Thesis Title: Rethinking the Architecture of the Web

Author: Liang Zhang

Ph.D. Thesis Approved to complete all degree requirements for the Ph.D. Degree in Computer Science.

Thesis Advisor: 
Date: 7/7/2016

Thesis Reader: 
Date: 7/7/2016

Thesis Reader: 
Date: 7/15/16

Thesis Reader: 
Date

Graduate School Approval:

Director, Graduate School: 
Date: 7/26/16

Copy Received in Graduate School Office:

Recipient's Signature: 
Date: 7/27/2016

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI website.
Rethinking the Architecture of the Web

A Dissertation Presented
by

Liang Zhang

to

The College of Computer and Information Science

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Computer Science

Northeastern University
Boston, Massachusetts

July 2016
To my family
**Contents**

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>ix</td>
</tr>
<tr>
<td>Abstract of the Dissertation</td>
<td>x</td>
</tr>
</tbody>
</table>

**1 Introduction**

1.1 Contributions ........................................ 5
1.2 Outline ................................................. 7

**2 Background**

2.1 Web browsers ............................................... 8
   2.1.1 Dynamic web pages .................................. 9
   2.1.2 Browser plug-ins ................................... 10
   2.1.3 HTML5 .............................................. 11
   2.1.4 Mobile web browsers ................................. 12
2.2 JavaScript ................................................ 13
   2.2.1 The web and beyond ................................ 13
2.3 Web servers ............................................... 14
   2.3.1 Web application servers ............................ 15
   2.3.2 Privacy ........................................... 15
2.4 Cloud services ........................................... 16
   2.4.1 Content distribution networks (CDNs) ............ 17

**3 Maygh: Rethinking web content distribution**

3.1 Motivation ............................................... 19
3.2 Maygh potential ......................................... 21
3.3 Maygh design ........................................... 22
   3.3.1 Web browser building blocks ....................... 22
   3.3.2 Model, interaction, and protocol .................. 23
   3.3.3 Maygh client ....................................... 25
   3.3.4 Maygh coordinator .................................. 28
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.5 Multiple coordinators</td>
<td>28</td>
</tr>
<tr>
<td>3.4 Security, privacy, and impact on users</td>
<td>30</td>
</tr>
<tr>
<td>3.4.1 Security</td>
<td>30</td>
</tr>
<tr>
<td>3.4.2 Privacy</td>
<td>31</td>
</tr>
<tr>
<td>3.4.3 Impact on users</td>
<td>32</td>
</tr>
<tr>
<td>3.4.4 Mobile users</td>
<td>32</td>
</tr>
<tr>
<td>3.5 Implementation</td>
<td>33</td>
</tr>
<tr>
<td>3.6 Evaluation</td>
<td>33</td>
</tr>
<tr>
<td>3.6.1 Client-side microbenchmarks</td>
<td>34</td>
</tr>
<tr>
<td>3.6.2 Coordinator scalability</td>
<td>36</td>
</tr>
<tr>
<td>3.6.3 Trace-based simulation</td>
<td>38</td>
</tr>
<tr>
<td>3.6.4 Small-scale deployment</td>
<td>43</td>
</tr>
<tr>
<td>3.7 Summary</td>
<td>44</td>
</tr>
<tr>
<td>4 Priv.io: Rethinking web applications</td>
<td>45</td>
</tr>
<tr>
<td>4.1 Motivation</td>
<td>45</td>
</tr>
<tr>
<td>4.2 Overview</td>
<td>47</td>
</tr>
<tr>
<td>4.2.1 Cost study</td>
<td>48</td>
</tr>
<tr>
<td>4.3 Design</td>
<td>51</td>
</tr>
<tr>
<td>4.3.1 Assumptions</td>
<td>51</td>
</tr>
<tr>
<td>4.3.2 Attribute-based encryption</td>
<td>52</td>
</tr>
<tr>
<td>4.3.3 Priv.io building blocks</td>
<td>53</td>
</tr>
<tr>
<td>4.3.4 Priv.io operations</td>
<td>54</td>
</tr>
<tr>
<td>4.4 Third-party applications</td>
<td>57</td>
</tr>
<tr>
<td>4.4.1 Application API</td>
<td>57</td>
</tr>
<tr>
<td>4.4.2 Managing applications</td>
<td>59</td>
</tr>
<tr>
<td>4.4.3 Security and privacy</td>
<td>59</td>
</tr>
<tr>
<td>4.4.4 Limitations</td>
<td>60</td>
</tr>
<tr>
<td>4.4.5 Demonstration applications</td>
<td>60</td>
</tr>
<tr>
<td>4.5 Discussion</td>
<td>62</td>
</tr>
<tr>
<td>4.6 Evaluation</td>
<td>64</td>
</tr>
<tr>
<td>4.6.1 Microbenchmarks</td>
<td>66</td>
</tr>
<tr>
<td>4.6.2 User-perceived performance</td>
<td>67</td>
</tr>
<tr>
<td>4.6.3 Small-scale deployment</td>
<td>69</td>
</tr>
<tr>
<td>4.7 Summary</td>
<td>69</td>
</tr>
<tr>
<td>5 Picocenter: Rethinking computation</td>
<td>71</td>
</tr>
<tr>
<td>5.1 Motivation</td>
<td>72</td>
</tr>
<tr>
<td>5.2 Long-lived mostly-idle applications</td>
<td>74</td>
</tr>
<tr>
<td>5.3 Picocenter Architecture</td>
<td>75</td>
</tr>
<tr>
<td>5.3.1 Interface to cloud tenants</td>
<td>75</td>
</tr>
<tr>
<td>5.3.2 Challenges</td>
<td>76</td>
</tr>
<tr>
<td>5.3.3 Architecture overview</td>
<td>76</td>
</tr>
<tr>
<td>5.3.4 Hub</td>
<td>77</td>
</tr>
<tr>
<td>5.3.5 Worker</td>
<td>79</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Overview of how content is delivered in Maygh, implemented in JavaScript (JS). A client requesting content is connected to another client storing the content with the coordinator’s help. The content is transferred directly between clients.</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>Maygh messages sent when fetching an object in Maygh between two clients (peer-ids p&lt;sub&gt;id1&lt;/sub&gt; and p&lt;sub&gt;id2&lt;/sub&gt;). p&lt;sub&gt;id1&lt;/sub&gt; requests a peer storing content-hash obj-hash&lt;sub&gt;1&lt;/sub&gt;, and is given p&lt;sub&gt;id2&lt;/sub&gt;. The two clients then connect directly (with the coordinator’s assistance, using STUN if needed) to transfer the object.</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>Overview of how multiple coordinators work together on Maygh. The mapping from objects to lists of peers is distributed using consistent hashing, and the coordinator each peer is attached to is also stored in this list. The maximum number of lookups a coordinator must do to satisfy a request is two: one to determine a peer storing the requested item, and another to contact the coordinator that peer is attached to.</td>
<td>29</td>
</tr>
<tr>
<td>3.4</td>
<td>Average response time versus transaction rate for a single coordinator. The coordinator can support 454 transactions per second with under 15ms latency.</td>
<td>37</td>
</tr>
<tr>
<td>3.5</td>
<td>Average response time versus request rate for multiple coordinators working in tandem, in two different placements of the coordinators across machines. Close-to-linear scaling is observed as more coordinator nodes are added (the one-machine set of coordinators show lower performance after 16 coordinators are placed on a single machine due to the effects of hyperthreading).</td>
<td>38</td>
</tr>
<tr>
<td>3.6</td>
<td>Cumulative distribution of five-minute average bandwidth required at the operator for the same trace as Figure 3.7</td>
<td>40</td>
</tr>
<tr>
<td>3.7</td>
<td>TOP: Bandwidth required at the operator, as normal (with no plug-ins or Maygh), with a 10% plug-in deployment, and with Maygh; a 10% plug-in deployment results in 7.8% bandwidth savings, while Maygh results in 75% bandwidth savings. BOTTOM: Request rate observed at the Maygh coordinator; the rate is almost always below 800 requests per second, and is easily handled by a four-core server.</td>
<td>41</td>
</tr>
<tr>
<td>3.8</td>
<td>Network traffic at the operator, relative to the trace without Maygh, for different minimum image sizes. The traffic consists of three components: images that are served normally because they are too small, images that are served normally because Maygh cannot find an online client able to serve it, and overhead caused by Maygh (downloading the client code and protocol overhead). If Maygh serves all images, the operator would experience 75% less traffic due to image downloads.</td>
<td>42</td>
</tr>
</tbody>
</table>
3.9 Network traffic at the clients, comparing amount of data uploaded to downloaded.

We observe that the client workload distribution of Maygh is “fair”: most clients are requested to upload an amount of data that is proportional to the amount they download. 43

4.1 Complementary cumulative distributions of total monthly (a) storage, (b) bandwidth, (c) requests per user (top) and resulting costs (bottom) for Facebook, Twitter, and Flickr users (note the logarithmic scale on the y-axis). Also shown is the distribution of (d) total monthly costs. 99% of users would have to pay no more than $0.95 per month in all three scenarios. 50

4.2 Diagram of login process for user Alice in Priv.io. Alice visits https://priv.io/, obtaining the Priv.io JavaScript. Upon entering her username and password, her browser contacts her cloud provider S3, verifies her password, and communicates with her cloud provider as well as the cloud providers of her friends. Note that the only communication with the main Priv.io server is fetching the original JavaScript. 55

4.3 Diagram of how Newsfeed uses storage in Priv.io. Each user stores their own content, and the user who creates each thread stores a feed file linking together all comments. 61

4.4 Priv.io encryption and decryption performance when run in different browsers. Shown is (a) AES encryption and (b) AES decryption for objects of different sizes. Also shown is (c) ABE key generation for keys with an increasing number of attributes (ABE encryption under policies of increasing lengths shows very similar behavior). Upper three lines denotes results from mobile devices, while the lower three are from desktop browsers. 65

4.5 Average Newsfeed loading time with varying amounts of content, when content is loaded from multiple friends. We observe that the loading time increases linearly with the amount of content, as expected. 68

5.1 Picocenter represents a unique point in the space between existing abstractions by allowing users to run arbitrary processes that are, in expectation, long-lived yet mostly idle. 72

5.2 Overview of Picocenter, with Picocenter components shown in red. Tenants submit applications via the hub and receive a DNS name; the hub then assigns these applications to be run on workers. Clients lookup a Picocenter domain via the hub and then interact with applications. Workers may swap out inactive applications to cloud storage, and swap them in again later when requests arrive. 77

5.3 Timeline for serving incoming requests for applications, both inactive and active. Inactive applications are revived while the DNS response is sent to the client; all network traffic for the application goes directly to the worker. 79

5.4 Architecture for supporting partial swap ins, ActiveSet, and on-demand memory loading in Picocenter. Processes are revived with their memory image and filesystem on a FUSE filesystem; as memory requests are made, they are handed to our FUSE process which either serves them locally (for pre-fetched content) or downloads them from S3. 86
5.5 Microbenchmarks for our test application in Picocenter (VM+Containers), compared to a native process running in an IaaS VM. The error bars are standard deviations, and the y-axis is in log scale. Containers (and therefore Picocenter) show 1.4% overhead across all tests, and 6.4% overhead on the UDP ping test.

5.6 End-to-end client performance for different applications in ”(h)ot” (application live and running on a worker), ”(w)arm” (application swapped out on a worker’s local storage), and “(c)old” (application swapped out to cloud storage) states. The hub and workers are located in Virginia, and the clients are located in different locations around the world, with varying average ping times to the Picocenter instance: Virginia (VA, 0.258 ms), Oregon (OR, 80.9 ms), Frankfurt (DE, 88.7 ms), and Tokyo (JP, 149 ms). Error bars represent the 5th and 95th percentiles of the overall time to complete the client’s request. Also shown are the times spent waiting on S3 (downloading checkpoints) and CRIU (restoring the application). Overall, Picocenter provides good end-to-end application performance, with average restore times of 111 ms when applications are “warm” and 186 ms when applications are “cold.”

5.7 Response latency for our strawman application when comparing Picocenter’s ActiveSet technique to two baselines: fetching memory only with reactive paging from cloud storage (without ActiveSet), and downloading full application checkpoints every time. Our strawman application is configured with a 64 MB memory size, and we vary the working set size between 4 KB and 8 MB. Note that both axes are log scale. The ActiveSet technique significantly outperforms the baseline approaches.

5.8 Response latency for our strawman application when different fractions of the application’s working set is missing from the ActiveSet. Our strawman application is configured with a 64 MB memory size and a working set of 8 MB. The latency remains low, even when large fractions of the application’s memory have to be fetched from cloud storage.
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Breakdown of browsing traces from the top 100 Alexa web sites. &quot;Cacheable&quot; refers to the fraction of bytes that are cacheable according to the HTTP headers.</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Average time (ms) to load first / second 50 KB objects using Maygh with RTMFP.</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>Average time (ms) to load first / second 50 KB objects using Maygh with our proof-of-concept WebRTC implementation.</td>
<td>35</td>
</tr>
<tr>
<td>4.1</td>
<td>Summary of required features (storage, messaging, REST API, object versioning, DNS support, and authentication) in Priv.io, and their current support by major providers. Amazon, Azure, and Google support all required services today.</td>
<td>51</td>
</tr>
<tr>
<td>4.2</td>
<td>Notation used in the description of Priv.io.</td>
<td>53</td>
</tr>
<tr>
<td>4.3</td>
<td>Subset of the Priv.io application API, covering API permissions, user information, storage, and communication. All methods are protected by permissions (users must permit applications to make API calls).</td>
<td>58</td>
</tr>
<tr>
<td>4.4</td>
<td>Average time taken to generate an ABE master and public key (set-up) and decrypt an ABE message (decrypt) in various browsers.</td>
<td>66</td>
</tr>
<tr>
<td>4.5</td>
<td>Average time (in seconds) taken to load the basic Priv.io code after login, as well as the Newsfeed and Chat applications, in various browsers. Android Chrome and Mobile Safari do not support Flash, which is used to make cross domain requests for Amazon SQS.</td>
<td>67</td>
</tr>
</tbody>
</table>
Acknowledgments

I owe too much to my advisor, Alan Mislove, for his patience, encouragement, support, and guidance over these years. I would like to express my deepest gratitude to Alan, for his support in research and beyond. This work would not be possible without him. To be advised by Alan is the best fortune in my academic life. I would also like to thank my Master’s mentor, Timothy Bickmore, for his advice and support, especially in the first two years of my graduate study.

I also want to thank my thesis committee, Ravi Sundaram, David Choffnes, and Dave Levin, for their insightful feedback and encouragement. Not only do they serve on my thesis committee, but also they are great collaborators. Also, I want to thank all my collaborators, James Litton, Frank Cangialosi, Theophilus Benson, Tudor Dumitras, Aaron Schulman, Christo Wilson, Fangfei Zhou, Eric Franco, Richard Revis, and Xiaolu Zhou. I enjoy working with them a lot.

I thank everyone who helped in Maygh, Priv.io, and Picocenter: I thank the anonymous reviewers and our shepherds, Katerina Argyraki, Ben Zhao, and Evangelia Kalyvianaki, for their helpful comments. I also thank Avleen Vig for his assistance with the Etsy traces, David Blank-Edelman and Rajiv Shridhar for their assistance with the Northeastern traces, Bimal Viswanath for the use of MPI-SWS servers, and Michael Mislove for his assistance with the New Orleans latency experiments. In addition, I thank the developers of the open source ArcusNode project. I also thank the Priv.io users and beta testers for their participation and patience.

I would also like to thank all of the friends I have made at Northeastern for making my graduate life so wonderful.

Last but not least, I want to express my deep gratitude to my family, for their unconditional love. I want to thank my parents, Zhiping and Rong, for their support of my graduate study. Also, I must thank my wife Yingying, for taking care of me and encouraging me to explore the unknown.
Abstract of the Dissertation

Rethinking the Architecture of the Web

by

Liang Zhang

Doctor of Philosophy in Computer Science

Northeastern University, July 2016

Dr. Alan Mislove, Adviser

Over the past two decades, web technologies have evolved dramatically and have changed what we see and how we interact with the web. The web browser today works akin to a software distribution platform, running web applications written in JavaScript and rendered with HTML and CSS. Web services, ranging from online social networks, to video hubs, to e-commerce sites, enable content sharing for billions of people on an unprecedented scale. However, despite these massive changes, the web is still implicitly built around the premise of a client–server architecture, where web browsers are clients and service providers play the server role. Therefore, popular service providers face significant monetary burdens, but frequently keep their services free for users. Instead, the service providers often monetize the content that users upload to support advertising. As a result, the traditional client–server architecture of the web that is widely used today has significant implications for the privacy, security, and economics of the entire web ecosystem.

In this thesis, we aim to rethink the basic architecture of the web by leveraging the advances in web browsers and cloud services. This thesis focuses on three fundamental aspects of web systems design: first, how content is being served? We address low-cost, scalable content delivery problem with Maygh, a system that builds a content distribution network from client web browsers, without the need for additional plug-ins or client-side software. The result is an organically scalable system that distributes the cost of serving web content across the users of a website. Second, can we provide users with greater control and privacy over their data? We give users control of their data and allow data to be shared by introducing Priv.io, a new approach to building web-based services that let users take full control of their data. Priv.io composes web services with user-provided storage and stitches applications together using web browsers for computation. And third, how can users run long-lived computations while their web browsers may go on and offline? To address the problem, we develop Picocenter, a hosting infrastructure that supports long-lived, mostly-idle applications
in cloud environments. As a result, Picocenter enables new types of applications that give users better control over their data while maintaining low running expenses. Taken together, the techniques in this thesis relax the strict client–server architecture of the web to allow end users to contribute resources, thereby better matching today’s workloads and enhancing user privacy.
Chapter 1

Introduction

The World Wide Web, also known as the web, is one of the most popular services on the internet today. Billions of internet users now have access to a wide range of web services, including news, search, communication, social networking, content sharing and more. For example, YouTube, a popular video-sharing website, has over a billion users, who watch hundreds of millions of hours every day. In addition, web clients sent 30% of emails in 2014, and e-commerce accounts for 7.8% of total retail sales today.

The recent rise in smartphones, PDAs, and tablet computers has expanded the web from traditional desktop computers to mobile devices. As of 2015, mobile devices accounted for 55.7% of all traffic of the leading websites in the U.S. Although some developers embrace native mobile apps for their performance and operating system services, web applications still attract users and developers for an installation-free experience and cross-platform support. Also, the web standards have encompassed many new features to overcome its shortcomings, such as application caches and location services.

Over the course of the 27 years that it has been in existence, the web has been through a series of technological innovations. From the first early static websites to today’s dynamic web applications, many technologies have been developed for the web to support its diverse applications. To understand why the web is designed as it is, we first provide a brief historical overview of the technological innovations that have enabled the web today.

The web started as a “static” system that consisted of two components: web browsers and web servers. Web servers were originally implemented by simply mapping web requests to files in local file systems and returning the static content to web browsers. Web browsers were implemented to render static web pages, fetching content from web servers and delivering it to users. Unlike other
files, web pages could include external content (e.g., images and audio) to be displayed on screen, and allowed users to navigate to other web pages via hyperlinks. However, in this first iteration of the web, users only had limited interactivity in the web, such as clicking links to navigate web pages. This simple static, client–server model proved to be insufficient to do many things people wanted to do, so it was extended with five key technological innovations, involving in both client-side and server-side evolutions.

On the client side, there are three key innovations. First, client-side programming was enabled with JavaScript. To address the lack of user interactivity in static web pages, JavaScript was introduced to add programmability to web pages running on the web browser. With JavaScript, web developers can dynamically insert, remove and modify HTML elements to respond to user events (e.g., mouse clicks, keyboard typing, and timer events) on web pages. As a result, browsing the web is no longer a static reading experience, but a rich, dynamic interaction; web pages have evolved from being static HTML to programs. Second, web pages became dynamic. At first, users had to explicitly refresh web pages to get updated content, because web pages are static. To address this issue, Document Object Model (DOM) and XMLHttpRequest were introduced to allow client JavaScript programs to fetch data and update web page content dynamically without refreshing. DOM defines an API for client scripts to interact with objects in web pages, and XMLHttpRequest is an API for client-side scripting languages to download data from or upload data to servers asynchronously. Due to these technologies, new web applications, such as Gmail and Twitter, have rich user experience and active user participation. Third, mobile devices gained desktop-class browsers. Mobile browsers used to support a subset of the HTML specification, and they rendered web pages differently due to the constrained mobile environment where storage, computation power, and network connectivity were limited. However, today, mobile devices are advanced enough to run desktop-class web browsers. In other words, web developers only need to apply the standard web technologies to build web applications that run on both desktop computers and mobile devices.

On the server side, there are two key innovations. First, dynamic web pages are generated with server-side web application frameworks. Although serving static web content plays an important part in web servers, generating dynamic web pages to cater individual user requests has become indispensable for building web applications, such as a customized Yahoo! portal home page. To address this, web developers invented server-side web application frameworks (e.g., CGI, PHP, Django, and Ruby on Rails) to boost development for web applications. These frameworks provide standard modules, including web session tracking, user cookie management, and object-to-relation database mapping. As a result, the server-side program became more complicated:
CHAPTER 1. INTRODUCTION

it evolved from organizing static files in local file systems to writing a program for a website. Second, web content is distributed via content distribution networks (CDNs). As more and more users from different geographic locations obtained access to the web, websites had two significant problems in distributing content: first, the workload for popular websites became more than a single group of servers could handle; and second, the latency for users to download web pages became an issue as web pages became more complex. As studies have shown, web content popularity follows Zipf’s law; web caching can provide tremendous benefits [66]. CDNs represent an elegant solution to both of these problems by serving as a distributed cache: a CDN system places proxy servers in geographically distributed data centers to cache web content. Instead of fetching the content from original servers, CDNs allow users to fetch content from proxy servers at a closer network distance.

All of these innovations together enable new types of web applications. For example, Facebook, a popular online social network (OSN) that has more than a billion daily active users, generates dynamic personalized content with PHP web servers, engages both desktop and mobile users with an interactive interface written in JavaScript with extensive use of dynamic web page techniques, and distributes significant amounts of user generated content (e.g., images and videos) with CDNs. In addition, this interactive web experience attracts users to communication sites (e.g., Gmail, Yahoo Mail and Slack), content sharing sites (e.g., videos on YouTube, photos on Flickr, and slides on SlideShare), and online collaboration sites (e.g., Wikipedia, a free-of-cost collaborated encyclopedia). User participation and collaboration are two distinctive properties of these web applications. Users actively produce content and consume content on these websites. In contrast to centralized content provider sites (e.g., CNN, New York Times and Yahoo), most content on OSNs and content sharing sites comes from the users themselves. These sophisticated multimedia, user-engaged applications used to be a privilege of desktop applications, but with recent web technology evolution, a web browser is the only software that a user needs for using these applications. These new web services are fundamentally different from the ones before; they have built rich client-side interactive interfaces in web browsers, served dynamically generated content, and distributed small to large data across users globally.

This is the web today: rich, dynamic, interactive web applications over both desktop and mobile browsers. Despite all these changes, the web still client–server. This has two significant implications. First, the architecture places significant monetary burden on the service providers, but does not place the same burden on clients. Users only pay for their bandwidth and devices to browse the web, but service providers must pay for the bandwidth and the necessary infrastructure to support all users coming to their sites. Most of these sites are “free”, and resolve to advertising for support.
CHAPTER 1. INTRODUCTION

their services. This model has leaded to complex tracking system that people have privacy concerns about [78][17]. Second, many services today rely on user-generated data, which contains privacy-sensitive information about users, such as photos and tweets. Because of the client–server architecture, the service provider sees all user data and activities, leading to many concerns. For example, data leaks to third parties [155][200]. Given these implications and concerns, this thesis focuses on three areas of the web: content distribution, user privacy, and beyond web browsers.

- **Content distribution**  Despite the dynamic nature of websites today, the serving infrastructure still follows the client–server architecture, i.e., web browsers can only fetch content from web servers. Although CDNs have been extensively used to help websites to distribute data, the situation is getting worse because more content is being increasingly uploaded to these services due to the popularity of OSNs [104][214]. Web caches and content distribution networks (CDNs) have been shown to exhibit poorer performance on OSN content [76][246], causing many OSNs to move from CDNs to highly-engineered in-house solutions [111][136][225]. As new web applications encourage user participation, both online social activities and user-generated content result in a workload that is challenging for CDN systems. As a result, website operators who wish to serve many users have a significant monetary burden in building their own content distribution systems or buying content distribution services from CDN providers.

- **User privacy**  Users today upload a wide variety of content to a broad range of free websites. In order to offer a service for free, service providers typically monetize user content, selling user data to third parties such as advertisers. Once the data is stored in service providers’ servers, users typically have no option to prevent service providers from accessing it. This can have negative consequences for users. For example, users have found it difficult to retrieve all of their data from providers due to services shutting down [154][158]. Furthermore, personal data may leak to the hackers or even the public because of security breaches in these websites [31][208]. As a result, by using these “free” web services, users have little control over their data or privacy.

- **Beyond web browsers**  Many web applications today, such as Gmail, offer offline mode, so that users can keep working on their local copy of the data when browsers are offline. Furthermore, some web applications (e.g., TiddlyWiki, a personal wiki) run entirely in the web browser and save data in local storage. These client-side only applications help users protect their privacy and data security, but most applications today still need dedicated long-
CHAPTER 1. INTRODUCTION

lived server-side processes for background processing, listening on incoming connections, synchronizing across user devices, and so on. However, user may need to manage their servers to run these applications, or rewrite them for customized programming environments (e.g., Google AppEngine or Amazon Lambda).

In this thesis, I aim to answer the question: can I relax the strict client–server architecture of the web to allow end users to contribute resources, thereby better matching today’s workloads and enhancing user privacy? The way I address this is by taking advantage of two recent trends. The first trend is the advanced development of web browsers. In addition to the innovations discussed above, current HTML5 standard introduces WebRTC and secure browser sandbox. The former technology implements direct browser-to-browser networks, which supports data exchanges at the edge without going through web servers. The latter creates isolated environments in the web browser for running untrusted code. The second trend lies in the rapid growth of cloud computing. Users today can buy storage, bandwidth, and computation from cloud providers at a fine granularity. Furthermore, to control these services, cloud providers offer APIs that are accessible in JavaScript running in web browsers.

1.1 Contributions

This thesis makes contributions in three areas of the web architecture: content distribution, data control, and cloud computation for end users; each one of these is described in more detail below.

Maygh: Rethinking web CDN We first aim at building a scalable content distribution system. Most of the popular websites today are operated by well-funded organizations, such as companies and universities. Content sharing and distribution places a substantial monetary burden on the website operator, who is required to supply the resources (serving infrastructure and network bandwidth) necessary to serve content to each requesting client. Spreading the distribution costs among unmodified user browsers—the idea that underlies the success of peer-to-peer systems such as BitTorrent [74]—is difficult over the web, as the web was designed with a fundamentally client–server architecture.

