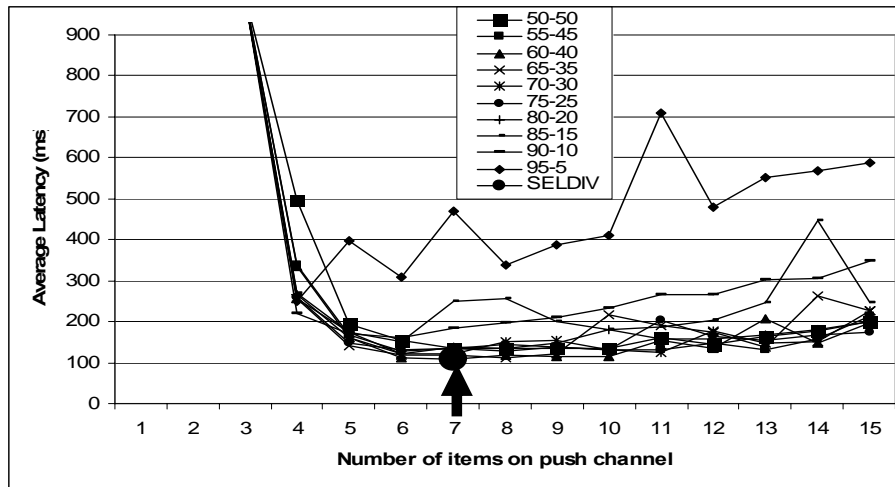
Figure 8 - Demonstrating the optimality of Algorithm 1 for document classification and bandwidth division. The arrow points to the single point found by the algorithm.

bandwidth was devoted to pull, the latency was suboptimal. In this scenario, SELDIV assigns the most popular 7 documents on the push channel, and allocates 63% percent of the bandwidth to push. The figure shows the algorithm's outcome with a circular point and an arrow pointing to it. The solution produced by SELDIV is better than any other point in the diagram. More specifically, SELDIV chose a split of 63/37 and the closest brute force curve in the figure is the 65/35 curve. The 65/35 line was also the lowest in the graph. SELDIV chose $k=7$ point as the number of push documents, which is also the minimum point on the 65/35 curve. Thus, SELDIV chose a better bandwidth split than the brute force approach and a document classification that was just as good.

Let G(k) be the average latency if the k most popular documents are placed on the push channel. The function G(k) is a weighted average of the average latency for pushed documents and the average latency for pulled documents. A graph showing an idealized G(k) from the Air Cache is shown in Figure 9. The function G(k) has a unique local minimum, which can be found by local search[29]. Figure 9 shows that the minimum of G(k) is to the right of the intersection of the push and pull curves. In this case, pulled documents would have lower latency than pushed documents. The actual curve that we obtained from our experiments is shown in Figure 10. Notice that the minimum of G(k) is to the left of the intersection of the push and pull curves, and thus pushed documents have lower latencies than pulled documents. Further, the minimum of G(k) occurs at a relatively small value of k, and thus complicated hierarchical schemes for the push channel may not be useful in this setting.
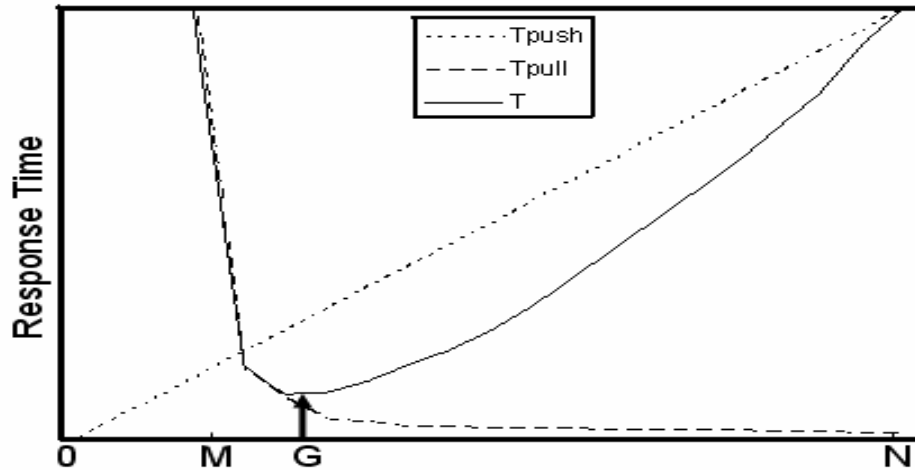
Figure 9 - Relation of Push and Pull latencies as number of items pushed is changed, according to Air Cache[47]
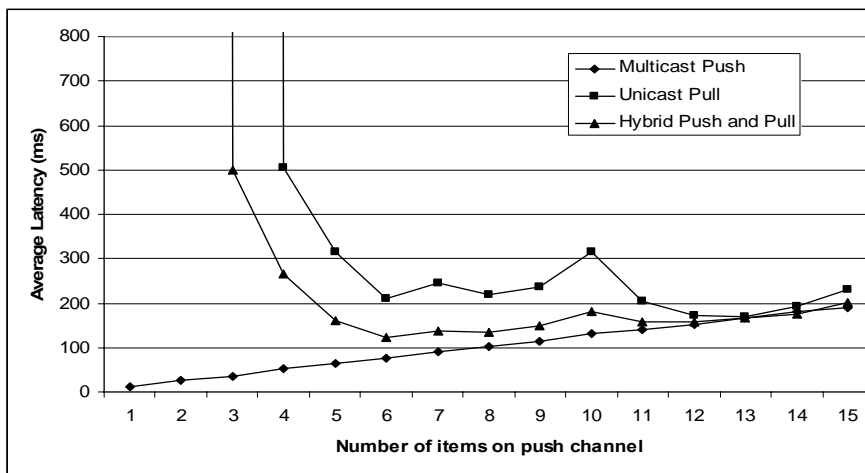

Figure 10 - Relation of Push and Pull latencies as number of items pushed changes according to our experiments

The location of the minimum is due to two complementary reasons. First, the most popular items are chosen for push and are also those to which a Zipf (or Zipf-like) distribution gives substantially more weight. Therefore, if a solution favors multicast push, it will also have the largest impact on the globally average delays. Second, the Unicast pull curve levels off and, from that point on, the exact choice of k has little impact on pull delay. In other words, pull delays are practically minimized at the point $\overline{k}$ where the pull curve flattens out. However, $\overline{k}$ precedes the intersection of the pull curve with the push curve, and so the overall minimum occurs before that intersection.

In conclusion, SELDIV was shown to be better than the best value returned by a brute force search. This shows that an algorithm which divides both documents and bandwidth at the same time provides a division similar to that which could be found doing a brute force search and provides the necessary information needed to generate a fully functioning hybrid data dissemination system. Furthermore, the integrated algorithms led to a behavior of the push and pull curves that differ qualitatively and quantitatively from previously published work, e.g., in terms of the relative behavior of push and pull delays.

4.1.3   To Multicast Pull or Not to Multicast Pull

In this experiment, we wish to compare the performance of using the multicast pull channel within our hybrid system architecture. We remained using the document selection algorithm for performing the document division, so the only difference between the compared architectures was the use of the multicast pull channel. This allows us to isolate the effectiveness of the multicast pull channel within the architecture, as it was a major new component of our improved architecture. The document popularity distribution is static in this experiment, meaning that the document that is the most popular at time t will be the most popular at time *t+1*, or that $\forall i \forall t : p_t(i) = p_{t+1}(i) = p_{t+2}(i)$ at all points of time in the future. The results of this experiment are shown in Figure 11.

Figure 11 shows the average latencies with multicast pull on, and with multicast pull off, for various values of θ. Remember that the higher the θ value, the more popular the popular items are and the more skewed the popularity patterns appear. This means that when the zipf θ is at 1, it is very much like having a mostly random pattern of access to documents. While having a completely random popularity pattern may seem like a good way to examine the effectiveness of a distribution scheme, the access patterns and popularity of documents in a real web server tend to follow a different pattern. There are usually a small set of documents which get the bulk of the requests (i.e., the main index (home) page of a site) and the other pages get requested with decreasing probability. This is reflected with a theta between 1.5 and 2.0.
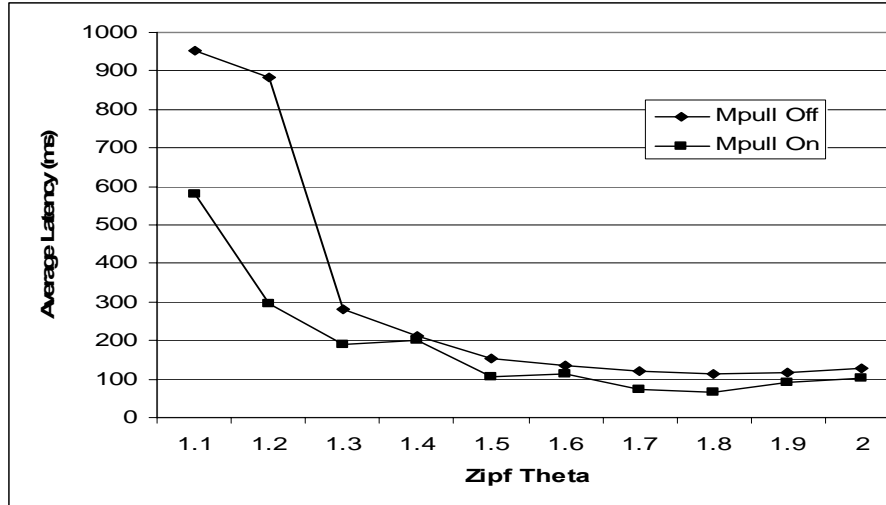
Figure 11 - Average Latency for Multicast Pull On versus Multicast Pull Off and Static Access Patterns

The basic goal of this experiment was to twofold. First, we wanted to check on how the two schemes compare under the most basic and unchanging environment. Our initial thesis was that the two schemes would be very close in performance when the popularities of documents did not change. Second, while we felt the two schemes would be close in performance, we wanted to make sure that having the multicast pull on would not adversely affect the system performance. We expected that using the multicast pull may present a little additional overhead that would cause it to perform a little worse than not having the multicast pull channel on. However, we did expect that the overall system variance would be lower when the multicast pull channel was used. The results we experienced were close to our initial thesis and using the multicast pull channel actually exceeded our initial predictions.

As Figure 11 shows, the two schemes were very close in performance for the higher values of θ, those between 1.4 and 2.0. While we expected the scheme without multicast pull to be faster, our experiment actually shows that using the multicast pull channel not only does not adversely affect the performance, but actually increases it slightly. In particular, we showed a 30% reduction in response time with a $\theta = 1.5$, where the response time decrease from 153.9ms with the multicast pull off to 107.6ms with the multicast pull on. Similar differences were found as the value of θ increased, with the reduction being between 5-35%. More interesting was the difference between the two schemes with θ being closer to one. For example, with a $\theta = 1.2$,

64

using the multicast pull channel provided almost a 66% reduction in average response times, from 879.6ms to 293.6ms.

The reason for this large of a change refers back to the meaning of a lower θ. A lower θ means more requests for a larger set of documents. Given our algorithm, when the popularity is more spread out, there are fewer documents that will find their way onto the push channel, but there will still be a large number of documents receiving multiple requests. By having the multicast pull channel available, the documents which are receiving multiple requests can be serviced with a single server send out, which lowers the stress on the server and allows it to handle requests faster. Thus, by having the multicast pull channel in use, the documents that are not hot enough for the push channel but are popular enough to cause the server to have to deal with a large load of requests are being handled more efficiently, which is lowering the response time for the clients.

