Just-In-Time Push Prefetching: Accelerating the Mobile Web

by

Nicholas D. R. Armstrong

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2011

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Web pages take noticeably longer to load when accessing the Internet using high-latency wide-area wireless networks like 3G. This delay can result in lower user satisfaction and lost revenue for web site operators. By locating a just-in-time prefetching push proxy in the Internet service provider's mobile network core and routing mobile client web requests through it, web page load times can be perceivably reduced. Our analysis and experimental results demonstrate that the use of a push proxy results in a much smaller dependency on the mobile-client-to-network latency than seen in environments where no proxy is used; in particular, only one full round trip from client to server is necessary regardless of the number of resources referenced by a web page. In addition, we find that the ideal location for a push proxy is close to the servers that the mobile client accesses, minimizing the latency between the proxy and the servers that the mobile client accesses through it; this is in contrast to traditional prefetching proxies that do not push prefetched items to the client, which are best deployed halfway between the client and the server.

Acknowledgements

To my family — my parents, grandparents, and especially my brothers — thank you. The breaks I spent with you kept me sane, and the support and encouragement you provided saw me through to the end.

To my supervisor, Paul Ward, the members of the Shoshin research group, and the students and faculty I interacted with along the way; thank you for your guidance and assistance in navigating University practices.

To my friends, for always listening to me rant about the problems I had difficulty solving, for providing perspective, and who never ceased to remind me why it was that I decided to do this; thank you.

And finally, thank you to Pravala Inc. and the Government of Canada, without whose financial assistance this degree would not have been completed.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Internet access is currently provided to mobile users through the use of wide-area wireless networks like those shown in Figure 1.1. Though these networks have similar raw data rates to local-area wireless access networks, the users accessing the network must share this capacity, causing each user to receive less than the full raw data rate. This sharing, as well as other technological and network conditions (including backhaul limitations) typically results in a high client-to-network latency. A detailed study of latencies experienced in 3G networks conducted by Fabini et al. [18] found that the client-to-network latencies were in excess of 50 ms when uploading data in the presence of a background data flow. This client-to-network latency is much higher than the latencies of residential cable or DSL connections that access the Internet.

Though network improvements can — and will — reduce some sources of latency, the only way to decrease the latency caused by spectrum sharing in mobile networks is to reduce the amount of sharing. This can be achieved by contracting the area covered by a cell, thereby allowing for an increase in the spatial re-use of network frequencies, or by increasing the number of frequencies

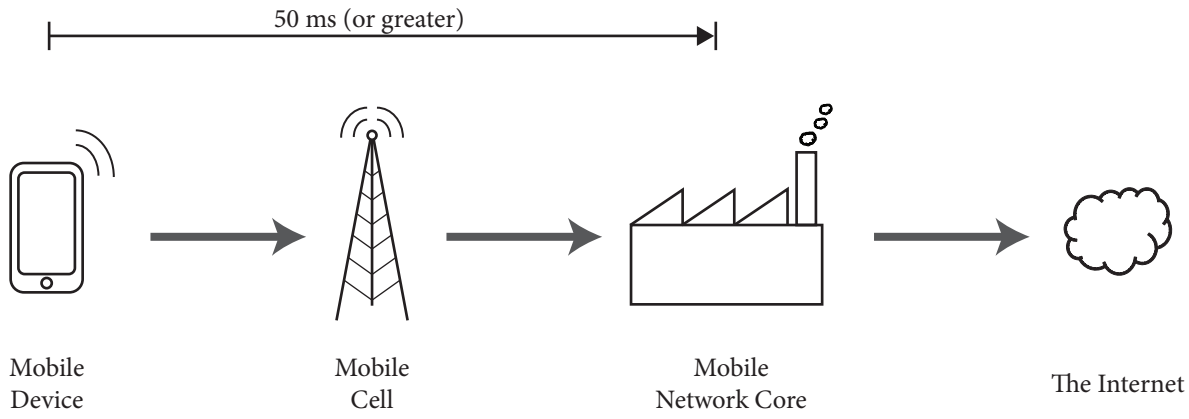


Figure 1.1: Simplified 3G network diagram

available for data transmission.

Though technologies like Wi-Fi and Femtocells cover smaller areas and therefore serve fewer users, it is unlikely that service providers will choose to blanket their current service areas with these technologies; if the coverage area of a cell tower is approximated as circular, each halving of the coverage radius quadruples the number of towers required, driving up the cost. Increasing the number of frequencies is not a viable option, as the number of frequencies suitable and licensed for cellular data transmission is limited. We therefore posit that wide-area wireless networks will continue to have higher latencies than wired and local-area wireless networks.

When we combine this high-latency wireless segment with the process of loading a web-page, we see perceivable increases in the time it takes to load a complete page. As web browsing forms the majority of network activity for smartphone users [50], increasing the speed at which pages load would result in an improved smartphone experience.

Multiple techniques have been proposed for increasing the speed at which pages load, from caching and prefetching proxies to content-delivery networks. However, most of these solutions are designed to be shared by multiple users, and as a result they are by definition on the far side

of the high-latency wireless segment and therefore do not improve page-load performance to the extent we desire. Furthermore, as these techniques work by moving content closer to the client, they magnify the unbalanced ratio between the high-latency wireless segment and the subsequent wired path to the server.

Two existing ways in which we can mask this delay — by having the client prefetch pages before the user navigates to them, and by having the server push the complete web-page, including resources, on the first request (limiting the traversals of the large latency segment to one) — have inherent characteristics that prevent them from being suitable for the mobile environment. In this work, we propose a just-in-time push proxy specifically targeted for wireless wide-area networks that combines non-speculative prefetching on the proxy with a push to the requesting client to produce an architecture capable of increasing the page-load speed on mobile devices. Architecturally, our system is similar to commercial multipart/related web accelerators; our design and implementation differs in that individual resources can be downloaded in parallel rather than in sequence. Furthermore, our just-in-time push proxy pushes prefetched content to a proxy daemon on the client, allowing operation with unmodified web browsers.

1.1 Contributions

This thesis presents a just-in-time prefetching push proxy that increases the speed of mobile browsing in wide-area wireless networks. The main contributions of this work are:

1. An analysis of the general technique used in architectures where a proxy server prefetches embedded resources and pushes them to a client, showing theoretical limits on the reduction in page-load time.

2. An implementation of the above architecture demonstrating the load-time reduction seen in our validation environment as well as in a live environment.

1.2 Organization

Chapter 2 gives an overview of the sources of delay in wide-area wireless networks, describes how latency affects the interaction between web browser and web server, and explores existing techniques for minimizing the effect of latency on page-load times. In Chapter 3, we describe the architecture for our just-in-time prefetching push proxy, and analyze its operation to understand the potential benefits of such an architecture. To measure our architecture's potential, Chapter 4 details an implementation of our architecture and presents results from a series of tests run with our simple implementation. We discuss future work in Chapter 5, and provide a brief outline of work related to ours that reduces page-load times through mechanisms other than caching and prefetching proxies. Finally, we present our conclusions in Chapter 6.

Chapter 2

Background

As noted in Chapter 1, the decreased page-load speeds experienced when web browsing using a mobile Internet connection are rooted in the increased latencies present in wireless wide-area networks in combination with the structure of web-pages themselves.

2.1 Sources of delay in wide-area wireless networks

In wireless wide-area networks, information (voice or data) transmitted from a mobile phone is captured by fixed antennas installed by service providers to cover geographical areas called cells. Assigned to these cells are specific frequencies on which they may transmit; neighbouring cells receive different frequencies to ensure that their transmissions do not overlap [51]. Active mobile devices in a cell are assigned a fraction of the cell's allotted frequencies to communicate with the cell base station; this fraction may be a time slice of a frequency band (for GSM networks), a unique code across the whole frequency set (for CDMA networks), or some other division [51]. When traveling between cells (roaming), the mobile device exchanges its fraction in the current cell for a

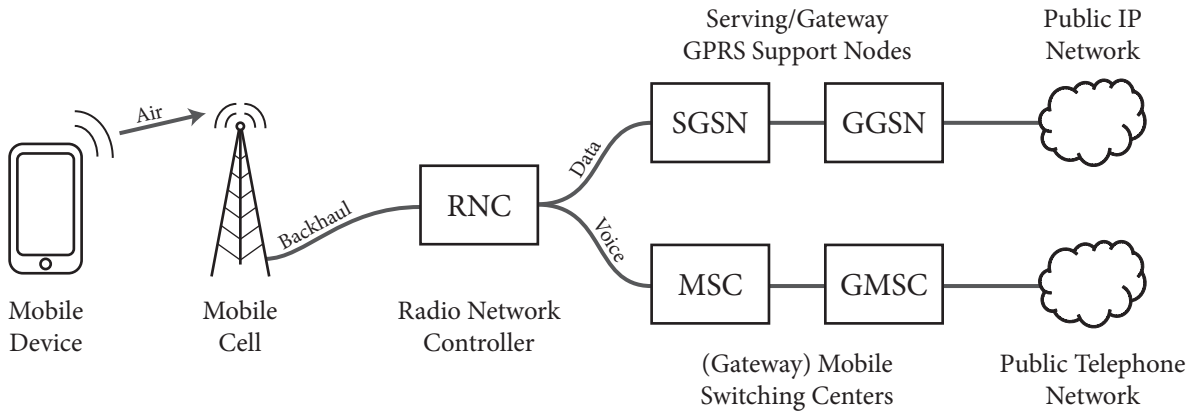


Figure 2.1: Detailed 3G network diagram

fraction in the new cell, freeing the fraction in the previous cell for re-use [51]. Inactive devices are not assigned any dedicated spectrum resources, instead sharing a random-access channel for signaling and paging.

If we follow the flow of data through a wireless wide-area network, we can identify a number of sources of latency in current mobile networks. A visual depiction of the elements in a GSM network is given in Figure 2.1 [58]. When a device wishes to transmit data, it first negotiates with the cell tower for a channel using a wireless MAC protocol; this is one source of latency. Once this channel is established, data can flow between the device and the cell tower; this transmission happens over spectrum that must be shared, another source of latency. After data is received by the tower, it is processed (more latency) and sent to a radio network controller (RNC) located in the mobile network core, where voice and data services are separated and routed to the appropriate network [58]. The communication between the cell tower and the RNC occurs over a high-capacity backhaul link. However, due to the rapid increase in the raw data rate a cell must handle to provide high-speed data service, not all cells have backhaul links of sufficient capacity to support all active clients in the cell [20], which can introduce additional latency.

If we go through that set of things that introduce latency again, we find that most of them can be reduced — processing time on the tower and RNC can be reduced with faster processors, and backhaul can be upgraded. Two components of this latency cannot be so easily improved — the latency incurred by sharing the channel with other users, and the latency resulting from the use of wireless MAC protocols. As a result, we expect latency on mobile networks to be a problem for the forseeable future; this work addresses the former source of latency, spectrum sharing.

Specifically, it is the way in which spectrum is shared that introduces latency. If each device were assigned dedicated frequencies for uplink and downlink, the latency over the air interface would be a mere 0.23 ms at the 35 km maximum range of a GSM tower [51]. As electromagnetic spectrum is a finite resource for which demand for prime frequencies outstrips supply, mobile networks must share these frequencies amongst all devices active in a cell's coverage area. For GSM networks, this is accomplished by splitting each frequency into 8 time-slots that repeat every 4.615 ms (a frame); virtual channels for transmitting data, voice, or signalling are constructed from sequences of these time-slots [51].

Voice and data channels are formed from time-slots using different assignment algorithms. Voice channels consist of a single consistent time-slot every frame, providing a guaranteed low latency (and low data rate) channel for communication with the tower [51]. Admission control is used by the network to ensure that assigned channels are reserved for the mobile device for the duration of the call; new calls, and active calls for users roaming into the cell's coverage area are only admitted by the network when an unused channel is available.

Data channels on the other hand are assigned dynamically to more closely match the needs of packet-switched data, and can be formed from multiple slots per frame to provide higher data rates [51]. The available data capacity of the cell is divided amongst all active users, providing access

for all users when the network is heavily loaded and higher speeds when the network is lightly loaded. However, since the assignment of these time-slots is not consistent, both jitter and latency are higher than with voice channels [51].

Though the particular details of each wide-area wireless network combine to produce its particular latency characteristics, at very fundamental level each network is making a trade-off between throughput and latency. If we increase the length of time each device is allowed to use a frequency, less signalling is required, less time is lost to guard space, and more data can be transferred – frequency use is more efficient. However, this also means higher latencies, as devices wishing to transmit must wait longer – there are fewer users of the frequency in each time period. Conversely, reducing the amount of time a device can hold a frequency increases the number of devices that can transmit in a time period, reducing the time a device must wait and thereby reducing latency; this also results in more time loss to guard space, more signalling, and a lower efficiency. Historically, throughput has been a larger problem than latency; ever-increasing demands for data are placed on wide-area wireless networks, and the designs of these networks have therefore been biased toward throughput. With the explosive growth in smartphones in recent years, it is unlikely that this bias will change – mobile networks face unprecedented capacity demands [11].

Technologies like Wi-Fi and Femtocells offer us a way to decrease sharing by reducing the geographical area covered by a cell. This decreases the number of users to whom the cell can provide service; decreasing the coverage area also allows for denser frequency re-use, providing the potential for additional network capacity. By decreasing sharing, networks could use less efficient protocols to provide lower-latency service while maintaining data rates similar to those available today. However, given that the typical range of a femtocell is 20 to 30 metres [27], compared to

the urban coverage range of 1 to 3 kilometres for a typical macrocell [10], it is unlikely that service providers will choose to blanket their existing coverage areas with these devices. Current trials instead use femtocells to boost signal coverage in indoor locations, and push the costs of the femtocell off to the consumer; the user must pay for both the femtocell and the Internet connection it uses to backhaul data to the carrier's network [53, 62]. As a result of this incentive structure, femtocells have seen limited adoption thus far — end users have been generally resistive to the notion of paying the start-up and operation costs for extensions to the carrier's network.

2.2 How latency affects page-load speed

Content on the web is formatted using the HTML markup language. Using a series of metadata tags, textual content is arranged and formatted, and media elements — images, video, and interactive elements — are placed within the page content. When a web browser encounters a web-page, it builds a structure known as a Document Object Model (DOM) tree; Figure 2.2 displays a sample DOM tree. When the browser generates a DOM element for which the HTML file does not contain the content — an image, for instance — the browser fetches that element from the provided URL and inserts it into the tree. These external elements are often referred to as *embedded* objects [52], even though they are not embedded within the page itself, but referenced from it. It is only once the browser downloads and renders all parts of this tree that the load of a page is considered complete.

In addition to embedded media elements, web-pages often reference style sheets (CSS, Cascading Style Sheets) and snippets of interactive JavaScript code. Style sheets provide an enhanced way of modifying the look of the content on a page, by setting font styles, drawing borders or backgrounds, and by positioning elements on the screen. Some of the attributes set by a style

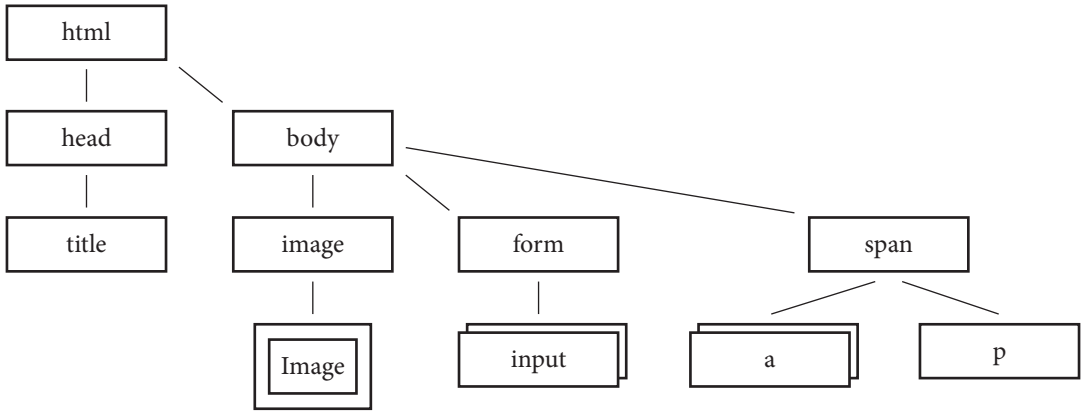


Figure 2.2: Document Object Model (DOM) tree sample

sheet support referencing media elements, like images; the browser fetches these resources when it applies the style. CSS also allows style sheets to reference additional style sheets; the browser fetches additional style sheets when it encounters the style sheet reference.

Most browsers also provide an environment for running JavaScript code, used to interact with the page's DOM tree. This allows developers to build interactive applications — like web-based email and banking applications — within the confines of a web browser. Since JavaScript can modify the DOM tree, it can insert elements into the page that result in the browser fetching additional elements.

When the browser needs to fetch a resource, it issues a HTTP GET request to the server listed in the item's URL. This request starts with a query to DNS for the corresponding server IP address, after which the browser opens a TCP connection to the provided IP address. Once the browser establishes a connection, it sends a plain-text HTTP request header to the server. This header specifies the resource desired, the host from which it is requesting from, the identity of the client making the request, and a small amount of other information.

Upon receiving this header, the server locates or generates the desired resource, and transmits

the resource back to the browser along with a response header; this response header provides the information necessary for the client browser to understand the response. After the client receives the response, it may close the underlying TCP connection, or can keep it alive and re-use it for future requests [49]. Regardless of the state of the underlying TCP connection, for each resource the browser must transmit a new HTTP request/response pair; only the TCP connection carrying these requests is reused.

For this reason, the time it takes to load a page is particularly sensitive to the latency of the network over which is being requested, as each element must be individually requested and therefore must experience a round-trip from the client to the server and back. If these requests were to occur serially, this would add one round-trip time to the total load time for each resource referenced by the page. To improve performance, all browsers load elements in parallel by creating multiple TCP connections to the server when loading a page; this allows the browser to overlap requests and decrease the load time for pages referencing a large number of elements.

However, as the browser does not know all of the resources at the time of its initial request — it starts by requesting only the root HTML page — there exists a minimum bound on the load time of the page, based on the organization of elements it references. For pages without embedded resources, that bound is 1 round-trip time (excluding connection initialization for TCP/SSL). If the page references additional elements, that bound increases to 2 or more, depending on the number of resources referenced by the root HTML page and the amount of parallelism. If those resources themselves reference resources, the minimum bound continues to increase.

