8-2016

A Holistic Approach to Lowering Latency in Geo-distributed Web Applications

Shankaranarayanan Puzhavakath Narayanan
Purdue University

Graduate School Form 30
(Revised 08/14)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Shankaranarayanan Puzhavakath Narayanan

Entitled: A Holistic Approach to Lowering Latency in Geo-Distributed Web Applications

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

SANJAY G. RAO

MITHUNA S. THOTTETHODI

T. N. VIJAYKUMAR

YU C. HU

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

SANJAY G. RAO

Approved by Major Professor(s): _____

Approved by: V. Balakrishnan 06/23/2016

Head of the Department Graduate Program Date

A HOLISTIC APPROACH TO LOWERING LATENCY IN GEO-DISTRIBUTED WEB APPLICATIONS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Shankaranarayanan Puzhavakath Narayanan

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2016

Purdue University

West Lafayette, Indiana

To my family, for the unconditional trust, love and support they have given me through all these years.

ACKNOWLEDGMENTS

Special thanks to Prof. Sanjay Rao, for being such a wonderful advisor, and for constantly encouraging me to push my abilities. I'm truly grateful to have received the opportunity to work with him for all these years. I'm thankful to my committee members, Prof. Charlie Hu, Prof. Mithuna Thottethodi, and Prof. T.N. Vijaykumar, for their advice and support throughout my graduate studies at Purdue University. My research has immensely benefited from their valuable ideas, inputs and suggestions. I'm grateful to Prof. Bruce Maggs for providing me with the opportunity to work with him and for guiding me through a significant part of my thesis. I'm indebted to my colleagues from Internet Systems Lab: Ashiwan, Mohammad, Yun, Yiyang, Xin, Ruben and Eric for their support and active involvement in all our research efforts. Last, but not the least, my sincere thanks to NSF, NetApp, and Google for supporting and funding parts of my research.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Puzhavakath Narayanan, Shankaranarayanan Ph.D., Purdue University, August 2016. A Holistic Approach to Lowering Latency in Geo-distributed Web Applications. Major Professor: Sanjay G. Rao.

User perceived end-to-end latency of web applications have a huge impact on the revenue for many businesses. The end-to-end latency of web applications is impacted by: (i) User to Application server (front-end) latency which includes downloading and parsing web pages, retrieving further objects requested by javascript executions; and (ii) Application and storage server(back-end) latency which includes retrieving meta-data required for an initial rendering, and subsequent content based on user actions.

Improving the user-perceived performance of web applications is challenging, given their complex operating environments involving user-facing web servers, content distribution network (CDN) servers, multi-tiered application servers, and storage servers. Further, the application and storage servers are often deployed on multi-tenant cloud platforms that show high performance variability. While many novel approaches like SPDY and geo-replicated datastores have been developed to improve their performance, many of these solutions are specific to certain layers, and may have different impact on user-perceived performance.

The primary goal of this thesis is to address the above challenges in a holistic manner, focusing specifically on improving the end-to-end latency of geo-distributed multi-tiered web applications. This thesis makes the following contributions: (i) First, it reduces user-facing latency by helping CDNs identify and map objects that are more critical for page-load latency to the faster CDN cache layers. Through controlled experiments on real-world web pages, we show the potential of our approach to reduce hundreds of milliseconds in latency without affecting overall CDN miss rates. (ii) Next, it reduces back-end latency

by optimally adapting the datastore replication policies (including number and location of replicas) to the heterogeneity in workloads. We show the benefits of our replication models using real-world traces of Twitter, Wikipedia and Gowalla on a 8 datacenter Cassandra cluster deployed on EC2. (iii) Finally, it makes multi-tier applications resilient to the inherent performance variability in the cloud through fine-grained request redirection. We highlight the benefits of our approach by deploying three real-world applications on commercial cloud platforms.

1. INTRODUCTION

User perceived end-to-end latency of web applications have a huge impact on the revenue for many businesses [1–4]. For e.g., Amazon finds that every $100ms$ of latency costs 1% in sales [3], while Google Search found that a 400 millisecond delay resulted in a 0.59% reduction in searches per user [5]. Beyond e-commerce, bringing the latency under 100 ms [6] would imply that the user cannot differentiate between whether an application is running locally or is making remote requests. Service level agreements (SLAs) on such interactive web applications often require bounds on the $90th$ (and higher) percentile latencies [7], which must be met while scaling to hundreds of thousands of geographically dispersed users.

Modern web applications have multiple constituent components including Application servers, Datastore servers (DS) and Content Delivery Network (CDN) servers. Further, one or more of these components are geo-replicated and often hosted on third-party service providers including commercial cloud datacenters and CDNs. The end-to-end latency of web application depends on both : (i) user to application server latency, and (ii) application and storage layer latency. Consequently, we have seen recent efforts that aim towards reducing the latency at these different layers. On the one hand, there is a wide-spread interest in geo-replicated datastores [7–14] which help bring data closer to the users. On the other hand, we see the emergence of new protocols such as SPDY [15], which aims at faster delivery of objects from the server to the client.