Another advantage of using the multicast pull channel that we hypothesized about was that all other things being equal, it would decrease the standard deviation of the observed latencies. The reason is that with multicast pull turned on, fewer requests will have to wait for extreme lengths to get results. Without multicast pull, clients for pull based documents are forced to wait in the order the request is received to get their results. This can cause longer response times for several clients awaiting responses, especially versus those clients that are getting the data quickly off of the multicast push channel. By using the multicast pull channel, if several clients request the same document, they are all serviced at the time of the earliest request. Hence, the average response times for that document are more in line with the quickness of the multicast push channel. Additionally, by servicing less requests in general (as 5 requests for the same document are serviced as a single request) there is a smaller queue in general at the server, which means no client is waiting too long for a single request to be serviced, which will lower the variance in user perceived request response times.
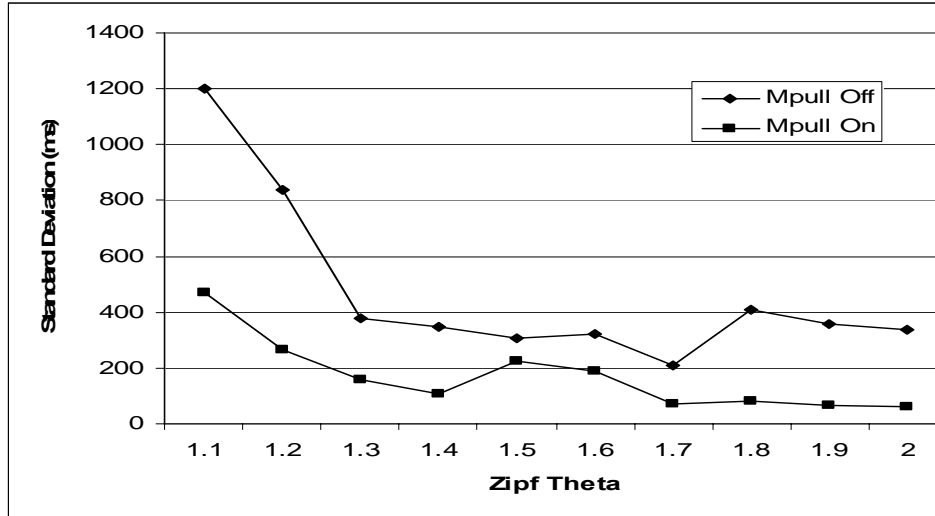
Figure 12 - Standard deviations in latencies for multicast pull on versus multicast pull off for static access patterns

Figure 12 shows the results of the experiments on the standard deviation variance between having the multicast pull channel on and having the multicast pull channel off. With a θ = 1.5, the standard deviation changed from 308ms to 226ms, while as mentioned above the latency also decreased. Likewise, when the θ = 1.9, we see the standard deviation change from 380ms to under 100ms, but the average latencies were very similar. This means that not only does the multicast pull channel lower latency, it also significantly lowers the standard deviation of response times making the response times experienced by clients more reliable. Thus, our original thesis that using the multicast pull channel would not adversely affect the performance by too much and would definitely lower the variance in response times is found to be true.

4.1.3.1 Difference in latency of channel types

One aspect we wanted to examine besides the average latency for overall system was to examine the effects of the different hybrid system setups on the actual different channel types (push versus pull). Figure 13 shows the same setup and experiment as was shown in Figure 11, but this time the latency of the different channel types (push versus pull) have been separated instead of combined into a single overall latency. There are three main observations to make from this figure. The first is that the response times for the push channels are the same in both systems. This intuitively makes sense for the following reason. The number of items on the push channel, and which items are on the push channel, is the same in both hybrid systems.
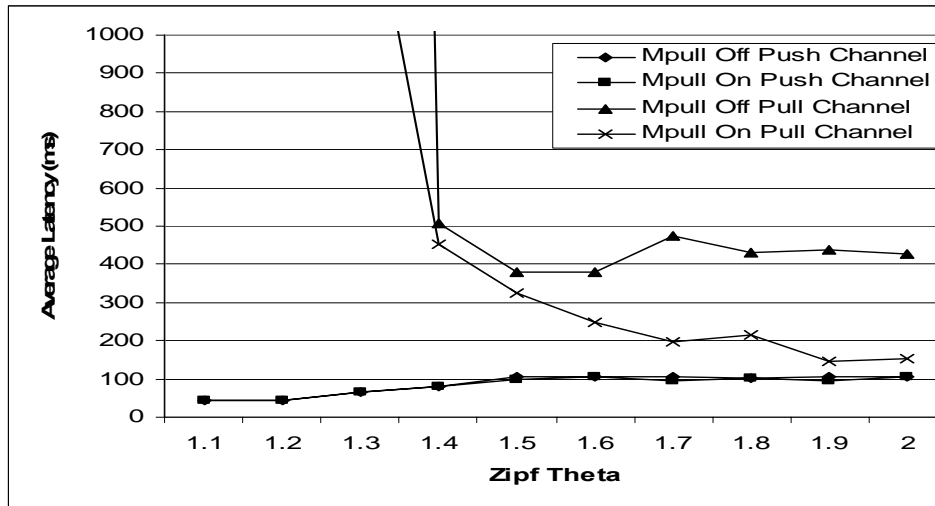
66

Figure 13 - Difference in latency of push and pull channels with multicast pull on versus multicast pull off

Therefore, the latency of that channel will be basically the same, because the same requests are being made for items on that same channel.

The second observation to make is that the difference in response times for items on the pull channels (that is, the unicast and multicast pull channels) is much larger than was shown in the overall latency graphs. Take the case of $\theta = 1.7$. In the overall latency graph, the difference between having multicast pull on or multicast pull off was only 15% but with latency looking at only the pull channel, the difference was almost 60%. The reason for this large difference is that the overall response time is dominated by response times for items on the push channel, and because of that the overall average latencies were driven low by the push average. With the response times for push items removed from the averages, the savings for pulled items is much higher and the usefulness of multicast pull it more apparent.

The final observation to make is that the savings for multicast pull on versus multicast pull off gets larger as the value of theta increases. The reason for this is that as theta gets larger, the number of items on the push channel is getting smaller, and the popularity of those items is getting larger. However, the popularity of the last few items left off the hot channel is also getting larger. Therefore, there will be more requests coming for items that are cold (those left off the smaller hot channel) and the server will have a larger load of overall requests but the similarity of those requests is larger. By having the multicast pull channel active, this load of

67

similar requests is more easily and efficiently handled, leading to lower average latency for items that are pulled.

4.1.4 Multicast Pull with Moving Hot Spot

In this next experimental set, we have a similar set up to the previous experiment with one major change; the popularity of the documents will no longer remain static. Instead, the popularities of the documents will change as time progresses, with the experiment attempting to capture the effectiveness of having the multicast pull channel on to help the server adapt to the changing popularities.

Our goal in this experiment set is to prove that having the multicast pull channel provides a major benefit when the access patterns of documents is changing over time. Our thesis is that when the document popularities shift, the server will have on the push channel documents that no longer belong there, and likewise have documents that are receiving a large number of quests still be serviced through pull based means. Additionally, document classification is a relatively expensive operation, requiring time linear in the number of documents, and the server can not afford to always be invoking the document classification algorithm. Therefore, using the multicast pull channel will provide a way for the server to service documents that should be pushed based on a semi-push based distribution channel, which will lower the strain on the server and should provide both better average latency for clients and a much lower variance in experienced response times.

Figure 14 and Figure 15 show the resulting average latencies resulting from gradual changes in popularity while varying one of θ or α and keeping the other parameter fixed to its default value. The most interesting feature in Figure 14 is that multicast pull is more helpful as θ increases. More precisely, the relative improvement one achieves in average latency when using multicast pull increases as θ increases. For example, when $\theta = 1.5$, multicast pull shows an improvement in average latency of 44.6% (from 217.9ms to 120.9ms), at $\theta = 1.7$ the improvement in average latency is 47.1% (from 228.9 ms to 121.1ms), and at $\theta = 2.0$ the improvement in average latency is 50.3 % (from 243.5ms to 121.2ms). The explanation is that for large θs, most of the probability is in the most popular items, so if a pull document should
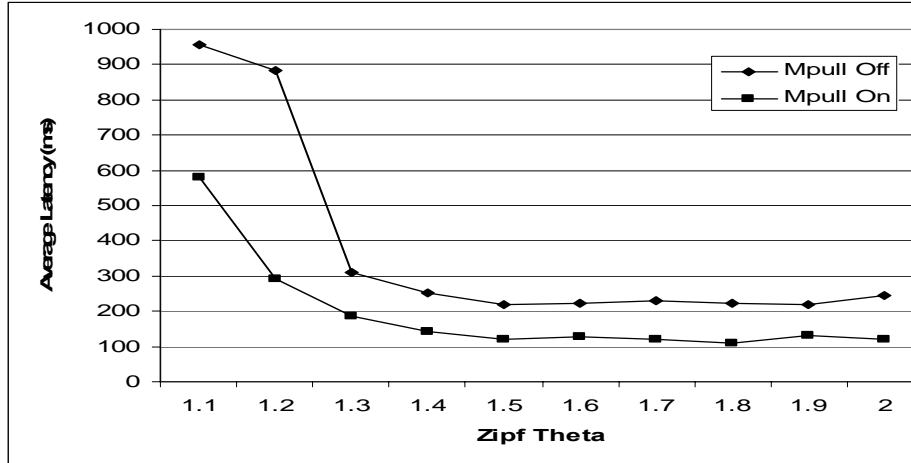
Figure 14 - Average latency for multicast pull on vs. multicast pull off for small move access patterns
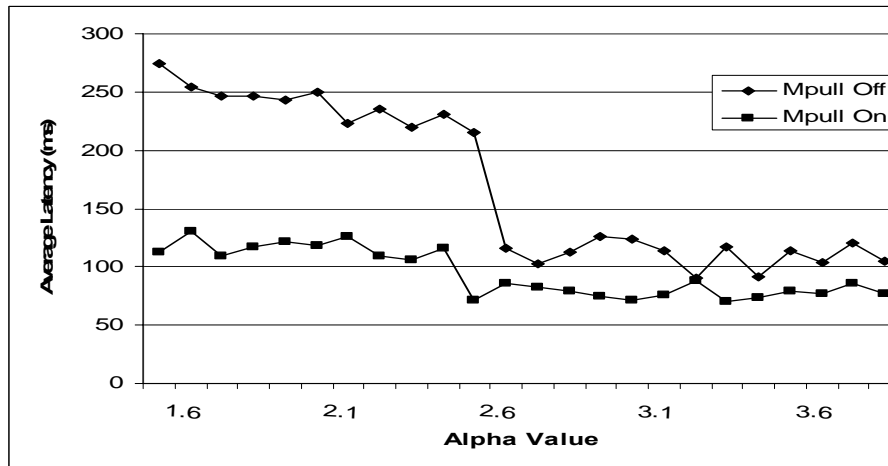

Figure 15 - Average latency multicast pull on vs. multicast pull off various α and small move access patterns

become more popular, it will receive many requests before the server next invokes the document classification algorithm.

The most interesting feature of Figure 15 is that the best choice of α for small moves is larger than it is for a static distribution. Recall that the optimal choice of α for a static distribution was in the range 2.0 to 2.5. In this experiment the optimal choice for α is in the range from 3.0 to 4.0. In this case, setting α = 2 has an average latency of 121.2ms while with α = 3.5 the average latency is 74ms, a decrease of 39%. The explanation is that as there is a shift in popularity, the popularity of the pulled documents will be greater than estimated, and thus obviously, pull should be even more over provisioned.
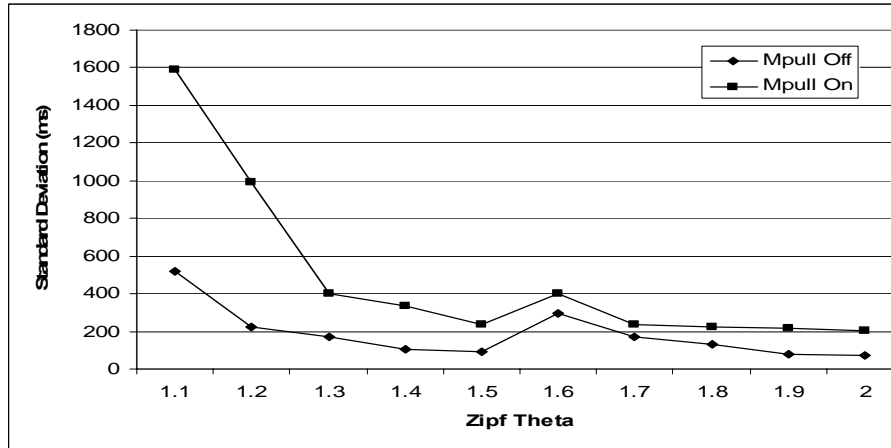
69

Figure 16 - Standard deviation on latency for multicast pull on vs. multicast pull off for small move access patterns

Figure 16 shows that once again multicast pull reduces the standard deviation of the observed latencies. For θ = 1.5, the standard deviation of the latencies decreases 60% from 235ms to 93ms. Note that the reduction in the standard deviation is greater than in the case of static access probabilities. The reason for this is because multicast pull provides some scalability when the pulled documents become popular.