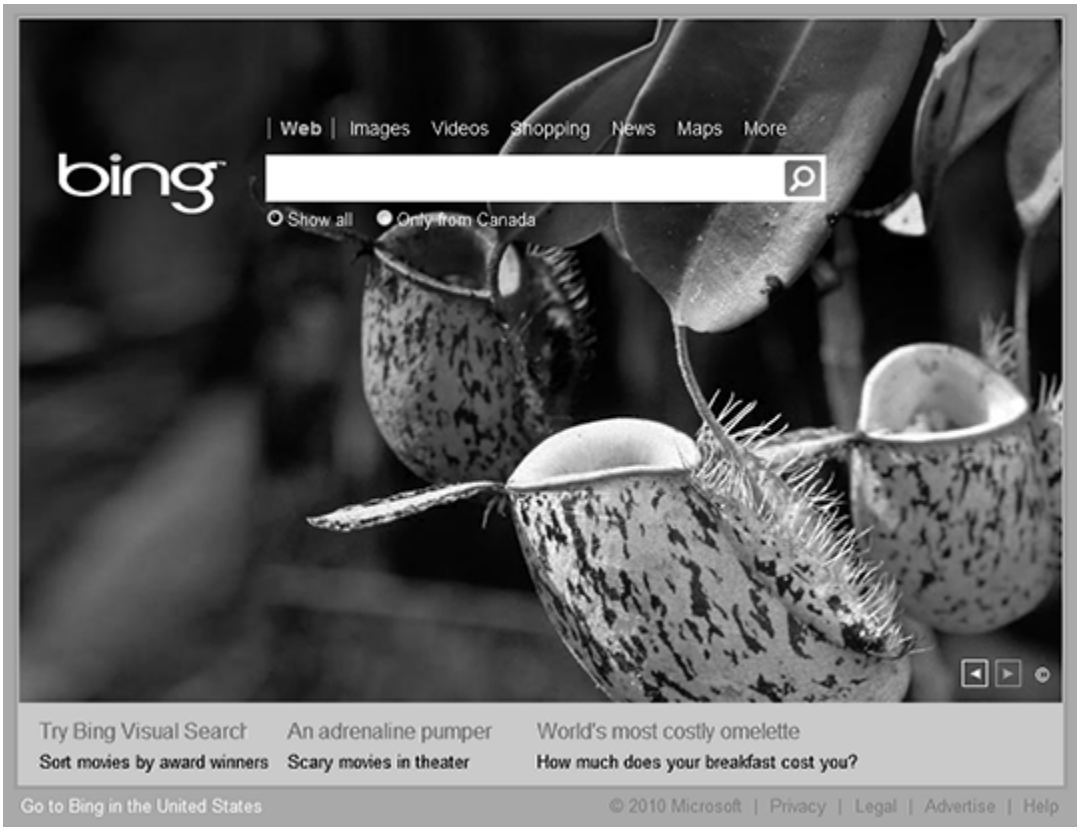
Furthermore, the HTTP protocol further limits the number of parallel connections to a single server to 2, to improve response times and avoid congestion [49]. This limit is commonly increased by modern browsers [6] — see Table 2.1 — but can still limit performance in instances where

Table 2.1: Browser connection limits

	Connections per Hostname	Maximum Connections
Chrome 8.0.562	6	30
Firefox 3.6.12	6	30
Internet Explorer 8.0	6	35
Opera 10.7	8	30
Safari 5.0.3	6	35
Android 2.2	4	4
iPhone 4.2.1	4	35
Opera Mini 5.1	11	30
Windows Phone 7	6	35

the browser needs to request a large number of small resources. In effect, this limit makes a large number of resources load in a similar fashion to a page with a deep resource structure, as the connection limit effectively batches requests into serially-loaded groups. As this limit is per hostname, some sites choose to load from multiple servers to avoid this limit; in this case, the browser will open connections until it reaches its global maximum (see Table 2.1).

When we observe this sequence of requests while loading an entire page, we see a waterfall-like pattern of HTTP requests. Figure 2.3 presents an illustrative example using the Bing home page. The load begins with a request for `www.bing.com`, which returns the root HTML document for Bing. As the browser processes the HTML file and builds the DOM tree, it encounters two scripts referenced by the page (`Shared.js` and `PostContent.js`) and makes a parallel request for these resources. After the browser retrieves the scripts, construction of the DOM tree continues, resulting in parallel requests for `h1.png` (the Bing logo) and `PitcherPlants_EN-CA.jpg` (the background image). As these requests are underway, construction of the DOM tree completes (this is the first vertical line in Figure 2.3). Once all of the items needed to display the page arrive and the page is fully rendered, the page load is complete; this occurs at the second vertical line in Figure 2.3.



URL	Size	Timeline
GET www.bing.com	7.3 KB	132ms
GET Shared.js	1.6 KB	50ms
GET PostContent.js	902 B	50ms
GET h1.png	3.8 KB	56ms
GET PitcherPlants_	79.6 KB	213ms
GET l?IG=634350b	42 B	107ms
GET HPImageArchi	2.6 KB	104ms
GET qsonhs.aspx	35 B	80ms
8 requests	95.8 KB	602ms (onload: 440ms)

Figure 2.3: Load timeline for the Bing homepage

The items loaded after the second vertical line in Figure 2.3 do not form part of the web-page proper; rather, they are intended to be loaded after the page finishes its primary load to provide additional, auxiliary features. This type of load is known as an asynchronous load, and is obtained by using JavaScript to modify the DOM tree to include these elements after the page has loaded. In Bing's case, these elements contain tracking and overlay features.

2.3 Minimizing the effects of latency on web browsing

Due to the web's sensitivity to the latency of the link between server and client, techniques have been developed to reduce the latency seen while web browsing. With reference to the mobile environment, we can group these techniques into four major areas: caching, network-side latency reductions, client prefetching, and resource bundling.

2.3.1 Caching

HTTP provides mechanisms for expiration and validation of cached content to facilitate caching [49]. When serving a resource, web servers have the opportunity to place an expiration tag on the response, which informs the cache of the duration for which it can serve the resource without requesting an updated copy from the server. When loading a page, the client browser can skip all network traffic for unexpired resources that are present in its cache (the browser may also opt to display stale resources in some circumstances). Instead of evicting the resource the cache may issue a conditional GET request to the server to inquire whether its cached copy is current. If the server determines that the browser has the most recent version, it sends the HTTP status code `304 Not Modified` [49], and the browser loads the resource from its cache. Otherwise, the server

transmits the new version of the resource along with the status code `200 OK` [49]. This eliminates unnecessary transfers of the resource across the network. Both of these mechanisms have the potential to improve perceived page load performance for previously visited resources, though only the former eliminates network traffic.

Locating a cache on the client can eliminate network latency for cached resources; all major browsers contain caches which save visited resources for a period of time for this reason. A recent study by Yahoo! found that 75–85% of page views on `yahoo.com` used browser-cached items [57], indicating the success of this technique. However, caches can only help once the resources have been previously loaded by the client; for the first load of the web site, they provide no benefit. For this reason, we must look to other techniques for increasing page-load speed.

2.3.2 Client prefetching

Client prefetchers increase the speed of web interactions by prefetching content prior to the user making a request for it. When a user views a page, a prediction engine generates a hint list of URLs the client is likely to request in the near future, along with a confidence score indicating the likelihood of the client accessing each individual URL [13]. The hint list can be generated on the client using the client's page view history [31, 34], on the server using information gathered from all accesses to the page [48, 56], or on a proxy in between the two using a combination of information [5, 15, 26]. In either case, the hint list is passed to a prefetching engine located on the client, which prefetches all resources above a certain confidence score and stores them in a local cache.

If the user decides to follow a link from the page they are viewing to a page the prefetching engine has fetched, the browser can load the page from cache. As the cache is located on the client

device itself and requires no network operations to access, the page loads rapidly. Alternatively, if the user accesses a page that the engine has not prefetched — either because the prediction engine computed too low of a confidence score for prefetching, or the user typed a URL into their browser — the page load occurs as it would normally, and sees no decrease in its load time. Setting the confidence threshold low results in the prefetching engine fetching more pages; this potentially results in a higher reduction in user-perceived latency, as the browser can access prefetched data without accessing the network at all.

Because a low confidence threshold implies more prefetched pages, it also results in a lower prefetch accuracy — pages that are prefetched and not subsequently viewed by the client. The prefetch accuracy is often expressed in terms of an extra data overhead, the number of bytes that the prefetching engine fetches in error for every byte prefetched and used by the client. Typical client prefetchers have extra data overheads in the range of 150% to 300% [31], though many choose not to report this value. Furthermore, the prefetching engine must be careful not to prefetch a page too far in advance of the user requesting it; the page displayed to the user is current as of the time of the prefetch, not at the time of the user access, and therefore may present stale data [5].

When applied to the mobile environment, this technique quickly proves unsuitable for typical web-pages because of the high cost of data transfer over current wide-area wireless networks. It can, however, be used successfully when there is a high expectation of use (pages the user accesses daily, for instance), or when the resources are known to be small (for example, news readers that fetch feed text only). Unlike resource bundling, described next, a client prefetcher must speculatively prefetch — it cannot prefetch only embedded resources. Because it is located on the client, on the client side of the high-latency mobile network segment, if it were to prefetch only embedded resources it would execute essentially in lock step with the browser itself, providing no performance

benefit at all. Since a client prefetcher that does not speculate provides no performance benefit, and speculative prefetchers increase the data transferred over the network, client prefetchers are not an appropriate way to minimize the effects of latency in spontaneous mobile web browsing.

2.3.3 Network-side latency reductions

Significant effort has been invested in reducing the total latency experienced when loading a page; with the exception of the techniques above, most of the focus has been on increasing page load performance by optimizing the network side of the web request's journey from client to server — the portion from mobile-network core onwards. Two major examples of techniques that optimize the network side of the web transaction are caching proxies and content-delivery networks (CDNs).

Similar to browser caches, caching proxies — which sit between the client and the servers it contacts — improve performance by saving server responses; clients can subsequently get the content from the cache rather than contacting the end server. Unlike browser caches, however, caching proxies can benefit multiple users, as all users share the same cache; individual users benefit from the accesses of other subscribers. By locating a caching proxy close to the clients, substantial performance gains can be obtained — effectively, the caching proxy appears to the client as a closer copy of the server for any resources it has cached. Further speed increases can be obtained by allowing the proxy to perform speculative prefetching [5], which works similarly to the client prefetching discussed earlier.

Content-delivery networks go one step further and *actually* relocate the source content close to the clients for which it is intended. At a high level, these networks ensure that people accessing a website access a copy that is close to them; users in Eastern Canada may access a copy in Montreal, while users in Western Canada may access a copy in Vancouver. To provide this service, content-

delivery networks locate large storage servers inside service-provider networks and at major network peering points. Web authors upload their resources to a CDN, and update their pages to reference CDN-provided resources; the content-delivery network's internal services automatically distribute the content to all of the nodes in their network. When a client makes a request for one of these resources, it receives the address for the closest CDN node (by way of anycast DNS), from which it can connect to and download the resource. As a result, the client can load content much faster.

In both of the above cases, however, these latency reductions do little to reduce the impact of the high-latency access segment seen in mobile networks. Typical deployment guidance for caching proxies, for instance, is to place them at the edge of access networks, close to the clients they serve [32]. In wireless wide-area networks, even if we place a caching proxy or content-delivery network node in the mobile-network core (reducing the network-core-to-server latency to effectively zero), the client still must traverse the high-latency access segment in order to reach the content, limiting the performance benefit of content-delivery networks and caching proxies for mobile clients. In fact, any technique that involves sharing at or past the mobile-network core and does not modify the nature of the traditional request-response HTTP interaction (through the use of a client push) is fundamentally not able to overcome this inherent latency to provide faster page-load times to mobile users.

2.3.4 Resource bundling

Instead of speculating on what pages a user will access in the future, we can instead follow the chain of resource references starting at the root HTML file, and deterministically identify resources in the same way that the browser does. If we perform this operation on the server, we can identify

all of the resources the browser will need to load a page prior to the browser itself requesting these resources; if we couple this detection with a push of the associated resources to the client, we can increase the page-load speed without speculation and its associated extra data overhead. Operationally, when a client requests a web-page's HTML file, the server returns the HTML page followed by all embedded resources in the same payload [52]. This allows the browser to receive all content for the page in a single round-trip to the server (excluding the round-trips necessary for TCP). We term this operation resource bundling, because the server is in effect bundling all of the resources needed to display the page into a single package, and streaming that package down to the client.

However, in order to support this mode of operation the server must be modified to use the `multipart/related` content type, be updated with a modified version of HTTP [52], or use another technology in HTTP's place [24]. Modifications to the client are also necessary in order for it to understand bundled responses, except for Internet Explorer and Chrome 14, which understand the `multipart/related` content type natively. Performance improvements are not available when accessing unmodified servers. Furthermore, for a server to bundle resources in this manner, all of the content for the page has to be present on that server. While generally true in the past, this is often not the case with modern web sites, where pages are built with parts from multiple servers — content servers, ad networks, analytics servers, and social networks being common sources of external resources. Because the server hosting the root HTML file cannot package resources from these servers, the reductions in page-load time available with resource bundling are limited.

Alternatively, resource bundling can be performed on a proxy between client and server, providing performance benefits to clients without requiring server modifications. This is the architecture used by the commercial web accelerator product offered by Openwave [44], as well as

work by Dong et. al [14]; specific details of these systems are given in Chapter 5. This is also the architecture we use for our just-in-time prefetching push proxy, discussed next.

Chapter 3

A Just-In-Time Prefetching Push Proxy

In order to overcome the limitations inherent in client prefetchers, network-side latency reductions, and when bundling resources, we investigate a just-in-time push proxy that combines non-speculative prefetching with a push to the requesting client from a proxy located within the mobile-network core, on the network side of the high-latency access link. This approach combines the benefits of client prefetching and resource bundling while avoiding the drawbacks that made those techniques unsuitable for mobile networks. Our system is targeted specifically to improving the end-user perceived performance of the spontaneous web-browsing sessions that form the majority of the data accessed by the device [50]. Our system differs from previous systems from Openwave [44] and Dong et. al [14] in that it supports simultaneous transmission of resources to the client rather than sequential.

Our just-in-time prefetching push proxy system consists of two elements: the mobile client and an in-network proxy located between the 3G network and the servers the mobile client can access (Figure 3.1). More precisely, our proxy sits right after the network's GPRS support nodes,

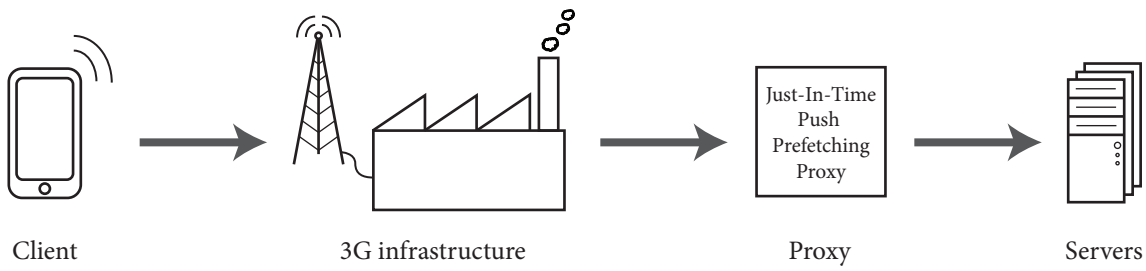


Figure 3.1: Just-in-time push proxy system architecture

which convert mobile data into IP for transmission on the public Internet; this allows us to address all forms of latency present in the 3G network. The client component of the system consists either of a proxy daemon that returns pushed resources to the unmodified system web browser once it makes a request for a resource with a matching URL; the web browser could also be modified to speak directly with the proxy.

When a user wishes to access a page, their mobile device sends the page request to the server that is intercepted by the proxy. The proxy performs the request on behalf of the client, and when the server sends its response, the proxy forwards the content on to the client. At the time of the server response, the proxy also scans the response for references to embedded content, and independently creates requests to the server for that content. Once the proxy fetches the independently requested content from the server, it forwards the response to the client, which uses the information when rendering the page. By prefetching only embedded content, our proxy avoids requesting content the client will never need; all content embedded in a page is necessary to completely display it on the client. Figure 3.2 shows this sequence of events.

If the client encounters a URL for which it does not have a matching resource in its cache — when the user browses to a new web site, for instance — the client will request the resource from the server. The client also makes a request to the server if our proxy fails to identify an embedded

resource — resources that are called by Javascript, for instance. These requests are intercepted by the proxy, which dispatches the request to the server, and returns the response to the client when it arrives. Depending on the precise sequence of events, this may also occur when the proxy has prefetched the content from the server but it has not yet been fully pushed to the client; in this case, the proxy drops the request it receives, and the client uses the pushed content when it arrives (the dashed arrow in Figure 3.2).

We can compare the actions of our just-in-time push proxy to a simple prefetching proxy that prefetches embedded resources but does not push them to the client. As described in Chapter 2, without the push to the client this proxy is located on the wrong side of the high-latency segment. This difference occurs at the final *forward* step shown in Figure 3.2. Rather than forwarding the response to the client immediately upon receiving it from the server as in the diagram, a simple prefetching proxy would have to wait until the request from the client arrived at the proxy before forwarding it on (this request is represented by a dashed line in Figure 3.2). Though this is faster than requesting the resource from the server, the delay while waiting for the client request on the proxy results in a slower page-load time compared to our push proxy.

3.1 Performance analysis

To understand how our system performs, let us begin by considering how our push proxy differs from the scenario where no proxy is present, as well as the scenario where a prefetching proxy is present but that proxy does not push the prefetch results to the client. For both types of proxy (push and non-push), we assume that they are located in the mobile-network core, placing them as close (in terms of latency) to the client as technologically possible. In particular, by locating the

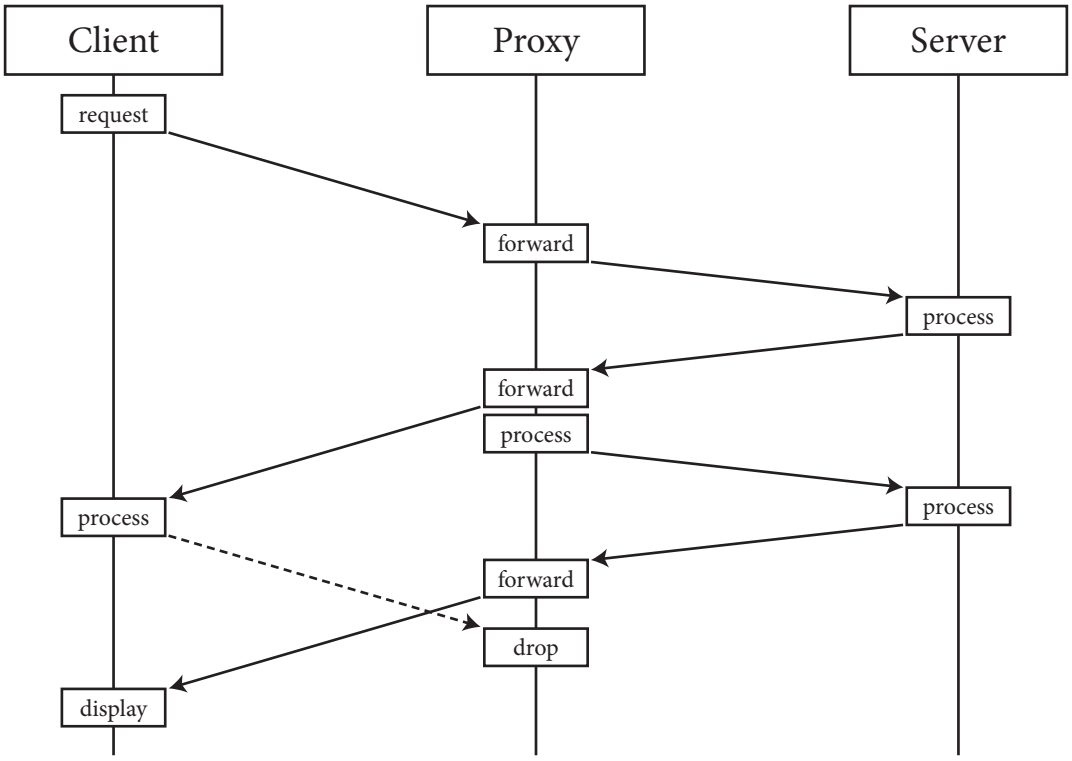


Figure 3.2: Sequence diagram for a just-in-time prefetch

proxy here we assume that the proxy is on the path from the mobile client to any server it wishes to access; this means that there is additional delay to transit the proxy, but no additional network latency.

3.1.1 Intuition

Intuitively, the time it takes for the client to retrieve the root HTML file describing a web-page should be the same whether or not we have a proxy (excluding processing time and queuing delay on the proxy), and whether or not the proxy pushes data to the client; in all cases, we must wait for the client to make a request for a page. It is only when the response to this request passes through the proxy that the proxy has the opportunity to predict what resources the client will request next, and take action on its predictions prior to the client making those requests.