To solve this problem, we have developed Maygh, a system that builds a content distribution network from client web browsers, without the need for additional plug-ins or client-side software. Users in Maygh send the content directly to others, relieving the site operator from having to serve the
CHAPTER 1. INTRODUCTION

content. By recruiting users’ web browsers, Maygh is an organically scalable system that distributes the cost of serving web content across the users of a website. Maygh enables web browsers to assist in content distribution using two techniques. First, Maygh uses the storage APIs supported by modern browsers to store content persistently on end user machines. Second, Maygh uses newly available browser-to-browser communication techniques to enable direct message exchange with other browsers. Maygh uses centralized coordinators to track the static content stored in browsers, serving as a directory for finding content.

Priv.io: Rethinking web services While Maygh results in a more scalable system, its primary beneficiaries are the service providers; Maygh does not change the fact that these service providers’ servers still receive all of the data that users upload. In fact, most of the free web-based services operate under a similar model: Users entrust the service provider with their personal information and content (e.g., their comments, photos, political and religious views, sexual orientation, occupations, identities of friends). In return, the service provider makes its service available for free by monetizing the user-provided information and selling user data to third parties (e.g., advertisers). In this model, users today have limited options of making their data private from the service provider.

To address this problem, we have built and deployed Priv.io, an alternate approach to implementing web-based services that provides users with control over their data, enhances privacy, and ensures practicality and reliability at low cost. In Priv.io, each user provides resources necessary to support their use of the service by purchasing computing resources (storage, bandwidth, and messaging) from cloud providers such as Amazon Web Services or Windows Azure. Thus, users retain control over their own data. Meanwhile, users still access the service using a web browser, and all computation is done within users’ browsers. Priv.io provides rich and secure support for third-party applications with social interactions like Facebook or Flickr. As a result, Priv.io provides an alternative model for users to pay web-based services without giving up their privacy.

Picocenter: Rethinking web computation In Priv.io, all computation on behalf of a user is done in the user’s browser (thus, keeping the user’s data private). Unfortunately, this means that a user can only participate when his or her browser is online; this dramatically limits the kinds of applications that can be built using Priv.io. Recent progress on open source alternatives of popular web services indicates users’ interest in hosting applications for themselves. The type of application that users are often interested to run can be described as being long-lived but mostly idle (we refer to these as LLMI), including personal email servers or web servers, or decentralized
social network nodes. These applications should be available 24 hours a day, but the vast majority of that time will be spent idly waiting for incoming messages.

LLMI applications make different assumptions from the ones that make by cloud providers. Traditional cloud providers assume applications are always active; this leads to ineffective use of resources, which in turn yields economic inefficiencies. Running LLMI applications in today’s cloud would require consuming a VM’s worth of resources in the cloud: resources the user must pay for, despite the fact that they would go unused the vast majority of the time. Moreover, because VMs consume resources whether active or not, many LLMI applications would limit the number of total jobs that the cloud provider could accept at any point in time. As a result, although end-users would benefit from being able to run LLMI applications in the cloud, they generally do not: today, long-lived cloud jobs are typically either mostly active or run by well-funded corporations.

To address this limitation, we develop Picocenter, a hosting infrastructure that provides a process-like abstraction to support LLMI applications in the cloud today. Compared to similar approaches such as Docker, Google App Engine and Amazon Lambda, Picocenter offers support for arbitrary computation and event handling (Lambda only supports a pre-defined set of event triggers), strong isolation between processes, and native support for long-lived processes that are automatically swapped to cloud storage when not active. Moreover, Picocenter can be deployed on top of today’s clouds, enabling incremental deployment through “resellers” of traditional virtual-machine-based clouds.

All together, this thesis is a first step towards decentralizing the client–server web architecture. We demonstrate three different approaches to let users contribute computing resources, thereby reducing the burden of service providers, and enhancing user data security and privacy in the web ecosystem. This thesis gives users who enjoy web services today or wish to control their services alternative system designs that work with today’s web infrastructure, run in modern web browsers without installation of additional software, and are compatible with legacy applications.

1.2 Outline

The rest of the thesis is organized as follows. Chapter provides background on the web in three areas, web browsers, JavaScript, and web servers. Chapter presents our work on Maygh, Priv.io, and Picocenter respectively. Chapter covers related work. Finally, we conclude this thesis in Chapter.
Chapter 2

Background

In this chapter, we provide the necessary background for this thesis. We start in the web browser, and discuss the technology advancements that enable the work in this thesis. Next, we review JavaScript, which is the language of the web. JavaScript began as an interpreted language, but had evolved to a powerful language that serves in the browser as well as the server. We then discuss about the dynamic content generation in web application servers, and the privacy concerns in recent web services. Last, we will discuss the rapid development in cloud services.

2.1 Web browsers

Web browsers are the most common tool for users to navigate the web. A web browser downloads web resources via the Hypertext Transfer Protocol (HTTP) protocol [100], then renders them on screen for users to read. Most web resources downloaded by the web browser are web pages. A web page is a text file written in HyperText Markup Language (HTML) [13], which is maintained by the World Wide Web Consortium (W3C) [26]. Despite the fact that web pages only have plain text data, they can include various types of resources from other locations, such as images from other websites. Other common resources are Cascading Style Sheets (CSS) [8] and JavaScript programs [30]. The former describes the presentation of a web page, and the latter allows developers to build programs running on the page in the web browser.

Due to the composition of external resources, the browser has to download all dependent resources for the user to see the complete look of the page. One immediate consequence of this design is that the page load time (PLT) increases as the number of external objects increases. The speed of opening web pages affects user experience in browsing the web [33, 161]. As a result, web browsers and web
CHAPTER 2. BACKGROUND

pages both try to optimize PLT [186][211][234]. For example, web browsers often load objects that are immediately visible on the screen first [186][211]; and web pages work to reduce the number of external objects. Also, we will discuss content distribution systems, such as CDNs that lower network latency, for boosting PLT in later sections.

Early web browsers delivered a “static” user experience that web pages were composed from static content (e.g., text and images) and users originally had a few interactions on these pages. This soon became a limitation as the web expanded its applications beyond the simple content presentation. To address this limitation, people had developed many technologies to extend the web. In the rest of this section, we will provide the background about the web browser’s four key technological innovations: dynamic web pages, HTML5, browser plug-ins, and mobile web browsers.

2.1.1 Dynamic web pages

In the early days of the web, users had to instruct web browsers to refresh a web page in order to get updates. Refreshing a web page involves a series of actions, including cleaning content on the screen, downloading new content, and rendering new content on the screen. In the refreshing process, the user cannot do anything but wait. Even worse is that the web page may not have any update when the refresh request arrives. To reduce the user waiting time, the web browser could send the local cached content’s timestamp via the HTTP if-modified-since header to the server; the server will return a short response of no modifications instead of the whole content if the content has no changes since the timestamp. However, this method only reduces time in downloading content; web browsers still have to check every embedded object even if only a small fraction has changed.

This issue is addressed by two new technologies in the browser: Dynamic HTML (DHTML) [10] and Asynchronous JavaScript and XML (Ajax) [130]. DHTML introduces the document object model (DOM) [144], a programming API for client-side JavaScript programs to modify HTML elements on a web page without refreshing the whole page. Also, the DOM allows JavaScript to receive user input, including events like clicking on an element, page scrolling, mouse moving and more. With DHTML, web developers are able to develop rich client-side user interactive applications without connecting to the server. For example, Maygh uses DOM to insert content loaded from other users to the web page. Meanwhile, Ajax introduces the XMLHttpRequest [226] object, an API for client scripts to transfer data between the server and the web page without user involvement in the browser. Although it was first designed to handle XML data as the name suggests, today it can process arbitrary objects. For instance, Priv.io loads various objects (e.g., text and images) from user
CHAPTER 2. BACKGROUND

provided storage via Ajax. As a result, web developers write new web pages with these techniques to update content in the browser dynamically.

With DHTML and Ajax, web pages have become dynamic. Compared to the web before it, websites built with these technologies have a rich user experience, and are responsive to users’ input. For example, websites often load content on-demand (e.g., users click buttons to load more content) instead of pushing everything to the browser all at once. Also, developers build new web applications with rich user interactions, bringing the experience of desktop applications to the web. In fact, many traditional desktop applications are being rebuilt for the web, thus becoming web applications. For example, Gmail’s web interface has, for many users, replaced more traditional email clients. The web application model simplifies the software deployment, because users only need web browsers to access their services. Also, web applications enable easy cross-platform development, since most platforms share the same web standard.

2.1.2 Browser plug-ins

Dynamic web pages open new possibilities to the web: massive user collaborations, simple deployment of web applications, and rich user interactions. However, they still fall short in some areas. For example, web applications cannot store data in local machines, instead, they solely rely on web servers to save data. If the connection between the browser and the server is broken, then the web application stops working. In addition, web browsers are designed to connect only to web servers because of the web’s client–server architecture. Thus, web browsers cannot listen to incoming connections, nor connect to other browsers directly.

To overcome these limitations, developers start extending web browsers via their plug-ins mechanisms, such as Netscape Plugin Application Programming Interface (NPAPI) [17] and Microsoft ActiveX [15]. Browser plug-ins allow web browsers to support features that are not defined in the web standards. For example, Flash Player [1] plays multimedia files and vector graphic animation on web pages in the browser. It later added peer-to-peer networking support for media streaming and data exchange. In Chapter 3, we leverage this feature to build a content distribution system by recruiting web browsers to help content exchange at the edge. Browser plug-ins bridge the gap between native applications and web applications. From the browser side, web developers can instruct client scripts to access functions exposed by the plug-in; on the plug-in side, they are native OS processes. Browser plug-ins are a way for web applications to access local operating system services.
CHAPTER 2. BACKGROUND

However, browser plug-ins are additional pieces of software that do not come by default with web browsers. Thus, users are responsible for the installation and maintenance of browser plug-ins. In fact, users often need to apply security patches for browser plug-ins [2], because security vulnerabilities in browser plug-ins could compromise the host operating system [87]. Due to the need of user involvement and security concerns, most browser plug-ins have a low penetration rate. Furthermore, many features (e.g., video playback and local storage) that used to be only found on browser plug-ins became standard in HTML5 [12] so that users have less need to install browser plug-ins.

2.1.3 HTML5

HTML5 [12] is the current version of the HTML standard. It introduces many new features (e.g., peer-to-peer networking and storage) to the web platform with an emphasis on programmability. In other words, web developers not only can use these new features as declarative HTML tags, but also control them with client-side scripts. HTML5 is one of the fundamental building blocks that we use for developing a web content distribution system (Chapter 3) and privacy-protected web services (Chapter 4). Below, we briefly overview features from the following three areas that related to this thesis: storage, communication, and computation.

- **Storage**  Web browsers cache content while users browse the web, but this caching system is not accessible to client-side scripts. To address this, HTML5 introduces Web Storage (a key-value storage) [239], IndexedDB (a transactional database) [236], and File (an API for file operations) [235]. These APIs allow client-side scripts to save data in local file systems so that web applications can reload data across browser sessions. We use these APIs for building a persistent content storage in Section 3.3.1.1.

- **Communication**  In the traditional web communication model, web servers passively wait for client requests. In other words, web servers cannot actively push content to clients, and clients cannot directly communicate with other clients. HTML5 introduces two new techniques: WebSocket [98] and WebRTC [28]. WebSocket provides secure full-duplex transmission channels between the web server and the client. Meanwhile, WebRTC enables peer-to-peer networking for real-time video and audio communication. It also has a data channel for general data exchange between browsers. We leverage the data channel for direct browser-to-browser content exchange in Section 3.3.1.2.
CHAPTER 2. BACKGROUND

- **Computation** JavaScript programs that run on web pages or iframes are single-threaded. The web page may become unresponsive if a JavaScript function is taking a long time. To address this, HTML5 introduces WebWorker [27] for executing long-running scripts without interrupting scripts that respond to user interactions. In addition, WebWorker defines shared workers for sharing computation threads across browser tabs. We use this feature for sharing browser-to-browser networking across multiple browser tabs as described in Section 3.3.3.

HTML5 supersedes many features that only browser plug-ins (e.g., Flash Player, Media Player and QuickTime) used to provide. From the developer’s point of view, HTML5 applications are more capable than the web applications before, and they are more akin to native applications than classic web page models. From the user’s point of view, HTML5 web applications work out-of-the-box as long as the user runs them in a modern web browser. Users do not need to worry about plug-in installation and maintenance. In fact, web applications not only attract traditional desktop users, but also have been adopted by mobile users, because most mobile platforms today have HTML5-compatible web browsers as well.

2.1.4 Mobile web browsers

From smartphones to tablet computers, mobile devices of different sizes and shapes now account for more than half of all web traffic in the U.S. [151]. Driving this massive web traffic, mobile web browsers work in a challenging environment. Mobile devices are usually powered by a battery, which means power consumption is critical in mobile computing. Also, compared to wired connections, mobile networks have different characteristics (e.g., higher network latency, smaller bandwidth, and higher packet loss rate).

Because of the constraints of mobile devices, early mobile web browsers behaved differently than desktop browsers. Depending on the mobile platform, mobile web browsers might run in constrained programming environments (e.g., J2ME [14], where they did not have access to all device resources). Mobile web browsers had developed techniques to save bandwidth and power. For example, Opera Mini [19] used proxy servers to reformat web pages and reduce image size. In the past decade, mobile devices have gained significant performance upgrades. Most mobile devices now run full-fledged web browsers, supporting the same HTML standard (e.g., HTML5) as their desktop counterparts. Meanwhile, developers, who wish to build web applications that run on mobile devices, still need to pay attention to peculiarities (e.g., screen sizes and device orientations) on mobile web browsers. Due to the development of client-side JavaScript frameworks, such as Bootstrap [6] and Backbone.js [4].
many web applications today can run naturally on both desktop and mobile devices. We develop Priv.io (Chapter 4) with the Bootstrap framework for creating an interactive user interface for both desktop and mobile users.

2.2 JavaScript

JavaScript is the de facto standard client-side scripting language on the web. First developed by Netscape [18], it was then adopted by other browser vendors. JavaScript was first designed for developers to program HTML objects dynamically on web pages. Typical use cases are client-side input validation, animation on web pages, and data transformation. Since the introduction of dynamic web pages, developers use JavaScript extensively for building web applications with rich user experiences that run on web browsers. JavaScript is the language that glues various web technologies, such as the DOM and XMLHttpRequests, together for creating interactive client-side applications. Gmail and Google Docs are examples of these complex web interface applications that run in JavaScript. Today, JavaScript is a critical component of the web, as many web applications simply cannot work without JavaScript.

JavaScript started as an interpreted language that ran on web browsers. Although it is easier to implement in this way, the interpreted execution model led to slow JavaScript performance. As client-side JavaScript programs get bigger in size and more complex in the web applications, the JavaScript engine needs to run code more efficiently. Over the past few years, many efforts, such as just-in-time compilation [35] and type systems [63, 228], have been made to JavaScript for performance improvements. Today, JavaScript performance is on the same order as other non-static typed programming languages [20, 22, 23] that are actively used in production environments. This thesis benefits from the JavaScript performance improvements. Our complex web service system Priv.io (Chapter 4) uses it not only to build the web interface but also to perform intensive cryptographic computations implemented in JavaScript, all in client-side web browsers.

2.2.1 The web and beyond

In order to run programs written in non-JavaScript languages on web browsers, developers have created new tools to transform traditional programs to the web by using web technologies. For example, Emscripten implements a JavaScript compiler back-end for the LLVM [140] compiler tool chain, so that any programming language that can be compiled to the LLVM intermediate
language can be compiled to JavaScript. Beside compilation, Emscripten has a POSIX emulated environment that runs on web browsers. As a result, POSIX applications can be recompiled to run on web browsers as well. Many projects\(^1\) have been ported to the web by Emscripten. In Section 4.6, we use Emscripten to compile Priv.io’s core cryptographic operations to JavaScript from the Ciphertext Policy ABE library \(^84\), which is written in C.

Due to the popularity of Emscripten, new techniques have been proposed for JavaScript performance optimization. asm.js \(^{29}\) defines a low-level subset of JavaScript that is similar to assembly language. Emscripten’s compilation output falls in this JavaScript subset. Because asm.js employs a simple execution model, JavaScript engines can further optimize the code for better performance. Meanwhile, SIMD.js \(^{108}\) aims to add Single Instruction Multiple Data (SIMD) types and operations to JavaScript, so that JavaScript can leverage related CPU instructions for a performance boost. These new JavaScript optimization techniques will have a significant impact on computation-intensive web applications such as Priv.io, because they archive close-to-native performance of JavaScript programs.

Although JavaScript was developed for browser-side scripting, it has made its way to other areas as well. For example, Node.js \(^{164}\) takes the JavaScript engine from the web browser, and applies it to the server-side web application development. Node.js allows JavaScript programs to access native operating system functions, such as listening to network sockets, accessing local file systems, and managing native processes. We build Maygh’s coordinator server (Section 3.3.4) on Node.js for its simplicity in network programming and active community support for browser-to-browser communications.

### 2.3 Web servers

While web browsers render content on the screen for users on the client-side, web servers serve this content on the server-side. In the early days of the web, web servers were more akin to file servers; each web page was mapped to a file on local file systems. As users demanded more dynamic content, static web servers could not meet the demand; new server-side technologies emerged for building dynamic web applications for massive users. In this section, we will cover the key technologies on the server-side for dynamic web applications, and the privacy concerns that come with it.

\(^1\)Emscripten keeps a list of applications that use Emscripten compilation techniques at https://github.com/kripken/emscripten/wiki/Porting-Examples-and-Demos
CHAPTER 2. BACKGROUND

2.3.1 Web application servers

Compared to static web servers which only serve static content, such as images and web objects (e.g., HTML, CSS, and JavaScript files), web application servers are built to serve dynamically generated content. To support complex web application today, web application servers have been through a series of technological evolutions. Early web server programming interfaces, such as Common Gateway Interface (CGI) [7], allowed native processes written in scripting languages (e.g., Perl [21] or Bash [5]) or compiled languages (e.g., C or C++) to handle web requests dynamically. However, these early web server programming models only had limited libraries in helping parsing web requests and generating web responses. This soon cannot meet the fast changing demands in developing server-side web applications.

To simplify web application development, developers have invented web application frameworks (e.g., PHP [20], Django [9], and Ruby on Rails [24]). These frameworks provide many features, including handling HTTP requests, templates, object-relational mapping, data binding and so on. One of the key features of these frameworks is browser session tracking. Because HTTP is a stateless protocol, the session tracking feature helps developers to save client states in building web applications. This feature also places a burden on the server-side, because the server now has to save all session states.

Since the rise of Ajax, client-side programs that run in the web browser keep their own states for rich user interactions. These rich interactive applications make tracking browser states less important for the server. As a result, Representational State Transfer (REST) [220] has been introduced to relax the burden of saving browser states in the server by asking clients to provide necessary information for fulfilling service requests. Many web services today allow developers to access their RESTful services via JavaScript programs running in the browser. Priv.io (Chapter 4) calls Amazon S3’s [55] RESTful storage APIs from the client-side JavaScript programs for accessing user-provided storage space.

2.3.2 Privacy

Since the rise of online social networks (OSNs), user-generated content and content sharing have become increasingly popular among websites. In particular, OSNs, such as Facebook [103] and Twitter [222], and content sharing services, such as Flickr [107] and YouTube [240], have attracted massive numbers of users to create and share a significant amount of content on their websites. On these websites, users are both content consumers and content creators at the same time. In contrast to
CHAPTER 2. BACKGROUND

traditional websites, most content on OSNs and content sharing sites comes from their users. Not only these websites collect user generated content, but they also track user activities across different websites with cookies [32], which is a small piece of data that web browsers send to servers for tracking user sessions. The reason why these websites obsess over user data is that many of them rely on advertising to provide monetary support for their services.

Collections of sensitive user data leads to user privacy concerns. User data can be used in many areas, for example, personalized recommendations (e.g., recommend a movie to user’s taste), targeted ads (e.g., display baby products to whom is going to be parents), public politics (e.g., predict elections) and much more. Once service providers have collected user data, users no longer have control over how service providers use them. Service providers may sell user ads, sell data to third parties, or leak data to the public if their systems are hacked. Today, users have difficulties in hiding their data from the service provider. To allow users to use the web services today while protecting their privacy, we developed Priv.io (Chapter 4) to protect user data and privacy by isolating service providers from directly accessing user data.

2.4 Cloud services

Today, website operators often host their services by renting computing resources from third-party cloud computing service providers, or simply the cloud. Cloud computing services have commoditized computing resources of various types, from early virtualized hardware resources (for example, virtual machines), to large data processing systems (for example, Amazon EMR [3], a hosted Hadoop framework). Compared to an in-house hosting infrastructure, cloud vendors have built their services for redundancy, scale, resilience, and security. Cloud computing has thus had to evolve to meet the increasingly wide range of user applications. Based on a few business models, cloud computing is organized into the following three types.

- **Software as a Service**  Since the introduction of dynamic web pages, many traditional applications have migrated to the web, offering as services, for example, Google Docs and Gmail. This model is called Software as a Service (SaaS). Unlike traditional desktop applications, applications employing the SaaS model do no sell software copies. Instead, SaaS providers manage applications on their servers, providing resources, such as storage, bandwidth, and computation, to host their services. In this model, users are free from the management burden
of software installation and maintenance, and simply access these services via the web browser. Also, users only need to pay for the resources they use, or a monthly subscription.

- **Platform as a Service**  Platform as a service (PaaS) vendors offer a development environment and hosting infrastructure for application developers to build and run their services. For example, Google App Engine [116] provides APIs for logging, search, and database access for a few supported programming languages (e.g., Python, Java, and Go). Behind these APIs, PaaS vendors optimize their services for scalability and efficiency, so that application developers can focus on building business logic, and scale their applications as needed. Compared to monthly subscription model of traditional web hosting services, PaaS changes users at a fine granularity. For example, users pay for the amount of storage spaces, queries of databases, bandwidth, and web requests.

- **Infrastructure as a Service**  Despite PaaS’s diverse APIs, applications, which require specific operating system environment or have to access to hardware, cannot run in PaaS’s customized environment. Due to hardware virtualization techniques, Infrastructure as a Service (IaaS) vendors sell their hardware resources as virtual instances, including virtual machines for computation, virtual routers for networking, and virtual disks for storage. By renting virtual hardware resources, users install operating systems, customize the environment, and maintain software stack.

The cloud makes web application development and operation easier. By relieving developers from having to build their own infrastructure, they can buy well-operated computing, storage, and networking services as needed. Also, the cloud has elasticity in scaling: as the number of users increase, developers can simply buy more cloud resources to support the growing demand. By renting virtual machines from cloud providers (e.g., Amazon AWS [57] and Google Cloud Platform [11]) based on demand, Picocenter (Chapter 5) provides elastic computing resources with a process-like running environment for end users to execute their programs in the cloud.

### 2.4.1 Content distribution networks (CDNs)

In the early days of the web, a small number of web servers could handle the light workload of a website. As the web became popular, websites, such as Yahoo and CNN, who generated lots of content, had attracted many users to browse their content on the web. Soon, the simple cluster of web servers could not fulfill the demand of serving content to massive users. Meanwhile, the
CHAPTER 2. BACKGROUND

Latency for users to download content had become an issue as web pages become more complex. As studies demonstrated that web content popularity followed Zipf’s law [66], web caching can provide tremendous benefits [66]. CDNs, as an example of early cloud services, rose to serve the content to users globally. A CDN system places proxy servers in geographically distributed data centers to cache web content. Instead of fetching the content from original servers, CDNs allow users to fetch content from proxy servers at a closer network distance.

Today, OSNs and personalized content recommendation derive decentralized workload patterns. Users are more likely to watch content from their friends or family, and also each user may have vastly different recommend content. As a result, CDNs find it hard to choose popular content to cache. Moreover, it is not realistic to cache all content in CDNs, because CDNs have limited storage space, but at the same time, there is a massive amount of user-generated content that needs to be served. For large well-funded organizations, they can pay CDN providers, such as Akamai [49] or Limelight [146] to deliver content for them, or they can build their own content distribution solutions. However, for other operators who do not have the resources find it difficult to serve a large number of users. In this thesis, we build Maygh (Chapter 3) to serve content by recruiting users’ web browsers to help content distribution. Maygh is an organically scalable system that distributes the cost of serving web content across the users of a website.
Chapter 3

Maygh: Rethinking web content distribution

Historically, the web was assumed to be client–server, placing the burden for distributing content on the service providers. In this chapter, we present the design and evaluation of Maygh, a system that automatically builds a CDN from visiting users’ unmodified web browsers. With Maygh, users visiting the operator’s web site automatically assist in serving static content to others for the duration of their visit; once they leave the site, this assistance ceases. Maygh requires no client-side changes and only minor modifications to the operator’s site, and can be deployed on the web sites and web browsers of today. The result is an organically scalable system that distributes the cost of serving web content across the users of a web site. We first presented Maygh as a conference paper in EuroSys’13 [245].

3.1 Motivation

Over the past two decades, the web has enabled content sharing at a massive scale. Unfortunately, the architecture of the web places substantial monetary burden on the web site operator, who is required to supply the resources (serving infrastructure and network bandwidth) necessary to serve content to each requesting client. Spreading the distribution costs among the users—the idea that underlies the success of peer-to-peer systems such as BitTorrent [74]—is difficult over the web, as the web was designed with a fundamentally client–server architecture. This situation exists despite the fact that significant amounts of web content are now generated by end users at the edge of the network, fueled by the popularity of online social networks, web-based video sites, and the ease
CHAPTER 3. MAYGH: RETHINKING WEB CONTENT DISTRIBUTION

of content creation via digital cameras and smartphones. Web site operators who wish to serve a large number of users are forced to make substantial investments in serving infrastructure or cloud computing resources, or pay content distribution networks (CDNs) such as Akamai or Limelight, to serve content.

Recent approaches have worked towards overcoming these limitations, aiming to allow end users to help the web site operator distribute the static objects (e.g., images, videos, SWF) that make up web pages. Examples include Akamai’s NetSession [42, 50], Firecoral [217], Flower-CDN [89], BuddyWeb [233], and Web2Peer [189]. All of these systems are built either using browser plug-ins that the user must install, or client-side software that the user must download and run. As a result, the set of users who can take advantage of these systems is limited to those that download the software or plug-ins; clients have little incentive to install the plug-in or software, and clients without it are served using existing techniques. With the most popular [159] of all FireFox plug-ins (Adblock Plus) being installed by only 4.2% of FireFox users, such techniques are still likely to provide only modest gains in practice.

In this chapter, we rethink the web content distribution system by leveraging recent technological innovations in the web browser. We build and evaluate Maygh, a system that automatically builds a CDN from visiting users’ unmodified web browsers. With Maygh, users visiting the operator’s web site automatically assist in serving static content to others for the duration of their visit; once they leave the site, this assistance ceases. Maygh requires no client-side changes and only minor modifications to the operator’s site, and can be deployed on the web sites and web browsers of today.

One might expect that peer-assisted content distribution would not work well when implemented only using web technologies, as such an implementation necessarily results in short average session times, small cache sizes, high churn, a limited computation model, and limited network stack access. We demonstrate that even in this challenging environment, a useful browser-assisted CDN can be constructed and deployed today, allowing operators who wish to distribute significant amounts of static content to spread the costs of distribution across their users.