Figures 17 and 18 show the results of a similar experiment to above but in this case we are looking at big moves in the popularity of an item. In this case, we again see that using multicast pull is a significant win for larger θ. As one would expect, for both methods, the latencies are higher than in the slowly moving hot spot experiment. Using multicast pull we see a reduction in average latencies. For θ = 1.5 the reduction is 45% from 351.8ms to 193.2ms, for θ = 1.7 the reduction is 40% from 263.4ms to 159ms, and for θ = 2 the reduction is 61% from 364.7ms to 141.8ms. We also see again that the over provisioning factor should be greater than for static documents.

Figure 19 shows that for big moves, multicast pull reduced the standard deviation of the latencies even more dramatically for small moves. For θ = 1.5 the standard deviation has decreased 63% from 562ms to 205ms, and for θ = 2 the decrease was 81% from 327ms to 63ms. The reason that multicast pull reduces the standard deviation more for big moves than for small moves is that the scalability that multicast pull provides becomes more important as the pulled documents become more popular.
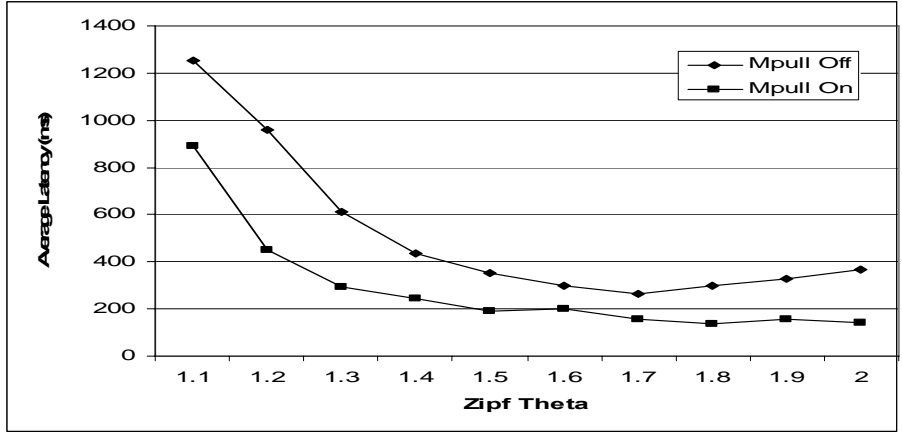
70

Figure 17- Average latencies for multicast pull on vs. multicast pull off for big move access patterns
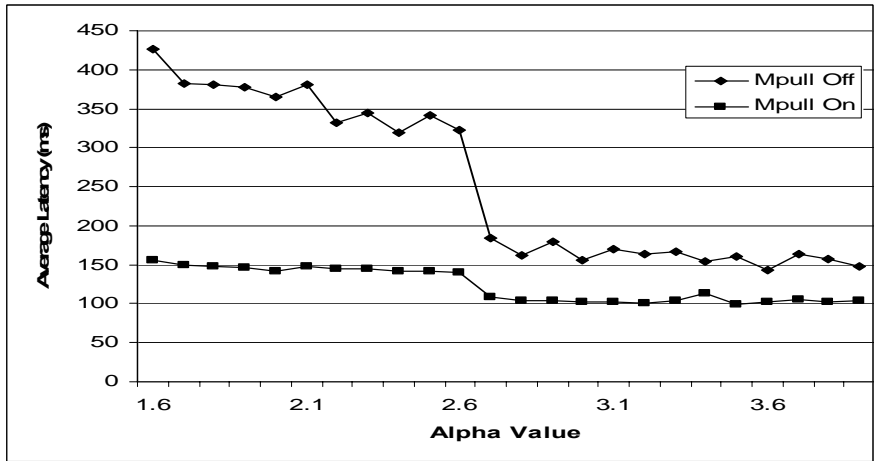

Figure 18 - Average latencies for multicast pull on vs. multicast pull off various α and big move access patterns
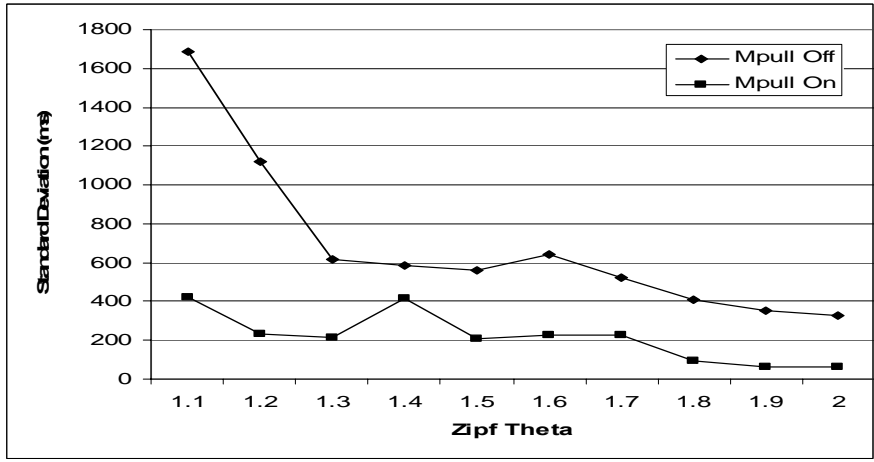

Figure 19 - Standard deviations for multicast pull on vs. multicast pull off for big move access patterns

71
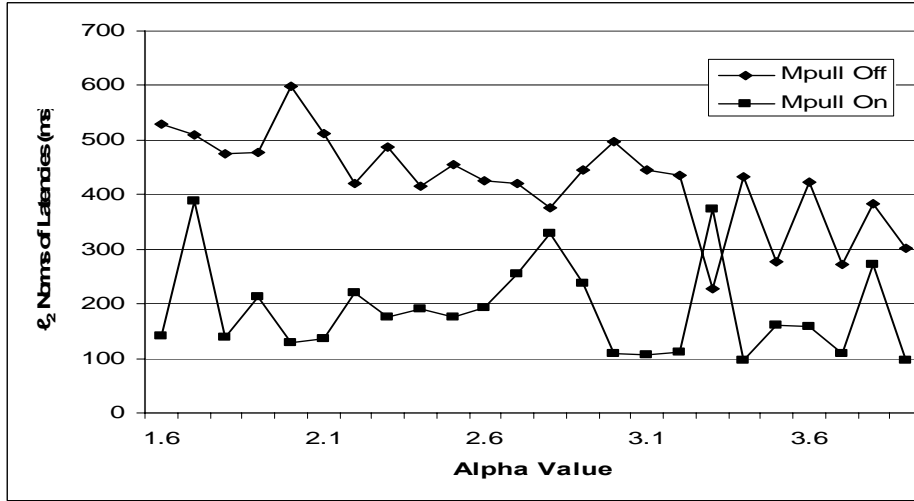
Figure 20 - ℓ2 norms of latencies for various α and small move access patterns
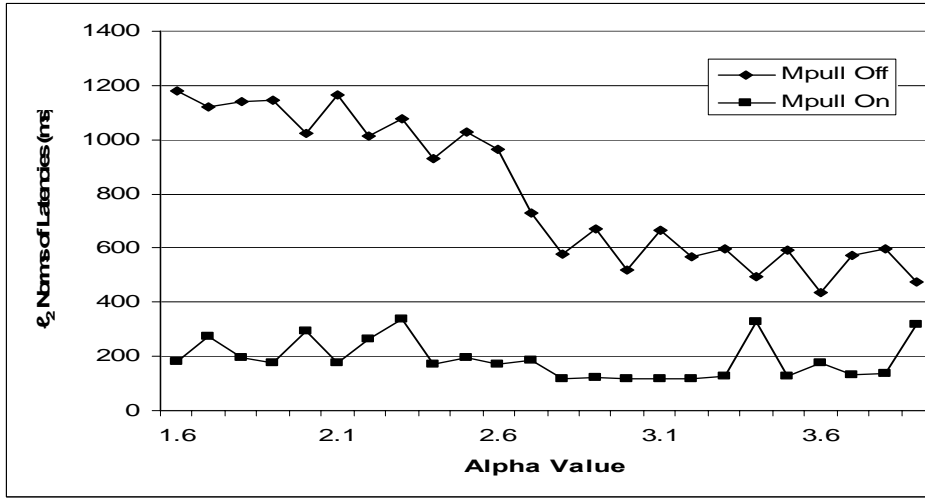
Figure 21 - ℓ2 norms of latencies for various αs and big move access patterns

Average latency is by far the most commonly used quality of service (QoS) metric in the literature. The metric is simple and intuitively appealing. However, it is also well known that average latency is generally not the ideal system metric in that the solution that optimizes average latency may starve some jobs. Allowing jobs to starve is considered bad system behavior. Ideally one would want a metric that balances the competing demands of optimizing for the average and avoiding starvation. The standard solution is to use the ℓ_p norm for small p.

The ℓ_p norm is $\left(\sum_{i=1}^{n} \frac{x_i^p}{n} \right)^{\frac{1}{p}}$. For example, the standard way to fit a line to collection of points is

72

to pick the line with minimum least squares, equivalently ℓ_2, distance to the points, and Knuth's TeX typesetting system uses the ℓ_3 metric to determine line breaks. The ℓ_p, $1 < p < \infty$, metric still considers the average in the sense that it takes into account all values, but because x^p is strictly a convex function of x, the ℓ_p norm more severely penalizes outliers than the standard ℓ_1 norm.

Figures 20 and 21 show what the effect of varying α has on the difference between having multicast pull on and off for the ℓ_2 norms latencies. We set $\theta = 2.0$. For $\alpha = 2$ and small moves, the ℓ_2 norm of the latencies decreases by 78% from 475.2ms with multicast pull off to 138.7ms with multicast pull on. At $\alpha = 3.5$ and small moves, the decrease is 42% from 277.1 ms with multicast pull off to 160.6ms with multicast pull on. We found similar results for big moves. These results are shown in Figure 20. For $\alpha = 2$, we see a reduction in the ℓ_2 of latencies of 71.5% from 1025.3ms to 292.8ms. For $\alpha = 3.5$ we see a reduction of 78% from 591.1ms to 128.64ms. Once again this shows the scalability provided by multicast pull. The server become highly loaded during these popularity shifts and causes an increased latency to be experienced by the system without multicast pull. The reason for this is that the system without multicast pull has no method in place to handle the increased load until the document selection thread is invoked.

4.1.5   Multicast Pull Advantage with varying Time between Reconfiguration

In this experiment, we examine the advantage of using multicast pull when the time between invocations of document selection changes. For this experiment, we set the θ to 1.5. Figures 22 and 23 show the results of the experiment. As one would expect, using multicast pull is more advantageous when reconfigurations are less frequent. The obvious reason is that it is taking the system longer to adjust to the changes in user preferences, and therefore there are many requests coming in that have to be handled through pull.

Using multicast pull, we observe a reduction in latency for small moves of 46% when the reconfiguration is every 10 seconds, 45% when the reconfiguration is every 20 seconds, and 55.6% when the reconfiguration is every 60 seconds. For big moves, we observe a reduction in latency of 46.4% when the reconfiguration is every 10 seconds, 50% when the reconfiguration is every 20 seconds, and 53.4% when the reconfiguration is every 60 seconds.
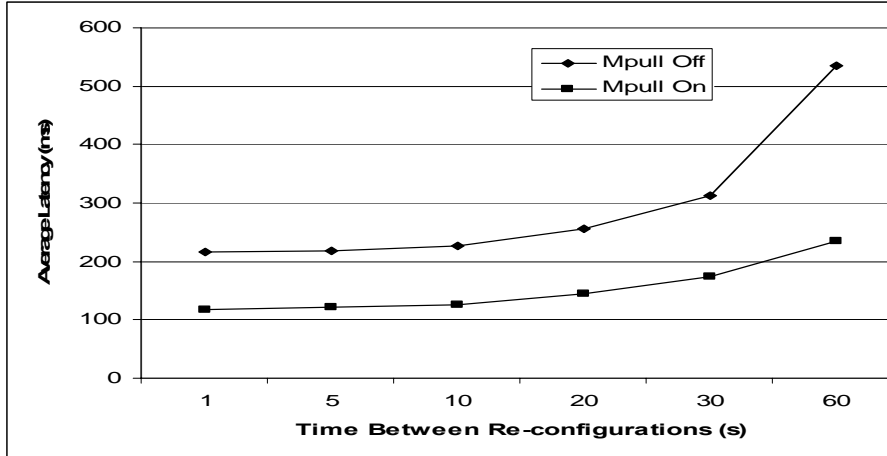
Figure 22 - Multicast pull on vs. multicast pull off for varying reconfiguration times in seconds for small move access patterns
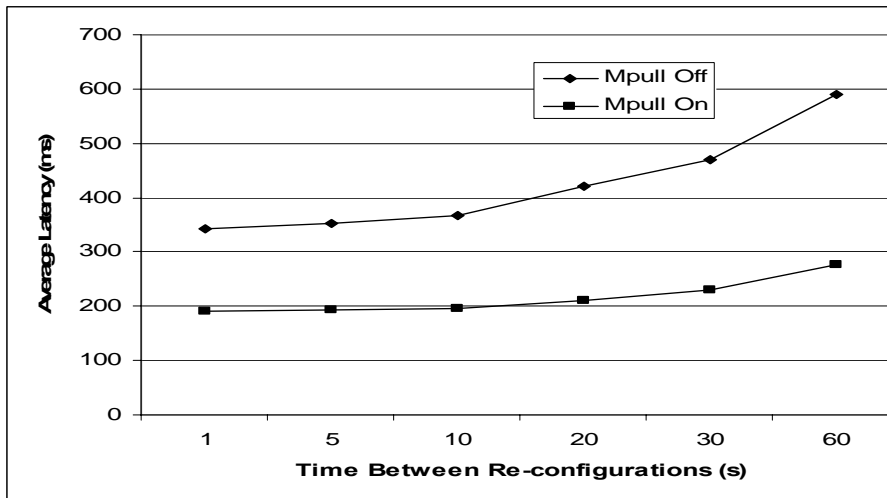
Figure 23 - Multicast pull on vs. multicast pull off for varying reconfiguration times in seconds for big move access patterns

Notice the trade off that exists between waiting too long to run the reconfiguration and the average latencies. The longer that is waited to reconfigure the system, the worse the response times for clients get. However, using multicast pull can help maintain lower latencies when the system is slow in adapting to changes in the request patterns.