We expect a prefetching proxy that does not push to the client to be faster than no proxy (again, setting aside processing time and queuing delay) as it sees the server's responses sooner than the client, and can therefore begin fetching embedded resources sooner. When the client determines that it requires an embedded resource, it makes its request to the server; this request is intercepted by the proxy and upon which it catches up with the proxy request already in progress. From the client's perspective, this appears as if the server moved closer to the client, as the proxy responds on the server's behalf for all requests after the first.

For us to see a benefit from a push proxy, we must find ourselves in the situation where the proxy can prefetch a resource prior to the client request reaching the proxy. If this were not the case, then a request from the client would already be waiting on the proxy, and using a separate mechanism to push the content to the client would provide no benefit. If we do find ourselves in the first case, then we expect to see performance proportional to the distance between proxy and

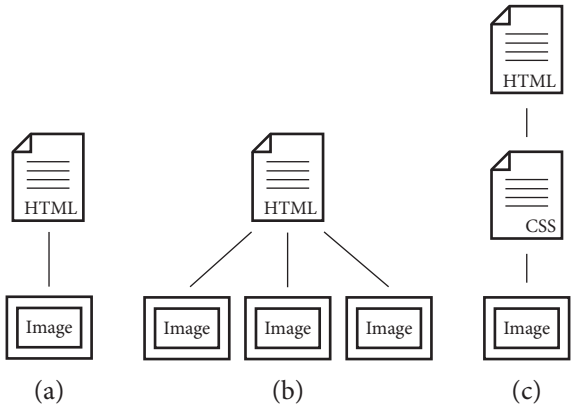


Figure 3.3: Page reference structures analyzed and tested

server. Considering the high-latency first hop in mobile networks and that the majority of network activity on a mobile device is web traffic [50], we believe that mobile users spend most of their time accessing servers for which push proxies are beneficial.

In the following sections, we examine three different page structures as diagrammed in Figure 3.3. The first structure consists of a page loading a single resource (Figure 3.3a). The second consists of multiple resources embedded in the root HTML (Figure 3.3b); in this case, the browser will open connections in parallel in order to download all resources. The final structure consists of a root HTML page that references a CSS file, which itself references an image (Figure 3.3c). In this case, because the browser does not know the next resource to load prior to loading the previous one, it cannot parallelize its requests. All three scenarios test resource arrangements seen on the web.

3.1.2 Loading the root HTML file

Based on our intuition in the previous section, we expect the time it takes to load the root HTML file to be equal to the time it takes for the client to make a request to the server and receive a response, passing through the proxy when present. This process is diagrammed in Figure 3.4.

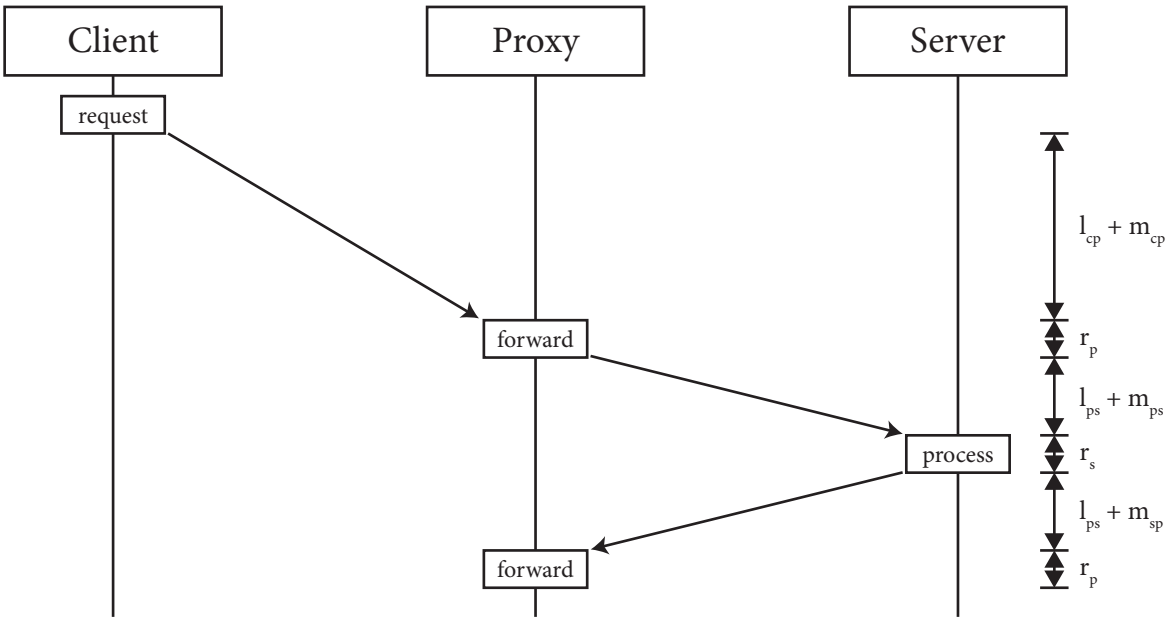


Figure 3.4: Sequence diagram for t_{root}

Here, we define the variable l to represent the latency between two positions in the network, and the variable m to represent the transmission time of the request and response. Processing time and queuing delays are represented by the variable r. Subscript pairs indicate participants and directionality. Therefore, the subscript cp models a quantity starting at the client and ending at the proxy, while a similar quantity between proxy and client would have the subscript pc.

Following the request as it travels to the server, the request must first be transmitted from the client to the proxy (or the position where the proxy would be if present, in the no-proxy case). This

quantity is formed from the latency of the path between client and proxy, l_{cp}, and the transmission time of the request over the client-proxy network, m_{cp}.

Next, we encounter some processing time and queuing delay at the proxy, represented by r_p. This quantity is equal to 0 in the no-proxy case, as no proxy is present to impose this additional latency. Regardless of the value of r_p, the request subsequently travels to the server, which takes $l_{ps} + m_{ps}$ — the latency of the path between the proxy position and the server, plus the transmission time of the request over the proxy-server network.

Upon reaching the server, the server locates the requested content and prepares a response, a process taking r_s. This response is sent to the client, taking $l_{sp} + m_{sp}$ to reach the position of the proxy. If the proxy is present, additional processing time of r_p is also encountered at this time.

It is at this point that the behaviour of the three cases begin to diverge beyond simple differences in processing time. For clarity, we represent this period of common behaviour in the quantity t_{root}, presented in Equation 3.1.

$$t_{root} = \left(l_{cp} + m_{cp|HTML}\right) + \left(2l_{ps} + m_{ps|HTML} + m_{sp|HTML}\right) + 2r_p + r_s \tag{3.1}$$

The transmission times in Equation 3.1 have been extended with the subscript $HTML$ to indicate that the quantity being discussed here is the root HTML file. In future sections, the subscript *resource* will be used to differentiate the transmission times for resources from the transmission times for the root HTML file

Equation 3.1 also makes some other assumptions about the network configuration in order to simplify this discussion. We assume that the latencies in the system are equal, so that $l_{ps} = l_{sp}$. Network bandwidths are not assumed to be equal. By placing our proxy in the mobile network

core, we assume that there is no additional path latency to travel through the proxy.

3.1.3 Loading a page with 1 embedded resource

To determine the load time for a page with a single resource, we must pick up where we left off in Equation 3.1 and determine the time $t_{resources}$ each configuration takes to load all of the resources embedded in the root HTML file. We start by examining the load pattern of a HTML file referencing 1 resource; Listing 3.1 presents an example of such a page.

Listing 3.1: HTML page with 1 resource

```
<html>
    <head></head>
    <body>
        <img src='X.png'/>
    </body>
</html>
```

We start our analysis of $t_{resources}$ at the proxy. With no proxy, the 'proxy' is just an arbitrary position in the network, and the response must continue all the way to the client before the browser can process the HTML file and generate a request for the embedded resource. Equation 3.3 describes this case; the quantity r_c represents the time it takes the client to process the received HTML file and generate a request for the embedded resource.

$$t_{resources|no} = \left(3l_{cp} + m_{pc|HTML} + m_{cp|resource} + m_{pc|resource} + 2r_c\right) + \left(2l_{ps} + m_{ps|resource} + m_{sp|resource} + r_s\right) \tag{3.2}$$

For both proxies, the proxy forwards the request on to the client while scanning it for embedded resources (see Figure 3.2). When the proxy encounters the embedded image reference, it generates

a request and sends it off to the server. The time taken for this scanning is represented by the parameter r_i. Once the proxy receives a response from the server, it is either returned directly to the client (if a request from the client has arrived in the interim), held on the server until the client request arrives, or proactively pushed to the client. Equation 3.4 details the time it takes to load the embedded resource with a prefetching proxy, while Equation 3.5 shows the time with our push proxy. Adding these equations to t_{root}, derived in the previous section, produces the total page-load time t_{load}.

$$t_{resources|no} = \left(3l_{cp} + m_{pc|HTML} + m_{cp|resource} + m_{pc|resource} + 2r_c\right) + \left(2l_{ps} + m_{ps|resource} + m_{sp|resource} + r_s\right) \quad (3.3)$$

$$t_{resources|pre} = \left(3l_{cp} + m_{pc|HTML} + m_{cp|resource} + m_{pc|resource} + 2r_c + r_p\right) + \max\left(r_p, \left(2l_{ps} + m_{ps|resource} + m_{sp|resource} + r_i + r_s\right) - \left(2l_{cp} + m_{pc|HTML} + m_{cp|resource} + r_c\right)\right) \quad (3.4)$$

$$t_{resources|push} = \left(2l_{ps} + m_{ps|resource} + m_{sp|resource}\right) + r_i + r_s + r_p + 2r_c + \left(l_{cp} + m_{pc|resource}\right) \quad (3.5)$$

Graphing these equations with a l_{ps} of 30 ms produces Figure 3.5. For the graphs displayed in this section, processing times r_s, r_p, r_c, and r_i were set to 0.2 ms, 2.2 ms, 4.6 ms, and 0 ms, respectively. These values were obtained experimentally (see Table 4.3). Transmission times m were calculated based on a 10 Mbit client-proxy link and a 100 Mbit proxy-server link, with a root HTML page of 56 bytes and resources 600 bytes in size. All values and are for illustration only; the general patterns we observe here are independent of the precise values of these parameters, as they do not affect the slope (they do not change with l_{cp}).

We see that the slope for our push proxy is half that of the no-proxy and prefetching proxy. For each additional millisecond added to the client-to-proxy latency, the no-proxy load time encounters

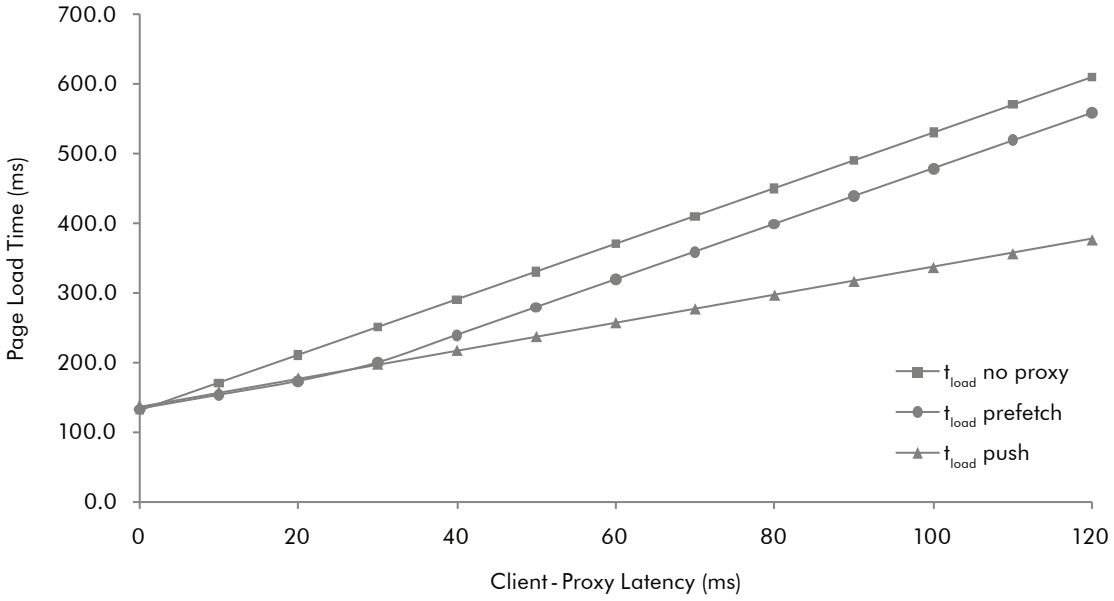


Figure 3.5: Total load time for page with 1 embedded resource

an extra millisecond of load time both when making a request to the server (an outbound traversal of the client-proxy network), and when it receives a response from the server (an inbound traversal). It does this twice, once to load the root HTML file and once to load the embedded resource. For the push proxy, the page load encounters an extra millisecond of load time per millisecond of additional client-to-proxy latency in its initial request and when the server transmits the resource back to the client; the proxy begins streaming the embedded resource back to the client in the same amount of time in all cases as the proxy-server latency does not change. We do not see the extra millisecond added to the response containing the HTML file, as this response is not on the critical path for the total page-load time.

The kink in the graph of the prefetching proxy load time occurs at l_{cp} = 30 ms. Before this point, the client's request for the embedded resource is waiting on the proxy by the time the prefetching proxy finishes retrieving the resource, making the prefetching proxy operate similarly to the push

proxy. After this point, the prefetching proxy must wait for a client request in order to return the resource it prefetched.

This results suggest that our push proxy can have significant benefits in the mobile environment — adding a prefetching proxy improves page-load time performance by moving resources closer to the client, but bears the same slope as no proxy. A push proxy, on the other hand, has a fundamental advantage on both other configurations, and becomes a better solution as the imbalance between the client-proxy and proxy-server latencies grows.

3.1.4 Loading a page with multiple embedded resources

In practice, web-pages rarely contain a single image; rather, they are formed with multiple images, all of which the client browser must request from the server. Listing 3.2 contains an example of such a page, which loads 3 images.

Listing 3.2: HTML page with multiple embedded resources

```
<html>
    <head></head>
    <body>
        <img src='X.png'/>
        <img src='Y.png'/>
        <img src='Z.png'/>
    </body>
</html>
```

When loading pages containing multiple embedded resources, web browsers typically create multiple TCP connections to the server so that they can download resources in parallel. This results in a series of overlapping request-response pairs; the HTTP specification also allows pipelining multiple requests in a single connection for a similar overlap, though browser support for this

feature is limited. Analytically, this load has a similar shape to the load pattern seen when loading a single resource — with no proxy, we see the same end-to-end load time as we did previously, while with both proxies we see the requests spawned earlier on the proxy and the response either returned directly to the client, held on the server until the client request arrives, or proactively pushed to the client.

The difference in page-load time between loading a single resource and loading multiple resources is the time it takes to transmit the extra resources, as we can see in Equations 3.6 – 3.8.

$$t_{resources|no} = \left(3l_{cp} + m_{pc|HTML} + m_{cp|resource} + m_{pc|resource} + 2r_c\right) + \left(2l_{ps} + m_{ps|resource} + m_{sp|resource} + r_s\right) + (n-1)\max\left(m_{pc|resource}, m_{sp|resource}\right) \quad (3.6)$$

$$t_{resources|pre} = \left(3l_{cp} + m_{pc|HTML} + m_{cp|resource} + m_{pc|resource} + r_p + 2r_c\right) + \max\left(r_p, \left(2l_{ps} + m_{ps|resource} + m_{sp|resource} + r_i + r_s\right) - \left(2l_{cp} + m_{pc|HTML} + m_{cp|resource} + r_c\right)\right) + (n-1)\max\left(m_{pc|resource}, m_{sp|resource}\right) \quad (3.7)$$

$$t_{resources|push} = \left(2l_{ps} + m_{ps|resource} + m_{sp|resource} + r_i + r_s\right) + \left(l_{cp} + m_{pc|resource} + r_p + 2r_c\right) + (n-1)\max\left(m_{pc|resource}, m_{sp|resource}\right) \quad (3.8)$$

In general, we expect the bandwidth of the client-to-proxy network to be lower than the proxy to server bandwidth, and therefore, the requests will be spaced apart by m_{pc}. If the opposite is true, the requests will be spaced by m_{ps}.

Graphing these equations with the parameters as before and 3 embedded resources produces Figure 3.6. Here, we see the same pattern emerge as we saw with the single resource scenario; the slope for our push proxy is half that of the no-proxy and prefetching proxy. In this instance, the total load time is slightly higher due to the increased transmission time for the extra resources.

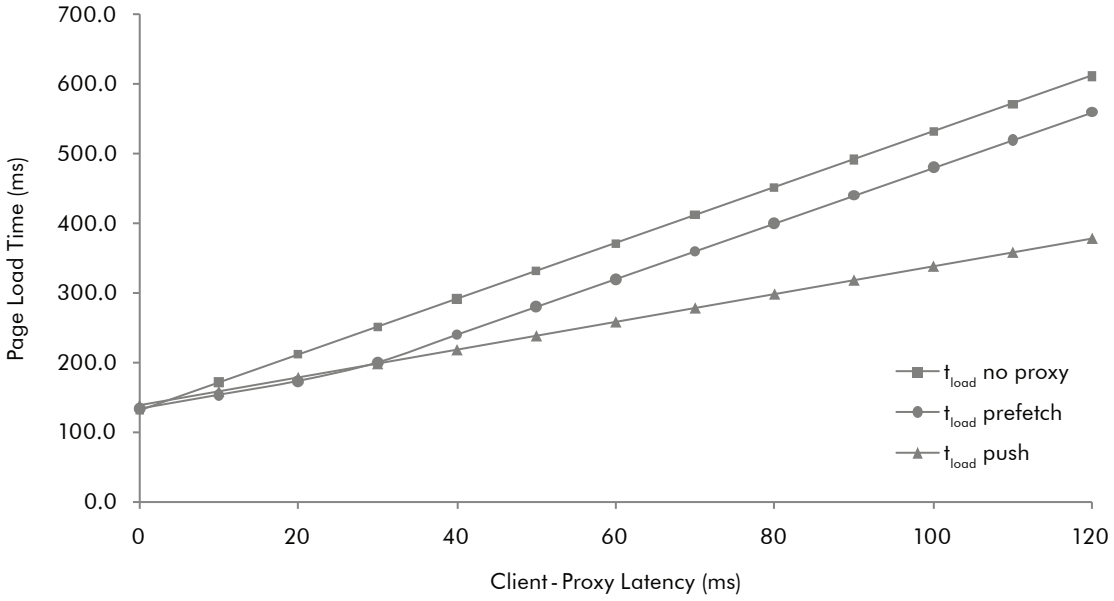


Figure 3.6: Total load time for page with 3 embedded resources

3.1.5 Loading a page with nested resources

An alternate resource arrangement occurs when the root HTML page loads resources that themselves load further resources, as is often the case. Listings 3.3 and 3.4 present an example of this structure. Unlike images, CSS files have the ability to load additional resources, be they images or other CSS files. We define depth d as the length of the longest chain of nested resources not counting the root HTML page (see Figure 3.7). Therefore, a HTML page with no resources has a depth of 0, as the browser does not have to perform additional requests to obtain resources. Unlike the previous scenario where the browser could overlap requests to the server to improve performance, because the browser cannot identify all necessary resources by loading only the root HTML file, each level of depth incurs an extra round-trip to the server.