Maygh enables web browsers to assist in content distribution using two techniques. First, Maygh uses the storage APIs [236, 239] supported by modern browsers to persistently store content on end user machines. Second, Maygh uses newly-available browser-to-browser communication

---

1 AdBlock Plus reports approximately 15 million active users [48], out of approximately 350 million Firefox users [160].

2 It is worth noting that Akamai’s NetSession reportedly has over 24 million users [42] (the NetSession software is typically bundled with video streaming software). However, because these users run Akamai’s software, web operators must still pay Akamai to use these clients to serve content.

3 “Maygh” is a rough phonetic translation of a Hindi word for “cloud.”
techniques [192,237] to enable direct message exchange with other browsers. Maygh uses centralized coordinators to track the static content stored in browsers, serving as a directory for finding content.

The remainder of this chapter is organized as follows: Section 3.2 explores the potential of Maygh by examining the fraction of static content on sites today. Section 3.3 details the design of Maygh, and Section 3.4 discusses the security/privacy implications. Section 3.5 describes the implementation of Maygh, and Section 3.6 evaluates Maygh. Finally, Section 3.7 summarizes.

3.2 Maygh potential

We begin by examining the potential for a system like Maygh that is able to assist in the delivery of static content. Over the past few years, dynamically generated web pages have become more prevalent. For example, advertisements are often targeted, online social networking sites customize pages for each user, and news web sites often provide suggested articles based on each user’s browsing profile. While most of these dynamically generated pages cannot be served by end users, a significant faction of resources embedded in the page (such as images, videos, and SWF) represent cacheable static objects. We present the results of a brief experiment to measure the fraction of bytes that such static content typically represents, suggesting the potential for Maygh to help distribute static content.

We conduct a small web browsing experiment. We first select the top 100 websites from Alexa’s [52] ranking (collectively, these sites account for 37.3% of all web page views [52]). For each of these sites, we use a web browser under automated control to simulate a browsing user, and we record all of the HTTP requests and responses. Starting with the root page of the site, the browser randomly selects a link on the page that stays on the same domain. The browser repeats this step five times, effectively simulating a random walk on the site of length five. To avoid any effects of personalization, we remove all browser cookies between requests. Finally, we repeat the experiment five times with different random seeds.

The result is shown at Table 3.1 aggregated across all sites by content type. We observe that 74.2% of the bytes requested are marked as cacheable based on the Cache-Control HTTP header.

This result, while by no means exhaustive, is inline with other studies [126,127] and suggests that reducing the bandwidth required to distribute static content is likely to provide significant savings to web site operators in practice.

We only consider content that contains no Cache-Control header or is marked Cache-Control: public, as recommended by RFC 2616 [100].
CHAPTER 3. MAYGH: RETHINKING WEB CONTENT DISTRIBUTION

<table>
<thead>
<tr>
<th>Content Type</th>
<th>% Requests</th>
<th>% Bytes</th>
<th>% Cacheable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>70.5</td>
<td>40.3</td>
<td>85.7</td>
</tr>
<tr>
<td>JavaScript</td>
<td>13.1</td>
<td>29.0</td>
<td>84.8</td>
</tr>
<tr>
<td>HTML</td>
<td>10.7</td>
<td>19.9</td>
<td>30.1</td>
</tr>
<tr>
<td>CSS</td>
<td>3.5</td>
<td>8.7</td>
<td>86.5</td>
</tr>
<tr>
<td>Flash</td>
<td>0.9</td>
<td>1.3</td>
<td>96.0</td>
</tr>
<tr>
<td>Other</td>
<td>1.3</td>
<td>1.0</td>
<td>45.7</td>
</tr>
<tr>
<td>Overall</td>
<td>100</td>
<td>100</td>
<td>74.2</td>
</tr>
</tbody>
</table>

Table 3.1: Breakdown of browsing traces from the top 100 Alexa web sites. “Cacheable” refers to the fraction of bytes that are cacheable according to the HTTP headers.

3.3 Maygh design

We now describe the design of Maygh. At a high level, Maygh provides an alternate approach to building a CDN for distributing static web content like images, videos, and SWF objects. Maygh relieves some of the load of serving content from the web site operator (hereafter referred to simply as the operator). Maygh works by allowing the client web browsers visiting the operator’s site to distribute the static content to other browsers. Maygh is compatible with dynamic web sites, as it can be used to load the static content objects that dynamic sites use to build their pages.

Maygh consists of two components: a centralized set of coordinators, run by the operator, and the Maygh client code, implemented in JavaScript that is executed in each client’s web browser.

3.3.1 Web browser building blocks

In the design of Maygh, we use technologies now present in web browsers. We assume that users have a web browser that supports JavaScript, and that users have JavaScript enabled (recent studies show that these assumptions hold the vast majority of web users [247]).

3.3.1.1 Persistent storage

To store content, the client-side Maygh JavaScript uses the storage APIs [236, 239] supported by modern browsers. In brief, these APIs allows a web site to store persistent data on the user’s disk. The interfaces are similar to cookie storage, in that they present a key/value interface and are persistent across sessions, but are larger in size and can be programmatically accessed via JavaScript. When a user fetches content in Maygh, the JavaScript places this content into the browser’s storage, treating the storage as a least-recently-used cache.
CHAPTER 3. MAYGH: RETHINKING WEB CONTENT DISTRIBUTION

3.3.1.2 Direct browser-to-browser communication

Maygh is designed to use either of two existing protocols that allow two web browsers to establish a direct connection between each other. These two protocols, described below, are largely similar and are intended to allow video and audio to be exchanged in a peer-to-peer (p2p) fashion between web browsers. They also allow application-level messages to be sent; it is this messaging facility that Maygh leverages to communicate between clients. Both protocols are built using UDP and support network address translation (NAT) traversal using STUN\(^5\) with assistance from the server if the client is behind a firewall.

Both protocols are built around a protocol server that assists in setting up direct connections between browsers. Each web browser generates a unique peer-id that other browsers can connect to with the assistance of the protocol server (the protocol itself handles the mapping from peer-id to IP address). In Maygh, the coordinator is implemented to also serve as a protocol server.

RTMFP \(\text{(Real Time Media Flow Protocol [192])}\) is a closed-source protocol that is built into the ubiquitous Flash plug-in\(^6\). RTMFP enables direct communication between Flash instances. All RTMFP packets are encrypted (each client generates a key pair), and RTMFP implements flow control and reliable delivery.

WebRTC \(\text{(Web Real-Time Communications [237])}\) is an open-source standard that is beginning to see adoption in popular web browsers\(^7\). WebRTC uses Interactive Connectivity Establishment\(^8\) for communicating with the protocol server and setting up peer-to-peer communication channels. WebRTC can be accessed using a browser-provided JavaScript library.

3.3.2 Model, interaction, and protocol

We assume that the content that Maygh serves is always available from the operator as normal, in case Maygh is unable to serve the content. We also assume that all content to be distributed by Maygh is named by its content-hash\(^8\); since we are focusing on static content, this can be accomplished in an automated fashion by the operator before publishing the content.

\(^5\)In brief, Session Traversal Utilities for NAT (STUN)\([183]\) enables two machines, each behind NATs, to establish a direct communication channel over UDP.

\(^6\)Adobe claims\([106]\) that the Flash player is installed on over 99% of desktop web browsers. RTMFP has been included in Flash since version 10.0 (released in 2008).

\(^7\)WebRTC is available starting in Google Chrome 23 and Firefox 18.

\(^8\)We avoid the worry of hash conflicts by using a hash function with a sufficiently large output space (e.g., SHA-256).
Figure 3.1: Overview of how content is delivered in Maygh, implemented in JavaScript (JS). A client requesting content is connected to another client storing the content with the coordinator’s help. The content is transferred directly between clients.

To use Maygh, the operator runs one or more coordinators and includes the Maygh client code (a JavaScript library) in its pages. The client code automatically connects the client to a coordinator and enables the client to fetch content from other clients later. Thus, the use of Maygh is transparent to users, who only observe web pages being loaded as usual. Users only participate in Maygh on a given web site as long as the user has a browser tab open to the site; once the user closes the browser tab, the participation ceases. Because the coordinator is run by the web site operator, the operator still receives a record of all content views and can use this information—as they do today—to target advertisements and make content recommendations.

3.3.2.1 Client-to-coordinator protocol

The Maygh client code communicates with the coordinator over either RTMFP or WebRTC.

Initial connection After the web page is loaded, the client initiates a connection with the coordinator. Once the RTMFP/WebRTC handshake is complete, the client informs the coordinator of the content that it is storing locally by sending a Maygh update message. This message contains a list of content-hashes. The client and the coordinator then keep the connection open via keep-alives.

Content request When a client wishes to fetch content, it sends a lookup message to the coordinator containing the content-hash of the content requested and the peer-ids of any other clients the client is currently connected to. The coordinator responds with a lookup-response message, containing a peer-id that is online and is currently storing that piece of content. If there are many

\[ \text{Clients are only able to communicate with others that support the same protocol (RTMFP or WebRTC); the coordinator ensures that only such clients are returned in the lookup-response.} \]
other clients storing the requested content, the coordinator attempts to select other clients that the
requesting client is already connected to, or that are close to the requesting client (e.g., by using a
geo-IP database).

Connect to another client When a client wishes to actually fetch content from another client, it
requests a connection to the specified peer-id. The functionality of establishing the direct connection
is handled by RTMFP or WebRTC with the coordinator functioning as a protocol server, and the
client is informed when the direct client-to-client connection is available for use. In brief, if either of
the clients is not behind a NAT, the connection can be made directly without coordinator assistance.
If not, the connection is established using STUN, with the coordinator assisting.

New content stored When a client has a new object stored locally, it informs the coordinator by
sending another update message. This message contains the content-hashes of any new objects
stored, along with the content-hashes of any objects that are no longer stored.

3.3.2.2 Client-to-client protocol

The protocol between clients also happens over either RTMFP or WebRTC. Once a direct
connection is established using one of these protocols, the client requesting the connection sends a
fetch message containing the content-hash that it wishes to download. The other client responds
with a fetch-response message containing the corresponding content. A timeline of the
messages exchanged in Maygh is shown in Figure 3.2.

3.3.3 Maygh client

The client-side code that implements Maygh is written in JavaScript. This code manages content
and makes Maygh easy to integrate into an existing web site. With RMTFP, the Maygh code also has
a small Flash object that conducts all communication (the Flash object itself is hidden, so the use of
Maygh does not alter the layout or appearance of the operator’s site). To deploy Maygh, the operator
includes a reference to the Maygh JavaScript in their web page, which causes the Maygh JavaScript
(as well as the Flash object, in the case of RTMFP) to be loaded and run.

The Maygh JavaScript code exports an API that the operator can use, shown below:

- connect(coordinator) Called when the web page is first loaded. Causes the Maygh
code to connect to the given coordinator and establish an open session.
Figure 3.2: Maygh messages sent when fetching an object in Maygh between two clients (peer-ids pid$_1$ and pid$_2$). pid$_1$ requests a peer storing content-hash obj-hash$_1$, and is given pid$_2$. The two clients then connect directly (with the coordinator’s assistance, using STUN if needed) to transfer the object.
CHAPTER 3. MAYGH: RETHINKING WEB CONTENT DISTRIBUTION

- **load(content_hash, id)** Called when the client code wishes to load an object. Causes Maygh to request the address of another client who is currently storing the given object, and then connect to that client, download the object, and verify its content-hash. If no peer has the content, if it cannot connect to the peer, if the content-hash is incorrect, or if downloading from the peer fails, the Maygh code loads content from the operator’s web site as normal. The id refers to the DOM ID of the object; Maygh will display the object once loaded.

In brief, the operator can load static content via Maygh by slightly modifying its existing pages. For example, if the operator loads an image using the HTML

```html
<img id="-id-" src="-src-"/>
```

it can instead load the image with Maygh using

```html
<img id="-id-">
<script type="text/javascript">
maygh.load("-hash-", "-id-");
</script>
```

where `-hash-` is the content-hash of the image. Due to security and privacy concerns, the operator decides what images are loaded with Maygh. Once loaded, Maygh will display the image as normal. Similar techniques can be used to load other static content objects like videos, SWF objects, and CSS.

The Maygh library is configured to maintain only a single connection to each site’s coordinator, even if the user has multiple browser tabs open to the same site (essentially, each site maintains a single connection with each browser, regardless of the number of tabs open). With WebRTC, this is done automatically using shared WebWorkers [238]. With RMTFP, this is accomplished using Flash’s LocalConnection, where the multiple tabs from a single site can communicate with a single Flash instance and share a single coordinator connection.

It is important to note that many of the content-loading optimizations that are present in the web today are compatible with Maygh. For example, many web sites pre-fetch images that the user is likely to view next, or fetch images using AJAX instead of HTML `<img>` tags. Both of these can be easily modified to load the content with Maygh, replacing the existing loading logic with a call to Maygh’s `load` function. Additionally, Maygh connects to other clients and can load objects in parallel, thereby avoiding additional latency on pages that have many objects loaded with Maygh.

27
3.3.4 Maygh coordinator

Maygh uses one or more centralized coordinators run by the web site operator. The coordinators have two functions: serving as a directory for finding other clients storing content, and serving as a protocol server for RTMFP or WebRTC.

Recall that once clients are connected to a coordinator, they inform the coordinator of any locally stored content (identified by content-hashes). The coordinator maintains this data in two data structures: First, the coordinator maintains a content location map, which maps each piece of content (identified by its content-hash) to the list of online peer-ids storing that content. Second, the coordinator maintains a client map, which maps each peer-id to the list of content that it is storing.

Maintaining these two maps allows the coordinator to ensure that the content location map contains only references to clients who are online. Whenever a client goes offline (either explicitly or through a timeout of the keep-alive messages), the coordinator determines the list of content that client was storing using the client map, and then purges that client’s record from each of the entries in the content location map.

The coordinator also keeps track of the number of content bytes each client has downloaded and uploaded for a configurable time period (e.g., each week). Doing so allows the coordinator to ensure that no client is asked to upload more than a configurable fraction upload_ratio of what it has downloaded. Maygh also provides a global upper bound upload_max on the total amount of content that any client is asked to upload, regardless of how much it has downloaded. For example, the operator could set upload_ratio to 1 and upload_max to 10 MB per week, ensuring that no client has to upload more than the amount it has downloaded, and never more than 10 MB each week.

3.3.5 Multiple coordinators

One of our goals is to allow web sites using Maygh to scale to large numbers of users. In such a deployment, it is likely that a single coordinator will quickly become a performance bottleneck. We could trivially allow the operator to run multiple coordinators in parallel, but clients who are connected to different coordinators would be unable to fetch content from each other (as each coordinator would not be aware of the content stored on clients connected to the other connectors). This has the potential to preclude much of the potential savings of Maygh from being realized.

Instead, we enable multiple coordinators to work in tandem and allow clients connected to different coordinators to exchange content. We assume that the operator has deployed a set of \( N \)
CHAPTER 3. MAYGH: RETHINKING WEB CONTENT DISTRIBUTION

Figure 3.3: Overview of how multiple coordinators work together on Maygh. The mapping from objects to lists of peers is distributed using consistent hashing, and the coordinator each peer is attached to is also stored in this list. The maximum number of lookups a coordinator must do to satisfy a request is two: one to determine a peer storing the requested item, and another to contact the coordinator that peer is attached to.

Recall from above that the single coordinator maintains two data structures: the client map and the content location map. We keep the client map exactly the same as before (each of our $N$ coordinators maintains a client map containing entries for its clients). However, we distribute the content location map across all of the coordinators using consistent hashing [133], effectively forming a one-hop distributed hash table [109]. Each coordinator selects a random coordinator-id from the same hash space as the content-hashes (e.g., by running the same hash function on its IP or MAC address). Each coordinator is then responsible for storing the content location map entries for which its coordinator-id is numerically closest to the content-hash key. Finally, for each peer-id in the content location map, we also store the coordinator-id that the peer is connected to. A diagram showing this distribution is shown in Figure 3.3.

Distributing the state of the coordinator in this manner allows for the multiple coordinators to have favorable scaling properties. Consider the operations that are necessary when a client issues a lookup message: The client’s coordinator receives that message, and can immediately determine
the coordinator who is storing that entry in the content location map, through the use of consistent hashing. The coordinator requests that entry from the remote coordinator, caches the result, and then returns the list of peer-ids to the client. When the client requests to be connected to another client, the coordinator uses the cached result to determine the coordinator to whom the remote client is connected, and then communicates with that coordinator to allow the two clients to directly connect. In fact, the establishment of the direct client-to-client connection proceeds exactly as in the single-coordinator case, except that each coordinator only sends STUN packets to its own client.

As a result, the coordinator is only required to send at most two messages to other coordinators in response to a lookup message, regardless of the number of coordinators that exist. As we demonstrate in the evaluation, this allows the performance to scale close to linearly with the number of coordinators, and enables Maygh to be used on sites that serve many thousands of requests per second.

### 3.4 Security, privacy, and impact on users

The design of Maygh changes many of the properties of the web, raising a number of concerns about security, privacy, and the impact that Maygh will have on users. We now address these questions, leading to a discussion of the deployments where Maygh is most appropriate.

#### 3.4.1 Security

We first examine how Maygh handles malicious users. There are two primary concerns: malicious users might attempt to serve forged content to other users, or might attempt to perform denial-of-service (DoS) attacks by overloading the coordinator or violating the protocol.

In order to detect forged content, all content in Maygh is self-certifying [209], since it is identified by its content-hash (see Section [3.3.2]). When a client receives content, the client immediately compares the hash of the content with its identifier. This enables the client to immediately detect and purge forged content; if forged content is detected, the client then downloads the content from the operator as normal.

In order to address users who attempt to violate the Maygh protocol (e.g., by claiming to have content stored locally that they later turn out not to have), Maygh uses similar techniques that are in-use by such sites today: Operators can block accounts, IP addresses, or subnets where malicious behavior is observed [206]. Additionally, since the coordinator is under the control of the operator,
existing defenses against DDoS attacks can be deployed to protect the coordinator similar to the operator’s web servers [137][152].

3.4.2 Privacy

Next, we examine the privacy implications of Maygh. We first note that the Maygh coordinator tracks the content stored in each user’s browser while the user is online, which could lead to privacy concerns. However, we note that the Maygh coordinator is run by the web site operator, who, today, is already able to log access requests and track downloaded content.

In Maygh, clients do receive information about views of content by other users (i.e., when a client sends or receives a fetch message for a piece of stored content, it can determine the IP address of the other user viewing that content). As a result, there may be sensitive content for which Maygh is inappropriate. In such cases, the operator can disable loading such content via Maygh, or allow the user to do so using privacy controls. Alternatively, the operator can add background requests to random pieces of content (sometimes referred to as cover traffic [101]), or can place content on clients before they have viewed it, in order to provide plausible deniability. Regardless, the privacy implications of Maygh, where users can sometimes infer information on the views of others, are similar to a number of deployed peer-assisted content distribution systems including Akamai’s NetSession [50] (used by services like the NFL’s online streaming [143]), Flower-CDN [89], Firecoral [217], and numerous IPTV systems [119] such as PPLive [203].

Additionally, it is difficult for an attacker to determine the contents of a specific users’ Local-Storage in Maygh. This is because, for each lookup request a user issues, the coordinator returns a single remote client that the coordinator believes is close to the requestor. As a result, the choice of which client is returned is the coordinator’s, and the coordinator is able to apply randomization or other techniques to limit the ability for the attacker to target specific users. Accounts or IP addresses that issue spurious requests can be banned or blocked in the same manner as they are on today’s web sites [206].

In order to ensure the users can only access content they are authorized to view, the coordinator can authenticate content requests in the same manner as existing web servers. If a client issues a lookup request for a content-hash it is unauthorized to view, the coordinator can simply deny the request. Moreover, using content-hashes for naming may enable Maygh to skip this authentication for many applications, as users can only request content if they know its hash. In fact, this is precisely the semantics that many web sites with sensitive content use today. For example, on Flickr, the URLs
CHAPTER 3. MAYGH: RETHINKING WEB CONTENT DISTRIBUTION

of images are obfuscated through the use of a per-image “secret” (analogous to our content-hash), but anyone who possesses the secret can construct the URL and download the image.

3.4.3 Impact on users

When deployed, Maygh reduces the bandwidth costs imposed on the web site operator. Our hope is that lowering these costs will both reduce the need for operators to rely on advertising revenue (often enabled by data mining end-user-provided content), as well as allow web sites to be deployed that are not currently economically feasible. For example, sites that deploy Maygh may choose to make Maygh opt-in, offering to show fewer ads to users who help with content distribution. Regardless, Maygh’s tracking of the amount of content uploaded and downloaded ensures that no user has to contribute more resources than they use, and we demonstrate in the next section that Maygh imposes an acceptable bandwidth and storage cost on the clients (since the load is distributed over all clients).

Most cable or DSL systems offer users asymmetric bandwidth, with more downstream than upstream bandwidth \[86\]. This configuration is unlikely to significantly affect Maygh, as we demonstrate in Section 3.6 that each user ends up serving content infrequently and the content served is small compared to the available upstream bandwidth. Reducing content page loading latency is typically important for web site operators, and using Maygh imposes increased latency on content requests. Moreover, additional latency in Maygh can be caused by clients with asymmetric bandwidth. However, operators can hide much of this increased latency from users by using content pre-fetching techniques (in fact, many sites already do so).

3.4.4 Mobile users

The Maygh design so far has focused on users who are using traditional desktop web browsers. However, users are increasingly accessing the web from mobile devices. As the mobile web browsers are quickly catching up with their desktop counterparts, it is likely that Maygh will work without modification once WebRTC support is provided to these browsers. However, a potential drawback is that the user session times are likely to be much shorter, as smartphones typically only allow users to have one “active” web page at a time. Moreover, smartphones present additional constraints, such as Limited battery life, data access charges, high RTTs, and slow CPUs. Previous work \[242\] has shown that mobile browser-based assistance can be implemented without significant battery consumption, and Section 3.6 demonstrates that per-user limits on the amount of data users are
CHAPTER 3. MAYGH: RETHINKING WEB CONTENT DISTRIBUTION

requested to upload does not significant impact the benefits of Maygh. However, we leave a full evaluation of the potential of Maygh on mobile devices to future work.

### 3.5 Implementation

Our full Maygh implementation is written using RMTFP, as this allows us to deploy Maygh onto a wide variety of web browsers and to obtain a userbase quickly [106]. Support for WebRTC is in progress, and a proof-of-concept implementation is described in Section 3.6.1.2.

The Maygh coordinator is written as a heavily-modified version of the open-source ArcusNode [59] RTMFP server. ArcusNode is written in JavaScript, built on top of the Node.js [164] framework. As a result, our coordinator consists of 2,944 lines of JavaScript, and 372 lines of C for the encryption and encoding libraries. Our coordinator supports working in tandem with other coordinators, as described in Section 3.3.5. The code is single-threaded, event-driven, and uses UNIX pipes for communication (between coordinators resident on the same machine) or TCP sockets (if not).

The client-side Maygh implementation consists of 657 lines of JavaScript (with a total size of 4.2 KB) and 214 lines of ActionScript (compiled into a 2.6 KB Flash object). We use the Stanford Javascript Crypto Library’s SHA-256 implementation [213] (6.4 KB) to implement hashing on the client side. Thus, each client has to download an additional 13.2 KB of code (which is likely to be cached, allowing each client to only have to download it once).

In some of the experiments below, we wish to simulate a large number of clients using Maygh. It is challenging to run thousands of web browsers and Flash runtimes at once; instead, we wrote a command line-based implementation of the Maygh client. This implementation, written on the Node.js [164] framework similar to our coordinator code, follows the exact same protocol as the Maygh JavaScript and Flash does when run within a web browser; from a network perspective, the messages sent are identical. However, the simulated clients run entirely at the command line.

To implement this client, we reverse-engineered the client-side RTMFP protocol. We then implemented the client on the Node.js framework. In total, it contains 2,900 lines of JavaScript (2,053 of which are shared with the coordinator implementation).

### 3.6 Evaluation

We now turn to evaluate the performance of Maygh. We center our evaluation around four questions:
CHAPTER 3. MAYGH: RETHINKING WEB CONTENT DISTRIBUTION

<table>
<thead>
<tr>
<th>Loaded to</th>
<th>Loaded from</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LAN BOSTON</td>
</tr>
<tr>
<td>LAN</td>
<td>229 / 87</td>
</tr>
<tr>
<td>Cable</td>
<td>771 / 283</td>
</tr>
</tbody>
</table>

Table 3.2: Average time (ms) to load first / second 50 KB objects using Maygh with RTMFP.

- What is the impact of Maygh on web clients, in terms of increased latency and network overhead?
- What is the scalability of Maygh? How many content requests per second can a set of coordinators support?
- What is the impact of Maygh on the operator, in terms of the reduction in network traffic? How much network overhead does Maygh incur?
- How does Maygh perform when deployed on a real web site to real users?

### 3.6.1 Client-side microbenchmarks

We now examine the impact that Maygh would have on end users. We examine the latency of fetching objects, the additional network traffic, and the storage requirements.

#### 3.6.1.1 Client-perceived latency

We first examine the client-perceived latency of fetching content. For this experiment, we deploy Maygh to a test web site with a coordinator located on our university’s LAN in Boston. We connect two clients running Google Chrome 13.0 with Ubuntu 11.04 to the web site and one of the clients fetches two 50 KB objects from the other. We measure the time required to fetch each entire object, and all results are the average of ten separate runs. We report the time taken from the requesting client’s perspective, including all messaging with the coordinator, connection setup with the other client, and hash verification. For all experiments, we configure additional command-line clients to create a background load of 200 fetch requests per second to the coordinator.

As we are interested in the latency experienced by clients in different locations, we run this experiment with a number of different configurations. We placed the client requesting the objects in two different locations: on the same LAN as the coordinator, and behind a cable modem in Boston.
Then, we placed the client serving the objects in three different locations: on the same LAN as the coordinator, behind a cable modem in Boston, and behind a DSL modem in New Orleans.

The results of this experiment are presented in Table 3.3. We report the average time taken for the first object separately from the second object; the first is higher because it includes connection setup (including STUN, in the cases of Cable–Cable and Cable–DSL) with the remote client. For the second object, the connection to the remote client is cached by the Maygh library. For all intra-Boston connections, the time taken to deliver the 50 KB object is under 320 ms, with a RTMFP connection setup overhead of approximately 300 ms. Fetching content from New Orleans to Boston is more expensive, but we expect that clients will be able to find another online client within their geographic region the majority of the time. Additionally, much of this latency can be hidden through the use of content pre-fetching (which many web sites already do) and parallel downloads (which most browsers already do).

### 3.6.1.2 Latency with WebRTC

Our latency results in the previous section suggest that there is non-trivial latency overhead when using RTFMP (recall that RTFMP is primarily designed for audio and video streams to be exchanged, not application-level messages). To determine how much of the latency is due to RTMFP protocol overhead—and is therefore not fundamental to Maygh—we implemented a proof-of-concept version of Maygh using WebRTC. We use Chromium 26.0.1412.0\[10\], which has initial support for WebRTC’s DataChannel. We repeat the same experiment as above, and report the results in Table 3.3. We observe that all cases, our prototype WebRTC implementation is significantly faster (typically around twice as fast as RTMFP). This result indicates that the performance of our RTMFP-based

---

\[10\]This build of Chromium imposes a rate limit on each DataChannel of 3 KB/s; we removed this rate limit to perform our experiments. Additionally, this build only supports unreliable channels; we implemented a reliable sliding-window-based protocol on top of this interface. Since version 31.0.1650, Chromium has supported reliable DataChannel over SCTP, and has allowed bandwidth control with SDP.