4.1.6   Report Probabilities

In Chapter 2.2.1, we mentioned the scheme we used to calculate the popularity of documents on the multicast push channel. Recall that the main issue with the multicast push channel is that the documents are not directly requested, so determining whether the items on that channel are still popular is quite a difficult task. Recall further that we proposed a solution to this problem that involved having the client send a request to the server with a probability that was inversely related to the previous popularity of an item. For example, if an item had 50 requests previously, we would only like to have between 1 and 5 requests appeared for that item, which would then be multiplied by the inverse factor and count as 50 requests each.

In order to determine the usefulness of our proposed push popularity scheme, we compare it to a solution we mentioned previously as the drop down method, which is found in a comparable work to our own. The solution for the push popularity problem proposed in prior work[29] was to occasionally drop each pushed document *i* off of the push channel so that clients would have to make explicit requests to *i*. However, there is a danger that these explicit requests for i could overload the server. Thus, in the prior work[46] it was recommended that i should be dropped as short of a period of time as possible.

The shortest possible time that the document can be dropped is one broadcast cycle. However, we show here that even such a short drop disrupts the server, while our proposed method does not suffer from such disruptions. Therefore, our thesis is that while both will provide a comparable accuracy for calculating the popularity, the drop down method will cause more overhead that our report probability scheme. Our belief is that when a document is dropped off the multicast push channel, there is a period of high request rates that will occur, given that the access patterns have not changed. Further, while the system will correct the items on the push channel, it will take several broadcast cycles to recover, while using our access probability method will not have such spikes and response time degradation.

Figure 24 shows the average latencies around the broadcast cycle T when the most popular item is dropped from the push channel. The figure shows performance degradation for about 5 broadcast cycles. Basically, looking at the graph shows that before the drop occurs, the system is in a steady state of response times. However, once the item is dropped down the clients are no longer getting requests off the push channel. Instead, they must make requests

75
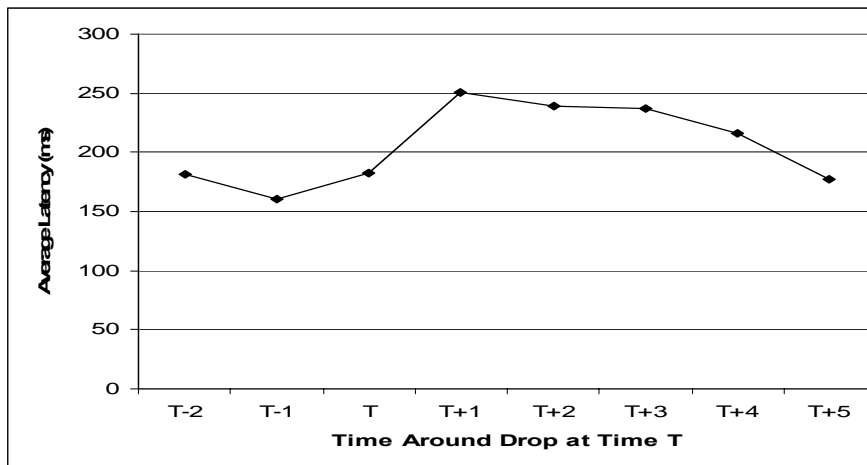
Figure 24 – Effect on latency of demoting an item
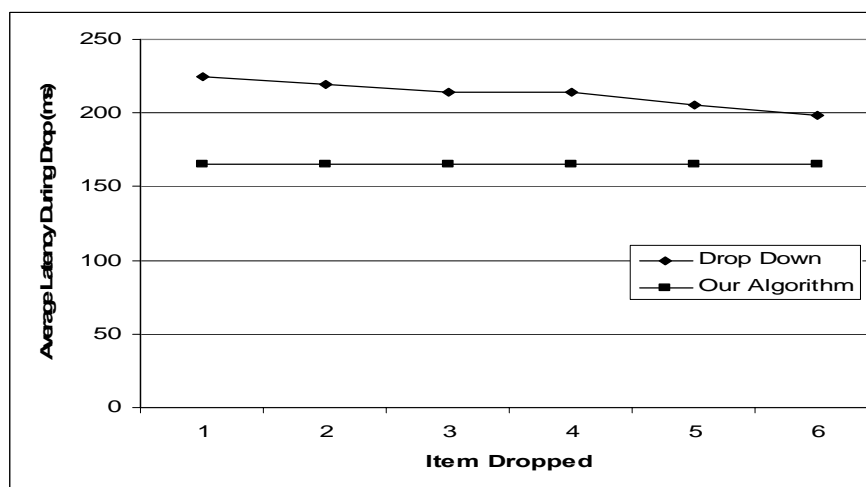
Figure 25 – Drop down method versus our probability method

directly to the server. Based on the Zipf distribution, as mentioned earlier, the bulk of requests were for items that were on the push channel. Therefore, dropping an item off the push channel causes a brief but substantial influx of requests to the server. This brief surge causes response times for requests during the given broadcast cycle and a few subsequent cycles to suffer while the server recovers and returns to its steady state.

Figure 25 shows the average latency over the next 5 broadcast cycles when the ith most popular document is dropped from the push channel for one broadcast cycle. The flat line represents the average response time using our method for push popularity. If the most popular document is dropped, then we see a 35% increase in average latency over the next 5 broadcast cycles. If the 6th most popular document is dropped, we see an 8% increase in average latency

over the next 5 broadcast cycles. This increase is in comparison to using the simple yet effective scheme we proposed of simply including a popularity estimator with the broadcast index.

4.2   EXPERIMENTS IN REAL WORLD ENVIRONMENT

In this section of experiments, we focus on operating in a real world environment and looking at the scalability of the system versus other distribution methods. The previous experiments we presented were done in an isolated environment and done in order to test several of the particular enhancements we created for the hybrid data dissemination architecture. The issue with those experiments, however, was that it can never be completely determined if given a real environment, the findings we have discovered will hold up. Additionally, none of the previous experiments clearly focused on the scalability of the system, which we will look at in this section.

The evaluation metric that we used in this new set of experiments is the amount of time (or the latency) that the clients experience between the time a request is made and the time that the full document result is generated and returned to the client user program. We measure this time in milliseconds. Because the system is in a real network environment, these times measurements will include network congestion and distance that data has to travel, as many of the clients are located across the globe. Unlike the previous experimental set, we do not need to see these as relative measurements of time. Instead, the measurements can be taken as the actual time it took to make requests and the resulting times can be seen as actual savings when comparing the different schemes.

4.2.1   Environment and Experimental Setup

4.2.1.1   Planet Lab

The environment we chose to use for our real world experiments was the Planet Lab network environment[44]. Planet Lab is a collection of "nodes" found throughout the world at various

universities and organizations. Each location donates several hardware servers that can be used to help create a real internal network. Each user is given a slice, which means that they can have memory on many of the servers, allowing users to run clients or servers as needed. This system of nodes operates over the Internet and therefore experiences all the traffic, congestion and distance delays found there within.

Operating in the Planet Lab environment has several pieces of variability that should be mentioned and are included in the upcoming experiment results. Because Planet Lab is accessed and shared by many universities, the operation of programs within the Planet Lab environment is also shared. This means the machine we use will not be 100% dedicated to operating our server, but could fluctuate and at certain times have more processor available than others. The same could be said for the bandwidth available; however, we introduce a filter on our server to only allow 100KB per second, to account for the variability that could occur. We believe that by running such a large number of experiments, with such a large number of users and for the amount of time the server is operating, much of this variability can be factored out.

In our use of the Planet Lab network, we operated under the following setup. We deployed the server on one of the planet lab nodes located at the University of Pittsburgh, on the East Coast of the United States. Both the server and any multicast channel servers were deployed in this single node. The reason we placed the multicast channel servers on the same box as the actual web server is that we envisioned this system for cheap and easy deployment. We do not expect the actual users to have access to multiple servers. Thus, we wanted to emulate what an actual user of our software would have available, which in this case was a single server box. Each client was run on a separate physical machine and was located in many different areas. These range from the West coast of the United States at the University of Berkeley, to nodes at the University of Tokyo in Japan, Uppsala University at Sweden, and University of Ioannina in Greece just to name a few examples. The point is that the nodes are very widely spread out, and since they operate over the internet, provide a nice set of locations to test as clients and provide a good estimate of how the system operates in a real world environment.

4.2.1.2 Experimental Servers

There are multiple server setups that we used in this set of real world experiments. The first server setup is a purely pushed based server setup. This server reflects an environment where no requests are made for documents and all documents must be sent out to ensure that each client receives all requested information. We expect that this will be a worse case response time allowable by a reasonable system. The reason is that this server should have unlimited scalability, since regardless of the number of clients the server never receives any requests, and thus never gets overloaded. In addition, the average latency should remain relatively flat as the number of clients is increased, since again adding a client does nothing to affect the server. We refer to this server as the *pure-push* server.

The second server setup is a purely pull based server setup. This server uses only unicast to communicate with the clients throughout the network. This most closely reflects the operation of a normal server that exists on the internet today, where TCP connections are made with the server for each request and the server responds to each request in turn. This server receives a request, queues the request until it is selected to be serviced, and retrieves the data and returns the results to the user. Since this most closely emulates the common internet, we expect the response times of this server to be low until a certain threshold of the number of clients is met. Once that threshold is crossed, we expect the response times to increase dramatically as each new client is added. Hence, we see this as providing a baseline for the minimal number of concurrent users a server should be able to handle, and thus is a baseline for our scalability factor. We refer to this server as the *pure-pull* server.

The third server setup that we employed was our fully functioning hybrid system architecture as described in the real world implementation section of Chapter 2.3. This includes both a server side middleware which sits in front of the server and the client side proxies on each of the clients. The multicast layer used is Hypercast, and as was mentioned above the multicast servers which Hypercast requires were located on the same physical server and the server side middleware. Both the document division algorithm and the request probability feedback mechanism we have developed are deployed within this middleware. As this setup is a direct implementation of our middleware, we expect that it will behave very similarly to our simulated experiments. We refer to this server setup as the *middleware* server.

Our thesis is that the middleware will have response times close to pure-pull when there are a low number of clients and lower than the pure-push when the number of clients is extremely high and well beyond the load that pure-pull can handle. Further, we believe that with the way our algorithm is designed, the middleware server will actually dynamically adapt itself to properly adjust to the correct flow of requests. For example, when the request rate is low, we do not expect to see much difference beyond network noise between the behavior of the *pure-pull* server and the *middleware* server. This is because only the most popular items will be placed on the push channel, and that will keep the latency low, and many requests will experience the same response time as those items on the unicast or pull channels.