The behaviour of the client browser in the no-proxy case is straightforward; after receiving the root HTML file, it makes a request for the embedded resource. Once the server responds with

Listing 3.3: HTML page with external CSS

```
<html>
    <head>
        <link  rel='stylesheet' href='level2.css'/>
    </head>
    <body></body>
</html>
```

Listing 3.4: CSS resource file `level2.css` importing additional CSS

```
.level2
{
    background-image: url('Y.png');
}
```

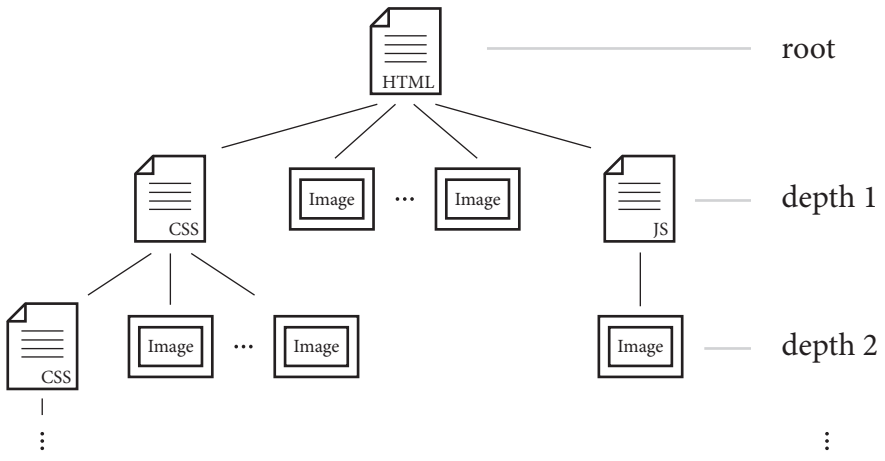


Figure 3.7: Reference structure for a page with nested resources

this resource, the browser parses it to identify any resources it references, and makes requests for those resources. This cycle repeats until the browser has identified and requested all resources. Equation 3.10 describes this case.

$$t_{resources|no} = \left(l_{cp} + m_{pc|HTML} + r_c\right) + d\left(\left(2l_{cp} + m_{cp|resource} + m_{pc|resource}\right) + \left(2l_{ps} + m_{ps|resource} + m_{sp|resource} + r_s\right) + r_c\right) \tag{3.9}$$

For both of the proxies, once the proxy receives a response from the server, it is either returned directly to the client, held on the proxy until the client request arrives, or proactively pushed to the client as in the previous scenarios. Instead of scanning only the HTML file on its way from the server to the client, the proxies also scan retrieved CSS or HTML files and generate requests for any resources referenced by those files. Equation 3.11 details the time it takes to load the embedded resource with the prefetching proxy, while Equation 3.12 shows the time with the push proxy.

$$t_{resources|no} = \left(l_{cp} + m_{pc|HTML} + r_c\right) + d\left(\left(2l_{cp} + m_{cp|resource} + m_{pc|resource}\right) + \left(2l_{ps} + m_{ps|resource} + m_{sp|resource} + r_s\right) + r_c\right) \tag{3.10}$$

$$t_{resources|pre} = \left(l_{cp} + m_{pc|HTML} + r_p + r_c\right) + d\left(2l_{cp} + m_{cp|resource} + m_{pc|resource} + r_c + \max\left(r_p, \left(\left(2l_{ps} + m_{ps|resource} + m_{sp|resource} + r_i + r_s\right) - \left(2l_{cp} + m_{pc|HTML} + m_{cp|resource} + r_c\right)\right)\right)\right) \tag{3.11}$$

$$t_{resources|push} = d\left(2l_{ps} + m_{ps|resource} + m_{sp|resource} + r_i + r_s + r_c\right) + \left(l_{cp} + m_{pc|resource} + r_p + r_c\right) \tag{3.12}$$

Graphing these equations with the parameters as before and a depth of 2 produces Figure 3.8.

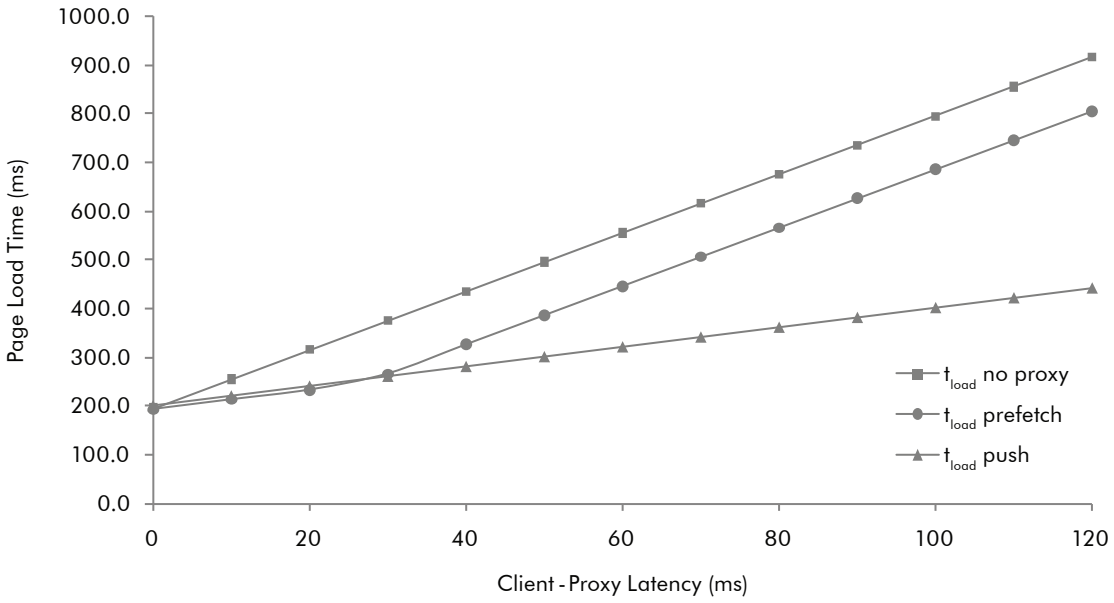


Figure 3.8: Total load time for page with embedded resources to depth 2

Again we see the general shape we saw in the previous situations — a linear increase in the no-proxy load time with a shallower increase seen on the push proxy. In this case, however, we see a larger difference between the slopes of the no-proxy and prefetching proxy configurations and the push proxy, as well as a larger gap between the parallel no-proxy and prefetching proxy lines.

As in the previous cases, the push proxy line has a slope of 2 ms/ms. The no-proxy and push proxy slopes, however, see an increase to 6 ms/ms. When resources load other resources, another traversal of the client-proxy network is added. Since each extra layer in depth adds an extra trip between client and proxy for the no-proxy and prefetching proxy configurations, we expect to see that the slope of these lines is proportional to the depth; in our scenario, $\Delta t_{load|no} = 2 + 2d$ ms/ms.

The improvement that the prefetching proxy sees over no proxy is also explained by the change in slope. Since the prefetching proxy and the push proxy have similar behaviour up to $l_{cp} = 30$ ms, the gap between these lines is the difference in the rises of the no-proxy and push proxy lines

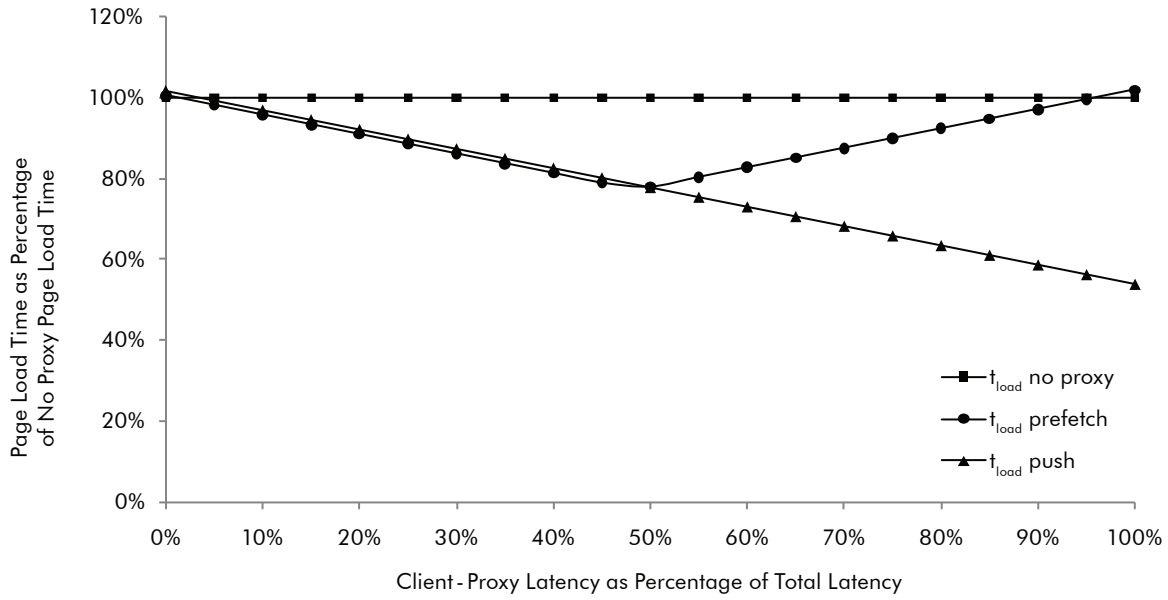


Figure 3.9: Page-load time for 1 resource, as ratio

between l_{cp} = 0 ms and l_{cp} = 30 ms. With an increase in the slope of the no-proxy line, we see an increase in the rise over this period and thus a larger performance improvement by the prefetching proxy.

3.2 Observations

From our analysis, we can see that just-in-time push proxies have the potential to improve the perceived performance of the web on mobile devices. In particular, their insensitivity to latency increases between client and proxy make them particularly suitable for mobile scenarios, where the high-latency first segment dominates the latency of the entire client-server path. Furthermore, push proxies maintain these benefits even when pages are structured similarly to how they are on the modern web, with multiple resources at different depths.

We can make some additional observations if we re-graph the preceding equations by expressing the client-proxy latency as a percentage of total latency, as Figure 3.9 shows. By holding the total end-to-end latency constant, this graph shows how the positioning of each proxy affects the magnitude of the benefit it provides. As the client-proxy and proxy-server latencies are an arbitrary distinction when no proxy is present, we see a constant load time for that configuration, and normalize the other configurations with respect to it.

Of primary note is the peak that occurs when the client-proxy and proxy-server latencies are equal; in the previous graphs, this occurred at an l_{cp} of 30 ms. To the left of this point, both proxies have identical performance, as we noted previously. To the right, the push proxy's performance continues to improve, while the benefits provided by the prefetching proxy begin to diminish. At both edges of the graph, the prefetching proxy takes more time to load a page than no proxy due to the extra time required to transit the proxy overwhelming the benefits provided by prefetching. The point at which prefetching is equal to no proxy is not at the very edges of the graph, but rather at the point where the proxy can improve load time enough to balance the time lost passing through the proxy. Since the push proxy does not need to wait for a request from the client to return prefetched resources, the push proxy exhibits this behaviour on the left edge of the graph only.

Therefore, we note the interesting difference that exists between the deployment guidance for a push proxy versus a typical caching proxy; while caching proxies should be placed as close as possible to the client in order to reduce the distance the content must travel [2], push proxies provide bigger benefit when they are placed as close to the server as possible. Deployment guidance for prefetching proxies that do not push content to an attached client is different still; it should be placed at the midpoint between client and server for optimum performance. While placing a prefetching proxy closer to the client maximizes the proxy-server distance, increasing the

performance difference over no proxy, it also reduces the lead time the proxy has for prefetching, resulting in more waiting by the client. Since the servers accessed by a client are not at a uniform distance from the client, this performance characteristic makes prefetching proxies difficult to place in practice.

Chapter 4

Evaluation

In order to see whether the potential performance benefits identified in the previous chapter exist in the real world, we have implemented our just-in-time prefetching push proxy. We feel this is a better solution than simulations or trace-based evaluation, as an actual implementation allows us to accurately capture the complex behaviour of modern browsers. Furthermore, we can use our implementation with 3G networks as deployed today, allowing us to evaluate the performance of our system in combination with any traffic management and optimization systems currently deployed by service providers. Due to this approach, however, our results are intrinsically dependent on the quality of our implementation; programming errors, operating system issues, and other network effects may cloud our results — all issues simulations avoid.

Though we used a desktop computer and desktop browser to run these tests, we believe that this setup is a reasonable approximation to the conditions encountered on the mobile web. First, 3G USB data sticks provide users with a way to access the mobile web from a portable computer using a desktop browser; our tests thus accurately represent this case. Second, smartphones and other

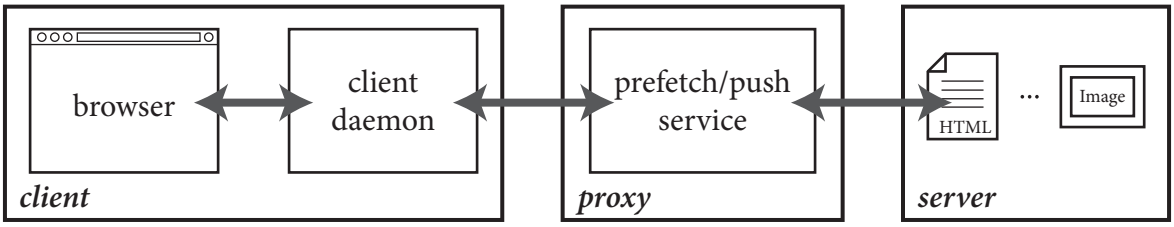


Figure 4.1: Implemented architecture of our just-in-time prefetching push proxy

devices with embedded mobile network cards typically use the rendering engines from desktop browsers; for example, the WebKit engine used in the Google Chrome desktop browser used for testing is also used in the Chrome browser for Android as well as Mobile Safari on iOS. We expect both desktop and mobile versions of the browser to operate similarly for this reason. Finally, as smartphones become more powerful, we expect the browsers on these devices to become more similar to their desktop cousins.

4.1 Proxy implementation

The client portion of our architecture was split into two components, the web browser and a client daemon, in our sample implementation. Figure 4.1 presents this architecture. By splitting the client in this manner, we were able to use an unmodified client browser. We used Google Chrome for testing; we also verified with other browsers during development. The browser was connected to the client daemon by setting the browser's HTTP proxy to our client daemon, which forced the browser to make all of its requests through our client daemon. The client daemon interacted with our proxy as described in the previous chapter.

We implemented both the client daemon and the prefetching push proxy using node.js [29] (version 0.2.5), an evented asynchronous network programming environment running under the

V8 JavaScript engine [59]. Though this environment would not be well suited to a commercial implementation of our push proxy, our testing found that node.js was sufficient for understanding the behaviour of our architecture with a single client. The client daemon connected to the network proxy over persistent TCP connections; each client daemon established a configurable number of connections to the proxy upon startup. We tested the proxy in two modes: *prefetch*, in which the proxy scanned all HTML and CSS documents for referenced resources and prefetched these resources to the proxy, and *push*, in which the proxy operated as in the prefetch configuration while also pushing prefetched resources to the client over one of the persistent TCP connections. The *push* mode realizes our just-in-time prefetching push proxy.

4.1.1 Proxy operation

When a user enters a web address into their browser, the browser forwards this request to the client daemon. The client daemon checks its pool of pushed resources to see if a resource with a matching URL is present; if it is, it begins streaming the data back to the browser. If the resource it is streaming is incomplete, it streams all available data and then streams the remaining data as it arrives. If no matching resource is present, the client forwards the request to the in-network proxy, and waits for a response. Once the proxy begins streaming data, the daemon forwards the data to the browser using the waiting connection. Once the daemon has streamed a complete request to the browser, the daemon purges the resource from its list of pushed resources. The client daemon also periodically scans its list of pushed resources looking for items that have not seen activity in the previous 300 seconds; these items are the result of erroneous predictions and are discarded by the daemon to keep memory usage low.

On the proxy, the service waits for incoming client requests. When a request arrives, the proxy

service checks its list of in-progress and completed requests to see whether there is a resource with a matching URL present for that client; if there is, it streams the data back to the client daemon. If no matching resource is present, the proxy requests the resource from the server. Like a client browser, the proxy uses HTTP 1.1 persistent connections to the servers it connects to, allowing it to eliminate TCP handshakes with servers it has connected to recently. We have limited the number of connections our proxy makes to any individual server to 6, matching the limit set in Google Chrome [6]. When a server responds to a proxy request, the proxy checks to see if it has a waiting client request; if it does, it forwards the data immediately. If no request is waiting, it caches the data (in prefetch mode) or pushes it to the client via its push channel (in push mode).

Whenever a HTML or CSS file passes through the proxy (and the proxy is in prefetch or push mode), the proxy also scans the resource to identify embedded resources. When the proxy identifies an embedded resource, it is placed on the proxy's prefetch queue and prefetched when sufficient information is available. Our implementation is able to process both uncompressed and compressed HTML responses, and identifies referenced scripts, images, i-frames, and style sheets and icons using the `<link/>` HTML tag. It also processes uncompressed and compressed CSS responses, identifying resources using the CSS `url()` syntax.

HTTP request headers

In order to maintain compatibility with existing browsers and web servers, the proxy generates the HTTP headers it sends using the HTTP headers sent by the client. As some web sites tailor content to the identity of the browser accessing the content, sending the client's header is important to ensure that the proxy receives the correct content. This practice sees widespread usage in the mobile web, where browsing to a site's homepage will often redirect the user to a mobile-optimized

Table 4.1: Extra resource fetches, before and after filtering unused CSS styles

	Extra Resources (Before)	Data Overhead (Before)	Extra Resources (After)	Data Overhead (After)
Google	0		0	
Amazon	66	+51%	40	+37%
The Toronto Star	198	+31%	53	+9.0%
New York Times	32	+2.0%	10	+1.3%
Google	2	+0.2%	2	+0.2%
Amazon	5	+6.8%	1	+0.3%
The Toronto Star	3	+0.3%	1	+0.2%
New York Times	28	+28%	0	

site by checking the `User-Agent` header. This header is also used to present appropriate formats of media elements to mobile clients.

Unused CSS styles

Initial testing of our implementation saw the prefetcher requesting a large number of resources that were not subsequently used by the client. We found that web authors typically create a single root CSS file for an entire web site, and reference this file from all pages on the site; this ensures that all portions of the web site look identical, but it also results in some of the styles in a CSS file going unused on any given page. As a result of our prefetcher not knowing which styles were used in the page, we were requesting all possible URLs. To resolve this issue, we amended our proxy to log all style identifiers it encounters during the parse of the root HTML file; the CSS parser checks this log when scanning and prefetches only resources for used styles.

Table 4.1 presents the effect of loading all resources referenced by a CSS file versus loading resources only for styles used by the page. In most cases, filtering out unused styles reduces the

number of extra resources fetched, reducing the volume of data fetched in error by anywhere from a few percent to an order of magnitude.

Cookie handling

A key aspect in the implementation of our proxy was the correct handling of cookies. Existing prefetcher implementations often ignore cookies, and prefetch content without providing a cookie; however, without sending user cookies to the server the response received by the prefetcher may not be correct. The need to pass along the client cookies effectively blocks prefetching attempts to domains with cookies, as the prefetcher must wait for the client's request to obtain the cookies needed to make its own request; this eliminates the gain available from just-in-time prefetching. Prefetchers integrated with the client browser do not suffer from this problem, as they can access user cookies directly.

Past studies have shown that roughly 30% of requests contain cookies [7], which could hamper the effectiveness of our prefetching scheme. To work around this problem, our proxy caches the cookies sent by the client browser, and re-uses these cookies for a short period of time on subsequent prefetch requests. When the prefetcher queues a prefetch request for a domain the client has not requested recently, the prefetch blocks until a client request for that domain arrives (containing a fresh cookie). The arrival of a client request for a domain unblocks all prefetch requests waiting on that domain. This allows us to provide the server with the correct cookie, while still performing prefetching in advance of the client.