---

Table 3.3: Average time (ms) to load first / second 50 KB objects using Maygh with our proof-of-concept WebRTC implementation.

<table>
<thead>
<tr>
<th>Loaded to</th>
<th>LAN</th>
<th>Loaded from</th>
<th>Cable</th>
<th>DSL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BOSTON</td>
<td></td>
<td>BOSTON</td>
<td>NEW ORLEANS</td>
</tr>
<tr>
<td>LAN</td>
<td>72 / 16</td>
<td>364 / 120</td>
<td>544 / 354</td>
<td></td>
</tr>
<tr>
<td>Cable</td>
<td>284 / 57</td>
<td>577 / 107</td>
<td>765 / 379</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3: Average time (ms) to load first / second 50 KB objects using Maygh with our proof-of-concept WebRTC implementation.
CHAPTER 3. MAYGH: RETHINKING WEB CONTENT DISTRIBUTION

implementation is likely a lower bound on the performance of (in-progress) complete WebRTC-based implementation.

3.6.1.3 Network traffic

We now turn to examine the network traffic overhead caused by Maygh. As discussed above, the Maygh code itself is 13.2 KB, although this will be cached by the web browsers for clients’ subsequent requests. To connect to the coordinator, the client sends a total of 1.3 KB, and the client sends an additional 0.6 KB for each content request and subsequent connect request. Thus, even for very small objects, Maygh imposes very little network traffic overhead. Moreover, the majority of the overhead comes from the downloading of the client code and the initial connection to the coordinator; this cost will be amortized over all objects in the page.

3.6.1.4 Client storage capacity

Our Maygh implementation uses the LocalStorage [239] browser storage API, which is by default limited to only 5 MB of storage per site (the other available storage APIs offer greater defaults; our results are therefore conservative). We next examine is the number of objects that can be stored in each user’s LocalStorage. Taking into account the 33% overhead induced by storing objects in base64 format, Maygh is able to store 3.3 MB of content per site in each user’s LocalStorage. However, as we demonstrate below using real-world data from Etsy, even using only 3.3 MB per site on each client still allows significant savings to be realized.

3.6.2 Coordinator scalability

We now turn to explore the scalability of the coordinator nodes. Our goal is to determine the rate of content requests (referred to as transactions) that can be supported by the coordinators. Each transaction consists of a lookup and lookup-response message and—if the requested content was found on another online client—a connect message followed by the coordinator-assisted connection establishment between the clients.

For these experiments, we run coordinator node(s) on a cluster of machines with dual 8-core 2.67 GHz Intel Xeon E5-2670 processors (with hyperthreading), connected together by Gigabit Ethernet. We then run simulated clients on similar machines in the same LAN. We configure each of the clients to make content requests every five seconds, and measure the throughput (in terms of the number
Figure 3.4: Average response time versus transaction rate for a single coordinator. The coordinator can support 454 transactions per second with under 15ms latency.

of transactions per second) across the coordinators. The clients are configured so that 70% of the transactions will have the content present on another client.

3.6.2.1 Single coordinator

Our first experiment examines the performance of a single coordinator process (i.e., a coordinator running only on a single core). In this experiment, we slowly increase the number of clients over time and record the number of transactions per second processed by the coordinator. We also record the request latency at the clients; as the coordinator reaches peak throughput, we expect the number of transactions per second to level off and the client-observed latency to increase sharply. The results of this experiment are presented in Figure 3.4. After 454 transactions per second, we observe that the response time increases above 15 ms and rises more sharply, indicating that the coordinator is having trouble keeping up.

3.6.2.2 Multiple coordinators

We now explore the scalability of the coordinators. We repeat the experiment from above, but deploy multiple coordinator nodes that work together to distribute content. The clients are randomly assigned to one of the coordinators and make requests for random content (i.e., there is no coordinator-locality in the requests).

We run two experiments, distributing the coordinators in two different ways: all resident on the same physical machine, and each located on their own machine. In all cases, we provide each coordinator with a dedicated CPU core. For each experiment, we slowly increase the number of clients present and calculate the number of transactions per second that can be supported before the average response time at the clients increases beyond 15 ms.
Figure 3.5: Average response time versus request rate for multiple coordinators working in tandem, in two different placements of the coordinators across machines. Close-to-linear scaling is observed as more coordinator nodes are added (the one-machine set of coordinators show lower performance after 16 coordinators are placed on a single machine due to the effects of hyperthreading).

The results of this experiment are shown in Figure 3.5. We observe close-to-linear scaling, as expected from our design. We also observe that the set of coordinators located each on their own machine show similar performance to the set of coordinators on a single machine, up to 16 coordinators. After this point, the performance of coordinators on a single machine increases more slowly, because coordinator processes begin to be assigned to the same physical core, though on different core threads (i.e., the effects of the hyperthreaded CPUs begin to become apparent).

Overall, we observe that the Maygh coordinators show very favorable close-to-linear scalability. With our 32-core server, the Maygh coordinators are able to support over 3,700 transactions per second, making it suitable for very large websites. In fact, in our simulations below using the access logs from Etsy, we observe a peak rate of 938 transactions per second; this load would require only a 4-core machine to host the coordinators.

3.6.3 Trace-based simulation

Our next evaluation concerns the benefits that Maygh can be expected to provide in practice to the web site operator.

3.6.3.1 Simulation data

Determining the benefits of Maygh in practice requires a real web access trace, as it is dependent upon the object request pattern, the object sizes, and the offline/online behavior pattern of clients. We use traces provided by Etsy, a large e-commerce web site that is the 50th most popular web site in the U.S. [51]. Etsy is a popular online marketplace for independent artists and designers; each seller
is given an online “store” where they can post items available for sale. Each item listing typically contain of a number of images, and Etsy currently distributes these using Akamai.

We obtained anonymized logs for seven days of accesses to the static image content on the etsy.com website, covering 205,586,135 requests to 56,084 unique objects from 5,720,737 unique IP addresses. The logs only include the busiest 18 hours of the day, from 6:00am–11:59pm EST. In total, the requests represent 2.77 TB of network traffic, or 395 GB per day. To estimate the overall fraction of Etsy’s bandwidth that is represented in our traces, we use the same random browsing methodology from Section 3.2 on Etsy’s site. We find that image content represents 85.6% of the total bytes served by Etsy, meaning any savings we report is likely to represent significant overall bandwidth savings.

It is important to note that our logs are what the web site operator sees, after the browser caching and any in-network (HTTP proxy) caching. Thus, any bandwidth savings that we report are ones that would be observed in practice.

3.6.3.2 Simulation setup

Simulating a Maygh deployment requires knowing how long the clients’ browser windows remain on the site (as this determines how long the client is available to serve requests to other users). The trace lists only HTTP requests, so we do not know the length of time that the client’s browser window remains open to the site. Therefore, we simulate the clients staying on the site for a random amount of time between 10 seconds and 30 seconds after each request. This short online window is likely to be conservative [139] for a storefront like Etsy; the longer that users stay online, the better the network traffic savings that Maygh can provide (since users are available to serve requests for longer).

We simulate the clients’ LocalStorage staying intact between sessions, as this would happen in practice. Unless otherwise mentioned, we set upload_ratio to 1 and upload_max to 10 MB. This means that no client is asked to upload more bytes than they have previously downloaded, and no client is asked to upload a total of more than 10 MB during the simulated week. In our bandwidth calculations, we also include the network traffic for clients to download the Maygh code, to connect to the coordinator, and to execute the Maygh protocol.

To compare Maygh to alternate approaches, we also simulate a deployment of a plug-in-based system (e.g., Firecoral [217]) to a random 10% of the Etsy users (recall from Section 3.1 that 10% is likely to be a higher fraction than would be observed in practice, as the most popular plug-in today is used by 4.2% of users). Due to the plug-in architecture, users who install the plug-in would only be
able to download from other such users; users who do not install it would continue to fetch content from the operator as normal. We simulate the plug-in system with a per-user 100 MB cache size, and allow users to serve to others regardless of the amount of data they have downloaded (i.e., we do not limit users to uploading only as much as they have downloaded).

3.6.3.3 Bandwidth savings

We examine the amount of network traffic at the operator under three configurations: normal (with no plug-in or Maygh), with 10% of users running installing the plug-in, and with Maygh. We record the network traffic experienced at the operator, aggregated into five-minute intervals over the course of the week-long trace.

The results of this experiment are presented in the top graph of Figure 3.7, showing the five-minute average bandwidth required at the operator with different configurations. Figure 3.6 presents the cumulative distribution of this same trace. We make a number of interesting observations: First, we observe that Maygh provides substantial savings: the median bandwidth used drops from 50.3 Mb/s to 11.7 Mb/s (a 77% drop). Second, we also observe that the 95th-percentile bandwidth—which often determines the price that operators pay for connectivity—shows a similar decrease of 75%, demonstrating that Maygh is likely to provide a significant cost savings to the operator. Third, we observe that the 10% plug-in deployment results in a median bandwidth decrease of 6.9% and a 95th-percentile bandwidth decrease of 7.7%; the savings is less than 10% due to cache misses and protocol overhead.

In the lower plot of Figure 3.7, we show the number of transactions per second that is experienced at the Maygh coordinator over the week. We observe that the average transaction rate is 482 transactions per second, with a maximum of 938. This shows that Maygh could be deployed at Etsy
CHAPTER 3. MAYGH: RETHINKING WEB CONTENT DISTRIBUTION

Figure 3.7: TOP: Bandwidth required at the operator, as normal (with no plug-ins or Maygh), with a 10% plug-in deployment, and with Maygh; a 10% plug-in deployment results in 7.8% bandwidth savings, while Maygh results in 75% bandwidth savings. BOTTOM: Request rate observed at the Maygh coordinator; the rate is almost always below 800 requests per second, and is easily handled by a four-core server.

with a small number of coordinators.

3.6.3.4 Bandwidth breakdown

We now turn to examine the breakdown of the network traffic at the operator. In order to explore the contribution that images of different sizes have on the performance of Maygh, we configure Maygh to only serve images larger than a given threshold. We then vary this setting, examining the resulting performance tradeoff. In this setup, the network traffic at the operator can be broken down into three classes:

1. Maygh overhead consisting of downloading the client code and Maygh protocol overhead.

2. Content served normally due to Maygh misses when the coordinator could find no online client able to serve a request.

3. Content served normally because it is too small when content is smaller than the Maygh threshold configured by the web site operator.

The results of these experiments are presented in Figure 3.8 for different settings of the minimum image size that Maygh will serve. The results are presented as stacked curves and are expressed in terms of the total network traffic that Etsy experienced without Maygh. As an example, if Etsy chose to configure Maygh to only serve content larger than 5 KB, they would experience only 27% of the network traffic that they did in the original trace. Of this traffic, 19% would be comprised of images below 5 KB that are served normally, 62% would be comprised of images larger than 5 KB.
CHAPTER 3. MAYGH: RETHINKING WEB CONTENT DISTRIBUTION

Figure 3.8: Network traffic at the operator, relative to the trace without Maygh, for different minimum image sizes. The traffic consists of three components: images that are served normally because they are too small, images that are served normally because Maygh cannot find an online client able to serve it, and overhead caused by Maygh (downloading the client code and protocol overhead). If Maygh serves all images, the operator would experience 75% less traffic due to image downloads.

that Maygh cannot find an online client to serve, and 19% would be comprised of Maygh overhead (primarily downloading the Maygh code).

We observe that Etsy’s workload consists primarily of small images; over half of the network traffic without Maygh is spent transmitting images smaller than 35 KB. As a result, the largest benefits of Maygh in our simulations only happen when the size threshold for being served Maygh is small. However, we note that this is an artifact of Etsy’s workload; Maygh has lower overhead with larger objects, so a workload with a larger objects is likely to perform even better.

Finally, we observe that, for small thresholds, the largest fraction of the network traffic at the operator is caused by Maygh misses. Examining the logs, we find that these misses are usually caused by the enforcement of the upload constraints at the clients. In other words, there are often clients online who can serve a given piece of content, but all of the clients have already reached their upload limit. Re-running the same experiment from above and removing the upload constraints at the client (meaning clients may be asked to serve more content than they download) causes the network traffic due to images at the operator to fall to 18% of the traffic that would be experienced without Maygh (compared to 25% with the upload constraints).

3.6.3.5 Client network traffic

We now examine is the network traffic that is imposed on clients. For each client in the trace, we record the total amount of data downloaded and the total amount of data that the client is requested to upload. We then compare the two, to examine how “fair” the distribution of the upload workload is across clients. For this experiment, we configure the Maygh threshold to be 5 KB.
CHAPTER 3. MAYGH: RETHINKING WEB CONTENT DISTRIBUTION

Figure 3.9: Network traffic at the clients, comparing amount of data uploaded to downloaded. We observe that the client workload distribution of Maygh is “fair”: most clients are requested to upload an amount of data that is proportional to the amount they download.

Figure 3.9 presents the results of this experiment, plotting the total amount of data uploaded vs. downloaded. We observe that the shape of the plot is largely defined by Maygh policy: no user is ever asked to upload more than they have downloaded, and no user is asked to upload more than 10 MB. However, even with those constraints, the workload distribution across users is quite “fair”: Most users are asked to upload in close proportion to what they have downloaded.

3.6.4 Small-scale deployment

As a final point of evaluation, we deployed our Maygh prototype on a small scale within our computer science department to examine how it would work with real-world users. We set up a coordinator within our department, deployed Maygh to a special version of our department’s web server, and then recruited users by emailing our graduate student population. The deployment ensured that all images on our department’s web site would be loaded via Maygh.

In total, over the course of our 3-day deployment, we observed 18 users use Maygh on Google Chrome, Firefox, and Safari, on machines running Windows, Linux, and OS X. These users browsed a total of 374 images. 90 (or 24%) of these images were served from another Maygh client. For the remaining 76% of the images, there was no other client online storing the image; they were fetched from the origin site as normal. While the network savings of Maygh is lower than in our simulations, it is due to our deployment environment: For the simulations, we considered what would happen if a large, popular web site deployed Maygh; in our real-world deployment, we had to manually recruit users and the size of our user population is dramatically smaller. However, the deployment demonstrates that Maygh is can be feasibly deployed to today’s web sites and web browsers.

\[11\] Our real-world deployment was covered under Northeastern University Institutional Review Board protocol #10-07-23.
CHAPTER 3. MAYGH: RETHINKING WEB CONTENT DISTRIBUTION

3.7 Summary

Over the past two decades, the web has provided dramatic improvements in the ability and ease of sharing content. Unfortunately, the client–server architecture of the web causes web sites who wish to share popular content to make substantial monetary investments in serving infrastructure or cloud computing resources, or pay organizations like CDNs to help serve content. As a result, only well-funded web sites can serve a large number of users.

In this chapter, we have presented Maygh, a system that distributes the cost of serving content across the visitors to a web site. Maygh automatically recruits web visitors to help serve content to other visitors, thereby substantially reducing the costs for the web site. A thorough evaluation of Maygh using real-world traces from a large e-commerce web site demonstrated that Maygh is able to reduce the 95th-percentile bandwidth due to image content at the operator by over 75%, providing a substantial monetary savings, and a small-scale deployment demonstrated that Maygh imposes little additional cost on clients and is compatible with the web browsers and sites of today. With Maygh, we have shown that we can significantly improve web content distribution systems to match today’s workload by using the latest technological advancements of the web without requiring users to install additional software.
Chapter 4

Priv.io: Rethinking web applications

Maygh solves the problem of distributing web content at low cost by recruiting user participation in direct content exchange. This fits the emerging needs of content sharing services, such as online social networks, particularly well. On these sites, users contribute significant amounts of content and share them with other users. However, user-generated content on these sites is still under the control of these services, and users have little control of what services providers can do with their data.

In this chapter, we leverage the power of the modern browsers again to solve this issue. We present Priv.io, a new approach for building web-based services that offers users greater control and privacy over their data. We leverage the fact that today, users can purchase storage, bandwidth, and messaging, but not computation from cloud providers at fine granularity: In Priv.io, each user provides the resources necessary to support their use of the service using cloud providers such as Amazon Web Services. Users still access the service using a web browser, all computation is done within users’ browsers, and Priv.io provides rich and secure support for third-party applications. We first presented Priv.io as a conference paper in COSN’13 [243].

4.1 Motivation

Users today have access to a broad range of free Web-based services (e.g., online social networks such as Facebook, microblogging services such as Twitter, content sharing sites such as Flickr). All of these services operate under a similar model: Users entrust the service provider with their personal information and content (e.g., their comments, photos, political and religious views, sexual orientation, occupations, identities of friends). In return, the service provider makes their service available for free by monetizing the user-provided information and selling the results to third parties.
(e.g., advertisers). Even though users are often provided with privacy controls on these sites, these controls generally only affect flow of information to other users or third-party applications; users today have no option of making their data private from the service provider. This model also makes it difficult for users to retrieve all of their data from the provider (e.g., if the provider closes the service [154, 158]) or remove their data entirely.

Researchers have investigated a number of approaches that provide users with greater control and privacy in such services, ranging from encrypting data uploaded to the provider [113, 179, 215] to dividing data between provider-hosted and user-hosted servers [69, 195] to implementing a fully decentralized system [70, 82, 90]. Unfortunately, none of these approaches have enjoyed widespread adoption, as they suffer from one or more of three general limitations:

- **Accessibility** Most proposals require users to install dedicated client software, such as desktop applications or browser plugins. As users typically access services from a variety of devices, these solutions require significant effort of the user (who has to install the software) and the developer (who has to build and maintain clients for various devices).

- **Reliability** Systems that rely on hosting content on end-user machines [70, 82, 90], home routers [153], or smartphones [202] maintain availability via replication. Unfortunately, such systems are known for suffering from fundamental reliability tradeoffs in dynamic environments [61].

- **Cost** Systems that require users to rent their own server from a cloud provider [195] or pay for subscription of the service [58] are likely to be too expensive for most users.

In this chapter, we rethink the architecture for building web-based services, with the goal of enhancing user privacy. We present Priv.io, an alternate approach to implementing web-based services that provides users with control over their data, ensures privacy, and avoids the limitations of practicality, reliability, and cost. In Priv.io, each user provides resources necessary to support their use of the service by purchasing computing resources (storage, bandwidth, and messaging) from cloud providers such as Amazon Web Services or Windows Azure. Unfortunately, having users purchase computation from cloud providers is not practical in Priv.io: at the finest granularity, users still must purchase an entire virtual machine for an hour, and having an always-on server is too expensive for most users. Instead, Priv.io is built entirely in JavaScript, and all computation is done

---

1 Of course, computation is also necessary on the cloud provider’s side to implement the storage and messaging abstractions. For the purposes of this chapter, computation refers to Priv.io and third-party application logic.
within the users’ web browsers while they visit the Priv.io web site. Priv.io works with unmodified versions of common web browsers such as Safari, Chrome, Firefox, and Internet Explorer, as well as browsers on mobile OSes including Android and iOS.

The result is a **confederated** service, where each user retains control over his or her own data. We demonstrate that services similar to Facebook, Twitter, and Flickr can be implemented in a confederated manner with very low monetary costs for most users. Thus, Priv.io provides users with an alternative to today’s model of paying for web-based services by giving up their privacy.

Priv.io provides strong guarantees of user privacy. Priv.io uses attribute-based encryption \[65, 69\] to encrypt all content stored on the cloud provider; this encryption is implemented in JavaScript within the user’s browser. Thus, only the users’ browsers ever see plaintext content. Priv.io also provides rich support for third-party applications (e.g., Farmville \[249\]) by providing an API that is implemented within the users’ browsers. Priv.io uses browser-based sandboxing to ensure that third-party applications can only access the data that users allow and cannot leak any user information to the application provider or other third-parties.

The remainder of this chapter is organized as follows. Section 4.2 presents a measurement study aimed at estimating the cost of Priv.io to users of a variety of today’s Web-based services. Section 4.3 describes the design of Priv.io and Section 4.4 details how Priv.io supports third-party applications in a secure manner. Section 4.5 presents a discussion of some issues that arise when deploying Priv.io. Section 4.6 presents an evaluation of Priv.io and Section 4.7 summarizes.

### 4.2 Overview

Recall that our approach is to implement a web-based service in a confederated manner, by having users provide the resources necessary to support their use of the service via cloud providers such as Amazon Web Services. While cloud providers typically offer bandwidth, storage, and messaging at relatively fine granularity, computation is still sold at a relatively coarse granularity (typically an entire virtual machine for an hour). As a result, even running the smallest of Amazon’s EC2 servers (t1.micro) would cost a user $14.40 per month \[53\], not including EBS storage and I/O costs. Moreover, running an entire virtual server is overkill for most users; most of the time, this server would sit idle.

\[\text{2We choose the adjective } \text{confederated } \text{rather than } \text{federated, as members in a confederation retain autonomy and are generally free to leave (e.g., the Articles of Confederation between the original 13 U.S. colonies).}\]
CHAPTER 4. PRIV.IO: RETHINKING WEB APPLICATIONS

The result is that cloud services can be practically used today to provide storage, bandwidth, and messaging, but not computation. Our insight in Priv.io is to use the user’s web browser to provide the computation needed while they use Priv.io. To relax the need of browsers for computation, we design a lightweight process-like environment for users to run computation in the cloud at low cost in Chapter 5. Doing so provides a number of benefits: Using a user’s web browser for computation reduces costs (since users do not need to purchase computation), reduces security concerns (since content is encrypted in the browser, no third-party sees unencrypted content), and is practical (since most cloud providers allow storage and messaging services to be accessed via HTTP). However, only using browser-based computation also presents a few challenges: it results in a system where users are not always online (if a user does not have a browser window open to Priv.io, computation cannot be done on their behalf) and only provides a restricted model of computation (browser JavaScript is sandboxed, and cannot access the local disk or have unfettered access to the networking stack).

4.2.1 Cost study

Before we describe how we address these challenges, we briefly estimate the cost to users if services such as Facebook were implemented in a confederated manner. In other words, if each user contracted with a cloud provider to pay for their use of services such as Facebook, what would the per-user costs be? We examine this question in the context of social networking sites, microblogging sites, and content sharing sites.

Unfortunately, estimating the per-user cost is not entirely straightforward, as data availability is scarce and the costs of optimizations and overhead are hard to estimate. As a result, our goal is not to deduce the exact costs, but rather, to provide a reasonable estimate.

Social networking: Facebook To estimate the cost of storing and serving Facebook content, we use a collection of 651,539 Facebook profiles\(^3\) from a large regional network [110]. The data includes all wall posts, status updates, photos, and videos uploaded by these users. We assume that photos are 64 KB [62], and that videos have a bitrate of 1.403 Mbps [166].

Unfortunately, our data set does not include how often content is viewed; the most detailed statistics on the viewing patterns of Facebook content come from the description of Haystack [62], Facebook’s photo serving system. We use the photo view distribution (Figure 7 in [62]) to parametrize

\(^3\)At the time of collection (2009), Facebook profiles were by default visible to all members of the same regional network.
our calculations. We assume that videos are viewed with the same popularity distribution as photos, but at 1/20th the rate.

**Microblogging: Twitter**  To estimate the cost of storing and serving Twitter content, we use a data set containing an almost complete set of all tweets issued up to September 2009 [77]. This data set contains 1,755,925,520 tweets issued by 54,981,152 users. We observe that the average size of a tweet (including all metadata fields) is 2551 bytes. Unfortunately, we do not have tweets view counts, but we estimate this by using the number of Twitter followers (subscribers) each issuing user has (i.e., every follower views every tweet).

**Content sharing: Flickr**  To estimate the cost of storing and serving Flickr content, we use a data set from early 2008 consisting of 2,570,535 Flickr users sharing 260,317,120 photos [150]. This data set contains all of the users in the large, weakly connected component on Flickr. We know how many photos were uploaded by each user, but we do not know the number of times each photo was viewed. Instead, we derive the view distribution from studies by Yahoo! researchers [227]. We assume that, on average, photos require 4 MB of storage, and 2 MB of bandwidth per view (photos are typically encoded on disk in multiple sizes [62]).

**Analysis**  We estimate the monthly per-user storage, bandwidth, and request costs for each of these three sets of users on Amazon’s S3 service. Figure 4.1 presents the complementary cumulative distributions of the total storage, bandwidth, and requests per user per month (top) and resulting costs (bottom). Figure 4.1(d) presents the overall cost per user per month.

We make a number of observations. *First*, the different systems show different cost characteristics: the cost of hosting Facebook content is dominated by bandwidth (due to the high average user degree, requiring distributing the same content to many friends), the cost of hosting Flickr content is dominated by storage (due to the high resolution of Flickr photos), and the cost of hosting Twitter content is dominated by requests (due to the small but frequent content). *Second*, we observe that for the vast majority of users, the total costs are quite tiny: for 99% of users, the monthly total costs are no more than $0.95 (Facebook), $0.88 (Flickr), and $0.23 (Twitter). *Third*, our calculations assume a

---

4 We note that Facebook receives 120 million new photos and 100 billion photo views per day [62]. Given that photos receive 29% of their lifetime views on their first day [62] and, at the time, Facebook users had an average of 130 friends [104], we estimate that newly uploaded photos receive 1.85 views per friend of the uploader on their first day.

5 At the time of publication, Amazon charges $0.095/GB/month for storage, $0.12/GB for outgoing bandwidth (with the first GB free each month), and $0.004 per 10,000 GET requests [56].

---
Figure 4.1: Complementary cumulative distributions of total monthly (a) storage, (b) bandwidth, (c) requests per user (top) and resulting costs (bottom) for Facebook, Twitter, and Flickr users (note the logarithmic scale on the \(y\)-axis). Also shown is the distribution of (d) total monthly costs. 99% of users would have to pay no more than $0.95 per month in all three scenarios.
nàïve design; optimizations such as content aggregation and caching are likely to provide lower costs in practice.

4.3 Design

We now detail the design of Priv.io, comprised of two components: Priv.io core and applications. Priv.io core provides libraries for accessing user information, manipulating the user’s data, and communicating with other users; most user-facing functionality is built as applications on top of Priv.io core. When user visits [https://priv.io/](https://priv.io/), the Web server returns Priv.io core’s JavaScript. This page allows a user to register, log in, control Priv.io settings, and install applications. It also serves as a container for hosting sandboxed applications, and provides libraries for these applications to use. Below, we describe the design of Priv.io core, followed by how applications are implemented (Section 4.4).

We begin by discussing the assumptions we make (Section 4.3.1), followed by the Priv.io building blocks (Sections 4.3.2 and 4.3.3). We then describe how these are used to implement basic Priv.io functionality (Section 4.3.4).