Similar to how the system will operate when the client request load (or the number of clients) is low, we expect that when the request load becomes very high, the *middleware* server will in the worst case operate as the *pure-push* does, though we expect the request times to be lower. The reason we expect lower request times are that while the bulk of the load may be forced to be placed on the multicast push channel, the remaining items will still be served over unicast, even if that total number of items is low. Additionally, the clients will receive, on average, the requests on the multicast push channel much quicker, because there are not as many items cluttering up the channel, as would be the case with *pure-push*. Thus, we expect this system to have the scalability factor similar to *pure-push* while getting the performance in request times similar to *pure-pull*.

Another server we tested as part of our real world experiments is a duplicate of the *middleware* server but with one major change, there is no multicast pull channel included. We expect the performance of this server to behave very similarly to the *middleware* server, except when the popularity patterns switch or change. In those cases, we expect the *middleware* server to be better. During normal operation, we expect this server set-up to be almost as good as the *middleware* server, though we expect as the number of clients is increased, the *middleware* server may perform slightly better due to having the multicast pull channel available. We refer to this server setup as the *no-mpull* server, standing for no multicast pull.

The final server we employ in this experiment is similar to the *no-mpull* server but again with one major change, we no longer use our document selection algorithm. Instead, we use a threshold for determining whether the document should be placed on the multicast push channel or on the pull channels. Basically, when it comes time to run the document selection, the number

of requests per item is checked. If the number of requests exceeds a prior defined threshold, the item is place on the multicast push channel; otherwise it is serviced over the unicast channel. We chose a threshold of 7 for the multicast push channel, as it seemed reasonable to use that any document having more than 7 requests is popular enough for the push channel. Also, based on checking probability patterns from various θs, 7 requests seem to place the documents into the upper tier of requested items. We refer to this server as the *hybrid* server.

Using the *hybrid* server, we expect it to perform much differently than the *middleware* and *no-mpull* servers. We expect that because it does not have an optimal split of popular and unpopular items, it will at best get lucky and perform as well as no-pull, but most likely be less than optimal one way or the other, which will affect the latency experienced in the system. Further, at some point the threshold may be too high to catch all the items that should be placed on the multicast push channel based on the incoming request rate and number of clients. In this case, the *hybrid* server will start to go into an overload state, which will make its response times become very unbearable. Unlike our system, which we feel will make the best of both *pure-push* and *pure-pull*, we expect *hybrid* to have a better scalability factor than *pure-pull*, both not any other system. Further, we expect the response time of *hybrid* to only be better than *pure-push*, but not to beat any of the *hybrid* systems using our algorithm, as the split is arbitrary and not optimal.

4.2.1.3   Experimental Clients

All clients were the same in each experiment. Each client consisted of a fully functioning version of the middleware client, without an actual web browser as the front end. Instead, where the socks proxy (as defined in Chapter 2.3) would be to capture and forward the client requests to the client proxy, there is an auto request generator which generates the requests for the clients. This application makes a request to the client proxy as the socks proxy would, which then relays the requests to the server side middleware proxy. The results are then returned to the client proxy, which compiles them in a complete binary form of the requested documents and passes that binary data collection to the request generator. The request generator will then write this data out to file, just as the socks proxy would pass it to the client front end which would then display it to the clients. Response times are then calculated from the time that the request generator creates the request until the time that the full file data has been written out to file.

Thus, it is the actual request time a client would experience when using the server setup for that experiment.

4.2.2   Experimental Parameters

The experimental parameters for the set of real world experiments can be found in Table 4. Most of these parameters are similar to the ones we used in the previous set of experiments, so we now describe those parameters which are different. The first difference is that the document sizes are no longer static. In this experiment, we used a wide range of document sizes, from .25KB to 2.5MB. The relation between popularity and document size is not determined, so we assigned the document sizes randomly to all documents, and allowed a random request generator to generate the set of requests to make, following a Zipf distribution with the Zipf θ set at 1.5, which has been found to closely mirror the actual distribution of requests on the internet[17]. This means that our algorithm and response times can in no way be associated with the request sizes, making it completely random which documents are being requested. The only pattern is that all clients follow a similar popularity module, so those documents which are popular on client A are also popular at client B, at all times during the experiment.

The multicast pull threshold was set to 3 for the middleware server. We felt that if a request received multiple requests (greater than 2) then it should be replied to over the multicast pull channel, to alleviate pressure immediately on the server. Due to the vast size and distribution of clients within the network, using the multicast pull should usually only be saved for extremely necessary conditions, since it is not as fast as unicast. However, since we set to threshold to 3, that ensured that at least 3 requests had to be pending for the item, so that the extra time the first requester has to wait for the item to be received over multicast is offset by the savings the second and third requesters save from getting serviced immediately. This makes sure that it should not adversely affect our server performance in any way.

Other relevant parameters are the reconfiguration period, which we set to 5 seconds to match a good value which was found in Figure 22. Because the requests take much longer to complete in the real environment, we set the total requests made per client to 5000 each. The hot spot is either static or moving drastically, as small and large moves were found to be similar in the simulation. The number of clients was varied from 1 to 160. Although Planet

Parameter	Value	Default
Document Size	0.25 - 2MB	Random
Zipf Parameter	1.5	1.5
Multicast Pull Threshold	3	3
System Bandwidth	100 KB/Second	100 KB/Second
Number of clients	1 – 160	1 – 160
Re-configuration Period	5 seconds	5 seconds
Total Available Items For Request	100 documents	100 documents
Total Requests Made (per client)	5000 requests	5000 requests
Hot Spot Movement Type	Off, Big	Varying
Alpha Parameter	2.0	2.0

Table 4 - Real World Experimental Parameters

Lab contains over 260 nodes, not all the nodes are available all the time and nodes tend to shot down over time. Additionally, nodes were difficult to access at times. We would wait to get 160 clients ready and started to begin these sets of experiments. The failure rates of nodes in the experiments stayed less that 20 nodes during all runs, which helps to further the realism of the experiments, since clients ceasing to send requests was factored in.

The system bandwidth set at 100Kbytes/second refers to the way that we limited the output of the server. Each component of the server, in particular the push and pull channels, are constantly trying to send out data. We decided to have each channel create an output queue within itself, and that channel is given a time slot in which it is permitted to send out all the data it has up to its limit, as determined by the method being used. By limit, we refer to how the scheme distributes its bandwidth between each channel, if multiple channels exist. For the pure-push and pure-pull schemes, 100% of the bandwidth is given to their respective channel types. This means during each second, the server can send out up to 100Kbytes of data, and then must sleep until the remainder of the second is up. This ensures a maximum bandwidth, to make sure all servers have an equal amount of available bandwidth.

For the multiple channel servers, we used our document selection algorithm (which was given as input 100Kbytes as the available bandwidth) to determine the amount of bandwidth

each channel would get. During each send round (once per second) the channels queue size was checked, and then the data permitted to be sent out and the sleep times were calculated. This ensured that no more that 100Kbytes were sent out at any given time period, matching the bandwidth available to the pure servers. For the hybrid server, the one not using our algorithm, we set the bandwidth division at 50%, with the remainder of the calculations running the same as the other multiple channel systems.

The cache size in these experiments was unlimited at the server (meaning all items were kept in memory at the server and there was no delay for the server to fetch the item from the backend). We did this to avoid any additional network issues with having to hit another server, and because it was just easier to create the server side proxy as part of the server. This would be similar to having a server which could prefetch all available documents and store them in virtual memory for quick servicing. With only 100 possible documents to request, this is a possible scenario. However, having all documents in memory actually aids the unicast side of the system, which means the unicast server setup is not hindered by fetch times. This is also true of any of the multiple channels systems pull based channels, which evens out the performance results.

On the client side, we turned the client cache off. We turned the cache off for these experiments because in the previous experiment set, the cache was also off, and we wanted an even comparison. While the cache is available, and we described how it can be used, we did not use it during these experiments. This hurts, in particular, all multicast based data dissemination, as multicast is not as fast as unicast. This also ensures that the effectiveness of our scheme (using multicast pull) is not based on getting lucky with items being in the cache (in fact, this would only have enhanced the effectiveness of our system). A final reason for turning the cache off is we did not want our experiments skewed or seemingly based on the cache scheme. We want to avoid any argument saying that with a different caching scheme, things would be different. We wanted to focus on the different server setups, and not on the caching scheme at the client.

4.2.3   Real world experimentation with static access patterns

In this first experiment, we varied the number of clients and ran the experiment for each server type.   The number of clients was varied from 1 to 160 clients, with the number of clients being increased in increments of 10 (i.e. 1, 10, 20…).   The location and identification of the clients within planet lab staying the same for each server setup, so the first client in the first server setup was the first client in the second server setup, and so on.   All of the clients were started at approximately the same time.   The first request that each client made included in it the start up time to contact the middleware server, get the initial startup information, and process that information.   The first client request also included the time to join into the multicast channel. This ensures that the average response times include the setup and connection time, which is some overhead that a hybrid data dissemination scheme faces.

The popularity of documents remains static during this set of experiments, as was done in the previous experiment for static access patterns.   All the servers were included in this set of experiments, and the document sizes were randomly generated, and based on this random generation several of the most popular documents (the documents which on average should receive the most requests) were over 1MB in size, so that it can not be said only small documents were used to skew the results.   We limited the bandwidth using the technique mentioned above, where there was a max number of bytes a given channel could output per second.   As a reminder the cache was also turned off during this experiment.

Figure 26 shows the average response times for each of the server setups, as the number of clients was increased.   For all the methods, except the pure-push method, the response times increased as the number of clients increased.   This was to be expected, and comes as no surprise in examining the results.   The pure-push method receives no requests on the server end, and thus is not affected as the number of clients increase.   While the number of clients does not affect the response times, the results for the pure-push are still much higher that all methods (until the unicast reaches overload that is).   The reason for this high response time is that the clients must wait, on average, for at least half the length of the broadcast cycle to get their requested item. Because each broadcast cycle contains every document this wait time can be very long.   Also, considering there is no feedback, there is no way to set up the documents in any order, so the broadcast is flat.   This means even the most popular documents will require a long wait.
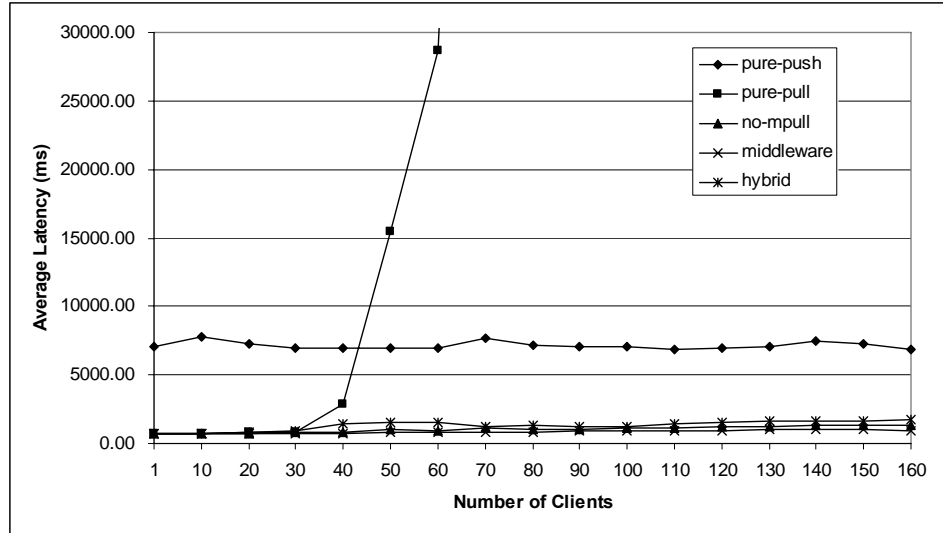
Figure 26 - Real world experiments for various servers and static document popularities

On the opposite end of the scale is the pure-pull channel. Notice that the pure-pull channel, as we hypothesized, performed extremely well until it approached an overload level, at which point the response times went beyond our allowable average of 30 seconds (note that 13 seconds was found to be the length of time a user will wait for a web site to respond before giving up, so 30 seconds is actually more than enough for a cutoff). The actual maximum lied somewhere between 50 and 60 seconds. Beyond 60 seconds the experiments were taking such a long time they were stopped, as the queue at the server was being forced to drop requests and that is not an allowable situation in our experiments. When that happens, we consider the server completely overloaded and to have unlimited response time, as is evident by the pure-pull curve going off the chart.