Our implementation also immediately proceeds without cookies for requests to static (cookie-less) domains. These domains are used by web developers to improve page-load speeds [60]. As the HTTP protocol requires the browser to upload all cookies for a domain on every request, hosting

Table 4.2: Effects of immediate requests for cookieless domains

	Waiting Requests	Average Load Time	Waiting Requests (immediate)	Average Load Time (immediate)
Amazon	10	2.4 s	0	2.3 s
The Toronto Star	85	9.0 s	5	9.5 s
New York Time	58	4.8 s	15	4.7 s

static content on a separate domain that never sets cookies can reduce the request size without affecting the operation of the primary domain. Table 4.2 presents a comparison of the number of requests that must wait on the server for a client request containing the appropriate cookies.

The results presented in Table 4.2 were obtained using the same 3G network used for our live page tests; full details of the experiment setup are given in Section 4.5.3. Counterintuitively, our results show no measurable benefit as a result of this optimization, despite a significant reduction in the number of requests that must wait on the proxy. Though the number of waiting requests is reduced, a corresponding reduction in total load time is not present, indicating that the requests do not lie on the page's critical path. This optimization has no effect on pages that load from a single domain (like the Google home page), as the cookies for the initial request can be used for all remaining requests. Though this optimization provides no clear benefit, we have left this optimization enabled for the tests in this chapter.

Request spacing

During testing, we identified a bug in the node.js HTTP library that resulted in intermittent data loss and delays when we made multiple HTTP requests in a row. This occurred in particular during HTML and CSS parsing in prefetch and push modes, when the proxy identified multiple resources

to prefetch and dispatched them to the server. Though this bug was intermittent — not all batched requests exhibited uncharacteristic slowdowns — it was reproducible and occurred in around 25% of cases where our proxy made multiple requests.

We attempted to fix the bug, but due to its complexity and intermittent nature were not able to identify its root cause. During our investigation, we did discover a workaround that allowed our testing to continue until a fix to node.js is available (our tests were run against node.js version 0.2.5). Our workaround spaces server requests apart by at least 1 ms apart by using the JavaScript `setTimeout()` function; this results in a minor increase to the proxy processing time, but eliminated the problem of intermittent server request delays.

4.2 Evaluation framework

As we are concerned with the *perceived* performance improvements our architecture provides, we can guide the evaluation of our push proxy using the response time limits developed through decades of research in human factors [9, 41, 42]:

1. **100 milliseconds** is about the limit for a user to feel that an action completed instantaneously.
2. **1 second** is about the limit for a user's task to remain uninterrupted; that is, the action and its response happen as part of a single event
3. **10 seconds** is about the limit for maintaining a user's focus on a single task; any longer, and they will attempt to perform other tasks while waiting (browsing in other tabs, getting a coffee, or abandoning the website altogether)

On the web, these periods of time begin when the user initiates a navigation to a new page — by following a link, choosing a bookmark, or typing in the address bar — and end when the requested page has finished loading. As discussed in Chapter 2, this occurs at the time the browser fires the `load` (`onload`) event. We used a browser extension inside Google Chrome to measure this period; this allowed us to capture the total page-load time, including non-network aspects like rendering.

4.3 Architecture validation

Our first set of tests validates the analysis presented in Chapter 3, and allows us to compare the theoretical gains predicted there to the actual gain seen with a real implementation. We ran all tests using Google Chrome 8.0.552.224 on Ubuntu Linux 10.04 x64; the client machine contained 3 GB of RAM and a 2.7 GHz Intel Core i3 E4500 CPU. During testing, we disabled Chrome's disk cache to ensure that it loaded all resources from the network. We also configured the server to indicate all pages and resources were uncacheable; this prevented Chrome from caching resources in its in-memory cache between page loads.

We constructed our validation system by placing our client machine in a disconnected Ethernet network consisting of the client machine, a proxy machine, a web server, and a router. Figure 4.2 diagrams our setup. Both the client and proxy machines ran our just-in-time prefetching proxy system; the server was an unmodified Apache web server. We used the netem Linux kernel module to vary the client-proxy and proxy-server delays in our network; we set the proxy-server delay 30 ms for all tests, while varying the client-proxy delay from 0 ms to 200 ms. We tested all of the configurations analyzed in Chapter 3 — no proxy, our prefetching proxy set to prefetch mode, and

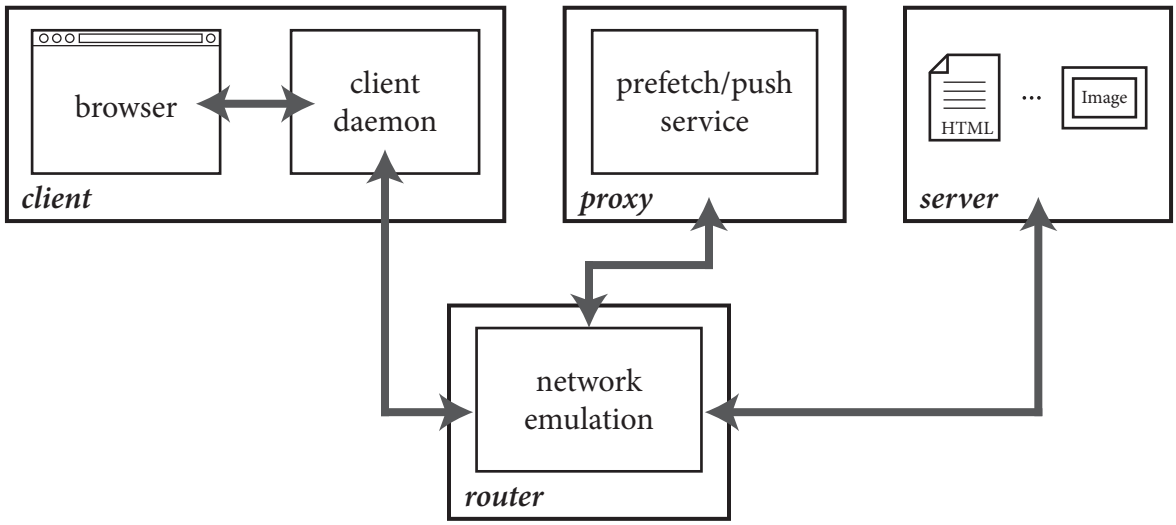


Figure 4.2: Architecture of the validation test system

our prefetching proxy set to push mode. Test runs consisted of 40 runs at each client-proxy delay.

We validated our implementation in three scenarios: a HTML page with 1 embedded image, a HTML page with 3 embedded images, and a HTML page with an image embedded at a depth of 2 (2 total embedded resources). In the first scenario, the HTML page was 56 bytes in size, and the embedded resource was an image 664 bytes in size. The second scenario had an HTML page of size 113 bytes, with the 3 embedded images averaging 609 bytes in size. For the last scenario, the HTML page was 119 bytes in size, referencing a CSS file 67 bytes in size, which referenced an image 247 bytes in size.

4.3.1 Processing time

We started our validation tests by measuring r_p, r_s, and r_c in our validation system; feeding these values into our analytical model allows us to compare our implementation with our expected results. To ensure we captured all components of each parameter, including processing and queuing delays,

Table 4.3: Validation environment processing times

Parameter	Value
r_s	0.2 ms
$r_{p\|up}$	3 ms
$r_{p\|down}$	1.5 ms
r_c	4.6 ms
r_i	0 ms

we used the Linux system utility `tcpdump` to monitor the packets entering and exiting the client (r_c), proxy (r_p), and server (r_s) machines, and measured the difference between entering and exiting packets during the load of our one resource validation page. Table 4.3 summarizes our measurements.

We measured r_p in two directions, both the time it takes the proxy to dispatch a request to the server after receiving a request from the client, and the time it takes to return a response from the server to the appropriate client. These are denoted as $r_{p|up}$ and $r_{p|down}$ (respectively) in Table 4.3. As our analytical model does not differentiate these quantities, we average the two values to form the r_p value supplied to our analytical model. Finally, as our proxy implementation is single-threaded, we have set r_i to 0 ms; r_i is subsumed into our measured r_p.

We expect these processing time values to be different in a production system offering service to multiple clients. The results presented here suggest that multi-client systems can provide page-load performance improvements if sized appropriately, but a deeper performance study is needed using an implementation explicitly designed to support multiple clients.

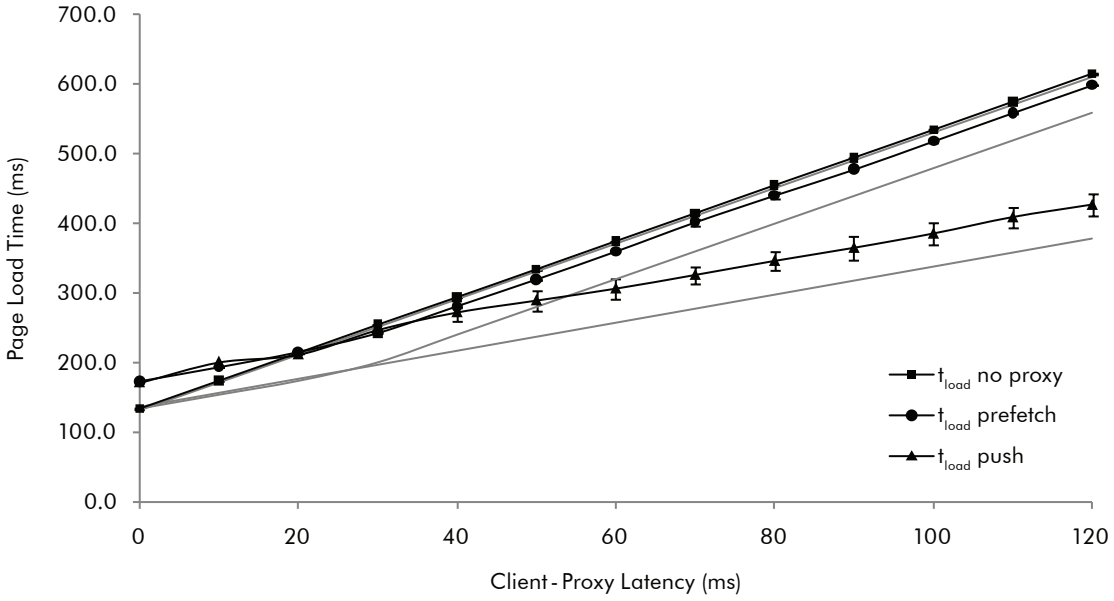


Figure 4.3: Actual load time for validation page with 1 embedded resource

4.3.2 Page with 1 embedded resource

Our first step measured load performance for a page with a single embedded image. As we can see in Figure 4.3, the measured page-load time matches our analytical model (screened in gray) for the no-proxy load; the proxied page loads take longer than expected by a constant amount. For pages with a single embedded resource, our push proxy reduced the total load time by 45 ms with a client-proxy latency of 50 ms, and by 149 ms with a latency of 100 ms.

The most unique portion of our graphs is the area around l_{cp} = 30 ms, where $l_{cp} = l_{ps}$. Figure 4.4 expands this area. Here, we expect to see the simple prefetching proxy change behaviour, and switch from the slope of the push proxy to the same slope as no proxy. Practically, this is the point where the simple prefetching proxy is able to prefetch resources to itself prior to the client requesting them. Without a mechanism to send these resources to the client as in the push proxy case, these resources must wait on the proxy until the client request arrives. As expected, we see

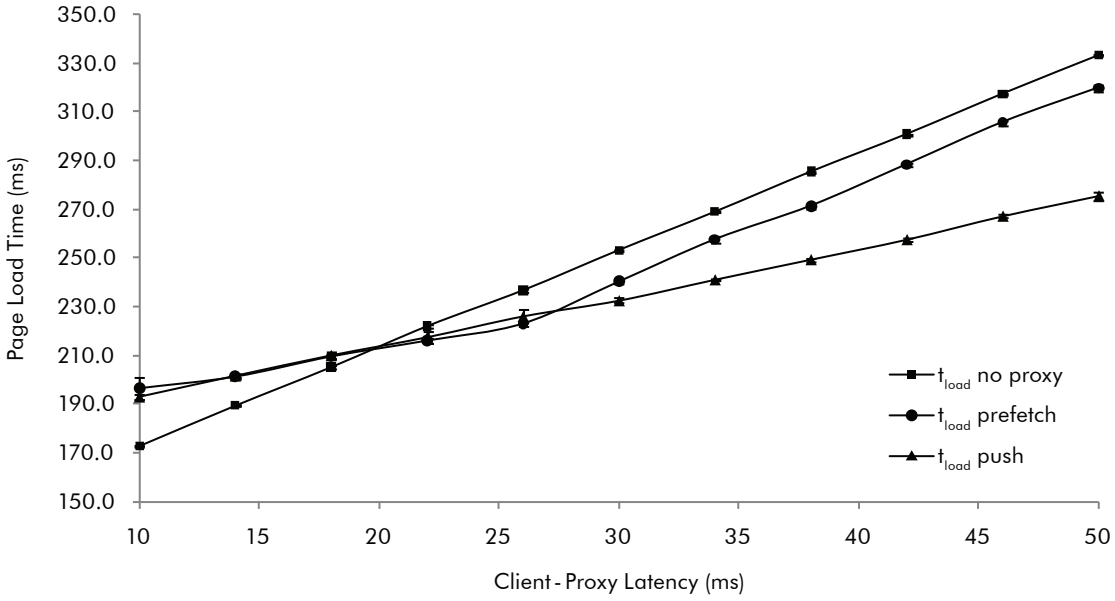


Figure 4.4: Actual load time for validation page with 1 embedded resource, 10–50 ms

the characteristic change in slope occurring around l_{cp} = 30 ms, confirming our analysis.

4.3.3 Page with 3 embedded resources

Figure 4.5 shows the measured page-load time for a page with 3 embedded resources. As in the previous case, our results have the same characteristic shape as in our analysis — the embedded images are loaded in parallel, with the increase in total load time due to the additional transmission and proxy processing time.

4.3.4 Page with embedded resources to depth 2

To determine how deep pages on the Internet are, we performed a brief survey of some popular web sites. Table 4.4 summarizes our results. We found that the median depth for desktop web-pages

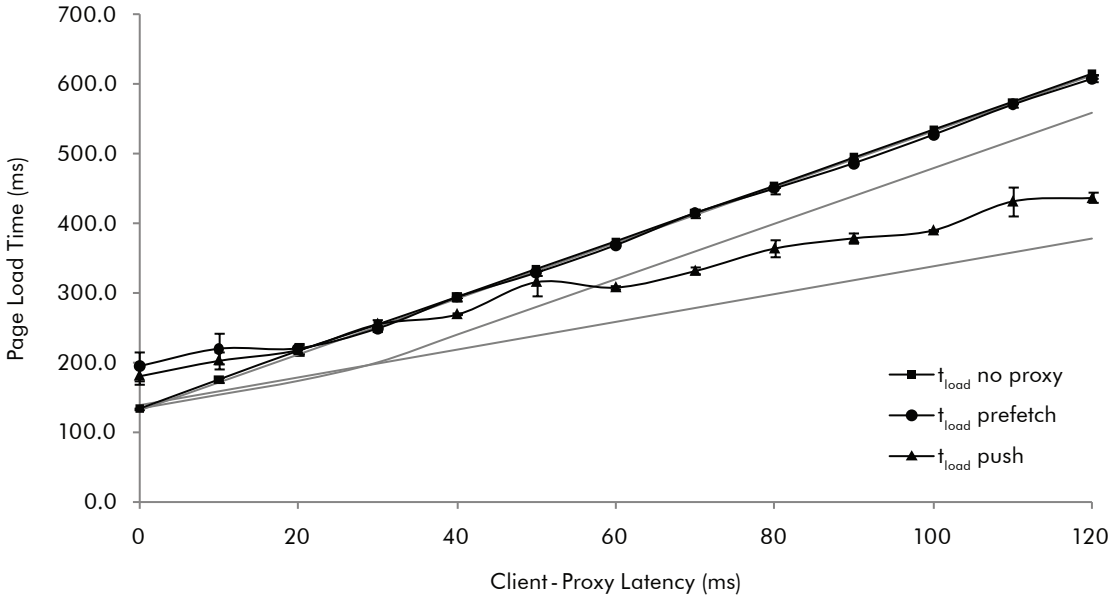


Figure 4.5: Actual load time for validation page with 3 embedded resources

was 2, while mobile web-pages (pages designed explicitly for mobile browsers) were evenly split between depths of 1 and 2. Here, we examine the behaviour of a page with an image resource embedded at a depth of 2.

Figure 4.6 shows that the measured page-load time for pages with deep resource structures is close to the load times predicted by our analysis. Our push proxy fares well here, following the much shallower slope predicted by our analysis. During the entire course of the test, the total load time for the proxied page stays under 500 ms; by the time l_{cp} = 100 ms, it loads in roughly half of the time of the no-proxy page.