4.3.1 Assumptions

The Priv.io core design includes three components: the Priv.io Web server, users’ Web browsers, and users’ cloud providers. We briefly overview the assumptions we make about each of these. We assume that some entity runs the Priv.io Web server (for now, our research group runs the server, but it could easily be run by a non-profit organization). As we will see later, the Priv.io Web server receives relatively few requests, and it is feasible to run such a server with few resources (for higher reliability, the site could be served using techniques like geo-replication or content distribution networks). We assume that users are running the latest version of a common Web browser with JavaScript and HTML5 support. We assume the security of DNS (i.e., that an attacker cannot modify Priv.io DNS entries).

We assume that the cloud provider provides certain services, listed below:

- **Storage/Messaging** We assume the provider offers both data storage and messaging (distributed queue) services.

- **REST API** We assume that operations can be performed via a REST API [193], enabling access to the API via JavaScript from the user’s browser.
CHAPTER 4. PRIV.IO: RETHINKING WEB APPLICATIONS

Table 4.1: Summary of required features (storage, messaging, REST API, object versioning, DNS support, and authentication) in Priv.io, and their current support by major providers. Amazon, Azure, and Google support all required services today.

- **Versioning** We assume that the provider supports storing multiple versions of objects.
- **DNS support** We assume that users can access their storage containers via DNS names (e.g., `bob.s3.amazonaws.com` maps to Bob’s storage).
- **Authentication** We assume that the provider allows permissions to be specified on stored objects.

Table 4.1 details which of today’s providers support these features; we observe that three providers exist that can support Priv.io today. We assume that the users’ cloud providers are honest-but-curious, meaning the providers faithfully implement the service that the users have contracted for (e.g., storing objects, retrieving the latest version of objects, delivering messages) but may attempt to decrypt data or messages. Finally, we assume that users’ cloud providers are available, meaning the providers do not close their service without warning (users are of course free to migrate their data to new cloud providers at any time).

4.3.2 Attribute-based encryption

Similar to other content-sharing systems such as Persona [69], Priv.io uses attribute-based encryption (ABE) [65]. In general, ABE dramatically simplifies key management when sharing content with multiple parties. To use ABE, users first generate an ABE public key and an ABE master key (the former is made publicly available and the latter is kept private). Users can then generate ABE private keys for each of their friends, where each ABE private key is generated with one or more attributes such as `friend`, `family`, or `yearBorn=1963`.

Users can encrypt content items using expressions over attributes, and only friends whose ABE private key satisfies the given expression are able to decrypt. For example, one such expression might
family \lor (\text{yearBorn} < 1980)

ABE is collusion-resistant \cite{65}, meaning users cannot collude to decrypt content that they could not decrypt separately. For a more detailed description of ABE, we refer the reader to the paper by Bethencourt et al. \cite{65}.

### 4.3.3 Priv.io building blocks

We now describe the building blocks used in Priv.io; a reference for the notation used is provided in Table 4.2. As is typical in Web-based services sites, users in Priv.io choose a username and password. Each user $u$ has an ABE master key $m_u$ and an ABE public key $P_u$. Each user also has a special ABE private key $p_u^{\text{self}}$ with the attribute $\text{self}$; this allows other users to encrypt messages for $u$ using the $\text{self}$ attribute, similar to more traditional public key encryption.

The Priv.io Web server serves two functions. First, it distributes the Priv.io JavaScript, CSS, and images to the users when they visit \url{https://priv.io/}. Second, it maintains the priv.io DNS domain, which serves as a directory for users’ cloud providers.

The Priv.io JavaScript provides libraries for using the REST APIs of the cloud providers’ storage and messaging services via XML HTTP Requests (XHRs). In order to use these APIs, though, the JavaScript must respect the default same-origin policy enforced by browsers (i.e., by default, the Priv.io JavaScript cannot make an XHR to alice.priv.io unless the HTML document was originally loaded from alice.priv.io). Priv.io addresses this problem in one of three ways: (a) providers such as Amazon’s S3 and Windows Azure allow users to specify a Cross-Origin Resource

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_u$</td>
<td>$u$’s ABE public key</td>
</tr>
<tr>
<td>$m_u$</td>
<td>$u$’s ABE master key</td>
</tr>
<tr>
<td>$p_u^v$</td>
<td>$u$’s ABE private key given to friend $v$</td>
</tr>
<tr>
<td>$p_u^{\text{self}}$</td>
<td>$u$’s special ABE private key with policy $\text{self}$</td>
</tr>
<tr>
<td>$C_u^{\text{user}}$</td>
<td>$u$’s credentials for accessing his cloud services</td>
</tr>
<tr>
<td>$C_u^{\text{friend}}$</td>
<td>$u$’s credentials, given to his friends, allowing limited access to his cloud services</td>
</tr>
</tbody>
</table>

Table 4.2: Notation used in the description of Priv.io.
CHAPTER 4. PRIV.IO: RETHINKING WEB APPLICATIONS

Sharing (CORS) policy, allowing such access, (b) systems like Amazon’s Simple Queuing Service provide a permissive crossdomain.xml file, allowing a small embedded Flash object to make cross-domain requests, or (c) other providers like DropBox allow a stub HTML file to be placed on the target domain, which is used to load the JavaScript in a separate iframe.

All Priv.io encryption and decryption is implemented in JavaScript; more details are provided in Section 4.6.

4.3.4 Priv.io operations

Registration When signing up with Priv.io, a user \( u \) visits [https://priv.io/] and provides their desired username, password, email address, cloud provider, and two sets of provider access credentials (e.g., AWS access/secret keys). The first set of credentials \( (C^{\text{user}}_u) \) are to be used by the user himself, while the second set \( (C^{\text{friend}}_u) \) are to be used by the user’s friends. Of the user-provided data, only the user’s username, email address, and cloud provider are uploaded to the Priv.io server (the email address allows the user to later change their cloud provider).

Meanwhile, the Priv.io JavaScript generates an ABE master key \( m_u \) and ABE public key \( P_u \), as well as an ABE private key \( p^{\text{self}}_u \) with the attribute self. Then, using credentials \( C^{\text{user}}_u \), the JavaScript creates two storage containers on the cloud provider: a publicly-readable container, and a private container that can only be read with one of the user’s two credentials. Finally, the JavaScript creates the user’s message queue, configured so that \( C^{\text{friend}}_u \) is only able to write to the queue.

Upon receiving the user’s registration request, the Priv.io server marks the username as assigned and sets up the user’s DNS entries. Each user has three DNS entries: [username].priv.io maps to the user’s public container, private.[username].priv.io maps to the user’s private container, and queue.[username].priv.io maps to the user’s message queue.

The Priv.io JavaScript then creates two files in the publicly-readable container: public_key containing \( P_u \), and credentials containing

\[
[m_u, p^{\text{self}}_u, C^{\text{user}}_u, C^{\text{friend}}_u]
\]

encrypted using the user’s selected password.

Login After a user is registered, login is straightforward. The user visits [https://priv.io/] and enters their username and password. The Priv.io JavaScript fetches [username].priv.io/credentials, and decrypts the file with the user’s password. If the password was correct, the
CHAPTER 4. PRIV.IO: RETHINKING WEB APPLICATIONS

Figure 4.2: Diagram of login process for user Alice in Priv.io. Alice visits [https://priv.io/](https://priv.io/), obtaining the Priv.io JavaScript. Upon entering her username and password, her browser contacts her cloud provider S3, verifies her password, and communicates with her cloud provider as well as the cloud providers of her friends. Note that the only communication with the main Priv.io server is fetching the original JavaScript.

login can proceed, as the JavaScript now has all of the credentials and keys needed to operate on the user’s behalf. It is worth noting that the only interaction with the Priv.io server is fetching the Priv.io root page; all others are with the user’s cloud provider. A diagram is provided in Figure 4.2.

**Friending** Priv.io is built to allow users to interact with friends, and friends need not share the same storage provider. Users can discover friends either through existing friends (e.g., users can browse the list of their friends’ friends), or via out-of-band means (e.g., users can exchange Priv.io usernames).

To become friends, users need to securely exchange ABE keys ($p^{vu}_{u}$) and credentials for their cloud providers ($C^{friend}_{u}$). To do so, let us assume that Alice and Bob wish to become friends in Priv.io. Alice first fetches Bob’s ABE public key from bob.priv.io/public_key. Then, Alice generates an ABE private key $p^{Bob}_{Alice}$ for Bob, with the attributes Alice assigns to Bob (e.g., colleague). Alice then stores $\left[p^{Bob}_{Alice}, C^{friend}_{Alice}\right]$ encrypted under Bob’s ABE public key with attribute self at the location alice.priv.io/friends/bob. Bob performs similar actions for Alice.

Bob then fetches alice.priv.io/friends/bob, and decrypts it using $p^{self}_{Bob}$. Bob is then able to write to Alice’s queue and read from Alice’s private container (using $C^{friend}_{Alice}$), as well as decrypt Alice’s shared objects (using $p^{Bob}_{Alice}$). Alice fetches bob.priv.io/friends/alice.
CHAPTER 4. PRIV.IO: RETHINKING WEB APPLICATIONS

and has similar privileges. Each of the two stores a copy of the newly-acquired credentials and keys in their own private storage encrypted under the policy self, allowing each to obtain them on subsequent logins. Finally, both remove the encrypted files from their public container.

If Alice and Bob are two hops away (i.e., one of Alice’s friends is also a friend of Bob), Priv.io automatically uses one of the intermediate friends to relay the request. Priv.io sends a message to the intermediate friend, who forwards it on to Bob; Bob is then automatically notified of Alice’s incoming friend request. Otherwise, Alice must tell Bob using out-of-band means that she has issued the request. Since the vast majority of friendships in online social networks are established between users who are friends-of-friends [149], we expect most friend requests to be able to be relayed.

**Default attributes** To simplify sharing, Priv.io generates private keys for friends with two default attributes, in addition to any user-provided attributes. The first attribute is @username, which allows users to share content with only a single user (e.g., if Alice wished to share content only with Bob, she could specify the policy @bob). The second attribute is @@, which is given to all friends. This attribute allows users to share content with all of their friends.

**Modifying friend permissions** Users in Priv.io may want to change the permissions given to friends, either to add attributes, remove attributes, or remove the friend entirely. Adding attributes simply requires generating a new ABE private key for the friend, and giving the friend the new key. Removing a friend is the same as removing all attributes from the friend.

Removing attributes from a friend requires re-keying. To simplify this process, Priv.io assigns an integral value to each ABE attribute, where the value is initialized to 0 and is incremented each time a user has that attribute removed. For example, consider user Alice with friends Bob and Charlie assigned the following attributes

\[
\begin{align*}
\text{Bob} & : \ @@=9, \ @bob, \ work=2, \ soccer=1 \\
\text{Charlie} & : \ @@=9, \ @charlie, \ work=2, \ it\_dep=3
\end{align*}
\]

Now, if Alice wishes to remove the work attribute from Charlie, Priv.io increments the work value to 3, and reissues an ABE private key to Bob with the attributes

\[
\text{Bob} : \ @@=9, \ @bob, \ work=3, \ soccer=1
\]

(note that Priv.io does not need to re-issue a key to Charlie). Any new content Alice shares with the work attribute is encoded with the policy work≥3 ensuring that only friends with re-issued keys
have access.

**Communication** Priv.io uses the messaging service of users’ cloud providers to enable communication with friends. After logging in, the Priv.io JavaScript connects to the user’s queue and processes any messages. While online, the Priv.io JavaScript remains connected the queue and continues to process any additional messages. The only Priv.io control messages that are sent are updating ABE private keys and friendship requests; all other messages are application-level messages and are delivered to the corresponding application (discussed in the following section). Friends do not need to be online for the user to send messages to them; cloud providers typically buffer messages for multiple weeks.

**Caching encryption policies** ABE operations are significantly more expensive than symmetric encryption operations. To mitigate the impact of expensive ABE operations, Priv.io is configured to use ABE to only encrypt and decrypt AES keys. Actual content objects are then encrypted under AES keys. Furthermore, Priv.io caches the AES keys used for each unique encryption policy; doing so allows Priv.io to only invoke expensive ABE operations when establishing friends, modifying friends, or using a new encryption policy.

### 4.4 Third-party applications

Almost all user-facing functionality in Priv.io is implemented as applications on top of the Priv.io core libraries. Similar to existing sites like Facebook, applications may be implemented by third parties, and need not be trusted. Applications are implemented using HTML and JavaScript, and are displayed to the user as part of the Priv.io Web page. Thus, the challenge in Priv.io is to provide rich support for third-party applications, while simultaneously providing strict guarantees of security and privacy for users. In particular, we wish to ensure that applications cannot leak user information back to the application provider or any other entity.

#### 4.4.1 Application API

Priv.io presents an API for applications to be written against. Since Priv.io is implemented entirely within a user’s browser, the API is implemented within the browser as well. Priv.io is designed to

---

6 Any previously-shared content will still be accessible to Charlie, as it was encoded with $\text{work} \geq 2$. If this is not desired, content can be re-encrypted with an updated policy.
### CHAPTER 4. PRIV.IO: RETHINKING WEB APPLICATIONS

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestPermissions(c)</td>
<td>Requests access for the application to methods c</td>
</tr>
<tr>
<td>getUsername()</td>
<td>Returns the user’s username</td>
</tr>
<tr>
<td>getFriends()</td>
<td>Returns usernames of the user’s friends</td>
</tr>
<tr>
<td>getFriends(u)</td>
<td>Returns usernames of friend u’s friends</td>
</tr>
<tr>
<td>getAttributes()</td>
<td>Returns the set of attributes assigned to the user’s friends</td>
</tr>
<tr>
<td>store(k, v, p)</td>
<td>Stores data v under key k, encrypted with policy p</td>
</tr>
<tr>
<td>retrieve(u, k)</td>
<td>Returns the value previously stored under key k in u’s storage; may return multiple versions</td>
</tr>
<tr>
<td>send(u, m)</td>
<td>Sends message m to friend u’s instance of this application</td>
</tr>
<tr>
<td>receive()</td>
<td>Receives any pending messages</td>
</tr>
<tr>
<td>delete(m)</td>
<td>Marks a previously received message as successfully processed</td>
</tr>
</tbody>
</table>

Table 4.3: Subset of the Priv.io application API, covering API permissions, user information, storage, and communication. All methods are protected by permissions (users must permit applications to make API calls).

Support social networking-like applications (Facebook, Twitter, and Flickr), but could also be used to build other applications (e.g., Web-based document editing, shared calendars, etc.). Applications in Priv.io are logically separate and cannot exchange data or messages.

Similar to the approach taken by services such as Facebook, applications must request and receive permission from the user to make various API calls. When requested, Priv.io presents a dialog to the user, identifying the application and the access that it desires. Priv.io records the user’s response, and then uses the specified policy to allow or deny API calls by the application.

A subset of the Priv.io application API is presented in Table 4.3 and is discussed below:

**User information**  Similar to the Facebook API, applications can request profile information about the current user or any of the user’s friends.

**Storage**  Applications are allowed to store and retrieve data from the user’s private storage container. Each application is given a storage folder, and is only able to access its own content (applications
CHAPTER 4. PRIV.IO: RETHINKING WEB APPLICATIONS

cannot read other applications’ data. When storing data, applications specify an ABE policy for encrypting the data (e.g., self for only the user, or family for all friends with the attribute family). Applications can also request to read data in friends’ containers written by another instance of the same application, but can only do so if the user is able to decrypt the data.

**Communication** Applications are allowed to send and receive messages to and from the same application run by friends. This is implemented by sending messages to the specified friend’s message queue, and reading from the user’s own queue. The Priv.io code multiplexes and demultiplexes messages, and buffers any incoming messages for an application until it is run.

4.4.2 Managing applications

Developers register Priv.io applications with the Priv.io Web server similar to user registration; each application is given a unique name (e.g., newsfeed). The Priv.io Web server makes the application available to users from a subdomain that is hosted by the Priv.io Web server (e.g., the Newsfeed app is available at [http://newsfeed.app.priv.io/](http://newsfeed.app.priv.io/)). The Priv.io Web server is responsible for serving all application HTML, JavaScript, CSS, and images.

Users install an application by providing Priv.io with the app name (e.g., newsfeed); Priv.io records all apps that a user has installed in the user’s private storage, along with their permissions, and reloads the list upon each login. Users can later remove an application by asking Priv.io to delete it. Priv.io then removes the application from the user’s list, and deletes any application-stored data and queued messages.

4.4.3 Security and privacy

Running third-party applications in Priv.io brings up two security concerns: First, can we restrict applications to only using the Priv.io API? In other words, can we prevent applications from accessing Priv.io JavaScript objects, or conducting attacks like cross-site scripting [162][163], frame hijacking [60], or frame busting [181]? Second, can we prevent applications from leaking user data obtained from the Priv.io API, either via XHRs or by loading DOM (document object model) objects?

In order to address these concerns, Priv.io sandboxes all third-party application code using iframes, loading each application in a separate iframe. Applications access the Priv.io API using the `postMessage` [123] feature in HTML5 to send API requests to the main Priv.io frame (the application’s parent frame). If the API request is allowed (based on user preferences), the
response is delivered back to the application via `postMessage` on the application’s iframe.\footnote{Applications cannot impersonate other applications (making messages sent via `postMessage` appear as if they are from another origin), since the `postMessage` mechanism is secured by the browser.} This mechanism prevents applications from directly accessing any Priv.io JavaScript objects.

However, iframes by default are allowed to load arbitrary content, meaning an application could leak user information obtained from the Priv.io API by loading DOM objects. For example, an application wishing to leak the information that user Alice is friends with Bob could request to load `http://malicious-domain.com/alice-bob.png`. To constrain applications from leaking data, each application’s iframe is loaded with a Content Security Policy (CSP). In brief, CSP allows a server to specify what client-side actions the pages it serves can take. Priv.io instructs the browser to disallow the application’s iframe from making any network requests other than to `[appname].app.priv.io` (which is hosted by the main Priv.io server). As a result, the application is constrained to only using the Priv.io API.

### 4.4.4 Limitations

Due to the architecture of Priv.io, there are a few applications that exist on sites today that cannot be replicated. For example, any operation that requires a global view of the user data (e.g., global search) is not possible, as there is no entity in Priv.io that can view all data. Other examples include applications that allow users to interact with random users who they are not friends with (e.g., ChatRoulette).

However, many services that might appear to require global information can usually be at least partially replicated. For example, a “friend suggestion” feature could potentially be implemented as an application that collects the structure of the user’s local network (friends and friends-of-friends) and suggests others the user likely knows \footnote{CSP is a new security mechanism provided in HTML5, and is supported by the latest versions of many browsers.}. We leave a more in-depth exploration of such techniques to future work.

### 4.4.5 Demonstration applications

To demonstrate that existing Web-based services’ functionality can be reproduced in Priv.io, we outline two applications that we have implemented.

**Newsfeed** Priv.io provides functionality similar to Facebook’s News Feed via the Newsfeed application. In the application, users start a thread by posting a comment, uploading a photo, or sharing a
Figure 4.3: Diagram of how Newsfeed uses storage in Priv.io. Each user stores their own content, and the user who creates each thread stores a feed file linking together all comments.

Newsfeed uses the communication API when a user comments on a friend’s feed. A message is sent to the user who owns the feed containing a reference to the comment; when the user owning the feed receives the message, Newsfeed adds the reference to the feed object, allowing other friends to then see the comment.

When users launch the Newsfeed application, it scans all of the friends’ feedlists, integrating all of the visible feeds into a single news feed. A diagram showing Newsfeed’s use of the storage API is presented in Figure 4.3.

Chat Priv.io allows users to “chat” by providing the instant messaging application Chat. The application is written entirely using the communications API. Users invite others to chat via an invitation message, and each chat message is broadcast to all other participants of the chat. As a result, Chat provides similar functionality to applications on existing sites, and could easily be
CHAPTER 4. PRIV.IO: RETHINKING WEB APPLICATIONS

extended to (optionally) archive conversations, allow file transfers, and so forth.

4.5 Discussion

We now discuss a few deployment issues with Priv.io.

Consistency and reliability In Priv.io, users only write to their own storage location, preventing a number of consistency problems. However, users may be logged in to Priv.io from multiple locations at once, exposing Priv.io to potential consistency issues due to multiple writers. To address this problem, Priv.io leverages the object versioning (described in Section 4.3) supported by the cloud provider. Specifically, when Priv.io writes an updated version of an object to the user’s storage location, it first checks to see if there is a newer version of the object present than the one that its pending write is based on. If such an object exists, Priv.io first downloads the updated object, and merges the two. Finally, Priv.io writes the new version of the object back, and deletes both of the previous versions. As a result, Priv.io itself and all applications must be able to perform merges on storage objects that may have diverged.

Priv.io allows each user to select the desired level of availability and durability for their content through the choice of their cloud provider. For example, on Amazon’s S3 service, users can choose between eleven 9s of durability or four 9s of durability for content, at different price points.

Reliable message delivery Because Priv.io is implemented within a browser, the user could decide to close the window at any time, thereby killing all Priv.io JavaScript. This property makes implementing reliable message delivery for applications particularly challenging, as a message may be delivered to an application, but the Priv.io window could be closed before the application finishes processing the message. To avoid such a scenario, Priv.io requires applications to explicitly call delete \(m\) on each message \(m\) after they have finished processing it. Only at that point is the message deleted from the cloud provider’s message queue\(^9\). Thus, Priv.io provides at-least-once delivery semantics for messages, and applications must be written to tolerate receiving the same message multiple times.

\(^9\)Message queues like Amazon’s SQS and Microsoft’s Azure Queue Service support similar semantics; if a recipient dies before marking a message as processed, the message is eventually delivered again.
CHAPTER 4. PRIV.IO: RETHINKING WEB APPLICATIONS

Security  To prevent man-in-the-middle attacks on Priv.io users, all interaction with the Priv.io Web server is over HTTPS. In the future, we aim to provide support for DNSSEC to ensure the integrity of Priv.io DNS entries as well (e.g., to prevent cache poisoning attacks).

Each user’s encrypted credentials file is stored in a publicly visible location (e.g., alice.priv.io/credentials). As a result, we are particularly concerned about brute-force password cracking attacks. To reduce the ability for an attacker to decrypt a user’s credentials file, we first choose a random initialization vector when encrypting the credentials and salt the password, preventing attackers from using rainbow tables [169]. Second, we use the PBKDF2 password strenghtener [132], which greatly raises the cost of a brute-force attack. Third, we require strong passwords for users in the form of pass phrases [177], which often possess more entropy than basic passwords.

An additional concern is whether an attacker can perform a man-in-the-middle attack during friend establishment, or intercept the exchanged friend credentials. We first note that all friend exchange information is encrypted with the destination’s ABE public key, making it unreadable by the attacker. Second, since each user is the only entity that is able to write to their public storage area, malicious users are unable to forge friend requests with credentials of their choosing.

Privacy  As privacy is of paramount concern in Priv.io, we now briefly analyze what an attacker can determine about other users. We first note that the attacker can only read the user’s public storage location; he cannot access the user’s queue or private storage location. There are three types of objects stored in the public storage area: the credentials file, the public_key file, and the temporary friend request files discussed in Section 4.3.4 (recall that these files are available while a friend request is outstanding). Thus, malicious attackers are able to determine if user $u$ is currently trying to become friends with user $v$. However, we note that attackers must guess the identity of $v$ correctly (attackers cannot “list” all request files), and the window of opportunity is likely to be short.

Incremental deployment  Signing up for Priv.io is more complicated than signing up for existing services like Facebook: with Priv.io, users must first sign up with a cloud provider, and then register for Priv.io using those credentials. Luckily, signing up with cloud providers is often relatively simple; for example, signing up for Amazon Web Services requires only filling out two Web forms (personal information and billing information), and much is carried over if the user already has an Amazon account. Regardless, we will continue to look for opportunities to lower the burden for signing up with Priv.io.
For social networking-like services, the network effect\(^\text{[10]}\) has resulted in entrenched service providers (e.g., Facebook); being the only one of your friends to be on a new social network is unlikely to provide significant benefits. Thus, it may be difficult to initially attract significant numbers of users to Priv.io without an established user base. However, we note that Priv.io is not limited to social networking applications, as current popular Web-based applications like Google Docs could easily be implemented in Priv.io and afforded the same privacy benefits. This approach may serve as a mechanism for attracting users, who later may also use the social networking features.

**Finer-granularity computation** The design of Priv.io is partially driven by the high cost of purchasing computation from cloud providers. However, as new cloud providers enter the market (e.g., resellers of existing cloud providers) or the price of computation drops, purchasing computation may become feasible. If so, this would open new opportunities for Priv.io, but may also come with a different set of privacy properties. For example, if users could purchase computation at a low-enough cost, Priv.io could easily interact with existing legacy systems that cannot be supported with in-browser computation (e.g., applications such as Google Mail—requiring SMTP—could be replicated in a confederated manner). However, doing so would allow the cloud provider to potentially view raw content (as incoming emails would be observed in plaintext by the user’s server). Regardless, we plan to explore ways of integrating purchased computation into Priv.io as future work.

### 4.6 Evaluation

We now present an evaluation Priv.io, covering both microbenchmarks and measurements of Priv.io performance under different workloads.

We have implemented a prototype of Priv.io that supports almost all of the features described thus far. Priv.io currently supports using Amazon Web Services as a cloud provider, with support for SQS and S3. Support for Windows Azure and Google Cloud Platform is in progress. Priv.io supports third-party applications; both of the applications described in Section 4.4.5 are implemented and installed by default for each user.

Since all of Priv.io is implemented in JavaScript, it is open-source and available to the research community at [https://priv.io/](https://priv.io/) The implementation is compatible with the latest versions of common desktop Web browsers, as well as browsers on Android and iOS. Alternatively, Priv.io can

---

\(^{10}\)The *network effect* describes the value of a network as the number of participants grows. In brief, it captures the notion that with each new user, the number of potential links increases, thereby increasing the value for all participants.
Figure 4.4: Priv.io encryption and decryption performance when run in different browsers. Shown is (a) AES encryption and (b) AES decryption for objects of different sizes. Also shown is (c) ABE key generation for keys with an increasing number of attributes (ABE encryption under policies of increasing lengths shows very similar behavior). Upper three lines denotes results from mobile devices, while the lower three are from desktop browsers.

also be implemented as native mobile applications, following the same cryptographic procedural to access user data and communicate with other clients. However, popular mobile OSes have limited support for creating nested sandbox environments in an application, which is required in implementing Priv.io applications. One possible solution is to embedded browser frames in a native mobile application. We leave this to future extensions.

The Priv.io core code represents 5,931 lines of JavaScript, excluding encryption and user in-
CHAPTER 4. PRIV.IO: RETHINKING WEB APPLICATIONS

<table>
<thead>
<tr>
<th>Browser</th>
<th>setup time (s)</th>
<th>decrypt time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safari</td>
<td>0.91</td>
<td>0.99</td>
</tr>
<tr>
<td>Firefox</td>
<td>0.63</td>
<td>0.36</td>
</tr>
<tr>
<td>Chrome</td>
<td>1.22</td>
<td>1.38</td>
</tr>
<tr>
<td>Android(Cr)</td>
<td>12.92</td>
<td>14.54</td>
</tr>
<tr>
<td>Android(FF)</td>
<td>14.63</td>
<td>13.08</td>
</tr>
<tr>
<td>iPhone</td>
<td>14.40</td>
<td>15.92</td>
</tr>
</tbody>
</table>

Table 4.4: Average time taken to generate an ABE master and public key (setup) and decrypt an ABE message (decrypt) in various browsers.

interface libraries. We use the Stanford JavaScript Crypto Library for all AES operations. We used Emscripten [241] to compile the Ciphertext Policy ABE library [84] (as well as other dependencies) into JavaScript. The resulting encryption library totals 621 KB. All of these libraries are static and can easily be cached by Web browsers.