One final observation to make about Figure 26 is the performance of all the multiple channel schemes. All three schemes work extremely well when compared to the pure methods. When the number of responses is low, all three channels behave in a similar manner to the unicast channel. This is not surprising, and in fact is expected based on the fact that all three schemes are using only unicast for data dissemination at the beginning. The middleware setup does use the multicast pull channel from time to time, but for the most part all requests are serviced over unicast. Thus, all three schemes behave similarly to pure-pull, but when pure-pull reaches its limit the multiple channel schemes do not suffer from the overload situation.
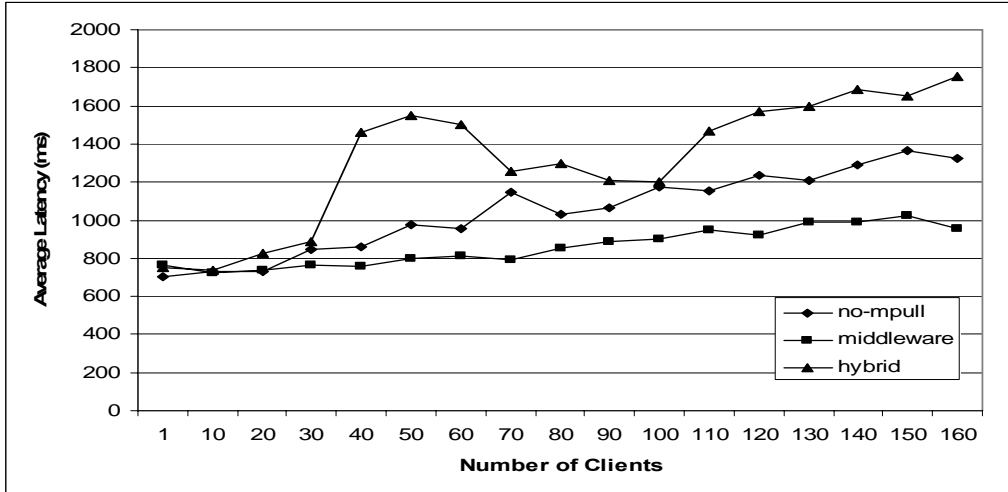
Figure 27 - Comparing multiple channel methods for static document popularities

Figure 27 shows the same data as in Figure 26, but this time focusing only on the three multiple channel methods. Here we can see the actual division between the average latencies of the different multiple channel schemes. The first interesting observation is that using the algorithm we developed for the division of documents performed much better than using the static threshold method for dividing documents. This can be seen for just about all clients' loads (beyond the first 30, when most of the documents are on the unicast channel). Looking at the figure, we see that even with a relatively low client load of 30 clients, no-mpull is 5% better and middleware is 13.7% better. This different gets larger as the number of clients is increased. When the number of clients is at 90, the difference becomes 11.6% and 23.3% for mo-mpull and middleware, respectively. When the number of clients is 130 the difference is 24.2% and 38% and when the number is 160, the difference is 24.5% and 45.5% for no-mpull and middleware.

The reason using our algorithm performs better is due to the way that the division works. First, there is the fact that using our algorithm also provides a bandwidth division to use. With the hybrid method using a 50/50 split of bandwidth, it is very inefficient and causes both the unicast channel and multicast push channel to be limited in their sending, building up queues of documents to send out. This increases the response time a user faces. For the lower number of clients, the unicast does not have the available bandwidth to service all the requests. This is because not many documents are popular enough to be put on the push channel but the push channel is holding 50% of the bandwidth. When the number of clients is higher, a lot of items

87

are put on the push channel.   Combining this with the fact that the push channel has many items on it, clients must now wait longer to get items off that channel, which contains the most popular items.   This leads to the discrepancy between the response times using our algorithm and the threshold algorithm.

Another interesting observation to make about Figure 27 is that, similar to the previous result set, it shows middleware provides the best average latency, especially as the number of clients increases.   Comparing between the two methods using our algorithm, middleware is always as good as no-mpull.   For 60 clients, middleware is 15.3% better than no-mpull.   This difference increases to 25.4% when the number of clients is 120 and 27.8% when the number of clients reaches 160.   This shows that using multicast pull is much better than not using multicast pull, even when the popularity of documents is not changing.   The reason for this difference is because of the shared documents on the multicast pull channel, which clients are listening to, and the decreased load on the server.

One final observation from Figure 27 is the behavior of hybrid.   As the figure shows, the response times of hybrid increase by a large amount from 30 to 50 clients, and then decreases from 50 to 100 clients, before increasing by a large amount again.   The reason for this behavior is the threshold number set for the document division algorithm.   As the number of clients goes from 30-50, the threshold is still too high for the right number of items on the push channel, and therefore the pull channel is having too large a load being placed on it.   However, around 50 clients, the number of requests for all the popular items breaks the necessary threshold and the push channel gets the correct items on it, and the response times decrease to around the performance of the other multiple channel schemes.   Once the number of clients gets high again, the threshold is now too low, and the bandwidth division is not close to optimal, and the response time for items on the push channel increases a lot, causing the overall response times to increase, as the graph shows.   This shows that using the correct document selection and bandwidth division, along with the multicast pull channel, provides much greater benefits for a multiple channel hybrid scheme.
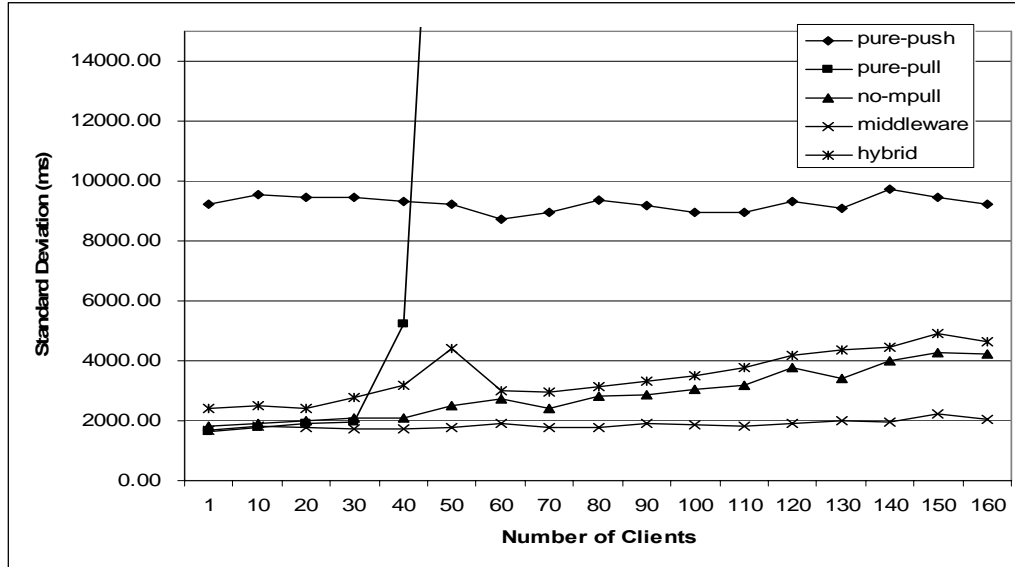
Figure 28 – Standard Deviation of real world server set ups with static document popularities

In addition to the response times, we also wanted to look at the variability in the request times experienced by the clients. We expected this to be the largest area of benefits provided by using the multicast pull channel within the multiple channel architecture. Figure 28 shows the results of the same experimental set shown in the previous 2 figures, but this time for the standard deviation of results. The larger the standard deviation, the more the range of results the user experiences varies from the average. Therefore, having a lower standard deviation makes the results clients experience closer in reality to the overall average. This also makes the system overall a more reliable system in delivering low latency for clients.

There are several observations to make from this figure. The first is that the standard deviations behave very similarly to the response times for the different schemes. All standard deviations increase as the number of clients is increases (except for push, which tends to have similar response times regardless of number of clients). Further, the unicast standard deviation scales just as high as the actual response time did. Secondly, all standard deviations are much higher than the associated response times.

Another observation to make is that the standard deviation of middleware is lower than the other two schemes. The reason for this is similar to the previous experiment. By using the multicast pull channel to respond to requests, along with the unicast and multicast push channels, the overall latency is lower, and clients receive documents quicker regardless of the channel that

89

is used. In the other multiple channel schemes, the difference between using the multicast push channel and unicast channel can be much greater, especially as the number of client increases and the wait on the unicast channel increases (or in the case of the pure hybrid, the wait on the multicast push channel increases). This causes a greater range of values for response times to be generated, which is reflected in the much larger standard deviation. This can be seen where for 30 clients, middleware is 24% better than no-pull and 37% better than using hybrid. This trend continues, where for 100 clients the differences are 31% and 39%, while for 160 clients the differences are 47% and 52% for no-mpull and the hybrid method, respectively.

One final observation to make is that using hybrid has, as was the case with the average response times, the worst performance of the three multiple channel schemes. The reason for this, as we clued into above, is that as the number of clients increases, the number of items on the multicast push channel, especially for hybrid, also increases. This causes the variance, as was the case with the response time, to start to reflect that of the multicast push channel. Because the standard deviation of the multicast push channel is high, so is the standard deviation of hybrid.

4.2.4 Real world experimentation with dynamic access patterns

The second experiment on the real world setup was a similar setup as the first experiment, except for this time the documents access patterns were dynamic, meaning that they change over time. The type of change we chose to impose was the large move type that was explained for the simulation based experiments. Recall that in the large move, the document popularities shift to a completely different set of documents. This means that what was the most popular at time t could be anywhere from the least popular to the second most popular item at time $t+1$, after the move has happened. In this experiment, we set the move time to be every 500 requests, as occurred in the previous set of experiments.

Unlike the previous experiment in the simulation environment, the change in popularity happens individually at each client. This means that the 500 requests are measured at each client as that client reaches each 500 request plateau. The change that occurs (which documents will be popular next) is the same at each client, however. The reason this piece of information is important is that it means clients that are closer to the server will switch choices of popular documents quicker. This means the popularity shifts are not completely happening at all times,
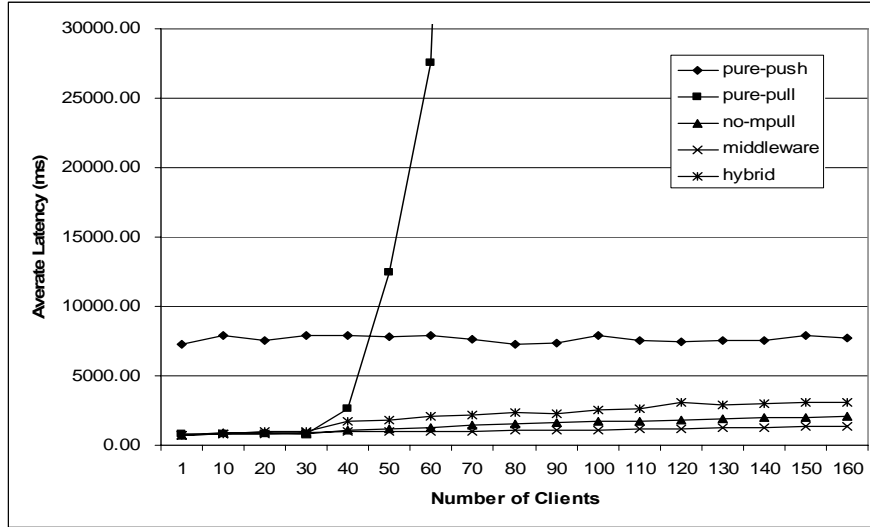
90

Figure 29 - Response times for all server setups with dynamic access patterns

so they will not be as severe as occurred in the simulated experiments, where all clients popularities could be shifted at the same time. Once again, the purpose of this experiment is to validate the results of the dynamic popularity shift experiments we ran in our simulation environment.

Figure 29 shows the results for the response times for all the server setups that were used during the experiment. As was the case in the previous real world experiment, the response time for the pure-push scheme stays relatively flat regardless of the number of clients that are used. This is an expected performance, as clients are still not making requests, so having the access patterns change should not change the performance of the push channel. This can also be seen if it is compared against the static scheme, since it has a similar response time as in that experiment. This also applies to the pure-pull method. Similar to the static request pattern experiment, the pure-pull server keeps a low request time until it reaches an overload point, at which time the requests scale to beyond our record keeping. Notice that the point at which this overload occurs is also the same as in the previous experiment. This too is expected behavior as the request pattern should have no effect on how the unicast channel behaves; the results support this line of thought.