4.3.5 Page with 10 embedded resources

Figure 4.7 shows the measured page-load time for a page with 10 embedded resources, a condition we did not analyze in the previous chapter. This page is similar to the 3 resources page tested

Table 4.4: Real-world page size examples

	Resources	Depth	Weight
Google	4	1	71 kB
Bing	13	2	127 kB
Amazon	64	2	682 kB
Facebook	55	2	561 kB
Twitter	69	2	780 kB
uWaterloo	18	2	784 kB
Wikipedia	39	2	737 kB
Yahoo!	43	2	699 kB
YouTube	28	2	652 kB
CBC News	71	3	912 kB
The Toronto Star	186	3	2.56 MB
New York Times	93	4	1.19 MB
Bing (mobile)	6	1	11 kB
Facebook (mobile)	29	1	120 kB
Google (mobile)	2	1	85 kB
Yahoo! (mobile)	10	1	212 kB
YouTube (mobile)	8	1	318 kB
Amazon (mobile)	15	2	70 kB
New York Times (mobile)	29	2	130 kB
Toronto Star (mobile)	23	2	45 kB
Twitter (mobile)	11	2	127 kB
CBC News (mobile)	24	3	135 kB

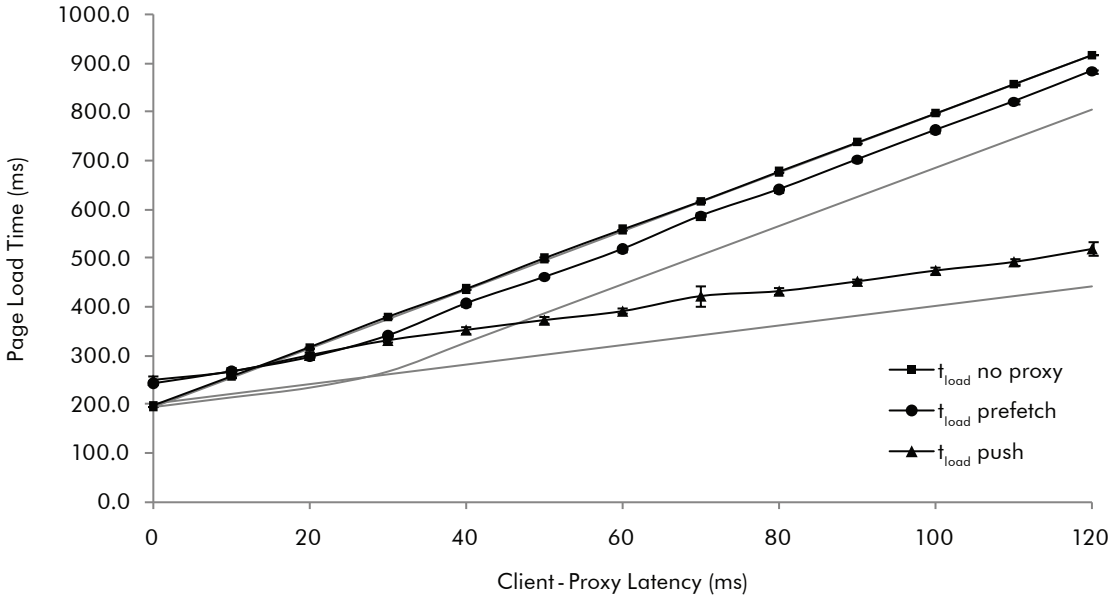


Figure 4.6: Actual load time for validation page with embedded resources to depth 2

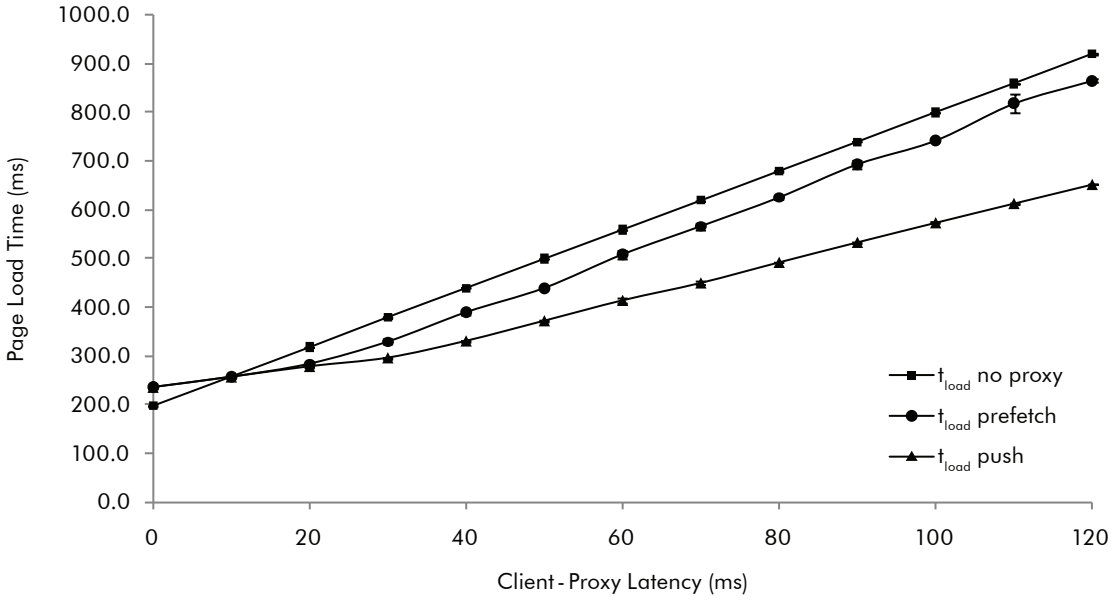


Figure 4.7: Actual load time for validation page with 10 embedded resources

previously, but because of the larger number of resources it triggers Chrome's connections-per-hostname limit of 6 [6]. As a result, we see an increase in the slope of the no-proxy and simple prefetching proxy lines, up to a slope of 6. This is similar to our results in the depth 2 case, where we see an increase in slope due to the extra page depth, here, the connection limit acts similarly to depth.

4.4 Real page performance

As we saw in the previous sections, our premise for improving page performance is sound; we can improve the load performance of pages with multiple resources, regardless of depth, by pushing these resources to the client prior to the client making a request for them. However, our tests covered only simple pages with small numbers of resources; what happens when larger, more complex pages like those seen on today's web are accessed?

In order to answer this question, we mirrored two popular web sites, Amazon and Facebook, onto our validation system. The mobile versions of each of these sites were tested, as our push proxy is intended to be used on mobile networks. As before, our validation system emulates the configuration seen in a service provider's environment, with a mobile client connected by a high-latency link to the mobile-network core, where the proxy is located.

For Amazon we mirrored the home page, which consisted of 15 resources and a depth of 2. Figure 4.8 presents our results. As we can see, our push proxy performs better than no proxy for client-proxy latencies in excess of 10 ms, with a page-load time reduction of 468 ms at l_{cp} = 100 ms. This is an easily perceptible decrease in Amazon's load time. t_{load} increased by 14 ms for each additional millisecond of client-proxy latency when no proxy was present, while it increased by a

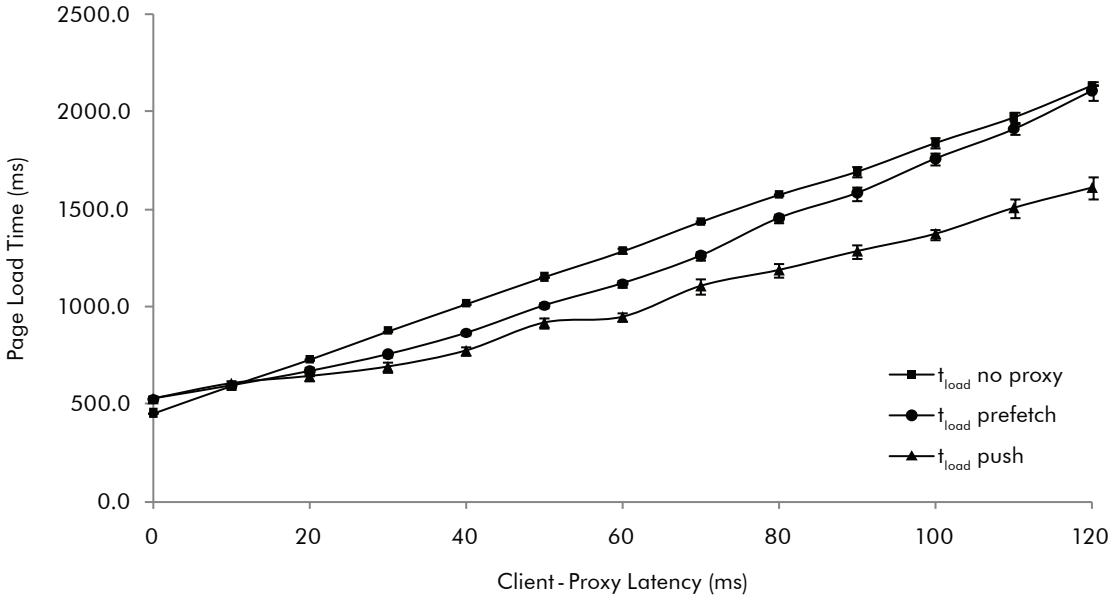


Figure 4.8: Load time for mirrored Amazon home page

shallower 9 ms/ms when accessed through our proxy.

We selected the *Celebrities on Facebook* fan page from Facebook, which resembles a user's wall page but is publicly accessible. This page consists of 29 resources, all loaded from the root HTML file (depth 1). Figure 4.9 presents our results. As was the case with Amazon, our push proxy performs better than no proxy for client-proxy latencies in excess of 10 ms. Due to the larger number of resources, however, we see a larger reduction in the total page-load time, with a decrease of 839 ms at l_{cp} = 100 ms.

Both pages here are influenced by an additional factor not explored in the previous sections; the browser's per-host connection limit (summarized in Table 2.1). For the browser used in testing, Google Chrome, that limit is 6 — only 6 resources can be loaded in parallel per domain. With 15 and 29 resources in the pages from Amazon and Facebook (respectively), Chrome must request the resources in stages, with 3 stages needed for Amazon and 5 stages needed for Facebook. The

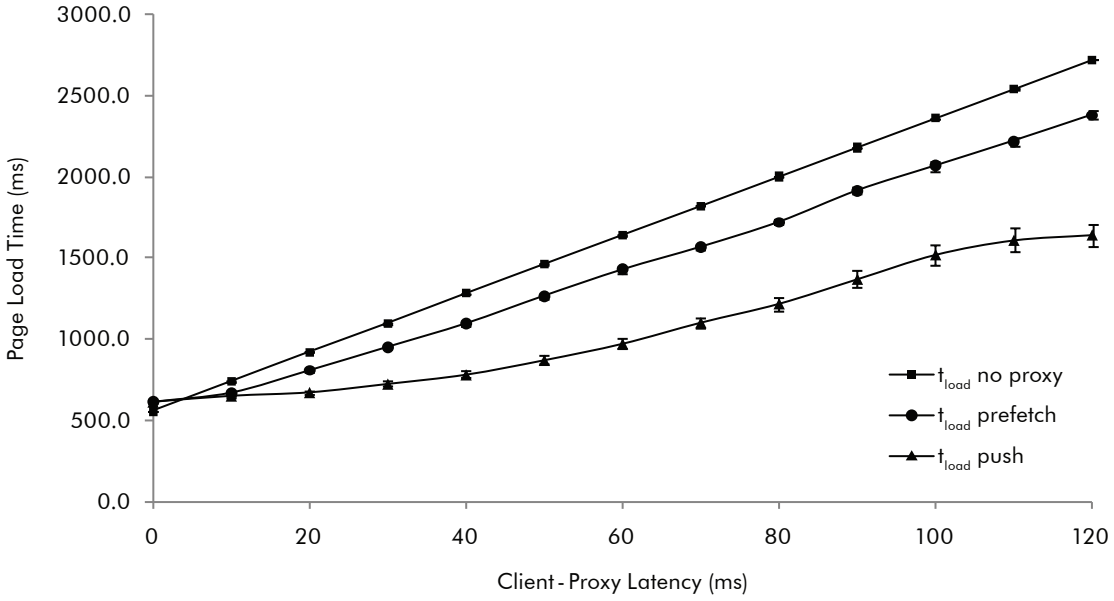


Figure 4.9: Load time for mirrored *Celebrities on Facebook* Facebook fan page

large number of resources combined with the connection limit serves to artificially increase the depth of the page. As we saw in Chapter 3, we expect slope to be proportional to depth, and that is reflected here; the page from Facebook has a higher slope than the page from Amazon, and the improvement seen with our proxy is larger as a result.

4.5 3G performance

The final test of our system is to see how it operates over an actual 3G network. Because we cannot access the mobile-network core, we cannot reproduce the scenario a mobile client would face; the client's traffic has to go out of its way to reach our proxy, rather than passing through it upon exiting the network core. This adds an additional path latency when the proxy is used that is not present with no proxy; as a result, in order for us to see a performance improvement our just-in-time push

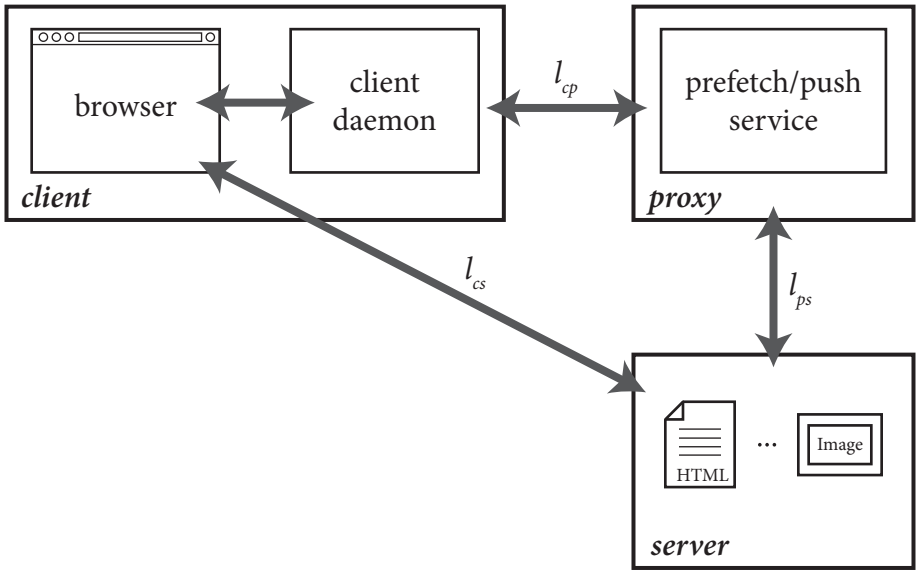


Figure 4.10: Additional path latency encountered in 3G test architecture

proxy must reduce the total load time by this extra path latency as well as the processing time of the proxy as we saw previously. Figure 4.10 diagrams this extra latency; note that $l_{cp} + l_{ps} > l_{cs}$.

4.5.1 University of Waterloo home page

The first page we measured was the University of Waterloo home page at `www.uwaterloo.ca`. This page consists of 18 resources and has a depth of 2; the total download size is 784 kB. Tests were performed using HSPA+ 3G data sticks from two Canadian mobile service providers, Rogers and Bell. Both sticks were capable of operation at 21 Mbps, though due to the other users on the 3G network it is unlikely these sticks reached this speed during our testing. Tests were performed on a weekday during work hours so that network loads similar to those encountered in day-to-day use were present.

We deployed our just-in-time push proxy on a server in our research lab for this test. This

Table 4.5: Network segment latencies when loading University of Waterloo home page

	l_{cp}	l_{ps}	l_{cs}
Rogers	53 ms	0.6 ms	52 ms
Bell	37 ms	0.6 ms	36 ms

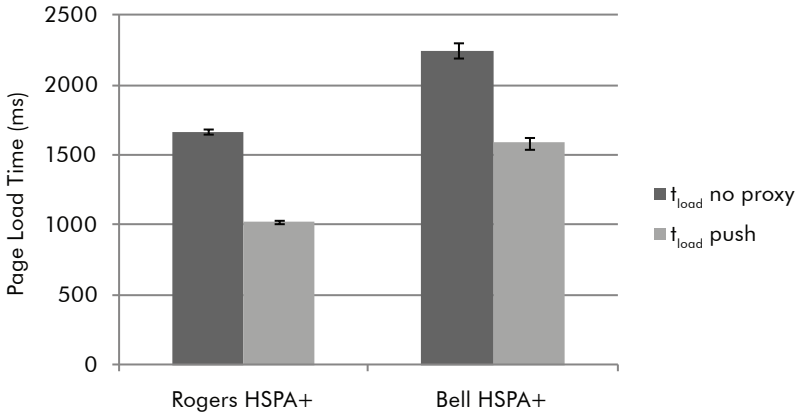


Figure 4.11: Load time for University of Waterloo homepage using 3G data stick

allows us to minimize the path latency difference between loads using no proxy and loads with our push proxy, simulating the situation where our push proxy is deployed into the carrier's mobile-network core (which likewise has minimal path latency). In this case, the additional path latency encountered in this test (not including processing time) was 1.6 ms. The latencies for each network segment are presented in Table 4.5.

Following the procedure outlined in Section 4.3.1, we measured the processing and queuing delay of our proxy using `tcpdump`. On this machine, our proxy took 24.2 ms to make a request to the server upon receiving a request from the client, and 0.9 ms to forward the response from the server to the client.

Figure 4.11 displays the load time for the University of Waterloo home page over both mobile networks, with and without our proxy. The magnitude of improvement seen on each network is

Table 4.6: 3G user bandwidth on Rogers and Bell networks

	Afternoon	Night
Rogers	939 kBps	1,210 kBps
Bell	537 kBps	739 kBps

quite perceptible to a user, consisting of a 645 ms reduction on the Rogers network and a 658 ms reduction on the Bell network. The load times on each network are quite different, however — the page loads in 1665 ms on the Rogers network, while it takes 2243 ms on the Bell network (an additional 578 ms). There are a number of possible explanations for this difference — network load, coding differences, backhaul limitations — so we ran an additional set of tests to see if we could determine the cause. Our results are presented in Table 4.6, which were taken during the workday as well as in the middle of the night when 3G networks are underutilized.

As we can see from Table 4.6, minimizing the number of users on the network by accessing the network in the middle of the night increases the bandwidth available to the user on each network, but does not address the difference between the two networks. To determine whether the difference could be because a tighter coding scheme was used on our Rogers 3G data stick as a result of being closer to a Rogers cell tower, we looked up the locations of the nearest cell towers to our testbed using Industry Canada's Spectrum Direct tool [28]. It showed that the closest Rogers tower was 376 m from our position, while the closest Bell tower was 1.84 km away, supporting this theory. Without access to the radio interface on the 3G sticks, we are unable to determine the precise coding rates used by our data sticks at the time of our tests. Alternatively, the differences we see could be the result of limited backhaul available to the Bell tower, which would result in a similar difference between networks; we believe this to be a less likely explanation, as in that case we would expect to see a larger difference in bandwidth between our day and night tests on the Bell network.

4.5.2 Validation pages

Our second set of tests measures the load times of the pages we used for validation when accessed via the Rogers 3G network. The test pages from sections 4.3 – 4.4 were uploaded to Amazon Cloudfront, a content-delivery network hosted by Amazon. As content-delivery networks move content ever closer to clients, they magnify the latency difference between the client-to-network and network-to-server segments; in some cases, the latter segment is nearly zero. This architecture provides a particularly big opportunity for our push proxy, as the larger the imbalance between the two segments, the higher the potential gain.

However, as we are not able to deploy our proxy into the mobile-network core nor along the route to the server (as was the case for the previous test), the test also encounters additional path latency. We sought to minimize latency by deploying our proxy to a Virtual Private Server (VPS) located at 151 Front Street in Toronto. This data center contains the major Internet peering point for southern Ontario, and most of the major service providers in our area peer with other providers at this location. Even with our server located here, we saw an additional 10 ms of path latency added when we used our push proxy; l_{cp} is 38 ms, l_{ps} is 12 ms, and l_{cs} is 40 ms. Using the procedure from Section 4.3.1, we measured the processing and queuing delay of our proxy on this VPS at 19 ms to make a request to the server upon receiving a request from the client, and 0.4 ms to forward the response from the server to the client.

As we can see in Figure 4.12, our just-in-time push proxy obtains modest reductions of 40 ms (1 resource) to 65 ms (3 resources) on pages with depth 1 and small numbers of resources. Our 10 resource validation page sees a smaller than expected reduction of 26 ms, though our depth 2 page sees a larger reduction as expected, of 134 ms. In all cases except the depth 2 resource, the reductions in page-load time are too small to be perceptible to an end user [9, 41, 42]; the difference

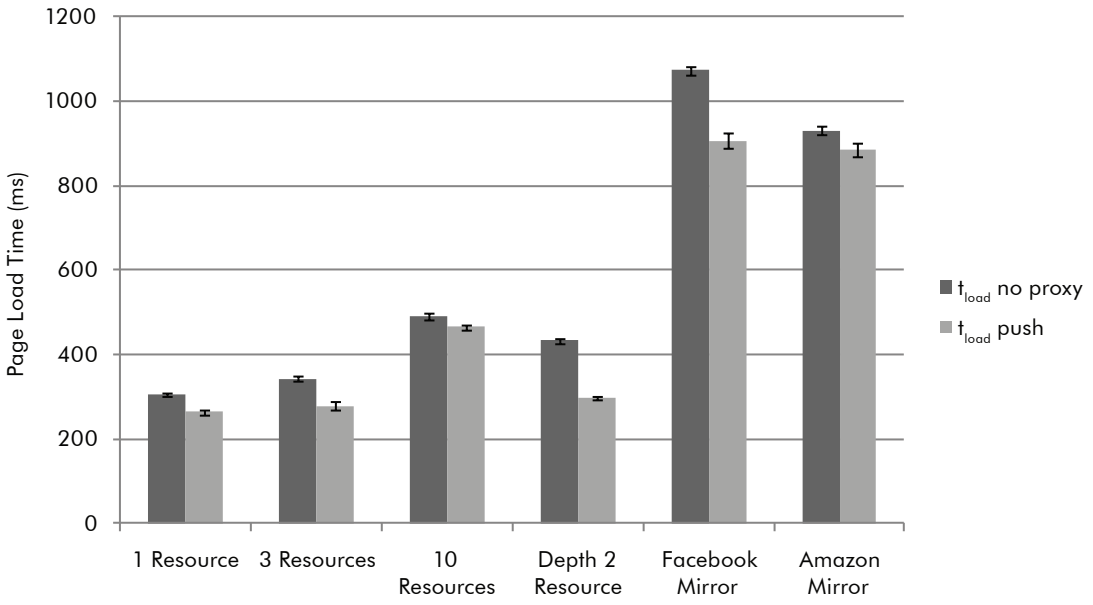


Figure 4.12: Load time for validation pages accessed using 3G data stick

in the depth 2 case is minimally perceptible. The reduction in performance here is a result of the extra path latency present here; with an extra 10 ms in path latency and 19.4 ms in processing delay, our push proxy must effectively overcome the no-proxy case's 39.4 ms head start on the first load to provide a benefit.