4.6.1 Microbenchmarks

Storage size Priv.io objects require storage on the user’s cloud provider and encounter overhead, due both to encryption metadata (initialization vectors, etc.) and the base64 encoding used. The fixed overhead of using AES encryption is 145 bytes, and the fixed overhead of using ABE encryption is 345 bytes plus approximately 370 bytes per policy attribute. The base64 encoding introduces an additional 33% overhead. The ABE public keys are 1184 bytes, the encrypted credentials file averages 1457 bytes, and each friend request file averages 2400 bytes.

Content loading latency Loading objects in Priv.io enjoys the benefits of the user’s cloud provider; we found the latency to be comparable to loading content from traditional Web sites. Using the us-east-1 Amazon Web Services S3 storage service and loading to a client located in Boston, we found the latency of loading 64 KB objects via Priv.io to be 154 ms.

Encryption and decryption We now examine the encryption and decryption performance in Priv.io. We first focus on AES encryption. Using the latest version\textsuperscript{11} of common browsers, we encrypt and decrypt objects of varying sizes using the AES library. We repeat each test 10 times, and report the average in Figures 4.4(a) and 4.4(b). We observe that AES encryption and decryption

\textsuperscript{11}Safari 6.0.4, FireFox 21.0.1 and Chrome 27.0.1453, all on OS X 10.8.3; Android Chrome 27.0.1453 and Firefox 21 on a HTC One X (AT&T); Mobile Safari 6.0 on an iPhone 5 running iOS 6.1.4. Mobile devices are connected via WiFi.
Table 4.5: Average time (in seconds) taken to load the basic Priv.io code after login, as well as the Newsfeed and Chat applications, in various browsers. Android Chrome and Mobile Safari do not support Flash, which is used to make cross domain requests for Amazon SQS.

time correlate linearly with object size, and are fast: for 100 KB objects, both are under 43 ms for all desktops and under 327 ms for all mobile devices.

We now examine the performance of ABE. There are four ABE operations that we need to consider: setup (generate public and master keys), gen_key (generate a private key), encrypt, and decrypt. Of these, the compute time of gen_key and encrypt depend strongly on the number of attributes; the compute time of the other two is relatively constant.

We first report the performance of setup and decrypt in Table 4.4. We observe that the performance ranges from under 1.4 seconds on desktop browsers to about 15 seconds on mobile devices. We next examine the performance of gen_key and encrypt, shown in Figure 4.4(c) (the two operations show almost identical performance, so we only present the results for gen_key for brevity). We observe a strong linear relationship with the number of attributes used, ranging from about one second for a single attribute on desktop browsers to 45 seconds for five attributes on mobile devices. We again note that the expensive nature of ABE operations is unlikely to impact users on a regular basis, as they are only necessary when adding/modifying friends, or encrypting content under a never-used-before policy (i.e., most sessions require no ABE operations).

4.6.2 User-perceived performance

We now examine the user-perceived performance of Priv.io. In evaluating Web services, the primary metric of interest is typically latency; we therefore focus on latency here. We are primarily concerned with three issues: First, what is the loading time of the basic Priv.io code when a user logs in, absent any applications? Second, what is the loading time of applications, both with and without subsequently loaded content? Third, what is the latency of sending application-level messages?
CHAPTER 4. PRIV.IO: RETHINKING WEB APPLICATIONS

Figure 4.5: Average Newsfeed loading time with varying amounts of content, when content is loaded from multiple friends. We observe that the loading time increases linearly with the amount of content, as expected.

Priv.io loading time We first measure the time taken for users to log in and load the basic Priv.io code, absent any applications. To measure this, we disable all applications and measure the time taken from when a user clicks “Log in” until the Priv.io code is fully loaded. We run this experiment on all of the browsers listed above, clearing the browser cache after each experiment, and repeating the experiment 10 times. The average loading time is shown in Table 4.5 under the “Priv.io” column. We observe that the loading time is quite fast, between 250 and 340 ms on desktops and between 0.6 and 2.8 seconds on mobile devices.

Application loading time Next, we explore the loading time of applications. We record the time taken to load the Newsfeed and Chat applications once the Priv.io code is loaded; in each instance, the applications are empty and contain no user data (we explore the loading latency when user data is present below). The results are presented in Table 4.5 under the application columns. We observe that the loading time is consistent across different applications, and ranges from 90 to 200 ms on desktops to between 360 and 980 ms on mobile devices.

Next, we explore the loading time of applications when user data is present. To measure this, we use the Newsfeed application, and create users with varying amounts of Newsfeed content from varying numbers of friends. Specifically, we load up to 15 Newsfeed items (i.e., one “page” of Newsfeed items), with each friend providing three items (i.e., we create one user who loads three items from one friend, another user who loads three items each from two friends, and so on). The average loading time is presented in Figure 4.5. We observe that the loading time increases linearly with the number of content items loaded, and that the desktop browsers are substantially quicker in loading content, as expected. However, we observe that the loading time is reasonable in all cases: below 515 ms for desktops and below 5.1 seconds for all mobile devices.
CHAPTER 4. PRIV.IO: RETHINKING WEB APPLICATIONS

Message latency  Finally, we examine the latency of sending application-level messages. To do so, we use the Chat application, measuring the time for a user to send a message to a friend, and for the friend to reply. Both sender and receiver logged in on the same machine and browsers (note that the message itself must be delivered via the cloud provider’s servers). We repeat this experiment in different browsers, and for each browser, we send 10 round trip messages and calculate the average. We find that the round trip time varies from an average of 637 ms on Chrome to 1.3 seconds on Safari\textsuperscript{12}, indicating that cloud providers’ messaging can easily be used for human-timescale communication.

Overall, our results indicate that Priv.io is practical on the desktop Web browsers of today, with most user-facing loading times on the order of a second. However, mobile devices present challenges for Priv.io, as their lower computational resources result in higher latencies. Our results show that Priv.io does work on these devices, and as they become more powerful, accessing Priv.io from them will become more practical.

4.6.3 Small-scale deployment

We have deployed Priv.io on a small scale within our department. Unfortunately, it is difficult to measure the primary benefits of Priv.io to our users: improvements in privacy and control over data. As of this writing, 28 graduate students and professors have joined Priv.io and are using the Newsfeed and Chat applications. There were a total of 88 friendships recorded, for an average of 3.82 friends per user. Our users have accessed the service using a variety of operating systems, browsers, and desktop/mobile devices (a total of 23 different User-Agent\textsuperscript{s}). In total, our users have posted 221 items to Priv.io, most of which are comments in the Newsfeed application.

4.7 Summary

With the increasing popularity of web-based services, users today have access to a broad range of free sites, including social networking, microblogging, and content sharing sites. Due to the client–server architecture of the web, site operators have unrestricted access to user data storing on their servers. In order to offer a service for free, service providers typically monetize user content, selling user information to third parties such as advertisers. As a result, users have little control over their data or privacy.

\textsuperscript{12}This latency could be further reduced if Amazon’s SQS supported Cross-Origin Resource Sharing, which would eliminate the need for a Flash-based work-around.
CHAPTER 4. PRIV.IO: RETHINKING WEB APPLICATIONS

In this chapter, we presented Priv.io, a new approach to building Web-based services using a confederated architecture. In Priv.io, each user is responsible for providing the resources necessary to support their use of the service; this is accomplished by contracting with cloud providers (for storage, bandwidth, and messaging) and by using the user’s Web browser (for computation). As a result, in Priv.io, users retain control of their own data, users are not required to reveal their information to any centralized entity, and users enjoy a highly reliable and available service. We demonstrated that implementing many popular services with Priv.io is both practical and affordable: Most users would pay less than $0.95 per month, and Priv.io works today on the latest versions of common Web browsers as well as (more slowly) on Android and iOS mobile devices.
Chapter 5

Picocenter: Rethinking computation

Priv.io demonstrated a new architecture for Web-based services based on user-provided cloud storage and application logic implemented in modern web browsers. The system relies on the browser sandbox and in-browser computation to share content. In situations where browsers’ capabilities are limited, such as mobile devices, Priv.io may not provide good performance, such as listening on sockets or using arbitrary network protocols. Also, not all application features can be implemented in the browser sandbox, such as listening on sockets. Even worse, once browsers are offline, most of Priv.io’s features stop working, meaning no message notification, background processing, and so forth.

In this chapter, we outline an alternative approach for cloud computation, based on a process-like abstraction rather than a virtual machine abstraction. The new model can help Priv.io applications continue working even when users stop running their browsers. It works by allowing users to execute their jobs in cloud processes at low cost. Users are billed only for the resources (CPU, memory) they use, making long-lived but largely-idle applications economically feasible. We present Picocenter, a hosting infrastructure for such applications that enables use of legacy applications. The key technical challenges in Picocenter are enabling fast swapping of applications to and from cloud storage (since, by definition, applications are largely idle, we expect them to spend the majority of their time swapped out) and maintaining security and isolation between applications. We first presented Picocenter as a conference paper in EuroSys’16 [244].
CHAPTER 5. PICOCENTER: RETHINKING COMPUTATION

Figure 5.1: Picocenter represents a unique point in the space between existing abstractions by allowing users to run arbitrary processes that are, in expectation, long-lived yet mostly idle.

5.1 Motivation

Over the past few years, website operators have turned to cloud computing services for hosting their web services. As a result, cloud vendors have commoditized computing resources, and build their cloud computing systems for redundancy, scale, resilience, and security. Although web services are among the first applications, cloud computing today has had to evolve to meet the increasingly wide range of user applications.

To adapt to users’ specific needs, there are now two primary models for cloud computing services, each tailored to a particular use case (Figure 5.1): First, Infrastructure as a Service (IaaS) systems, such as Microsoft Azure and Amazon EC2, offer the greatest generality—users can run arbitrary computations and have arbitrary network-facing applications—but are most effectively used by applications that are either mostly active (e.g., a popular web server) or short-lived (e.g., finite but intensive computation). The generality of IaaS comes at the cost of users having to assume the responsibility and overhead of launching, running, and managing an operating system to support their applications. Largely to address these challenges, Platform as a Service (PaaS) systems, such as Google App Engine (GAE), Heroku, and Amazon Lambda, emerged to facilitate cloud application development. Inverse to IaaS, PaaS systems limit generality—users must program their applications according to a constrained API using managed languages (typically permitting only event-driven code that cannot bind to new ports)—and are most effectively used by applications that are idle most of the time.

Unfortunately, neither of these models of cloud computation are well-suited for supporting a class of services that are what we call long-lived but mostly idle (LLMI). LLMI applications include a variety of services, such as personal email and web servers, decentralized social network nodes [69, 70, 221], per-user mobile cloud offload servers [92, 93, 184, 188, 201], personal cloud-based storage services [199], or rendezvous services [41]. Because these are LLMI, PaaS would seem to be
CHAPTER 5. PICOCENTER: RETHINKING COMPUTATION

the best fit, yet we are not aware of any PaaS API that supports such applications. There are many binary executables publicly available that implement these services, making IaaS a reasonable fit, but running them in an always-on VM would unnecessarily waste cloud resources (and thus cost more than necessary), as they are mostly idle. Moreover, in both current offerings, the user overhead to start an application is high: IaaS users must set up and manage a VM, while PaaS users must reimplement their service within the cloud provider’s limited API.

In this chapter, we rethink computation for end users by leveraging cloud computing resources. We present Picocenter, an intermediate point between IaaS and PaaS systems that efficiently supports LLMI processes in a cloud environment. More concretely, Picocenter simultaneously (1) allows users to deploy arbitrary binaries (without reimplementing to meet a constrained API) in a more cost-efficient manner than existing services, and (2) allows cloud providers to increase their revenues by enabling a new set of users and applications.

Picocenter provides two key abstractions that make it a unique point in the space of cloud computing solutions. First, to tenants (those who launch applications), it appears that they have sole access to an OS, while in reality there may be many users running in isolation within the same OS. This virtualization is similar to approaches like Docker [91], unikernels [38], and OSv [131]. Unlike these and PaaS systems, however, Picocenter supports arbitrary computation (including forking which is not supported by unikernels and OSv), arbitrary network protocols (AppEngine only supports HTTP), and event handling (Amazon Lambda [54] supports only a pre-defined set of Amazon event triggers). Similar to systems like Docker [91], Picocenter leverages prior work on Linux containers [145] to provide a client execution interface that supports legacy applications, ensures security and isolation, and, most importantly, scales to support a workload where a large number of applications are alive, but not actively running.

Second, to clients (those who interact with applications), Picocenter provides the abstraction that services are always up and running because they quickly respond to requests, though in reality Picocenter may swap entire processes off of a running machine and store it in long-term storage (such as Amazon S3). We achieve this by supporting checkpoint and restore—based on Linux’s Checkpoint/Restore in Userspace (CRIU) [147]—as well as dynamic page loading from cloud storage. To avoid costly delays when restoring applications, we develop an ActiveSet technique that learns the set of commonly accessed memory pages when requests arrive; we first prefetch those pages from cloud storage before fetching additional pages on-demand. Our ActiveSet technique represents a prediction of the memory pages that are likely to be accessed the next time an application is restored.

While each of these two goals have been addressed separately in prior work, we are the first to
CHAPTER 5. PICOCENTER: RETHINKING COMPUTATION

achieve them in tandem. Picocenter is therefore the first system that has had to tackle the performance challenges that arise when checkpoint/restore is applied to containers (we detail our implementation in Section 5.5).

The remainder of this chapter is structured as follows. In Section 5.2, we present an overview of LLMI applications. We describe Picocenter’s design in Section 5.3, its approach to enabling efficient swapping of applications in Section 5.4, and its implementation in Section 5.5. In Section 5.6, we present microbenchmarks and an end-to-end evaluation on Amazon EC2 which show that Picocenter is able to provide the abstraction that users’ processes are always running, even when the limited computing resources demand that they be swapped out when idle. Finally, we summarize in Section 5.7.

5.2 Long-lived mostly-idle applications

Long-lived, mostly idle applications are typically network-based services, such as personalized email servers, individual users’ web servers, IRC/XMPP chatting servers, or nodes in decentralized applications [69, 70, 221]. These LLMI applications all have several common characteristics:

- **Long-lived:** The applications typically provide network-based services, and often run for long periods of time.

- **Mostly idle:** The applications are largely I/O-bound, and spend most of the time waiting for incoming messages.

- **Short response time:** The applications may have latency requirements to responding to incoming messages (e.g., web servers or chat servers), meaning incoming messages should not be queued for long periods of time.

- **Low per-request processing:** The time required to process an individual message or event is typically short; the applications do not become CPU-bound for long.

There is no obvious PaaS-style “platform” or constrained API upon which to base all LLMI applications; some bind to multiple ports (such as an HTTP/HTTPS server) or fork many processes (such as BIND and vsftpd). Therefore, ideally, any arbitrary binary application could be run in an LLMI fashion. Moreover, to reduce resource consumption for cloud providers (and cost for users), LLMI applications would ideally consume as few resources as possible while they are not actively running. Picocenter provides these two features by drawing from a wide range of prior work.
CHAPTER 5. PICOCENTER: RETHINKING COMPUTATION

5.3 Picocenter Architecture

In this section, we present the design of Picocenter’s architecture and describe its components.

5.3.1 Interface to cloud tenants

We begin by describing what Picocenter looks like to cloud tenants. To begin using Picocenter, a tenant should provide:

- **Tarball** A filesystem containing any executables, configuration files, and libraries the application needs, as a tar file. Picocenter provides a base Linux image, so the tenant only needs to provide any missing or modified files.

- **Initial process** The initial process within the provided filesystem that should be executed, and its arguments; this process is free to launch any child processes, etc.

- **Pico-configuration** A list of the external protocols and ports the application will bind to.

For example, the application binary might be a web server such as Nginx, and the filesystem image would contain the binary and the set of documents Nginx should serve (along with Nginx’s configuration /etc/nginx.conf), and the list of external protocols and ports would be: TCP port 80. It is important to note that the tenant can specify particular ports or simply a number of ports (if a specific port is not needed).

If Picocenter accepts the tenant’s application, it will begin executing the application and will return to the tenant a unique DNS name where the application can be reached (on the ports specified at submission time). If the application becomes idle, Picocenter may swap the application (and its filesystem) to either local disk or remote cloud storage (e.g., Amazon’s S3) in order to allow for other active applications to be run. We refer to applications that are swapped out as *inactive* applications. However, Picocenter will maintain the illusion to the tenant that the application is always running: when the tenant attempts to access the application again in the future, or a long-lived timer in the application expires, Picocenter will transparently retrieve it from disk or cloud storage and restore the application where it left off. We refer to this as *reviving* the application.

Thus, to a tenant of Picocenter, it appears that their application is alive and running the entire time, but during its lifetime, it may in fact be swapped out and swapped back in on a different physical machine. The tenant is primarily billed for the time when the application is actively running, so LLMI

\footnote{Our requirements are similar to what is contained inside of a Docker image.}
CHAPTER 5. PICOCENTER: RETHINKING COMPUTATION

applications are likely to be significantly more affordable when compared to an always-running VM approach.

5.3.2 Challenges

Next, we motivate our design decisions by discussing the challenges with a process-based cloud computing model.

**Transparent checkpoint and restore**  Lowering operational costs for the cloud provider allows for savings to be passed on to clients. Picocenter will need to be able to support application checkpointing and swapping, and later restoring on a potentially different machine. This can happen both as the set of active applications changes, as well as when the load on different hosting machines varies. To ease the burden on cloud tenants, migration must be transparent to application processes.

**Backwards compatibility**  Picocenter needs to work with the rich set of applications that exist today, and not require any source code modifications or recompilation.

**Partial swap ins**  Since applications are swapped-out to cloud storage, downloading an application’s entire memory image before resuming it would likely carry a large latency penalty. Instead, Picocenter must support *partial swap ins*, where the hosting machine can restore the application with only a portion of its memory having been downloaded.

**Application storage**  Applications are presented with a private filesystem, consisting of a base image, user-provided data, and application-created files. This filesystem must be maintained as the application is swapped to cloud storage and restored (potentially on a different hosting machine).

5.3.3 Architecture overview

We present an overview of the Picocenter architecture in Figure 5.2, detailing the interactions between Picocenter and external entities. There are two external entities: the application owners, *tenants*, who deploy services in Picocenter and *clients*, who interact with the applications hosted on Picocenter using existing applications and devices.

At a high level, Picocenter is internally composed of two components. The first is a logically centralized component called the *hub* that performs global resource allocation, decides where to push new applications and where to route packets for swapped-in applications (e.g., the location of active
applications, as well as metadata about inactive applications). The hub serves as a central point of management control in Picocenter.

The second is a virtualization layer akin to a modern hypervisor called the worker, that actually hosts the tenant’s applications. There are multiple workers that work with the hub, and each worker performs local resource allocation. The worker is assigned with applications to run by the hub, but determines when to swap out running applications and performs optimizations to ensure that active applications maintain a short response time.

5.3.4 Hub

The hub manages global resources within Picocenter: allocating IP addresses and ports, assigning domain names, and assigning applications (both new and revived) to workers. To do this, the hub must be informed of any new applications to be launched as well as when applications are swapped to cloud storage: it receives requests from tenants to deploy new applications, it receives DNS requests from clients, and it receives updates from workers when applications are swapped out to cloud storage.

In the discussion below, we describe the hub as a single physical machine, as this is how it is implemented in our prototype. However, all of the functionality of the hub could be easily parallelized across multiple machines for both redundancy and scalability. In essence, the function of the hub in Picocenter is both as a cluster manager (similar to Borg [43], Omega [156], and Mesos [64]) as well as a load balancer (similar to HAProxy [121]). While we did not implement our hub prototype using

Figure 5.2: Overview of Picocenter, with Picocenter components shown in red. Tenants submit applications via the hub and receive a DNS name; the hub then assigns these applications to be run on workers. Clients lookup a Picocenter domain via the hub and then interact with applications. Workers may swap out inactive applications to cloud storage, and swap them in again later when requests arrive.
CHAPTER 5. PICOCENTER: RETHINKING COMPUTATION

any of these systems, some of the functionality could likely be outsourced to them. We leave a full exploration of a parallelized hub implementation to future work.

New applications As shown in Figure 5.2, the hub interfaces with the tenants and accepts new tenant applications subject to the constraints specified in the application’s configuration file. The hub searches for a public IP address with the appropriate ports free and for a worker with sufficient resources. For example, if the tenant requests port 80, the hub searches through all IP addresses owned by Picocenter, for one whose port 80 is unassigned. If no available IP address can be found, the hub either launches a new worker or refuses to accept the application.

To guide its decisions of which worker to assign the application to, the hub maintains state for each worker: what IP addresses it offers, what applications it is currently running, and whether or not the machine has asserted that it is capable of accepting new applications. This information is used by the hub’s scheduler to determine which worker is available to accept new tasks. The hub’s scheduler can support a range of scheduling disciplines from LRU to Lottery Scheduling [47] when deciding which available worker to pick for a task.

Assuming the hub does find an available IP address and a worker with sufficient free resources, it generates a unique DNS name; stores the mapping between the name and the public IP address; informs the worker that it should begin swapping in the application; and returns the DNS name (and any requested, unspecified ports) to the tenant.

To ensure that future incoming requests are always routed to the appropriate worker, the hub contains a DNS resolver that responds to DNS queries. Any network traffic to the application will thus be routed directly to the appropriate worker, and do not need to transit the hub. The TTLs for the DNS responses are set to the minimum amount of time that an application must reside on a worker before being swapped-out. We set this value to one minute in our prototype; while it may seem unlikely that today’s cloud-based applications would be idle for more than one minute, recall that we are targeting LLMI applications that are, by definition, idle for long periods of time.

Picocenter swaps a new application in when it is accepted (rather than immediately storing it in cold storage) so that the application can perform its own initialization, such as listening on sockets and starting timers.

---

2This process is similar to systems that provide cluster management and orchestration functionality like Kubernetes [138], except that the hub in Picocenter is managing at the level of applications and network ports; typical cluster management systems manage at the level of containers, and therefore simply need to assign a unique IP address to each container. In Picocenter, multiple applications may be assigned the same IP address as long as their port reservations do not conflict.
Figure 5.3: Timeline for serving incoming requests for applications, both inactive and active. Inactive applications are revived while the DNS response is sent to the client; all network traffic for the application goes directly to the worker.

**Reviving applications** Applications can be revived from cloud storage in one of two ways: when an application timer expires, or when a packet arrives for the application.

In both situations, the hub needs to revive the application by assigning it to a worker. To do this, the hub needs to know where the application is stored in cloud storage and which worker to swap the application to. The hub makes these decisions using the same mechanism as when a new application is launched: the hub looks for a worker that has sufficient free resources and has an IP address with the application’s used ports free. Similar to the case above, if no workers are available to accept a recently invoked task, then the hub spins up a new worker (or drops the request).

**5.3.5 Worker**

The worker is responsible for running the tenant’s applications, swapping in and swapping out applications, performing address translation for incoming traffic, and monitoring application resource utilization for billing purposes.

To accomplish these goals, the workers interact with the hub as follows. The hub informs the
workers of new application assignments. In turn, the workers inform the hub periodically as to whether or not they have sufficient capacity to accept new applications, and they also inform the hub when they have swapped out an application they were previously assigned to cloud storage (thereby releasing the application to be assigned to another worker).

Initially, all workers are willing to accept new applications. When an application needs to be run (either a new application, or an application to be revived), the hub chooses a worker (as described above) and provides it the required information for running the application. Namely, the application’s network configuration, the location of the application’s data in cloud storage, and the Pico-configuration. Using this information, the worker is able to successfully run the application and route its network traffic.

We provide more details on the worker implementation and hosting infrastructure in Section 5.5.

5.4 Quickly Swapping in Processes

Picocenter provides the abstraction that all registered applications are running all the time by providing low response time to client requests, even if the application has been idle for months or longer. However, so as not to unnecessarily waste cloud resources, Picocenter does not keep all registered applications running or even resident on worker machines at any given time; most of the time, an LLMI application’s memory state is “swapped out” to long-term cloud storage, such as Amazon’s S3. Picocenter maintains this abstraction by swapping in processes from long-term storage to execution on a worker quickly: to appear to always be running, our target is to swap in processes on the order of a few client’s round-trip-times (RTTs)—the time it takes for the DNS response to reach the client, and for the client to send the request to the application. This way, in the time it takes for the client to receive the DNS response and initiate a TCP handshake, the process would have been loaded (Figure 5.3).

Challenges  While swapping processes and memory pages has received extensive prior study across a wide range of domains, including VM migration [67, 68, 75, 176, 190], “just in time” unikernels [40] and process migration [95, 97], the LLMI applications we consider represent a unique set of challenges: First, whereas most work in VM migration involves moving a running VM from one machine to another, we seek to pull a frozen process into a running state quickly enough to remain unnoticeable to users. Second, unlike most prior work in PaaS and unikernels, we do not constrain the size or type of user applications, and thus we must support applications that can fork and
listen on multiple ports. To meet both of these ends, we extend prior work on partial migration \[68\] to quickly load precisely what a process needs to begin handling a request based on a given port.

### 5.4.1 Swapping out applications

The workers are responsible for judiciously swapping out enough applications in order to keep sufficient resources free for active applications. To this end, the worker constantly monitors its load (e.g., memory pressure, CPU utilization, or bandwidth consumption; in our implementation, we use memory pressure, but some combination thereof may be most suitable in practice). When a machine’s load goes above a high watermark, the worker informs the hub that it can no longer accept new applications, and it begins swapping out the most idle applications to cloud storage in an LRU manner. When the load drops below a low watermark, the worker then informs the hub that it can again accept new applications, and stops swapping out applications. By maintaining a separation between high and low watermarks, we ensure that workers do not oscillate between these two states.

Recall that when an application is assigned to a worker, the worker agrees to be responsible for the application for a fixed period of time (one minute in our prototype). This agreement is reflected in the DNS TTL that is returned to the client, mapping the application’s DNS name to that worker. At times, a worker may discover that its memory pressure is too great, but that it cannot swap any applications to cloud storage due to this restriction.

In order to allow a worker to relieve its memory pressure without violating this agreement, we allow workers to also swap applications to their local storage instead of cloud storage. The mechanism for doing so is identical to swapping to cloud storage, except that the worker does not inform the hub that it has swapped out the application. As a result, incoming network traffic may arrive at the worker for a locally swapped-out application. In this case, the worker revives the application from local disk, rather than cloud storage.

### 5.4.2 Swapping in applications

Picocenter swaps in applications as a result of an invocation event, such as an incoming DNS query for the given application’s hostname, or a triggered timer the application registered. Because applications register timers with workers (and with the hub, should a worker swap an application to cloud storage), we can easily predict timer events and simply swap in applications before they fire, so that the application is ready to run precisely when it needs to be. Incoming packets, however, are more difficult to predict, such as an incoming email to a personalized email server; to maintain
CHAPTER 5. PICOCENTER: RETHINKING COMPUTATION

our abstraction, we must react and swap in the relevant application quickly enough to avoid human perception.