Another observation to make from Figure 29 is that again, similar to the previous experiment, the three multiple channel schemes perform much better than the pure pull and pure push schemes. There is a difference, however, in how the three schemes compare to one another,
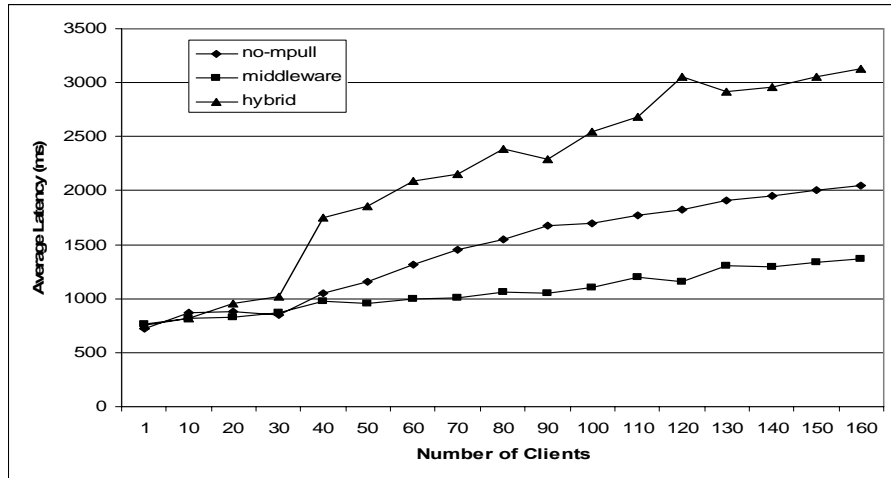
91

Figure 30 - Comparison of three multiple channel scheme for dynamic request patterns

which is shown in Figure 30. The difference in performance is much greater when the request patterns are dynamic than when they are static. This can be seen through the numbers, where for 30 clients, where using our algorithm produced similar results with and without the multicast pull channel, but hybrid was 15.2% worse average latency than both schemes. However, when the number of clients is increased to 60, middleware produces results which are 23.6% better than no-mpull and over 50% better than hybrid. This pattern continues, where for 90 and 160 clients the difference between using middleware and no-mpull is 37.1% and 33.3%, respectively. For comparing middleware against hybrid the differences are 53.8% and 56.3% for 90 and 160 total clients, respectively. This shows that using the multicast pull channel in combination with our algorithm produces much better average latency than the other methods, especially for dynamic document popularities, which confirms the findings in the simulation experiments.

One final observation about Figure 30 is that all three schemes continue to increase as the number of clients increase, and the resulting times in this experiment are higher than in the previous experiment with static access patterns. This is expected, as when the popularity is shifting, the unicast channel will receive more requests than it would when it has the division correct at all times. This increases the response times at all clients, however using the multicast pull channel helps to soften the load. Also notice that unlike the previous experiment, there is not a mini bump in the response time pattern of the pure hybrid channel. We believe that unlike previously, where the threshold kicked in and sorted the channels correctly, this time the
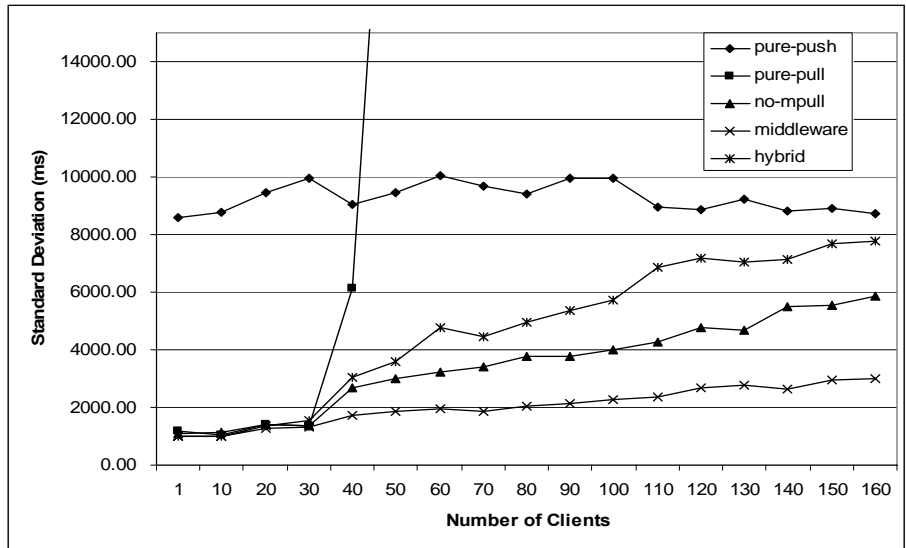
92

Figure 31 - Standard Deviation in response times for different server setups and dynamic document popularities

changing access patterns caused the pure hybrid to constantly have the wrong number and types of items on the multicast push channel, increasing the response times for all clients.

Figure 31 shows the variability in response times for the different server setups, again in the same way as the previous experiment. Similar to the previous experiment, the standard deviation of the pure push channel is rather flat and the standard deviation of the pure pull channel scales off the charts when the threshold for number of clients is hit. Also, like in the previous experiment with static patterns, using the multicast pull channel helps to decrease the amount of variance in the system. In this case, it is actually much more effective in decreasing the standard deviation, since it does not suffer from brief periods of having the wrong items on the push channel and having to service all requests one at a time over unicast. Looking at the numbers confirms this fact, as middleware is 2.4%, 39.5%, 42% and 48.7% more effective than no-mpull for 30, 60, 90 and 160 clients respectively. Likewise, comparing middleware against hybrid produces differences of 15.7%, 59.1%, 59.7% and 61.1% for 30, 60, 90 and 160 clients. Thus, the experiment confirms our hypothesis that using the multicast pull channel is effective in both lower average response times and lowering the overall standard deviation in response times experienced by clients.

93

4.3 EXPERIMENT SUMMARY

In this chapter, we looked at different experiments to measure the effectiveness of our hybrid system architecture improvements. We performed these experiments in two different environments, a simulated environment on a mostly secluded local area network and in a full application test over the internet through the use of the planet lab environment. All experiments measured their results in milliseconds, with the simulated results being more for relative comparison and the real world environment being for actual comparisons. We now summarize the results of our experiments.

The following observations can be found in common between the two experimental sets:

- Using the multicast pull channel does not adversely affect the performance of the hybrid system architecture; in most cases it actually improves it.

- The standard deviation of the results is much lower when using the multicast pull channel as compared to not using the multicast pull channel

- The main area that the variance in times is found is in the response times of the pull channel times, where using the multicast pull channel further helps to lower the average response time experienced by clients.

- When there is a large move in the popularity of the documents, the effectiveness of the multicast pull channel is at its best. In this case, the average latency and standard deviation are much lower than when not using the multicast pull channel. The reason for this is that the documents on the push channel are not correct. When the move occurs all the items on the push channel are wrong, so using the multicast pull channel is even more effective in getting the request queue empty, thus lowering response times for unicast requests as they come in.

The following results can be gathered from the simulation area experiments:

- The effectiveness of using the multicast pull channel is much larger when the zipf theta is lower and the document popularities are static.

- When there is a small move in the popularity of the documents, the effectiveness of the multicast pull channel is greater in both terms of response times experienced by clients and the variance of results. The reason is that when the shift occurs, before the document

94

selection is run, there is at least one document which is getting more requests than it should, for which the multicast pull channel is handling all the requests with a single send out.

- We showed that using a previously popular method for push document popularity, which was to drop an item off the push channel to test its popularity, caused spikes in response times when items were dropped, where our scheme did not suffer from those response time spikes.

The following results can be gathered from the real world experiments:

- Using a pure multicast push approach causes response times which are much larger than any of the multicast channel hybrid schemes but does provide unlimited scalability for the system

- Using the pure-pull approach provides excellent results until a threshold on the number of requests that can be handled is met, at which point the result times scale towards infinity.

- Both the pure push and pure pull schemes do not suffer any noticeable difference in performance when the document popularities are static or dynamic in nature.

- Using a threshold to decide which documents to place on the push channel and which documents to place on the pull channel can create a very non-optimal split of documents which causes response times to increase over using our algorithm in a hybrid system

- Using an even division of bandwidth, which would be the case without an integrated algorithm, causes increased response times and variance as the number of clients is increased, which is the case in the pure hybrid system we experimented with.

- Using our algorithm with and without the multicast pull channel provided better results in a real world environment than using a typical hybrid system.

- Using the multicast pull channel provided the same benefits in the real world experiment as it did in the simulated experiments. This was the case for both static and dynamic document popularity patterns.

Overall, our experiments succeeded in validating the claims that we had made about the improved architecture for hybrid systems that we developed. Using our document selection algorithm provided a better split of documents and bandwidths than was the case with other

methods we examined, as we hypothesized. We were also able to show that the results from the simulations were not solely from having similar sized documents or an isolated environment. While using the Planet Lab environment did cause certain variability to be introduced, the overall pattern of response times held to the simulated results. This meant that the system with the multicast pull on provided better results (in terms of lower response times) than the system with the multicast pull off. This alludes to the fact that in a real world implementation, using our full architecture will provide the benefits that we have explained within this work, and would be useful for expanding the scalability cheaply for a given server.

5 CONCLUSION AND FUTURE WORK

5.1 CONCLUSION

In this dissertation, we have proposed a new architecture for hybrid data dissemination architecture. Our architecture includes a third distribution channel called the multicast pull channel. Our main goal was to both provide scalability, while minimizing the response times experienced by. In particular, we accomplished the following tasks during the entirety of our work:

1. We designed an algorithm called SELDIV that will close to optimally solve the document selection problem. The document selection problem is deciding which items should be placed on the multicast push channel, and which items should be requested over the unicast channel.

2. We provided a way to divide bandwidth between the push and pull channel. This is important because if there is not enough bandwidth for the pull channel, it will get easily overloaded. If there is not enough bandwidth for the push channel, the latency for the popular documents will be high; this will in turn cause overall system latency to be high. Hence, an appropriate balance must be maintained. We accomplished this with an integrated algorithm for both the document selection and the bandwidth division.

3. We proposed to add a multicast pull channel to the multi-channel hybrid architecture. The multicast pull channel is used to both enhance scalability and performance, while keeping the variance of client latencies low.

4. We examined several methods for dividing both the documents and the bandwidth for the new channel we have proposed. We evaluated the pros and cons of these methods, and explain our rationale for the method that we adopted in our system.

5. We provided a fully functional multi-channel hybrid data dissemination system. We discussed the design decisions that we made in developing this system.

6. We performed empirical and analytic analysis of our architecture and algorithms in a simulation environment, where we can isolate the individual pieces of our architecture.

7. We performed experiments in a real world environment on Planet Lab, and found that the results matched those that we obtained in our simulation environment.

8. We provided an implementation of our hybrid architecture that can be used in a wireless environment.

We accomplished the goal of this thesis: *to create a server system that provides scalability at a low cost, while still keeping the response times experienced by clients low*. We believe this work provides a foundation for the mass adoption of hybrid data dissemination systems.

5.2 FUTURE WORK

The architecture that we propose in this dissertation is a multiple channel hybrid data dissemination system that uses three distinct channels to deliver data to clients. This architecture is an improvement over the previous hybrid system architecture. But there is always room for further enhancements. Some of these possible enhancements include:

1. *Operating with dynamic web content.* We never specifically addressed the issue of dynamic content, and how our system would deal with such content. If the dynamic content is unpopular data, then using the pull channels for accessing this content should not provide any issues, as our architecture keeps the pull channels relatively open. If the dynamic portion of the content is popular enough, it will be placed on the push channel. Clients will be able to continually access the updates to the dynamic data as they appear over the multicast push channel. In this way, our system is already configured to handle dynamic content, and should not be adversely affected by its implementation at a web site. Although various optimizations for dynamic data are imaginable.

2. *Using our improved architecture with edge caching.* If a collection of clients are using edge caching, our middleware can be used in conjunction with edge caching to enhance performance. If the dynamic content can be considered popular, then it will be

continually delivered to all clients. This fits directly into the edge caching scheme, where the static portion of the page can be kept at the cache (thus alleviating more pressure at the server) and the dynamic data itself is pushed to the cache. The edge cache then has constant access to the dynamic data for all clients using the edge cache. Additionally, the architecture still provides other channels over which documents that are not popular, or not kept in the cache, can be fetched. Edge caching fits in perfectly with our architectural design due to the nature of how it behaves in concurrence with how our architecture delivers data.