We would expect to see larger reductions for Amazon and Facebook, as both of these sites contain a larger number of resources and demonstrated larger page-load time reductions within our validation environment; indeed, that is exactly what we see here. With 29 resources, the Facebook mirror page sees a 166 ms reduction in page load time, an easily perceptible improvement. The Amazon mirror, with only 15 resources, sees a smaller improvement of 46 ms.

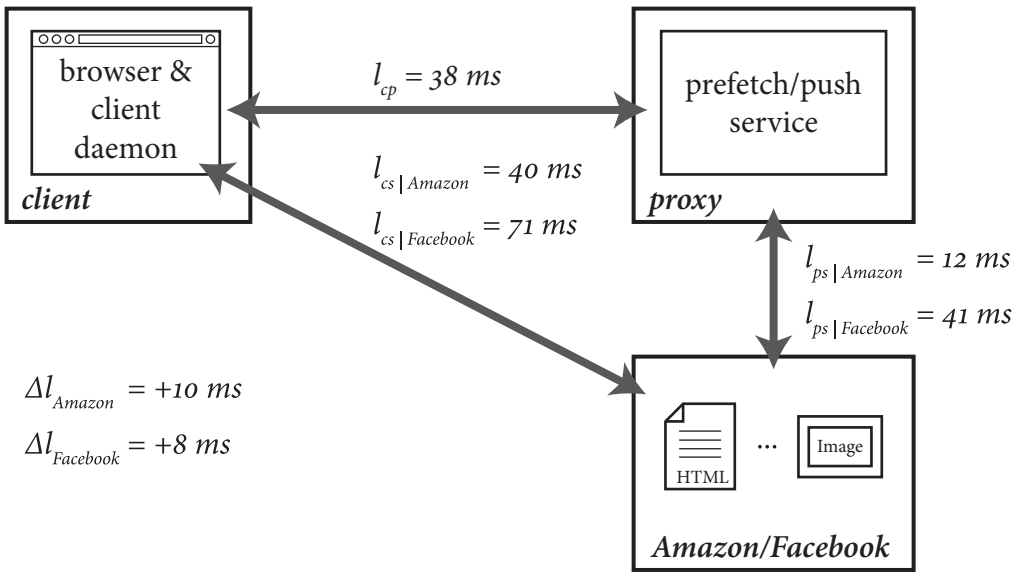


Figure 4.13: Network segment latencies when loading live pages

4.5.3 Live pages

Our final tests consist of accessing the live copies of the mirrored pages we made of Amazon's and Facebook's websites through our proxy using the 3G data stick. As in the last test, we used the Rogers 3G network, and placed our proxy on the same Virtual Private Server in Toronto. Figure 4.13 presents the latencies encountered for each website. The total additional time encountered with our proxy includes both the network latency detailed in Figure 4.13 as well as the processing and queuing delay on the proxy of 19.4 ms as before.

Another contributing factor to the performance of these live sites is our embedded resource identification accuracy; how accurately and completely our proxy predicts which resources are needed to display the page. In previous tests, our accuracy was high as the only resources were embedded directly into the HTML or CSS files, and our proxy can detect both types of resources. In practice, alternate ways of loading resources are available to page authors that our simple proxy

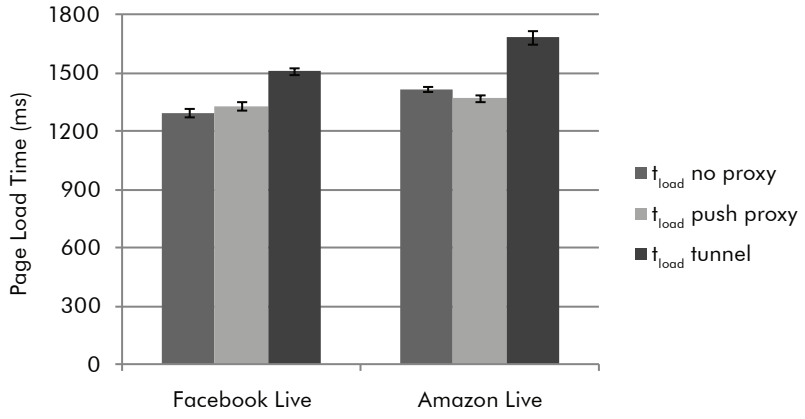


Figure 4.14: Load time for Amazon and Facebook using 3G data stick

implementation is unable to detect, with loads via JavaScript being the most significant. By missing a resource, the proxy forces the client to spawn an end-to-end request that can decrease the page-load speed and result in performance similar to the no proxy case.

Figure 4.14 presents the results of the live page test. For both Amazon and Facebook we see that the push proxy takes a similar amount of time to load the tested pages than loading the pages without our proxy present, with a 35 ms increase in load time when accessing Facebook and a 49 ms decrease in load time when accessing Amazon. As a result, in this instance there is no clear benefit for running our proxy. This does not appear to be the result of differing prefetch accuracies, as both Facebook and Amazon have acceptable prefetch rates of 78% and 56%, respectively.

We theorized that the lack of improvement in this case was due to the additional path latency incurred by the proxy not being located in the mobile network core. To confirm our theory, we created a SSH tunnel between the client and our proxy, which routed the traffic from the client through the proxy machine (though not our proxy software); this equalizes the path latency between the no-proxy and push proxy cases. Tunnelling the traffic through the proxy machine resulted in a processing delay of 1.5 ms (upsteam) and 0.2 ms (downstream); this compares to the

19 ms (upstream) and 0.4 ms (downstream) of processing delay when passing through our proxy software on the same machine.

As we can see in Figure 4.14, when the extra path latency is eliminated our proxy again shows performance improvements, of 178 ms for Facebook and 317 ms for Amazon. This underscores the importance of deploying our solution in the mobile network core — when the proxy is moved off of the path between client and server, it becomes much more difficult to realize a performance gain.

4.6 Implementation

Although we are primarily concerned with performance, we excluded client prefetching as a viable option for the mobile environment in Chapter 2 because of the amount of extra data transfer it requires. To see how effectively our simple just-in-time push proxy implementation works in practice, we measured the number of resources it was able to prefetch , the number of prefetched resources that were subsequently used, and the amount of data transferred to the client in error, as a percentage of the total page size. The measures of detection rate and prediction accuracy provide a measure of the effectiveness of our HTML/CSS scanning engine, while the data overhead allows us to compare our solution with the overheads encountered in client-based prefetchers, which typically have overheads in the range of 150% to 300% [31]. Figure 4.7 contains our results.

Resource detection rates are moderately low in our test set, with an average of 51% of resources detected in desktop pages and 55% detected in mobile pages. Though we cannot obtain 100% detection — we can never detect the load of the root HTML file before it occurs — a higher resource detection rate would allow our proxy to obtain larger decreases in page-load time. In

Table 4.7: Prediction accuracy and extra data overhead

	Resources	Prefetched	Extra Resources	Data Overhead
Amazon	196	144	75	+62%
CBC News	168	72	14	+2.1%
Facebook	68	9	3	+0.6%
Google	9	1	0	
The Toronto Star	268	192	24	+17%
Twitter	98	46	13	+5.2%
uWaterloo	22	18	4	+310%
Wikipedia	44	26	6	0.9%
Yahoo!	78	51	32	+27%
YouTube	57	25	22	+23%
Amazon (mobile)	16	9	0	
CBC News (mobile)	37	27	7	+1.2%
Facebook (mobile)	32	25	0	
Google (mobile)	8	3	1	+0.4%
Toronto Star (mobile)	31	25	2	+0.2%
Twitter (mobile)	6	4	0	
Yahoo! (mobile)	55	37	3	+3.5%
YouTube (mobile)	8	1	1	+0.2%

particular, if the resources our push proxy reduces the load time for are not on the critical path for the page load, we may see no change in the total page-load time; the larger the number of resources we miss, the higher the likelihood that we do not shorten the critical path. We also note that our proxy did not prefetch a web site's `favicon.ico` file unless it was referenced in the root HTML file; this icon is displayed in the tab's tile bar and is typically loaded after the page finishes loading. Removing both the favicon and the root HTML file from our results above results in a minor increase in the average detection rate, up 6%.

Increasing the detection rate will likely require moving beyond simple scanning of the HTML and CSS sources. Taking the YouTube mobile page as an example, our proxy prefetched none of the 8 required resources. A manual scan of the page indicates that this page builds itself almost exclusively with JavaScript, which our simple scanner is unable to process. Unfortunately, there are no standardized ways of loading resources with JavaScript, and therefore, any scanner would likely have to execute the JavaScript in order to determine which resources it loads. Furthermore, as noted at the beginning of this chapter, our proxy discards detected resources in unused styles. While this detection is accurate when CSS styles are applied statically, when JavaScript is used to apply a CSS style (as is often the case with interactive elements), we likewise are unable to detect the resource. In addition, our scanner was also unable to identify resources in CSS resource blocks places within a page's HTML; this does not seem to be a widespread practice in the pages in the test set.

Prediction accuracy was moderately good, with 67% of prefetched resources subsequently used by the browser across all desktop pages, and 89% of prefetched resources used across all mobile pages. A brief survey of the erroneously fetched resources indicated they were the result of unused styles; though our proxy attempted to filter these out, it was unable to remove them all.

The disparity between the prefetch accuracy for desktop and mobile pages appears to be the result of the page size and structure; mobile pages tend to have less interactivity and complexity, and as a result our proxy was better able to distinguish used styles from unused styles. As our proxy is designed for use primarily in the mobile environment, the 89% accuracy it received with the pages in our mobile test set is positive.

Compared to client based-prefetchers, our push proxy achieves lower data overheads, with the mobile test set sporting an average byte overhead of 0.7%. The desktop set is higher, at 45%, primarily due to the particularly large overhead during the load of the University of Waterloo home page, which had a byte overhead of 310%. Our investigation of this result determined that the increase was almost exclusively due to the erroneous load of a single resource, a 780 kB image. The actual load size for the page was 252 kB, resulting in a high overhead. The extra resource load (and indeed, all extra resource loads on this page) were the result of our proxy fetching a style sheet not intended for our browser, Google Chrome; that image and the associated style sheet are intended for Internet Explorer 7. As our proxy is unable to identify `<!--[if IE 7]><![endif]-->` and similar conditional comments, extra data is fetched when using browsers other than the browser specified by the condition. Decreasing the data overhead would require identifying conditional includes of this form and comparing the browser's `User-Agent` header to the include to determine whether or not the specified include should be fetched.

Chapter 5

Future and Related Work

5.1 Future work

An important piece of future work would be to develop an implementation of our system for current generation smartphones and tablets, to ensure that the results we saw here also hold true for these systems. In particular, should the low-power processors placed on these devices spend a high proportion of their time rendering pages rather than loading them (a situation that is not true of the desktop computer in our tests in Chapter 4), the benefits gained by our system may not play out in practice once we factor in the page rendering time. Though this is likely to be a transient condition — mobile devices are becoming ever more powerful — it may be a barrier to our system becoming adopted at the current time.

As we noted in Chapter 4, for simplicity our implementation uses a TCP connection to connect the client daemon to the proxy. Numerous studies have explored TCP's poor performance in the mobile environment [17], and as we have designed this system specifically to overcome latencies

present in mobile wireless connections, it would be prudent to use a protocol more suited for wireless transmissions. Since our system requires software on both client and proxy, and our proxy sends all web requests over the channel that connects the two elements, we have the opportunity to use a wireless-optimized protocol without having to interoperate with old servers and without having to modify client application connection libraries; the changes can be implemented solely within our architecture.

Though our push proxy was able to achieve high prediction rates on our validation test pages, real-world pages are much harder to predict accurately. Our predictor scanned only HTML and CSS files looking for referenced resources; however, the page author can also reference resources dynamically using scripts embedded in the page. This technique is used during the initial page load, where the browser determines the right resources to load based on user's identity (or their browser's identity, when used to work around browser limitations); it is also the key feature of AJAX web applications, where it allows for deferred loading [55].

Another situation impacting our predictor's accuracy is the caching of resources that web browsers perform. By caching the resource, the browser can load it directly from its cache, without going to the network (or go to the network only to validate that the resource has not changed on the server). On the proxy, however, we have no access to the list of resources that the browser has cached, and must retrieve all embedded objects from the server and push them to the client — even objects that the browser has previously cached. For this reason, some of the systems listed in Chapter 2 disable the browser cache [31, 36] or ignore the issue altogether.

We believe that both of these problems can be resolved through a more detailed modeling of the client browser on the proxy. For maximum compatibility, running a headless copy of the client web browser on the proxy server would allow the proxy to execute JavaScript code against the

browser DOM, ensuring that the prefetcher requests all dynamically-inserted resources, and would also allow the browser to build its cache using the same logic used on the client.

A more fundamental limitation of all implementations of our system is the use of HTTPS by the client browser to securely connect to a web site. With HTTPS, the operating system encrypts traffic at the transport level, which means that proxy servers placed between client and server are unable to read the content that passes through them. As a result, our push proxy is unable to scan the responses for embedded resources. Identifying ways to work with HTTPS is a significant area of future work, especially as it must ensure that the security provided by HTTPS is not compromised by the presence of side-channels (for instance, our push proxy should not know the identity of the 'security image' used on some banking websites, as this image is intended to be private between client and server).

5.2 Related work

Due to the large number of components in the web architecture, the speed at which a page loads can be improved in a number of different areas. Our system increases performance by reducing the number of network traversals the client must make while loading a page; other areas for improvement include changing the design of the page, changing the way pages are served by the server, preprocessing the page within the network, modifying the design of the underlying connection, and extending the client browser to perform prefetching or increase rendering speed.

5.2.1 Similar systems

Research by Dong et. al [14] investigated a system in which pages requested by a client were fetched from a server by a proxy and then compressed into a TAR archive in order to reduce the latency present in mobile web browsing. Their research is only preliminary, however — no description is given for how they identify all of the components of a web page, nor how the client decodes the resulting archive for display. Furthermore, their results are based on simulation, rather than an implementation of their system. Architecturally, this system is similar to our own, in that content is fetched by a proxy and pushed to a mobile client; numerous issues need to be addressed before it reaches the capabilities of our system.

Commercial web accelerators perform a range of optimizations that fall in many of the following categories; many of these optimizations are proprietary and not documented publically. One such optimization appears superficially similar to ours, and is described by its manufacturer Openwave as "When a user requests a URL, [Openwave's] Web Optimizer creates a compressed multipart/related document to return to the handset." [44] Multi-part HTTP documents, denoted by the MIME type `multipart/related`, consist of a root document and accompanying files identified by URL, and have been traditionally been used by email clients for rich text emails as well as by Internet Explorer to save complete web pages (the MHTML file type). Though further details on Openwave's system are not available, we posit that their Web Optimizer packages all of the resources for a requested page into a single MHTML document for transmission to the client, which can then display the entire page immediately. In this manner, it acts similarly to our own system; the proxy fetches the necessary content on behalf of the client without waiting for it to make corresponding requests, improving the perceived page load time by reducing the number of client–network round-trips. It is unclear from the Openwave documentation how many multi-part documents are created; if only

a single document is created, all resources must be loaded sequentially due to the contiguous nature of multipart/related documents. Though some technical differences exist between our system and Openwave's, architecturally both systems can obtain the same benefits when operating in a mobile environment.

However, as Openwave's Web Optimizer is a client-less product the use of this optimization is dependent on mobile browser support of multi-part documents. At the current time, only BlackBerry OS 5 and earlier support multi-part documents; on the desktop browser side, only Internet Explorer and Opera do. Support for MHTML documents has recently been added to WebKit; in the future, we may see support by Safari (iOS), Chrome (Android), and BlackBerry OS 6 and above. We believe that this optimization is of limited use at this time, though a client daemon analogous to our own could be created to act as an intermediary. A similar system, in which a multi-part document is prepared following an initial handshake with the client browser to confirm compatibility and containing the root HTML file is described in US patent 2005/0144278; we know of no commercial implementation of this patent.

5.2.2 Page improvements

The best way to improve the speed at which a web site loads over a mobile network connection is to design the site specifically for mobile clients; by eliminating images and other media elements, the number of round-trips between mobile client can be reduced. Companies like Facebook, Google, and the New York Times offer mobile sites for this reason. As shown in Chapter 4, even mobile sites can see an improvement with our just-in-time push proxy. An alternative to eliminating images is embedding them within the page itself using data URIs [39]. Resources embedded this way do not require the browser to make a separate request for the resource data, eliminating a client-server

round trip; both the iOS and Android browsers support this technique [6].

In addition to changing how a page references resources, both Google [25] and Yahoo [60] have developed tools that audit web sites and provide a list of recommendations to increase the page-load speed. These recommendations include ways to optimize caching through the use of far-future expires headers, minimizing upload size, and optimizing the way the browser lays out and requests a page [25, 60]. Using these techniques, reductions in page-load times of 25–50% are obtainable [54]. Google has recently packaged some of the above practices into an Apache module, mod_pagespeed, that performs these optimizations automatically [25].

5.2.3 Server improvements

There are a myriad of ways to reduce the time it takes a server to process a request for data, adapted to the specific type of resources being served by the web server (static files, dynamically generated pages, web applications, or streaming media). A full coverage of these techniques is not presented here, as the time it takes a server to serve the same content to a mobile client and a desktop client is equal; these techniques improve desktop and mobile page-load times equally.

We do make specific mention of the work done by Bhatti et al. [4] and Olshefski and Nieh [43] to incorporate a real-time measure of the user-perceived latency into the design of the web server itself. In particular, Bhatti et al. recommend using earliest deadline first scheduling on web servers, based on the limits of human perception (discussed in Chapter 4) and the actual delay between client and server [4]. In the mobile environment, this would result in mobile clients being served before desktop clients, as delaying a desktop client by an imperceptible amount in favour of a mobile client could result in modest speedups on the mobile client.

Another avenue for improving page load performance on the server is to increase the TCP

initial congestion window past the standardized maximum of 4 segments [37]. Work by Dukkipati et al. showed that increasing the initial TCP congestion window to 10 segments on the Google front-end servers reduced the average HTTP latency by 10% [16]. As this increase only results in a 0.5% increase in retransmissions [16], we believe this approach has merit for mobile clients.

Research by Serbinski and Abhari [52] proposed a system in which a custom-built HTTP server anticipates client requests for embedded objects also located on the server and sends them to the client automatically. Their approach differs from previous work in that their engine only predicts elements embedded in the current page, rather than using a probabilistic approach [52]. A daemon running on the client acts as an intermediary between an unmodified client browser and the server, much like our client daemon acts as an intermediary between our proxy and the client. This system showed reductions in page load time of up to 73% in their tests [52].

This system acts similarly to our system, except that the prefetching logic of our proxy is colocated on the server, removing the need for a separate proxy machine. However, as a result of this colocation only servers that have this logic added will see a corresponding reduction in load time; in contrast, by having a separate proxy all pages accessed by a mobile client can see better page-load performance with our system. Furthermore, our system is able to prefetch page components from third-party servers, an important feature for current web pages, which are often formed using components from third-party analytic, advertisement, or social media servers.