Let us take a step back and ask: how quickly does an application need to be swapped in to be unnoticeable to an external client?

Recall that each application is given a public-facing DNS name, but we may change which public IP address it is NAT-ed to as it moves between being active and inactive. Therefore, the first step a client must take to initiate a new connection is often to issue a DNS query for the hostname. Once the hub selects a worker to revive the application, we have roughly the RTT to the client before there will be any external input to the application. We can leverage this time—typically on the order of tens to hundreds of milliseconds—to begin swapping in (similar to the way TCP handshake offloading is used in some hypervisors [40]), but even this is not enough to swap in the entire memory space if the application is on cloud storage.

**Partial swap ins** A crucial component to maintaining our abstraction of “everything is always running” is swapping in only the portions of the process’s memory that are needed. For non-trivial applications, this is far less than the entire memory space and it is often somewhat predictable. As a motivating example, we instrumented the Nginx web server in Picocenter to record memory reads and writes when serving uniform requests to a variety of files. We observed that (a) on average, less than 1% of the valid memory space was actually accessed on any request, and (b) the set of pages accessed showed over 90% overlap across all requests.

Picocenter uses two mechanisms to efficiently load only what a process needs: reactive faulting and prefetching.

**Reactive page faulting** Picocenter monitors all memory pages that each application reads and writes to. When an application tries to access a page that has not been loaded into its runtime memory space, Picocenter captures this as a “Picocenter-level fault,” pauses the thread that caused the fault, downloads the data from storage, and resumes the thread. Because handling a Picocenter-level fault involves transferring a small amount of data from cloud storage to a machine (4KB), it is a latency-bound operation. Thus, we can swap in more than one page at a time without significantly impacting the overall performance: we segment the data into sets of $B$ pages, so that, given a page fault on page $p$, we swap in pages $B \cdot \lfloor p/B \rfloor$ to $(B + 1) \cdot \lfloor p/B \rfloor - 1$.

While these Picocenter-level faults are less expensive than loading an application’s entire memory, they are far more expensive than traditional OS page faults: the OS needs only pull from local storage,
while Picocenter may have to pull from cloud storage, such as Amazon’s S3. Thus, Picocenter seeks to minimize the number of Picocenter-level faults (to optimize runtime performance), while still loading as few pages as possible (to optimize swap in performance). To this end, Picocenter employs a preemptive component as well.

Prefetching with ActiveSet  When swapping out an application, the local machine stores the page access information in cloud storage, along with whatever kind of event invoked the application in the first place (e.g., the port of an incoming network connection or memory address of the timer that fired). Over time, Picocenter develops a model of which memory regions are often used for a given application when processing different events. When swapping in an application, Picocenter preemptively swaps in those memory pages that are most likely to be accessed as a result of serving that request. In the Nginx example, preemptive, partial swapping reduces the amount of application state that needs to be transferred from cloud storage to a machine by two orders of magnitude. Thus, tracking and predicting these accesses in Picocenter is likely to provide significant performance benefits.

Concretely, Picocenter maintains, for each application, what we call the ActiveSet of memory pages: data that we believe are most likely to be needed by that application to handle a future invocation event. Workers monitor what memory pages applications access, and use this to update the ActiveSet. In our prototype, the ActiveSet consists of all pages that have been accessed during the application’s most recent invocation. When swapping the application in from cloud storage, the worker first downloads an index file describing which pages are in the application’s ActiveSet and then prefetches those pages and makes them available to the application. When swapping out an application, the worker adds or removes pages from the process’s ActiveSet as necessary and writes it to the application’s index file in cloud storage.

The ActiveSet may have false positives (it believed a page would be needed, but it was not) and false negatives (it failed to anticipate that a page was going to be needed). False positives can result in a slower swap in, as the worker has to download more data from cloud storage than strictly necessary, but do not affect the application as it is running. Updating the ActiveSet over time mitigates false positives. False negatives, on the other hand, result in an application trying to access pages that we have not swapped in. For these pages, we fall back on our reactive scheme.

Advanced ActiveSet  Our approach permits more sophisticated definitions of the ActiveSet, such as a dependency graph among pages (encapsulating the notion of order of accesses), or pages that
CHAPTER 5. PICOCENTER: RETHINKING COMPUTATION

have been accessed $k$ out of the last $n$ times the application was active (for parameters $k \leq n$).

Because Picocenter supports applications that fork and listen on multiple ports, such as an email server or a web server handling HTTP (port 80) and HTTPS (port 443), it may be beneficial to track page accesses as a function of port number. For example, when processing an HTTP request, Nginx accesses $\sim 100$ memory pages; for HTTPS requests, Nginx accesses these same pages, plus an additional $\sim 300$ pages. Ignoring port numbers may result either in swapping in more (for HTTP) or fewer (for HTTPS) pages than necessary, resulting in slower processing times.

Picocenter could be extended to support multiple ports by maintaining more than one ActiveSet for each application: one for each listening port, and one that summarizes the most common across all ports. When a DNS query arrives, the worker would begin by loading the ActiveSet common across all of the application’s ports, as it would not yet know the port number—in the case of Nginx, this would constitute the $\sim 100$ pages common to HTTP and HTTPS processing. When the first transport-layer packet arrives, such as a TCP SYN packet, the worker would then swap in the pages corresponding to that port’s ActiveSet. There are several possible further extensions, such as estimating the likelihood of a particular port being accessed, but we leave this to future work.

5.5 Implementation and Discussion

Before presenting the evaluation of Picocenter, we describe the implementation and deployment challenges that we addressed in Picocenter.

5.5.1 Implementation

We implemented the hub and worker on Linux. The hub implementation consists of 757 lines of Python. The hub also uses Twisted (a Python framework) to implement the DNS functionality, and MySQL to store information about where applications are running or are swapped out. The hub maintains persistent TCP connections to all workers, used for job assignment and load reporting.

The worker leverages Linux containers [145] (LXC) as a basis for hosting client applications on workers. Containers are an operating system-level approach to virtualization that provides each container (essentially, a group of processes) with a virtualized environment. Thus, with containers, processes in different containers are unable to interact other than via the network. Containers have been successfully used—via projects like Docker [91]—as a basis for simplifying deployment of applications to hosting machines.
CHAPTER 5. PICOCENTER: RETHINKING COMPUTATION

Limitations  Our prototype implementation of Picocenter has a few important limitations. First, our prototype does not keep track of application timers, and currently only revives applications when network traffic arrives. Second, our prototype does not implement per-port ActiveSet, and because all of the applications in our evaluation were accessed over a single port, it does not affect our evaluation’s conclusions.

5.5.2 Swapping implementation

We make use of several useful tools that have been built on top of Linux containers to facilitate our implementation of swapping. Clients’ memory changes over time; it would be wasteful to copy down the entire memory space in order to simply store it back to the cloud. Instead, we build on the existing work of the Checkpoint/Restore from Userspace (CRIU) project to revive processes such that they only require the downloading of pages that are accessed and to only produce new data for subsequent restores for pages that were modified. This, in effect, gives us the ability to provide incremental checkpoints with on-demand loading of pages. When swapping out a previously-revived process, our modified CRIU only writes out those pages that were modified since the application was most recently revived. We store the resulting checkpoint, which necessarily only contains a page-level diff from the previous one, alongside the existing image in cloud storage. This approach enables quick swapping out of processes.

However, successfully leveraging containers for use in Picocenter presented significant challenges. First, supporting ActiveSet in Picocenter requires the ability to (1) observe the memory accesses (reads and writes) that clients make; (2) partially load a client’s memory image; and (3) dynamically load pages from cloud storage in the case of page faults. We enabled support for all of these by leveraging Linux’s Filesystem in User Space (FUSE) [105]. When restoring a client, we invoke the modified CRIU and supply the FUSE-mounted filesystem as the source for memory images. As CRIU runs and subsequently when the process is revived, the operating system will load pages from this file whenever a page fault occurs; since it is a FUSE file, our FUSE process will be called whenever this occurs. At this point, our process can fetch the appropriate memory pages from either local disk or from cloud storage. Leveraging FUSE also allows our process to monitor memory usage patterns (since all pages read and written will result in a FUSE call the first time they are accessed). When a client is being swapped out to local or cloud storage, the set of pages accessed are recorded along with the memory image of the process, allowing us to implement ActiveSet. Our FUSE process represents 1,906 lines of C/C++ code, and a diagram of its operation is provided in Figure 5.4.
Second, support for CRIU [147] with containers is still in “beta” status, and completely implementing Picocenter required us to fix bugs, add basic support for FUSE filesystems, and develop workarounds to enable pages to be lazily loaded; we contributed these modifications back to the respective projects. Overall, our modifications to CRIU comprise 683 lines of C, and, for the interested reader, we describe the details of these modifications in Appendix A.

Third, applications may write to local files, but we may swap an application off of one machine and onto another. We thus needed to implement support for each container to maintain a consistent view, not only of its memory, but of its own filesystem even as it is swapped in and out across different machines. Picocenter efficiently provides each application with a private filesystem based off of a predefined Linux image by using btrfs [250]. In essence, btrfs allows us to record all changes to the application’s filesystem, and the worker stores this diff in cloud storage alongside the application’s CRIU state. When reviving an application, the worker first re-creates the application’s filesystem by replaying the diff.

5.5.3 Deployment issues

Before presenting the evaluation of Picocenter, we briefly discuss a few issues that a real deployment of Picocenter would face.
CHAPTER 5. PICOCENTER: RETHINKING COMPUTATION

Catching “early” packets  One challenge that arises when launching new applications (or reviving existing applications) is that client packets could arrive at the worker before the worker has completed launching (reviving) the application. If not properly handled, the worker may drop the packet or send back a RST, potentially confusing the client or causing a timeout. To address this, our worker implementation uses the iptables NFQUEUE feature, which allows us to enqueue arriving packets. Once the application is fully alive, the worker releases all packets from the queue.

Resource accounting and billing  One of the key benefits of Picocenter is that applications are charged primarily when they are active; this enables largely-idle processes to be supported in a cost-efficient manner. We envision that Picocenter applications would be charged at two rates: $R_a$ for times when the application is active and $R_i$ for times when the application is inactive (on local or cloud storage), with $R_a \gg R_i$. Similar to Amazon Lambda, we envision that this rate would be charged per memory GB over time. Thus, as an application’s memory demands change over time, it is charged more or less. While we briefly estimate the cost of running a few applications in §5.6, we leave a full exploration of accounting and billing to future work.

Applications  Picocenter is designed to run most existing applications (including both multi-threaded and multi-process applications), but there are certain types of applications that Picocenter is a poor choice for hosting. Examples of such applications are computationally-intensive applications (scientific computing, Bitcoin mining, etc). While our evaluation focuses on LLMI applications, we note that there may be applications in between these two extremes. We hypothesize that there may be a class of applications, such as an email server that typically services requests once every few minutes, that may be more cost-effectively served on an over-provisioned machine that keeps the ActiveSet cached locally (“warm” in our benchmarks). In the event that the machine is unable to handle requests quickly enough, the tasks that were idle the longest could be moved onto another machine. In this case, tighter guarantees on response time can be provided and sold at an intermediate rate. We leave the exploration of this space to future work.

There are a number of common applications that users are likely to use often. Similar to the way existing cloud provides support VM “images,” Picocenter can provide pre-configured Picocenter images for these common applications. Doing so would relieve users of the burden of having to create their own configurations for applications to be used in Picocenter.

---

3 Amazon Lambda has a simple single rate of $0.00001667 per GB-second (as of this writing), because Lambda does not support swapping of inactive applications.
CHAPTER 5. PICOCENTER: RETHINKING COMPUTATION

**Competition for ports**  Because the hosting infrastructure for Picocenter has a limited number of IP addresses, certain ports may be in high-demand for applications (e.g., TCP port 80). We note that most applications today can support applications being run on non-standard ports (e.g., HTTP and HTTPS), but there are a few applications that are required to be run on specific ports (e.g., TCP port 25 for applications that need to interact with legacy SMTP servers). We believe that this non-uniform value of ports can be addressed by charging different rates $R_a$ and $R_i$ for applications that require such high-demand ports.

**Security**  As Picocenter will be supporting applications hosted by mutually distrusting tenants, Picocenter needs to provide the same level of security and isolation between applications as today’s cloud infrastructure does. Because Picocenter relies on Linux containers, Picocenter inherits the isolation between applications that containers provide (which is the same way that Docker instances are isolated from each other). Specifically, Picocenter relies on cgroups to enforce resource usage limits, Linux namespaces to provide isolated namespaces for each container and their filesystems, seccomp to limit system call access, and kernel NAT support to provide private network addresses to different applications.

**ActiveSet for filesystems**  As described so far, the ActiveSet technique is limited to applications’ memory images; however, it could theoretically also be applied to applications’ filesystems. If this were done, it could speed up the time required to revive an application by removing the need to download the entire btrfs diff before reviving the application. This could be accomplished by mounting the application’s filesystem via FUSE (to monitor accesses and handle faults), but doing so would require carefully indexing the btrfs diffs so that the requested data could quickly be located. We leave fully exploring this feature to future work.

### 5.6 Evaluation

We now present an evaluation of Picocenter. We frame our evaluation around four key questions:

1. **What is the low-level overhead of running applications in containers in Picocenter?**
2. **How quickly can Picocenter revive real-world processes from cloud storage?**
3. **How does the ActiveSet technique help to reduce application reviving time?**
4. **How does Picocenter perform with a challenging real-world application that has somewhat unpredictable memory access patterns?**
5.6.1 Evaluation setup

We evaluate Picocenter on a real-world cloud provider, Amazon EC2. We configure both the hub and worker machines to use \texttt{c4.large} instance types, with 3.75 GB of memory. All application swapping is done to Amazon’s S3.

Unless otherwise stated, our prototype is parameterized (as described in Section 5.3) to fetch blocks of $B = 32$ pages on every page fault, and we add a page to an application’s ActiveSet if it was read from or written to in its most recent previous invocation.

We present results evaluating Picocenter on three different, popular real-world applications: the lighttpd web server (version 1.4.36) \cite{lighttpd}, the BIND DNS server (version 9.10.2-P4) \cite{bind} and the PostFix email server (version 3.0.1) \cite{postfix}. We choose these three as they represent very different types of applications that users may wish to run in Picocenter. Preparing these applications for use in Picocenter only required preparing a tarball containing the binary, associated libraries, and necessary configuration files (similar to the process for preparing a Docker container).

5.6.2 Microbenchmarks

We begin by examining the low-level performance characteristics of the hosting infrastructure in Picocenter. Since Picocenter applications are implemented as processes running inside of Linux Containers, we refer to this setup as \textit{VM+Containers}. We also compare Picocenter to the same application running as a normal Linux process in an EC2 virtual machine, representing a prototypical IaaS deployment. We refer to this setup as \textit{VM}. We expect a slight performance hit when compared to VM, due to the overhead of using containers; our goal is to quantify this overhead.

We examine the performance overhead of three broad classes of computation tasks: running without requesting any kernel resources (e.g., math operations and accessing memory), making system calls, and making network requests. To measure these three, we implemented a simple application which (a) reads/writes one 4KB page of memory in a byte by byte fashion, (b) makes system calls for \texttt{gettimeofday}, \texttt{mmap}, and \texttt{munmap}, and (c) performs a UDP ping to a machine on the same network, and waits for a reply. We measure how long each of the operations takes, and repeat the experiments 1,000 times. We run these experiments on EC2 machines.

The results of this experiment are shown in Figure \ref{fig:5.5} broken down by each experiment and system. We observe that the overhead of using containers in Picocenter is quite low: across all experiments, the average performance overhead is only 1.4%, and in no experiment is the overhead statistically significant. This is consistent with prior work \cite{prior_work} that has shown the overhead of
CHAPTER 5. PICOCENTER: RETHINKING COMPUTATION

Figure 5.5: Microbenchmarks for our test application in Picocenter (VM+Containers), compared to a native process running in an IaaS VM. The error bars are standard deviations, and the $y$-axis is in log scale. Containers (and therefore Picocenter) show 1.4% overhead across all tests, and 6.4% overhead on the UDP ping test.

Running Linux containers is low. Moreover, we observe that the action that has the highest overhead in Picocenter is the UDP-level ping; this takes an average of 174 us versus 162 us for the VM process (an overhead of 6.4%). The reason for this somewhat higher overhead is the virtual network device that the container uses, necessitating extra processing in the kernel.

Overall, we observe that the usage of containers in Picocenter has very low overhead when compared to running VM-based IaaS processes. Thus, the results indicate that running applications in Picocenter has the potential to provide similar performance to running applications in traditional VMs. In the following section, we take a closer look at end-to-end application performance to see if this holds true for application-level benchmarks.

5.6.3 Swapping performance

We now turn to examine end-to-end application performance in Picocenter. In these experiments, we set up and run each of our three test applications. For lighttpd, we configure it to serve pages of 1KB and request the pages using curl on a separate machine. For BIND, we configure it to serve DNS queries for a test domain, and request a DNS A record using dig from a separate machine. Finally, for PostFix, we configure it to accept mail for a test domain, and send an email to a test account using Python’s smtplib from a separate machine.

We want to explore how clients in different locations perceive the latency of swapping in applications from cloud storage, compared to accessing applications that are already running. Thus, we run the hub and workers in EC2’s Virginia datacenter, and run clients in EC2’s Virginia, Oregon, Tokyo, and Frankfurt datacenters. We repeat each experiment 50 times and report the median, as well as the 5th percentile and 95th percentile performance (shown as error bars).
Figure 5.6: End-to-end client performance for different applications in “(h)ot” (application live and running on a worker), “(w)arm” (application swapped out on a worker’s local storage), and “(c)old” (application swapped out to cloud storage) states. The hub and workers are located in Virginia, and the clients are located in different locations around the world, with varying average ping times to the Picocenter instance: Virginia (VA, 0.258 ms), Oregon (OR, 80.9 ms), Frankfurt (DE, 88.7 ms), and Tokyo (JP, 149 ms). Error bars represent the 5th and 95th percentiles of the overall time to complete the client’s request. Also shown are the times spent waiting on S3 (downloading checkpoints) and CRIU (restoring the application). Overall, Picocenter provides good end-to-end application performance, with average restore times of 111 ms when applications are “warm” and 186 ms when applications are “cold.”
Figure 5.6 presents our results from this experiment. For each application, we compare the “cold”, “warm”, and “hot” performance (recall that cold represents applications that are stored on cloud storage and retrieved, warm represents applications that are stored on worker-local storage and retrieved, and hot represents applications that are already running on a worker). As a reference, we also include in the caption the average ping times between clients in different regions and our workers; we naturally expect that clients who are further away will have slower end-to-end performance characteristics in all applications due to the RTTs required by the various protocols. For example, even when applications are running, completing a PostFix transaction takes 750 ms for the client located in Oregon, while it takes 1,375 ms for the client located in Tokyo.

We make four key observations. First, we observe that swapping in applications from cloud storage has a surprisingly low performance penalty. Comparing the “hot” bar to the “cold” bar, which represents Picocenter’s overhead of reviving an application, reveals swap in times between 140 ms (lighttpd and BIND) and 200 ms (Postfix). The difference in swapping times is due to the applications having different numbers of processes, memory sizes and layouts, and amounts of process state.

Second, we observe that this overhead can often be dwarfed by the end-to-end performance of the protocol itself. For example, consider the case of PostFix with the client located in Frankfurt. When the application is swapped out to cold storage, sending an email to it takes 1,034 ms; however, even if the applications was already running on the worker, sending the same email would take 805 ms. Thus, in this particular case, the relative client-perceived overhead of swapping the process completely out to cloud storage is low.

Third, we note that there is significant variance in the total restore time, evidenced by the 95th percentile error bars on the cold and warm total bars. This high variability is largely due to S3’s variance in fetching content; we found that S3 fetch times can occasionally vary by up to an order of magnitude or more, even when fetching data of the same size from the same S3 bucket. Conversely, we found CRIU restore times to be largely consistent.

Finally, we observe that when reviving applications from cloud storage, Picocenter has about 70 ms of overhead not accounted for by CRIU and S3 (these are most clearly seen with the Virginia clients). This overhead is due to the setup cost of the btrfs snapshot, which takes about 70 ms on our Linux snapshot. This could potentially be reduced by applying ActiveSet to the application’s file system, but we leave this to future work.

Overall, these results show that Picocenter is able to process application-level end-to-end requests quickly, even when applications are swapped out completely to cloud storage. In these experiments, Picocenter restores “cold,” swapped-out processes in 158–219 ms and “warm” processes in 101–
CHAPTER 5. PICOCENTER: RETHINKING COMPUTATION

Figure 5.7: Response latency for our strawman application when comparing Picocenter’s ActiveSet technique to two baselines: fetching memory only with reactive paging from cloud storage (without ActiveSet), and downloading full application checkpoints every time. Our strawman application is configured with a 64 MB memory size, and we vary the working set size between 4 KB and 8 MB. Note that both axes are log scale. The ActiveSet technique significantly outperforms the baseline approaches.

122 ms. Recall that this overhead only occurs when a client contacts an application that is swapped out; once the application is restored, it is in the “hot” state until it goes idle again and the worker eventually swaps it out.

5.6.4 ActiveSet

Next, we evaluate the impact that ActiveSet has on the performance of swapping in an application from cold storage. Recall that swapping in must be done quickly in order to maintain the illusion to clients that all applications are running all the time, and that ActiveSet helps achieve this abstraction by transferring the pages that are predicted to be needed to handle a request.

Strawman application To provide for controllable experiments with ActiveSet, we implement a strawman application. This application allocates a block of memory of \( N \) pages. Each time it receives an incoming network request, it accesses \( M < N \) of these and subsequently returns a response to the client. The value of \( M \) represents the application’s working set size. To allow for different access patterns (and working set predictability), the application is also configured to “shift” the pages it accesses by \( S \) pages on each request. For example, if \( S = 0 \), the program will access the same \( M \) pages every time it is revived; if \( S = 1 \), it will access \( M - 1 \) of the same pages it accessed previously, and one new page; and when \( S = M \), the process will access different sets of pages on each request. We measure the client-side latency for the application to respond.
Figure 5.8: Response latency for our strawman application when different fractions of the application’s working set is missing from the ActiveSet. Our strawman application is configured with a 64 MB memory size and a working set of 8 MB. The latency remains low, even when large fractions of the application’s memory have to be fetched from cloud storage.

ActiveSet performance  We begin by exploring how ActiveSet improves the speed of reviving applications. To do so, we create two baseline implementations of Picocenter: 

First, we create a version of Picocenter that disables ActiveSet, and loads each memory page reactively.

Second, we create a version of Picocenter that performs a full download of the application’s memory image before reviving it, obviating the need for reactive page faulting. We run an experiment using our strawman application, configuring it to have a fixed 64 MB memory size ($N$), and we vary the working set size between 4 KB and 8 MB ($M$). For these experiments, the application accesses the same set of memory pages each time (i.e., $S = 0$). We repeat each experiment 20 times and report the median latencies in Figure 5.7.

We make three observations from these results. 

First, we observe that using ActiveSet provides significantly lower latency than the other techniques. The latency does increase with the size of the working set, but stays under 250 ms even when the application has an 8 MB working set size. 

Second, we observe that, as expected, the time to revive an application using the full checkpoint download is independent of the working set size; this is because there is no reactive faulting to cloud storage. However, the cost of downloading and restoring a full checkpoint is quite high: over 400 ms in this experiment.

Third, we observe that the latency for using only reactive paging performs between ActiveSet and full checkpoints for small working set sizes, but eventually becomes more expensive than full checkpoint once our strawman’s working set size grows beyond 2 MB. Reactive paging becomes expensive because the memory accesses end up generating a number of sequential round-trips to cloud storage to fetch all of the memory pages required. These results collectively show the benefits of applying ActiveSet to quickly swap in cold applications.
CHAPTER 5. PICOCENTER: RETHINKING COMPUTATION

Errors in ActiveSet In the experiment above, the ActiveSet prediction was always correct (i.e., the process always used exactly the memory pages in its ActiveSet). Next, we explore how ActiveSet performs when its prediction is incorrect. To do so, we use the “shift” parameter ($S$) to vary the overlap between the current working set and the set of memory in the ActiveSet. Because our two baselines are not affected by this shift, we examine only Picocenter with ActiveSet here. For this experiment, we configure our strawman application with a memory size of 64 MB and a working set size of 8 MB.

We vary ActiveSet’s prediction error from 0.5% to 100%, and present the results in Figure 5.8. We can immediately observe that when ActiveSet’s prediction is accurate (the left part of the graph), the response time is consistently low. Once ActiveSet misses 10% or more of the true working set, the response time increases linearly as the application is required to make multiple sequential requests to fetch memory from cloud storage. Comparing these results to the previous section’s baseline that downloads the entire checkpoint, we note that it is better to download a full checkpoint once the ActiveSet accuracy falls below 80%, for this strawman application. These results collectively show that ActiveSet need not have near-perfect prediction accuracy to significantly improve performance over full or reactive-only checkpointing.

Extreme case: ClamAV As an example of a challenging real-world application, we also evaluated the ClamAV anti-virus application. ClamAV is a natural fit for Picocenter; users can direct newly downloaded files to be scanned by the application (thus requests are likely infrequent), and its processing time is typically relatively short (on the order of seconds). At the same time, however, ClamAV effectively stress-tests our ActiveSet design; its memory footprint is 300 MB, and its ActiveSet is typically 54 MB (18% of the total memory footprint). Moreover, as one asks it to scan different files, the memory access patterns will likely differ significantly as the application accesses different parts of its internal index of malicious signatures.

We ran ClamAV in Picocenter on a machine in Virginia, and have a client in Oregon serve files for it to scan. Each time we query ClamAV, we randomly generate 10 files of 50 KB each for it to scan. We repeat the experiment 10 times and report the median. Overall, we find that the latency to scan all files is 5.16 seconds when ClamAV is actively running (“hot”) and is 7.49 seconds when ClamAV is swapped out to cloud storage (“cold”). Thus, even with ClamAV’s challenging memory

\[4\text{Recall that with ClamAV actively running, not all of its memory is necessarily resident (some may still be swapped out to cloud storage, and will cause reactive Picocenter-level faults if accessed). For comparison, if ClamAV is entirely memory-resident, we find that the latency to scan all files is an average of 3.36 seconds.}\]
CHAPTER 5. PICOCENTER: RETHINKING COMPUTATION

access, Picocenter’s use of ActiveSet largely hides the fact that the application was fully swapped out to cloud storage (almost all of the additional latency for the latter case is due to requests to cloud storage for memory pages that are not contained in the ActiveSet). Indeed, if we resorted to using the full checkpoint approach (downloading the entire application before reviving it), we found the latency for the same experiment would be 9.61 seconds.