3. *Decreasing the size of the push channel broadcast.* One of the key features of our system is the multicast push channel. In addition to data, this channel also contains an index which is used to determine the contents of the channel. It may be possible to exploit this index to decrease the size of the channel. If the items on this channel are large, the channel may use a lot of bandwidth across the network (for all nodes in the multicast tree). If the data is not changing, and no new clients require the data, it may be better to only send information in the index alerting clients of this fact. It may also be possible to only send the changes to the documents instead of the entire document each time. The questions this will pose are how to decide which portions of data to send, how to handle new clients, and how to handle different caching abilities at clients.

4. *Combining multiple instances of our middleware.* In this dissertation we focused on multiple clients but a single server. However, having many servers use our architecture may provide an opportunity to combine multicast channels. If a client is forced to maintain twenty different multicast connections (from ten different servers, for example) it may overtax the ability of the client. By having servers share multicast channels, it allows clients to connect to more servers, with less work to maintain the connections. This would be a particular issue for a client proxy with multiple users behind it. This would also solve a major issue with multicast channels, which is that a client may not be able to maintain many connections at a time without running out of memory.

5. *Using our architecture in combination with other systems.* As we have previously mentioned, our multiple channel hybrid data dissemination architecture could be used by any system to scalably and quickly disseminate data to a large number of clients. While in this dissertation we did not test it with other systems, we envision it working well as a

distribution layer if needed. This could include as part of a subscriber based system, were constantly updated items could be sent over the multicast push channel, the multicast pull channel for updates that pertain to a small number of users, and unicast for the updates that only affect a couple of users. The use of our system can therefore be extended beyond the design as a server side proxy and become a pure distribution layer.

6. *Replacing the underlying dissemination system with a peer-to-peer network.* The work we presented in this dissertation focused on the idea that the Unicast channel would be used for the one to one communication, and that the push based dissemination based on multicasting. It may be possible, however, to replace the underlying network with a peer-to-peer network. Instead of requests coming over Unicast to the server, the requests could be spread out throughout the network. Only if the data could not be found would the clients need to directly request from the server. The push-based channel would still be push-based, except it could rely on the underlying peer-to-peer network to flood the push data to all nodes, in a similar way the multicast push channel worked. This way, all clients are receiving the most popular data, but requesting the less popular data now has chances to be answered in places other than the main server. This would further help to increase the scalability of the solution, and by possibly having the data located at a close by client, lower the latency over all for the system.

In addition to other improvements and uses of our architecture, we identify a couple of open questions that are related to this dissertation:

1. What happens if clients have a large difference in bandwidth available and processing speed? Should the system slow down sending out data to the lowest ability client? We believe this can be accomplished by the multicast channel instead of our system, but may still need to be considered.

2. Is it possible to add more multicast channels to further improve the performance of the architecture? We believe that using more multicast channels will only cause more load on the client and provide no benefits in terms of variance or latency. The multicast pull channel is currently utilized only a small percentage of time, another multicast channel would not seem to add anything to the architecture.

BIBLIOGRAPHY

1) Abadi, D. and Carney, D. and Cetintemel, U. and Cherniack, M. and Convey, C. and Erwin, C. and Galvez, E. and Hatoun, M. and Hwang, J. and Maskey, A. and Rasin, A. and Singer, A. and Stonebraker, M. and Tatbul, N. and Zing, Y. and Yan, R. and Zdonik, S. *Aurora: A Data Stream Management System* (Demo). In Proc. SIGMOD, 2003.

2) Acharya, S. and Muthukrishnan, S. *Scheduling on-demand broadcasts: New metrics and algorithms*. In Proc. ACM/IEEE MobiCom, 1998.

3) Akamai: The trusted choice of online businesses. http://www.akamai.com

4) Aksoy, D. and Franklin, M. *RxW: A scheduling approach for large-scale on-demand data broadcast*. IEEE/ACM Transactions on Networking. Vol. 7, No.6, 1999.

5) Aksoy, D. and Franklin, M. *Scheduling for Large-Scale On-Demand Data Broadcasting*. In Proc. INFOCOM, 1998.

6) Alouf, S. and Altman, E. and Barakat, C. and Nain P. *Estimating membership in a multicast session*. Performance Evaluation Review, (Proc. of ACM SIGMETRICS), 31(1), 2003.

7) Altinel, M. and Aksoy, D. and Baby, T. and Franklin, M. and Shapiro, W. and Zdonik, S. *DBIS Toolkit: Adaptable Middleware for Large Scale Data Delivery*. In Proc. ACM SIGMOD, 1999.

8) Azar, Y. and Feder, M. and Lubetzky, E. and Rajwan, D. and Shulman, N. *The Multicast Bandwidth Advantage in Serving a Web Site*, In Proc. 3rd NGC, 2001.

9) Babcock, B. and Datar, M. and Motwani, R. *Load Shedding Techniques for Data Stream Systems*. In Proc. of the 2003 Workshop on Management and Processing of Data Streams. 2003.

10) Bansal, N. and Coppersmith, D. and Suiridenko, M. *Improved approximation algorithms for broadcast scheduling*. In Proc. of 7th Annual ACM-SIAM Symposium on Discrete Algorithms. 2006.

11) Barish. G. and Obraczka, K. *World Wide Web Caching: Trends and Techniques*. IEEE Communications Magazine Internet Technology Series, May 2000.

12) Beaver, J. and Chrysanthis, P.K. and Pruhs, K. and Liberatore, V. *To Broadcast Push or Not and What?* In Proc. of MDM2006, 2006.

13) Beaver, J. and Chrysanthis, P.K. and Pruhs, K. and Liberatore, V. *Improving the Hybrid Data Dissemination Model of Web Documents*. World Wide Web Journal (to appear)

14) Beaver, J. and Morsillo, N. and Pruhs, K. and Chrysanthis, P.K and Liberatore, V. *Scalable Dissemination: What's Hot and What's Not*. 7th International Workshop on the Web and Databases, 2004.

15) Beaver, J. and Pruhs, K. and Chrysanthis, P.K. and Liberatore, V. *The Multicast Pull Advantage in Dissemination-based Data Delivery*. Proceedings of Third Hellenic Data Management Symposium, 2004.

16) Bertosi, A. and Pinotti, M. and Ramaprosad, S. and Rizzi, R. and Shashanka, M. *Optimal Multi-Channel Data Allocation with Flat Broadcast per Channel*. In Proc IPDPS. 2004.

17) Breslau, L. and Cao, P. and Fan, L. and Phillips, G. and Shenker, S. *Web caching and Zipf-like distributions: Evidence and implications*. In Proc. INFOCOM, 1999.

18) Burns, R. and Rees, R. and Long, D. *Efficient Data Distribution in a Web Server Farm.* IEEE Internet Computing, Vol. 4, No. 5, 2001.

19) C`aceres, R. and Douglis, F. and Feldmann, A. and Glass, G. and Rabinovich, M. *Web proxy caching: The devil is in the details.* In Proc. ACM SIGMETRICS Workshop on Internet Server Performance, 1998.

20) Cao, F. and Singh, J.P. *MEDYM: Match-Early and Dynamic Multicast for Content-based Publish-Subscribe Service Networks*. In Proc. of the 6th ACM/IFIP/USENIX International Middleware Conference, 2005

21) Castro, M. and Druschel, P. and Ganesh, A. and Rowstron, A. and Wallach, D.S. *Security for structured peer-to-peer overlay networks*. In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation, 2002

22) Castro, M. and Druschel, P. and Kermarrec, A. and Rowstron, A. *SCRIBE: A large-scale and decentralized applicationlevel multicast infrastructure.* IEEE Journal on Selected Areas in communications, 2002.

23) Chiu, D. and Hurst, S. and Kadansky, M. and Wesley, J. *TRAM: A Tree-based Reliable Multicast Protocol.* Sun Microsystems, No. TR-98-66, 1998

24) Chrysanthis, P.K. and Pruhs, K. and Liberatore, V. *Middleware Support for Multicast-based Data Dissemination: A Working Reality.* Proc. of the 8th IEEE International Workshop on Object-oriented Real-time Dependable Systems, 2003.

25) Chu, Y-H and Rao, S, and Zhang. H. *A Case for End System Multicast*. In Proc. ACM SIGMETRICS, 2000.

26) Cormode, G. and Muthukrishnan, S. *What's hot and what's not: Tracking frequent items dynamically*. In Proc. of Principles of Database Systems, 2003.

27) Dolev, D. and Mokryn, O. and Shavitt, Y. *On multicast trees: Structure and size estimation.* In IEEE INFOCOM, 2003

28) Dykeman, H.D. and Ammar, M. and Wong, J.W. *Scheduling algorithms for videotex systems under broadcast delivery*. In Proc. International Conference on Communications, 1986.

29) Foltz, K. and Xu, L. and Bruck, J. *Scheduling for Efficient Data Broadcast over Two Channels.* In Proc. ISIT, 2004.

30) Fredrix, E. *Cheney slip sends Net surfers to anti-Bush site.* http://www.cnn.com/2004/ALLPOLITICS/10/06/debate.website.ap/index.html.

31) Google web search engine. http://www.google.com

32) Gross, D. and Harris, C. H. *Fundamentals of Queueing Theory*. John Wiley & Sons, Inc. Third Edition. 1998.

33) Jannotti, J. and Gifford, D. and Johnson, K. and Kaashoek, F. and O'Toole. *Overcast: Reliable Multicasting with an Overlay Network.* In Proc. USENIX OSDI. 2000.

34) Kenyon, C. and Schabanel, N. and Young, N. *Polynomial-Time Approximation Scheme for Data Broadcast.* In Proc. STOC, 2000.

35) Krishnamurthy, B. and Wills, C. and Zhang, Y. *On the use and performance of content distribution networks.* In Proc. ACM SIGCOMM Workshop on Internet Measurement, 2001.

36) Krishnamurthy, B. and Rexford, J. *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement.* Addison-Wesley. 2001.

37) Li, B. and Guo, J. and Wan, M.: *iOverlay: A lightweight middleware infrastructure for overlay application implementations*. In Proc. of the 6th ACM/IFIP/USENIX International Middleware Conference, 2004

38) Liberatore, V. *Caching and Schedulingfor Broadcast Disk Systems.* In Proc. ALENEX, 2000.

39) Liebeherr, J. and Wang, J. and Zhang, G. *Programming Overlay Networks with Overlay Sockets*. In Proc. NGC 2003.

40) Napster. htpp://www.napster.com

41) Nonnenmacher, J. and Biersack, E.W. *Scalable feedback for large groups*. IEEE/ACM Transactional Networking. 7(3), 1999.

42) Pendarakis, D. and Shi, S. and Verma, D. and Waldvogel, M. *ALMI: An Application Level Multicast Infrastructure*. In Proc. 3rd USENIX Symposium on Internet Technologies, 2001.

43) Penkrot, V. and Beaver, J. and Sharaf, M. and Roychowdhury, S. and Li, W. and Zhang, W. and Chrysanthis, P.K. and Liberatore, V. and Pruhs, K. *An Optimized Multicast-based Data Dissemination Middleware: A Demonstration*. In Proceedings of 19th International Conference on Data Engineering, 2003.

44) PlanetLab - An open platform for developing, deploying, and accessing planetary-scale services. http://www.planet-lab.org/

45) Ripeanu, M. and Foster, I and Iamnitchi, A. *Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design*. IEEE Internet Computing Journal special issue on peer-to-peer networking, vol. 6(1), 2002.

46) Rosenzweig, P. and Kadansky, M. and Hanna, S. *The Java reliable multicast service: A reliable multicast library*. Sun Microsystems, No. SMLI TR-98-68, 1998.

47) Stathatos, K. and Roussopoulos, N. and Baras, J.S. *Adaptive Data Broadcasting Using Air-Cache*. In Proc. WOSBIS, 1996.

48) Stathatos, K. and Roussopoulos, N. and Baras, J.S. *Adaptive Data Broadcast in Hybrid Networks*. In Proc. VLDB, 1997.

49) Stoica, K. and Morris, R. and Karger, D. and Kaashoek. F. and Balakrishnan, J. *Chord: A scalable Peer-to-peer Lookup Service for Internet Applications.* In Proc. ACM SIGCOMM, 2001.

50) The Apache Software Foundation. http://www.apache.com/

51) Wang, J. *A survey of Web Caching Schemes for the Internet*. ACM Computer Communication Review. Vol. 25, No. 9. 1999.

52) Yang, B. and Garcia-Molina, H. *Designing a Super-peer Network*. IEEE International Conference on Data Engineering, 2003

53) Yang, B. and Vinograd, P. and Gracia-Molina, H. *Evaluating GUESS and Non-Forwarding Peer-to-Peer Search*. In Proc. ICDCS, 2004