5.2.4 Network improvements

Future wide-area wireless technologies, like LTE-Advanced, are expected to bring higher raw and user data rates along with lower latencies [21]. As discussed in Chapter 1, due to the shared-usage nature of wide-area wireless technologies, we believe these technologies will continue to lag the

page-load speeds available with local-area networks. Femtocells and Wi-Fi provide an alternative to wide-area wireless networks [27]; by reducing the transmission range, these technologies reduce the number of users they can serve and therefore increase the fraction of the raw data rate that is available to individual users. Though expensive on a per-unit-area basis [10], the data rates and latencies available with these technologies are similar to those found in wired networks, resulting in similar page-load time.

SPDY is an experimental proposal proposed by Google for minimizing latency on the web. SPDY replaces HTTP with a new application-layer protocol that allows for multiplexed streams, prioritized requests, header compression, server hinting/pushing, and mandatory security that results in a 27 – 60% reduction in page load times in their testing [24]. Though SPDY's benefits come from the confluence of its features, the features with the most in common to our system are server hinting/pushing. Using these features, a proxy could be constructed that runs a prefetching engine similar to our own and then pushes the result to the browser without a separate client daemon. Furthermore, unlike our system the hint mechanism of SPDY could be used to search the browser's cache to ensure that already-cached resources are not re-fetched by the proxy. Architecturally, such a system could obtain the same benefits when operating in a mobile environment as our own system.

We have not seen any evidence that such a system has been constructed; existing research focuses on other performance-enhancing aspects of SPDY. In addition, mobile browsers would have to be updated with support for SPDY; at the current time, no mobile browsers support SPDY, and only Chrome supports SPDY on the desktop.

5.2.5 In-network processing

Given the small screen sizes present on mobile devices, a popular technique for improving page-load times on mobile devices is to transcode web content by reducing the size of embedded resources, summarizing text, and adapting the page layout [8]. Proteus, a mobile web adaptation system developed by Caetano et al. [8], performs all of these functions, obtaining compression ratios of up to 87%. Mowser, developed by Bharadvaj et al. [3], transcodes and filters requests bidirectionally, allowing the end server to provide scaled-down resources without requiring the proxy to transcode them on behalf of the client; it also ensures that only content the mobile device is able to display is requested from the server. Other transcoders [22, 38] show similar results, with significant reductions in the amount of data transferred between proxy and client.

In addition to simply transcoding information, the higher processing power available to networked servers has been used to offload page processing and rendering from the mobile browser into the network [30, 33, 46, 61]. This approach has two major advantages: complex pages can be rendered rapidly on the server, and the simplified form that results is often smaller and faster to transfer than the original page. A commercial example of this technology is the Opera Mini browser, available for smartphones and feature phones [47].

Opera Mini optimizes the mobile web experience by relocating the browser rendering engine to a proxy server 'in the cloud'; the Opera Mini application that resides on the mobile client acts only as an input/output component with no logic of its own. The rendering engine and display client communicate using a single TCP connection using a proprietary protocol known as OBML (Opera Binary Markup Language) [45].

When a user enters a web address into Opera Mini, their request is passed to the Opera Mini proxy. The proxy requests the page from the end server, and continues to request and render the

page as information arrives from the server. Once the page has been rendered, the rendered form is compressed and transmitted to the Opera Mini client that made the request, which displays the page. This process saves CPU usage on the mobile device and can reduces bandwidth by up to 90% [46]; other research [47] shows improvements of 27 – 67%. Perceived page load times can also be reduced as a result of a reduction in the page transmission time, the increased render speed of the server (more processing power), and by reducing the number of client–network round-trips.

Our just-in-time push prefetching system is similar to Opera Mini as it also improves the perceived page load time by reducing the number of client-network round-trips; as this segment is proportionally large in wireless wide-area networks, measurable reductions can be obtained. Unlike Opera Mini, our system runs only a fetching engine and not an entire rendering engine on the proxy; as a result, we require a full browser on the mobile client. Our proxy does not transcode or compress any content; the amount of data transmitted across the wireless link is similar with or without our proxy.

As a result of running the rendering engine on the proxy, Opera Mini has some notable drawbacks that our system avoids by virtue of it using a full client browser. First, once the page is loaded a static snapshot of the page is sent to the client; after this snapshot is sent, all scripts are paused on the server and no further updates are sent unless the user interacts with the page. This prevents some common web techniques like timed AJAX updates from functioning [46]. Furthermore, if the user clicks on a page element the click event must be dispatched to the server, the click handler executed, and a new snapshot prepared and sent to the Opera Mini client for display. Given the high-latency first hop on wide-area wireless networks, this process can easily take 100 ms or more – a delay that is certainly noticeable to users. Finally, though Opera Mini uses a secure connection to communicate with its proxy, as all rendering happens on the proxy

the end-to-end security semantics of SSL are broken; should the Opera proxies be compromised, the secure traffic of many users could be viewed (in contrast, our proxy has no ability to see into secure data streams).

5.2.6 Client improvements

In Chapter 2, we discussed prefetching web content as a strategy for reducing mobile page-load times. Prefetching can be applied to other aspects of the client-server interaction as well. For instance, prefetching — or pre-resolving — the DNS names of servers that the browser may access saved an average of 250 ms in tests performed by the Google Chrome team [23]. Furthermore, as each DNS request is typically small (hundreds of bytes), this approach adds minimal extra data overhead even when the prefetching engine fetches unnecessary DNS entries. For this reason, recent versions of the Firefox, Safari, and Chrome browsers support DNS pre-resolving.

Another intriguing approach is to 'prefetch' a TCP connection to the servers the browser predicts the client will access in the future, without making a request for content [7,12]. Since the time it takes for TCP's three-way handshake to occur can be a significant portion of the total time it takes to download a small resource, especially on high-latency links like those found in mobile and satellite environments, this approach can offer measurable performance benefits. However, the utility of this approach is more limited than either DNS or content prefetching, as origin servers typically close idle connections after tens of seconds or minutes [12].

Significant effort has been invested in recent years to improve the speed at which pages load within the browser, with approaches including improving the execution speed of Javascript engines [19, 35], increasing parallelism in layout and rendering [1, 40], and caching style and page fragments to save future recomputation [63]. Though Badea et al. [1] were able to increase page-

load speed by a factor of 1.84 with their parallel browser, as their tests were run in an offline environment it is difficult to compare the benefits available via browser improvements with our own network-based improvements.

Chapter 6

Conclusions

We have presented a just-in-time prefetching push proxy that increases the speed of mobile browsing in wide-area wireless networks. This architecture consists of an in-network proxy that proactively prefetches resources embedded in the HTML pages loaded through it, and a client daemon that the in-network proxy pushes prefetched resources to.

Our analysis of the push proxy showed that pushing content to the mobile client provides a fundamental decrease in page-load times. This decrease is due to the push proxy's much smaller dependence on the client to proxy latency, as compared to no proxy or a traditional prefetching proxy. This decrease holds for sites with multiple resources, including those resources reference other resources. Our analysis shows that a push proxy can achieve a significant decrease in page-load times given the client-proxy latencies encountered in mobile networks; however, this proxy must be deployed within the mobile network core (or elsewhere along the path from client to server) to provide this benefit. As our architecture is compatible with existing web browsers and servers, we built a sample implementation to evaluate our design; this evaluation showed that our

architecture performs as expected in our validation environment, and shows promise when using currently-deployed 3G networks to access the Internet.

Unlike typical proxies, where the largest performance gains occur when the proxy is close to the client, our push proxy performs better when placed closer to the server. This makes it ideally suited for the web today, where techniques like content-delivery networks bring content ever closer to the client yet cannot overcome the high-latency client-to-network segment present in mobile networks.

References

[1] C. Badea, M. R. Haghighat, A. Nicolau, and A. V. Veidenbaum, "Towards parallelizing the layout engine of Firefox," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism*. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–1.

[2] G. Barish and K. Obraczke, "World wide web caching: Trends and techniques," *IEEE Communications Magazine*, vol. 38, no. 5, pp. 178 –184, May 2000.

[3] H. Bharadvaj, A. Joshi, and S. Auephanwiriyakul, "An active transcoding proxy to support mobile web access," in *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 118–.

[4] N. Bhatti, A. Bouch, and A. Kuchinsky, "Integrating user-perceived quality into web server design," in *Proceedings of the 9th International World Wide Web Conference on Computer Networks*. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 2000, pp. 1–16.

[5] C. Bouras, A. Konidaris, and D. Kostoulas, "Predictive prefetching on the web and its potential impact in the wide area," *World Wide Web*, vol. 7, pp. 143–179, June 2004.

[6] Browserscope.org, "Browserscope web browser profiles," 2011. [Online]. Available: http://www.browserscope.org/

[7] R. Cáceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich, "Web proxy caching: The devil is in the details," *SIGMETRICS Performance Evaluation Review*, vol. 26, pp. 11–15, December 1998.

[8] M. F. Caetano, A. L. F. Fialho, J. L. Bordim, C. D. Castanho, R. P. Jacobi, and K. Nakano, "Proteus: An architecture for adapting web page on small-screen devices," in *Proceedings of the 2007 IFIP international conference on Network and parallel computing*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 161–170.

[9] S. K. Card, G. G. Robertson, and J. D. Mackinlay, "The information visualizer, an information workspace," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Reaching Through Technology*. New York, NY, USA: ACM, 1991, pp. 181–186.

[10] V. Chandrasekhar, J. Andrews, and A. Gatherer, "Femtocell networks: A survey," *IEEE Communications Magazine*, vol. 46, no. 9, pp. 59 –67, 2008.

[11] Cisco Systems Inc., "Cisco visual networking index: forecast and methodology, 2010-2015," June 2011. [Online]. Available: http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360_ns827_Networking_Solutions_White_Paper.html

[12] E. Cohen and H. Kaplan, "Prefetching the means for document transfer: A new approach for reducing web latency," *Computer Networks*, vol. 39, no. 4, pp. 437 – 455, 2002.

[13] J. Domenech, J. Sahuquillo, J. A. Gil, and A. Pont, "The impact of the web prefetching architecture on the limits of reducing user's perceived latency," in *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 740–744.

[14] W. Dong, X. Chen, S. Xu, W. Wang, and G. Wei, "Proxy-based object packaging and compression: a web acceleration scheme for UMTS," in *5th International Conference on Wireless Communications, Networking and Mobile Computing*, September 2009, pp. 1 –5.

[15] S. Drakatos, N. Pissinou, K. Makki, and C. Douligeris, "A context-aware prefetching strategy for mobile computing environments," in *Proceedings of the 2006 International Conference on Wireless Communications and Mobile Computing*. New York, NY, USA: ACM, 2006, pp. 1109–1116.

[16] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin, "An argument for increasing TCP's initial congestion window," *SIGCOMM Computer Communications Review*, vol. 40, pp. 26–33, June 2010.

[17] H. Elaarag, "Improving TCP performance over mobile networks," *ACM Computing Survey*, vol. 34, pp. 357–374, September 2002.

[18] J. Fabini, W. Karner, L. Wallentin, and T. Baumgartner, "The illusion of being deterministic – Application-level considerations on delay in 3G HSPA networks," in *Proceedings of the 8th International IFIP-TC 6 Networking Conference*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 301–312.

[19] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and

M. Franz, "Trace-based just-in-time type specialization for dynamic languages," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2009, pp. 465–478.

[20] H. Galeana-Zapién and R. Ferrús, "Design and evaluation of a backhaul-aware base station assignment algorithm for ofdma-based cellular networks," *Transactions in Wireless Communications*, vol. 9, pp. 3226–3237, October 2010.

[21] A. Ghosh, R. Ratasuk, B. Mondal, N. Mangalvedhe, and T. Thomas, "LTE-advanced: Next-generation wireless broadband technology."

[22] J. G. S. Gonzalez, V. J. S. Sosa, A. R. Montes, and y J. Carlos R. Olivares, "Multi-format web content transcoding for mobile devices," in *Seventh Mexican International Conference on Computer Science, 2006*, 2006, pp. 109 –115.

[23] Google, "DNS prefetching (or pre-Resolving)," 2008. [Online]. Available: http://blog.chromium.org/2008/09/dns-prefetching-or-pre-resolving.html

[24] ——, "SPDY: An experimental protocol for a faster web," November 2009. [Online]. Available: http://www.chromium.org/spdy/spdy-whitepaper

[25] ——, "Web performance best practices," 2011. [Online]. Available: http://code.google.com/speed/page-speed/docs/rules_intro.html

[26] P. Gulati, A. Sharma, A. Goel, and J. Pandey, "A novel approach for determining next page access," in *First International Conference on Emerging Trends in Engineering and Technology, 2008*, 2008, pp. 1109 –1113.

[27] S. Hasan, N. Siddique, and S. Chakraborty, "Femtocell versus Wi-Fi – A survey and comparison of architecture and performance," in *1st International Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace Electronic Systems Technology, 2009*, May 2009, pp. 916 –920.

[28] Industry Canada, "Geographical area search — Spectrum Direct," August 2011. [Online]. Available: http://sd.ic.gc.ca/pls/engdoc_anon/web_search.geographical_input

[29] Joyent, Inc, "node.js: Evented I/O for V8 JavaScript," 2010. [Online]. Available: http://nodejs.org

[30] J. Kim, R. A. Baratto, and J. Nieh, "pTHINC: A thin-client architecture for mobile wireless web," in *Proceedings of the 15th International Conference on World Wide Web*. New York, NY, USA: ACM, 2006, pp. 143–152.

[31] R. Klemm, "WebCompanion: A friendly client-side web prefetching agent," *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, no. 4, pp. 577 –594, 1999.

[32] T. M. Kroeger, D. D. E. Long, and J. C. Mogul, "Exploring the bounds of web latency reduction from caching and prefetching," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*. Berkeley, CA, USA: USENIX Association, 1997, pp. 2–2.

[33] A. M. Lai, J. Nieh, B. Bohra, V. Nandikonda, A. P. Surana, and S. Varshneya, "Improving web browsing performance on wireless pdas using thin-client computing," in *Proceedings of the 13th International Conference on World Wide Web*. New York, NY, USA: ACM, 2004, pp. 143–154.

[34] K. Lau and Y.-K. Ng, "A client-based web prefetching management system based on detection theory," in *Web Content Caching and Distribution*, ser. Lecture Notes in Computer Science, C.-H. Chi, M. van Steen, and C. Wills, Eds. Springer Berlin / Heidelberg, 2004, vol. 3293, pp. 129–143.

[35] S.-W. Lee, S.-M. Moon, W.-K. Jung, J.-S. Oh, and H.-S. Oh, "Code size and performance optimization for mobile JavaScript just-in-time compiler," in *Proceedings of the 2010 Workshop on Interaction between Compilers and Computer Architecture*. New York, NY, USA: ACM, 2010, pp. 6:1–6:7.

[36] T. S. Loon and V. Bharghavan, "Alleviating the latency and bandwidth problems in WWW browsing," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*. Berkeley, CA, USA: USENIX Association, 1997, pp. 20–20.

[37] M. Allman, S. Floyd, and C. Partridge, "Increasing TCP's initial window," RFC 3390, 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3390.txt

[38] A. Maheshwari, A. Sharma, K. Ramamritham, and P. Shenoy, "Transquid: Transcoding and caching proxy for heterogenous e-commerce environments," in *Proceedings of the Twelfth International Workshop on Research Issues in Data Engineering: Engineering E-Commerce/E-Business Systems, 2002*, 2002.

[39] Masinter, L., "The "data" url scheme," RFC 2397, 1998. [Online]. Available: http://www.ietf.org/rfc/rfc2397.txt

[40] L. A. Meyerovich and R. Bodik, "Fast and parallel webpage layout," in *Proceedings of the 19th International Conference on World Wide Web*. New York, NY, USA: ACM, 2010, pp. 711–720.

[41] R. B. Miller, "Response time in man-computer conversational transactions," in *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I.* New York, NY, USA: ACM, 1968, pp. 267–277.

[42] J. Nielsen, *Usability Engineering.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[43] D. Olshefski and J. Nieh, "Understanding the management of client perceived response time," in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems.* New York, NY, USA: ACM, 2006, pp. 240–251.

[44] Openwave Systems, "Openwave Web Optimizer," February 2011. [Online]. Available: http://www.openwave.com/sites/default/files/docs/solutions/WebOptimizer_021011_0.pdf

[45] Opera Software, "Opera Mini FAQ," 2011. [Online]. Available: http://www.opera.com/mobile/help/faq/

[46] ——, "Opera Mini: Web content authoring guidelines," February 2011. [Online]. Available: http://dev.opera.com/articles/view/opera-mini-web-content-authoring-guidelines/

[47] S. Østen, "Analysing the compression of Opera Mini™ traffic," Master's thesis, Oslo University College, 2008.

[48] V. N. Padmanabhan and J. C. Mogul, "Using predictive prefetching to improve world wide web latency," *SIGCOMM Computer Communications Review*, vol. 26, pp. 22–36, July 1996.

[49] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol – HTTP/1.1," RFC 2616, 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2616.txt

[50] Sandvine Inc., "2010 mobile Internet phenomena report," 2010. [Online]. Available: http://www.sandvine.com/downloads/documents/2010MobileInternetPhenomenaReport.pdf

[51] J. H. Schiller, *Mobile Communications*, 2nd ed. Addison-Wesley, 2003.

[52] A. Serbinski and A. Abhari, "Improving the delivery of multimedia embedded in web pages," in *Proceedings of the 15th International Conference on Multimedia*. New York, NY, USA: ACM, 2007, pp. 779–782.

[53] N. Shetty, S. Parekh, and J. Walrand, "Economics of femtocells," in *IEEE Global Telecommunications Conference, 2009*, December 2009, pp. 1 –6.

[54] S. Souders, *High Performance Web Sites*, 1st ed. O'Reilly, 2007.

[55] ——, *Even Faster Web Sites: Performance Best Practices for Web Developers*, 1st ed. O'Reilly, 2009.

[56] W.-G. Teng, C.-Y. Chang, and M.-S. Chen, "Integrating web caching and web prefetching in client-side proxies," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 5, pp. 444 – 455, May 2005.

[57] T. Theurer, "Performance research, part 2: Browser cache usage – exposed!" January 2007. [Online]. Available: http://yuiblog.com/blog/2007/01/04/performance-research-part-2/

[58] 3GPP, "High speed packet access (HSPA) evolution; frequency division duplex (FDD)," 3rd Generation Partnership Project (3GPP), TR 25.999, Mar. 2008. [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/25999.htm

[59] S. Tilkov and S. Vinoski, "Node.js: Using JavaScript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, pp. 80–83, November 2010.

[60] Yahoo Inc., "Best practices for speeding up your web site," 2010. [Online]. Available: http://developer.yahoo.com/performance/rules.html

[61] S. J. Yang, J. Nieh, S. Krishnappa, A. Mohla, and M. Sajjadpour, "Web browsing performance of wireless thin-client computing," in *Proceedings of the 12th International Conference on World Wide Web*. New York, NY, USA: ACM, 2003, pp. 68–79.

[62] S. Yun, Y. Yi, D. Cho, and J. Mo, "On the pricing of femtocell services," in *Proceedings of the 5th International Conference on Future Internet Technologies*. New York, NY, USA: ACM, 2010, pp. 1–4.

[63] K. Zhang, L. Wang, A. Pan, and B. B. Zhu, "Smart caching for web browsers," in *Proceedings of the 19th International Conference on World Wide Web*. New York, NY, USA: ACM, 2010, pp. 491–500.