Summary We note that the ultimate success of Picocenter is a function not of how much memory the process allocates in total, but of the process’s working set. In practice, many VMs actively use a small fraction of their overall allocation [68]; we have observed similar trends with individual processes. These results show that Picocenter is able to swap a process from cloud storage to running quickly enough to be negligiable for most applications—even when the working set of the process is quite large. Moreover, we plan on exploring additional techniques to improve the accuracy of the ActiveSet predictions; the results shown here use only a simple heuristic that predicts an application will use exactly the memory pages it used in the previous request.

5.6.5 Cost

As a final point of evaluation, we briefly estimate the cost to tenants of running applications in Picocenter. We consider a strawman Picocenter provider that simply uses existing cloud services (e.g., Amazon’s AWS) to provide the infrastructure for Picocenter, effectively acting as a reseller. The actual cost will depend on a large number of factors, but our goal here is to simply provide a rough estimate.

We begin by estimating the charging rates for active and inactive applications based on the EC2 and S3 costs of today (i.e., we estimate the cost of leasing EC2 machines to the Picocenter provider). To estimate $R_a$, the rate charged per GB of memory for active applications, consider the c4 generation of EC2 instances on AWS: each of these present a cost of $0.00052/GB-minute ($22.46/GB-month), so we expect $R_a$ to be on this order of magnitude. To estimate $R_i$, the rate charged per GB of memory for inactive applications, we draw directly from the S3 costs of today, providing an expected $R_i$ of $0.0000007/GB-minute ($0.03/GB-month).

The cost to a tenant would be dependent on the memory size of the application, the size of its working set, and the fraction of time that it is active. If we consider a user running an Nginx instance

---

In particular, our estimate does not account for many costs to the provider, including development, personnel, and any profit margin, but the provider would also likely be able to use techniques like Reserved Instances to obtain much cheaper rates.

96
(average memory usage of 65 MB) that is active 10% of the time and uses 10% of its memory on each invocation, we can derive that the cost of running this application in Picocenter would be approximately $0.0165/month. Compared with a dedicated VM (cheapest cost of $9.36/month), this represents a 99.82% savings.

5.7 Summary

Cloud computing services today enable end users to purchase various computing resources at a fine granularity. For example, Priv.io presented in chapter 4 uses user-provided cloud storage to save service data. However, both PaaS and IaaS cloud computation models are ill-suited for long-lived, mostly-idle (LLMI) applications. Running such applications in today’s IaaS cloud is economically inefficient, and prohibitive for typical end-users, as they are charged disproportionately to how much work their jobs perform. On the other end of the spectrum, PaaS models restrain the set of applications that may be run, precluding many useful LLMI applications, such as a personalized email server or a personalized mobile offloading application.

In this chapter, we presented Picocenter, a hosting infrastructure for end user LLMI applications. Picocenter is a general computation model for users to run their applications in the cloud. Our evaluation on a real-world cloud infrastructure shows that Picocenter represents a unique operating point compared to existing service models. Compared to PaaS, Picocenter is not constrained to a particular programming model—we showed that non-trivial applications can be run by just using their off-the-shelf binaries. Compared to IaaS, Picocenter is able to efficiently swap applications to and from cloud storage—we showed that Picocenter incurs low latency overhead when swapping applications from cloud storage to running. LLMI applications in Picocenter thus consume fewer resources, allowing users to be charged only for what they use, and allowing cloud providers to accept more jobs. As a result, Picocenter enables server-side support for user applications working within the client–server architecture of the web and beyond.

Current IaaS providers charge by the hour; should they move to a usage-based charging model we believe that these IaaS services will still be a poor fit for LLMI due to operational overheads of maintaining/managing VMs.
Chapter 6

Related Work

In this chapter, we describe prior work related to the topics presented in this thesis. To rethink the architecture of the web, we have presented three systems, Maygh, Priv.io, and Picocenter, covering three areas, content distribution, privacy, and computation respectively. With respect to these systems, the related work has been grouped into three sections detailing (a) work that tries to solve the problem of content distribution, (b) work that seeks to protect user privacy and secure the web, and (c) work that tries to enhance computation.

6.1 Content distribution

We began this thesis with rethinking the architecture for web content distribution. We presented Maygh, a system that builds a content distribution network from client web browsers, without the need for additional plug-ins or client-side software. Thus, we first describe related approaches in optimizing existing CDNs. Then we review non-CDN approaches, including peer-to-peer content distribution systems and cooperative web caches.

6.1.1 Optimizing CDNs

CDNs like Akamai, Limelight, Adero, and Clearway have emerged as a commonly-used platform for web content delivery. CDNs offload work from the original website by delivering content to end users, often using a large number of geographically distributed servers. While much work on CDN architectures has examined different server selection strategies \[44, 198, 230\] and content replication algorithms \[71, 224\], most of it is not directly applicable to Maygh. CDNs generally assume a relatively stable list of servers under centralized control; in Maygh, each visiting web client
is effectively a server. It may be possible to use content replication techniques to pre-fetch content onto users’ machines; however, we leave such techniques to future work.

Alternatives to centralized CDN have been built that use resources under multiple entities’ control to accomplish the same task (e.g., Coral [102], CobWeb [212], and CoDeeN [173, 232]). These solutions are impressively scalable and well-used, but they fundamentally rely on resources donated by governments and universities, and are therefore not self-sustaining in the long run. In fact, the Coral system quickly overwhelmed the bandwidth resources available on PlanetLab within the first year of deployment [217], and was forced to enforce fair sharing and reject some download requests.

Other approaches have explored allowing end users to participate in CDNs, including Akamai’s NetSession [50], Flower-CDN [89], BuddyWeb [233], Squirrel [125], and Web2Peer [189] (client-side applications that assist in content distribution to other clients), as well as Firecoral [217] (a browser plug-in that serves content to other Firecoral users). While the goals of these systems are similar to Maygh, all require the user to download and install a separate application or plug-in, significantly limiting their applicability and userbase in practice. WebCloud [242] allows users’ browsers to participate in content distribution without requiring any client-side changes; however, it requires that CDN-server-like redirector proxies be deployed within each ISP region.

Additionally, recent work [204] has demonstrated that information flow patterns over social networks (called social cascades) can be leveraged to improve CDN caching policies. This work is complementary to ours, and suggests that Maygh will perform especially well in systems like online social networks. Finally, other recent work [172] has examined the benefits of allowing ISPs to assist CDNs in making content delivery decisions. This approach is also similar in spirit to Maygh, but focuses on optimizing the server selection strategies employed by ISPs today.

6.1.2 Non-CDN approaches

Maygh can be viewed as approximating p2p content exchange through web browsers. Much previous work has focused on building standalone p2p systems for content storage and exchange [83, 134, 142] or avoiding the impact of flash crowds [197, 207]. However, unlike Maygh, almost all work on p2p systems assumes a full networking stack and is incompatible with being run inside a browser. Recent work [80] has moved towards leveraging resources on web clients to provide better scalability for web-based services, but is primarily focused on the services themselves, rather than content distribution.

Additionally, researchers have exploited the use of social relationships in p2p systems for other
CHAPTER 6. RELATED WORK

applications, such as file sharing [175], email [114], DHT routing [157], and social networking [70, 90]. As people have close ties in social networks may share the same common interests, it is more likely for them to share interests or being mutually trusting. Maygh leverages similar properties as these systems, but is designed for a different service.

Besides p2p systems, there has been a long history of work that examines corporative web proxy caches, where proxy caches on the same side of a bottleneck link corporate to serve each other’s misses [99, 229]. While these systems have the same net effect of Maygh (the reduction of load on the operator), their motivation is usually to lower network usage at the edge of the network, rather than at the operator. As a result, unlike Maygh, cooperative caches must be deployed and configured by network administrators at the edge and are not under control of the operator.

6.2 Privacy

Later in this thesis, we presented Priv.io, a new approach to building web-based services that offers users greater control and privacy over their data. Since Priv.io relies on the web browser to provide an isolated environment to perform untrust computation, we first review studies in protecting privacy in the web browser.

6.2.1 Enhancing web browsers

Over the past decade, web browsers have become significantly more advanced. Researchers have explored using process-based models to isolate misbehaving web pages [180], have examined moving beyond the same-origin policy of privilege separation [34], built systems that allow third-party code to execute while providing security and privacy guarantees [81, 124], and implemented iframe-based sandboxing [123]. We leverage many of these advances and techniques in the design of Priv.io.

6.2.2 Web services

Today, most of the web services are implemented as centralized services. In this section, we review work that tries to modify existing web services or building new web services for better privacy protection. Many groups have explored ways to provide greater user privacy by re-architecting existing web services. Developers have built subscription-based services such as app.net [58], which promise to not show ads in exchange for a yearly fee; unfortunately these simply replace one centralized provider with another. Systems have also been built that guarantee sandboxing of
CHAPTER 6. RELATED WORK

third-party applications [223], but these do not address hiding information from the service provider. Finally, researchers have developed approaches that enable sharing of provider-hosted content among different providers and with the user’s local machine [112]; however, these do not address the issue of privacy from the centralized provider.

Others have explored retaining existing centralized providers, but hiding certain information from the provider. For example, researchers have explored encrypting uploaded content [113, 179], encrypting social relationships [216], and keeping data on user-managed devices [231]. It is unclear whether existing providers are amenable to these solutions (as they directly impact the providers’ revenue stream), and deploying them independently risks users being banned by the provider.

In parallel, researchers have also explored new approaches that operate via the web. For example, Persona [69] (which inspired our design, and in particular, our approach for encrypting content) stores encrypted user data on user-contracted storage services. Similar approaches include Vis-à-Vis [195] (storing data on group-based EC2 machines), Confidant [141] (storing data on friends’ machines) and others [153] (storing data on users’ home routers). Unfortunately, all of these solutions require client-side changes in order to work, and assume that they have a less-restricted model of computation than is available to JavaScript within the browser. In contrast, Priv.io uses many aspects of these systems’ design, but does so without requiring any client-side changes and supports potentially untrusted third-party applications.

Others have explored separating web-based services from user data. For example, W5 [135] proposed an architecture that separates web service developers from providers that execute service code and host user data in a secure manner. While the vision of Priv.io and W5 are similar, to the best of our knowledge, W5 has not been deployed nor has any providers become available. BStore [79] provides a generic file system-like interface for web applications, allowing users flexibility in the location of their data. Priv.io stores data on cloud providers using a number of techniques that were proposed in BStore. However, BStore is focused on providing file storage, while Priv.io also deals with challenges of sharing data with others, supporting third-party applications, and demonstrating that existing services can be replicated in a confederated manner. In addition, Mylar [45] stores encrypted user data in the server while web applications to perform computation over encrypted data, such as keyword search. Compared to Mylar, Priv.io pushes all computation over user data to web browsers, without making changes to the server’s software stack. Furthermore, Priv.io provides social activity operations for building web applications, which is not supported on Mylar.
6.2.3 Decentralized network systems

In contrast to centralized services, researchers have presented systems that implement services in a decentralized fashion. These include PeerSoN [70], Diaspora* [90], Safebook [82], Contrail [202], and others [37]. While similar to Priv.io in goals, all of these approaches require client software to be downloaded, and also generally face challenges in ensuring availability [61]. Others have designed protocols [221] that allow users to host their data on dedicated, secure servers of their choosing. A more detailed overview of the tradeoffs of decentralized architectures is provided in [196].

6.3 Computation

Finally, to provide a general computation system for web applications, we presented Picocenter, an alternative approach for cloud computation based on a process-like abstraction rather than a virtual machine abstraction, thereby gaining the scalability and efficiency of PaaS along with the generality of IaaS. In this section, we review prior work in the areas of virtualization, containers, other work that seeks to extend PaaS applications, and migrating and checkpointing VMs and processes.

6.3.1 Hardware virtualization

We begin with work that supports computation with hardware virtualization. Typical hardware virtualization solutions such as Xen [170], VMWare ESX Server [46], or KVM [148] allow users to run operating systems and applications of their choice. Despite this, many cloud applications today only run one service (e.g., Apache or MySQL) in a virtual machine (often to simplify management and isolate failures). As a result, several prior systems have sought to reduce the overhead of running a full-fledged OS for a single application. For example, OSv [131] and unikernels [38] (e.g., Mirage [39]) improve resource utilization by reducing guest operating system size, providing a library-OS-like environment for user programs. However, these systems do not provide easy support for swap ins. The closest work on supporting swap ins for unikernels is Jitsu [40], which demonstrated that unikernels can be started in a matter of milliseconds. However, Jitsu assumes that the applications are stateless and that new ones can be spawned when requests arrive; with Picocenter, we aim to support the swapping of stateful applications and provide fast restore times.
CHAPTER 6. RELATED WORK

6.3.2 Software environments

Compared to hardware virtualization that operates on the hardware abstraction layer, software solutions for supporting computation execute user programs in customized environments. There are two types of such environments, operating system containers and dedicated runtime. Operating system containers serve as a lightweight method of virtualization compared to hardware virtualization; examples include Docker [91], LXC [145], VServer [205], BSD Jail [174], and Solaris Zones [88]. This technique modifies the hosting operating system kernel, using kernel-level isolation techniques to make it appear to process groups that they have the machine to themselves. Picocenter leverages containers in order to provide isolation between client processes while allowing backward compatibility with existing applications; Picocenter extends Linux containers with support for checkpoint/restore as well as partial swap ins.

In parallel, building on top of the operating system layer, dedicated runtime creates software environments for its user programs. Cloud providers today offer dedicated runtime in their PaaS systems, such as Amazon Lambda [54] and Google App Engine [116]. Picocenter is distinguished from PaaS systems in that it permits arbitrary applications written in arbitrary languages. PaaS services typically require that clients write code using a limited subset of managed languages (e.g., Java, Go, or Python); this likely simplifies the provider infrastructure necessary to support these services, but also greatly limits the kinds of services that PaaS systems can support. Another recent trend is hosting applications in a virtual machine; ZeroVM [248], for instance, uses Google’s Native Client [72]. However these techniques are not designed to run applications from multiple users, and do not handle resource allocation or swapping.

Additionally, other researchers have developed picoprocesses [118], a “stripped-down virtual machine without emulated physical devices, MMU, or CPU kernel mode” [187]. Essentially, picoprocesses are process-like objects that use a small interface to access various hardware resources (network, memory, etc). Picoprocesses have been previously used in the context of redesigning the architecture of web (in the Embassies [117] project), moving from a web based on HTML and Javascript to one based on the picoprocess interface [96]. We experimented with a Picocenter design based on picoprocesses, but found them to have significantly higher overhead than our eventual (CRIU-based) approach.
6.3.3 Pre-paging and migration

In Section 5.4, we presented ActiveSet, an prefetching algorithm for quickly swapping in processes. ActiveSet algorithm builds on a long line of research on Pre-Paging [210] and VM live migration [67,175,190] which predict and prefetch working sets using historical information. Other work [68] has explored consolidating desktop machines onto centralized hardware, and migrating the working set of each VM to the user’s machine. Yet others [95,97] have explored approaches to process migration that lazily copies memory in a similar manner as our reactive page faulting techniques. Unlike these approaches, our execution environment provides us with sufficient visibility into memory access-patterns and allows us to determine working sets at a much finer granularity. This finer granularity reduces the size of data and transfer times. Other approaches to migrating virtualized resources such as networks of VMs and other infrastructure [36,94,191] take on the order of minutes to hours to complete, unfortunately the interactive nature of the set of Picocenter’s target applications demand faster techniques.

6.3.4 Checkpoint and restore

Picocenter extends the study of process checkpoint and restore [120,128,168], which retrieve states of processes in operating systems and restore processes from the saved states. Picocenter leverages this work by using the Checkpoint/Restore in Userspace for Linux project [147], while extending CRIU to be able to restore processes with memory loading on-demand, and to also be able to make incremental checkpoints of Linux containers.

Other work has explored full virtual machine checkpoint and restore, including DreamServer [218] and Sloth [219]. In these systems, idle VMs (e.g., VMs with no incoming requests for 30 seconds) are suspended, and are restored when they receive an incoming request (typically within 2 seconds). While Picocenter shares many goals with these approaches, our underlying approach is fundamentally different; Picocenter deals with sets of processes, while DreamServer deals with entire VMs. Thus, Picocenter is significantly more lightweight (our restore times are an order of magnitude lower), and it also relieves clients of the burden of VM management.

6.3.5 Code offloading

In parallel with the work above, there has been a line of research investigating the potential for offloading computation from less-powerful devices to more-powerful devices (e.g., from mobile devices to the cloud). Notable examples include UCoP [188] (targeting laptop devices) and
MAUI [93] (targeting mobile devices). While these share some similarities with Picocenter, the set of assumptions they make and challenges they face are quite different. For example, such offloading systems typically must develop techniques to efficiently divide code between local and remote resources, but are not concerned with swapping idle applications to cloud storage.
Chapter 7

Conclusion

The web today is a popular platform running a wide variety of services, such as search, online social networks, news, content sharing, communication, live streaming, business automation, and massive collaboration. However, these new services that are still built on the client–server architecture bring new challenges to the web (e.g., users lose control over their data). In this thesis, we revisited the architectures of the web by allowing end users to contribute computing resources, and thereby enabling them to control their services and data. We leveraged recent developments in web browsers and cloud computing for building new solutions to the challenges. In particular, we focused on three areas where relaxing the strict client–server nature of the web could provide significant benefits to websites and end users: content distribution, user privacy and cloud computing for end users. In the following sections, we recap the high-level contributions of this thesis and discuss potential future research directions.

7.1 Summary

This thesis made contributions in rethinking the architecture of the web in three areas: content distribution, user privacy, and computation for end users.

We began this thesis in rethinking the architecture of the web. Today, the web’s architecture places a substantial monetary burden on the website operator; as a result, most operators are forced to monetize their users by selling advertisements to third parties. We have addressed this by introducing Maygh [245], a content distribution network built from client web browsers, without the need for additional plug-ins or client-side software. Maygh helps the website operator to distribute sharing content that users create by recruiting users to serve content to others. The result is an organically
CHAPTER 7. CONCLUSION

scalable system that distributes the cost of serving web content across the users of a website. Through simulations based on real-world access logs from Etsy (a large e-commerce website that is the 50th most popular website in the U.S.), microbenchmarks, and a small-scale deployment, we demonstrated that Maygh provides substantial savings to site operators, imposes only modest costs on clients, and can be deployed on the websites and browsers of today. In fact, it would reduce network bandwidth due to static content by 75% and require only a single coordinating server should Maygh were deployed by Etsy.

Maygh solves the problem of distributing web content at low cost by recruiting user browsers to help content exchange. This meets the increasing demands from content sharing services, as users contribute significant among of content and share them with other users. However, users today have little control over the data they exchange over the web, and are forced to reveal their data to third parties (e.g., Facebook and Twitter) if they wish to exchange content with friends. To rethink the architecture for building web-based services with the goal of enhancing user privacy, we presented Priv.io \[243\], a new approach for building web-based services. We leverage the fact that today, users can purchase storage, bandwidth, and messaging from cloud providers at fine granularity: in Priv.io, each user provides the resources that are necessary to support their use of the service using cloud providers such as Amazon web Services. Users still access the service using a web browser; all computation is done within users’ browsers; and Priv.io provides rich and secure support for third-party applications. An implementation demonstrated that Priv.io works today with unmodified versions of common web browsers on both desktop and mobile devices; Priv.io is both practical and feasible, and is cheap enough for the vast majority users.

Finally, long-lived but mostly idle (LLMI) applications, such as Priv.io, are prohibitively expensive to run on the cloud today. In Priv.io, we got around this by leveraging the computation available in web browsers, but this has significant downsides (e.g., not being able to process messages while users are offline). As a general solution for end users who wish to run applications on the web and beyond, we presented Picocenter \[244\], a hosting infrastructure designed to support LLMI applications. Compared to similar approaches such as Docker, Google App Engine, and Amazon Lambda, Picocenter offers support for arbitrary computation and event handling (Lambda only supports a pre-defined set of event triggers), strong isolation between processes, and native support for long-lived processes that are automatically swapped to cloud storage when the process is not active. The key technical challenge in Picocenter is enabling fast swapping of applications to and from cloud storage (since, by definition, applications are largely idle, we expect them to spend the majority of their time swapped out). We developed an ActiveSet technique that prefetches the applications
CHAPTER 7. CONCLUSION

predicted memory working set when reviving an application. An evaluation on EC2 demonstrates that using ActiveSet, Picocenter is able to swap in applications in under 250 ms even when they are stored in S3 while swapped out. Moreover, Picocenter can be deployed on top of today’s clouds, enabling incremental deployment through resellers of traditional virtual-machine-based clouds.

7.2 Extensions

So far, we have presented new solutions that relax the strict client–server constraints of the web for content distribution, user privacy, and cloud computation. However, recent trends in video streaming, JavaScript cloud programming, and mobile developments have shown new directions for extensions of this thesis. In the following paragraphs, we briefly outline a few possible extensions to the work presented in this thesis.

In this thesis, we presented Maygh, a content distribution system that specializes in distributing content as a whole, such as a photo or a web object. However, recent trends in video streaming and complex web applications require CDNs to deliver large objects (e.g., videos, mobile applications or large, complex JavaScript client-side programs). Also, new requirements such as loading partial data and short latency are critical for applications like video streaming. Other features, such as resuming downloads from breakpoints or downloading from multiple sources, are necessary to support these applications. These new applications raise new challenges for in-browser CDNs that manage peer-to-peer network across browser sessions. In essence, it remains an open challenge to extend Maygh to support video streaming, large objects, or even live videos.

Today, many cloud vendors offer new models to run JavaScript programs in the cloud [16, 54], allowing developers to build both browser-side and server-side applications with a single programming language. As an alternative model, Picocenter supports a JavaScript runtime for end users to run their applications in the cloud. For example, Priv.io could use Picocenter to offload its browser-side computation to the cloud when the web browser is offline. However, the security model in the web browser is significantly different from the one in the cloud. Compared to the cloud environment, JavaScript programs running in the web browser are subject to the Same-Origin policy and other API restrictions. Building a new security model for such cross-domain JavaScript applications is crucial for data security and user privacy. This direction could be an extension of both Picocenter and Priv.io.

Recent rapid development in the mobile platform opens new directions to extensions of systems presented in this thesis. Both Maygh and Priv.io work on today’s HTML5 mobile browsers, and
CHAPTER 7. CONCLUSION

Picocenter can run arbitrary Linux processes, including service applications that support mobile applications. However, due to the constraints in mobile environments (e.g., short battery life, high RTT, limited connectivity, low bandwidth, congested channel, and slow CPU), Maygh and Priv.io may not be ideal in the mobile environments. For example, Maygh clients running on cell phones may have a slow content upload speed because of limited mobile networks; Priv.io takes longer in cryptographic operations on mobile due to slower CPU. Last but not least, both Maygh and Priv.io may lead to higher power consumption on the mobile platform because their extensive networking and computation. Despite these constraints, Maygh and Priv.io can be extended to work better on mobile by considering battery and network conditions in operation. For example, these systems could serve other users in Maygh when the device is connected to a Wifi network with good network condition. Similarly, Priv.io may offload computation to the cloud with Picocenter.

Going beyond mobile web browsers, Maygh and Priv.io can be reimplemented as native mobile applications. Compared to web environments, native mobile environments allow applications to create arbitrary socket connections, run background jobs, execute native binary code and more. Working as a native mobile application, Maygh can be a system service that helps other applications running on the device to distribute content, while Priv.io can use background processing for message updates, or use push notification to inform other users. Despite these advantages, mobile applications have complex security frameworks. On the web platform, Maygh and Priv.io make use of web security primitives (e.g., iframe sandbox, Same-Origin policy, cross-origin resource sharing). But working as a native mobile application, Maygh and Priv.io need to develop similar security primitives to secure the system.

All of the source code presented in this thesis is available to the research community at

https://github.com/leoliangzhang/
Appendix A

Supporting ActiveSet in CRIU

Picocenter uses CRIU for checkpointing and restoring containers. Ideally, swapping in a process would be as simple as loading the process’s ActiveSet into memory and restoring the container. Unfortunately, CRIU can only restore a container after all of its pages written during a previous checkpoint are populated into memory. To modify CRIU to support ActiveSet, it was essential that not every page of memory was read before the process was restored.

To restore, stock CRIU spawns a new process and performs a “premap” whose size corresponds to each virtual memory area (VMA) that was a part of the previously checkpointed process. It subsequently reads data from the checkpoint data file to restore the data for these mappings. To support ActiveSet, rather than creating an anonymous mapping and then filling it with data, we perform a file-backed mmap with the appropriate offset and length in the checkpoint data file. For pages that were occupied, this conceptually consists of merely changing a read into a mmap each time the CRIU process attempted to read pages into the VMA. However, we were obligated to retain the unoccupied pages, so we still performed the initial premapping for every VMA. Each time we came across a memory area that traditionally would be read off of consecutive bytes on disk, we perform the file-backed mmap over this potential subset of the memory region. The Linux kernel will automatically split the VMA, if necessary, into anonymous unoccupied areas and consecutive file-backed areas. Since CRIU stores consecutive allocations on disk consecutively, the result is few regions.

Since each of these mappings are private, the region is essentially initialized with the contents of the file whenever a fault occurs, whereas private mapping semantics ensures that written pages are no longer associated with the backing file. In these cases, the written pages will be persisted as part of a subsequent checkpoint.
Once all of the memory maps have been completed, as the nascent process finishes restoration, each memory area must be remapped to the memory location it occupied before the previous dump. To accomplish this, stock CRIU performs a `mremap` over each original memory area. However, when the VMA data is sent to the CRIU parasite code, which runs in the context of a protean form of the restored process, we no longer have access to all book-keeping (and individual sub-mappings) done in the original restore process. While this could be rectified with additional information passing, our understanding of the mappings is especially confounded because some of these splits took place in the kernel, occasionally in surprising ways. We cannot simply perform a `mremap` over the entire original memory area, because `mremap` is not capable of remapping VMAs that span more than one mapping. To rectify this, we query the kernel to find all extant mappings for the process, which now includes areas that were internally split, and perform `mremap` calls for each area as it exists in the kernel that form subsets of the original mapping data.

The result is we were able to extend CRIU to support checkpoint/restore of containers while simultaneously allowing lazy loading of memory pages as the process runs.
Bibliography

[8] CSS. https://www.w3.org/TR/CSS/
[10] Dynamic HTML. https://www.w3.org/Style/#dynamic
BIBLIOGRAPHY


[27] Web Workers. https://www.w3.org/TR/workers/


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


[85] Cross-Origin Resource Sharing. [http://www.w3.org/TR/cors/]


[87] P. Ducklin. “Pwn2Own” competition pops Flash, Reader and four browsers, pays out over $550K. [https://goo.gl/rrnU4V]


[90] Diaspora*. [http://www.joindiaspora.com/]

[91] Docker. [https://www.docker.com]


BIBLIOGRAPHY


[104] Facebook Statistics. [http://on.fb.me/UtWB0](http://on.fb.me/UtWB0)


[122] HTML5 Geolocation API. [http://dev.w3.org/geo/api/spec-source.html](http://dev.w3.org/geo/api/spec-source.html).


BIBLIOGRAPHY


124
BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


[236] W3C Indexed Database API. [http://www.w3.org/TR/IndexedDB](http://www.w3.org/TR/IndexedDB).


[255] ownCloud. [https://owncloud.org/](https://owncloud.org/)
BIBLIOGRAPHY