Despite these efforts, lowering web-page latencies remains a challenging proposition, primarily due to the complexity of the web-pages and its deployment ecosystem. In particular, each page is typically composed of tens to hundreds of inter-dependent objects arriving from different application components, which have varying impact on the end-to-end latency of the web-page. The goal of this thesis is to develop frameworks that can simultaneously reduce both user to application latency as well as the application and storage

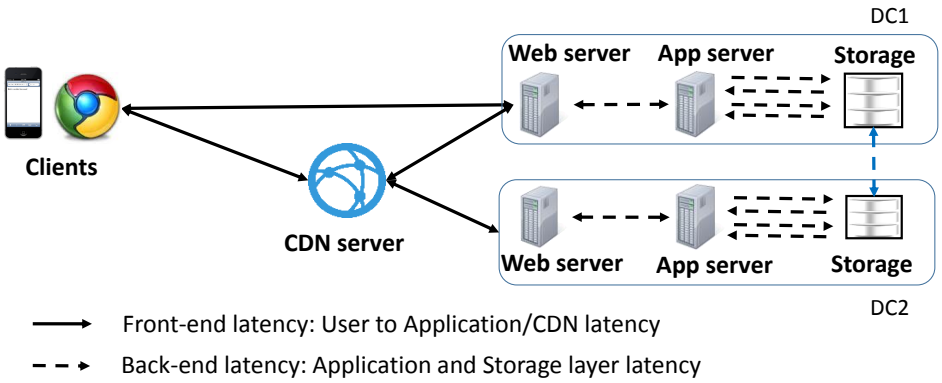


Fig. 1.1. Ecosystem of modern web applications

layer latency. The rest of this chapter presents in further detail, the complexities involved in serving web-pages, the challenges in lowering web-page latencies, and the contributions of this thesis towards addressing these challenges.

1.1 Modern web applications

Figure 1.1 shows the ecosystem of a typical web application comprising of the various components involved in serving a web-page to a user. Downloading a web-page is a complex activity and requires multiple request-response round-trips from multiple domains to fetch all the required objects in the page. When the user loads a web-page, the client browser resolves the domain address for the page URL, and sends a request to the corresponding web server (client-facing server). The web server forwards the request to the application servers, which query the storage servers to retrieve the meta-data required to compose an initial rendering of the web-page and responds back to the client with the initial HTML. The browser parses the initial HTML and initiates requests to fetch the different objects in the page. Further, a large fraction of these objects are often served from the Content Delivery Networks (CDN) servers that are located closer to the user.

Clearly, there are two sources of latency that contribute to the user perceived end-to-end latency.

User to web server (front-end) latency, which includes downloading the web-pages, parsing them, retrieving further objects requested by dynamic JavaScript executions. As described earlier, these objects can arrive either from the application server or from the CDN servers.

Application and Storage layer (back-end) latency, which includes retrieving the meta-data required for the initial rendering, responding to user queries and fetching content from the storage servers accordingly. Further, most real-world web-pages are multi-tiered applications, requiring interaction between multiple components to serve a web-page. Typical web applications have client facing web servers, business logic servers that determine the objects required to compose the web-page, and storage servers that store the meta-data along with the actual content. Also, composing a web-page often requires multiple calls from the business logic servers to the storage servers.

1.2 Emerging trends in lowering web application latency

The push towards reducing web-page latencies has brought forth many efforts from both academia and industry to develop tools and techniques to reduce web application latencies along both the sources described above.

Tackling front-end latency: To tackle the front-end latency, we have seen the emergence of new protocols such as SPDY [15] (expected to be a key part of the HTTP 2.0 standard) which tries to reduce page-load latency by employing a combination of techniques like compression, prioritization and multiplexing the delivery of objects between the client and server. SPDY allows multiple, simultaneously multiplexed requests over a single connection, saving on round trips between client and server, and preventing low-priority resources from blocking higher-priority requests.

Tackling back-end latency: Back-end latency has been typically handled by scaling and resource provisioning at the application servers. The cloud industry already provides mechanisms to scale up or down the number of server instances in commercial cloud platforms. Since these applications serve thousands of geographically dispersed users, retriev-

ing content from the storage servers for such geo-distributed applications can have significant latencies. Consequently, a number of datastores that replicate data across geographically distributed datacenters (DCs) have emerged in recent years [7–14]. A distinguishing aspect of such cloud datastores is the use of algorithms (e.g., quorum protocols [7, 9], Paxos [8, 13, 14]) to maintain consistency across distributed replicas. This is necessitated given these datastores often store application content (e.g. user information, application meta-data, etc.,) that require stronger consistency.

While prior works help reduce the latency at a specific component, they often have different impact on the end-to-end user perceived latency. Hence, despite these efforts, web-page latencies continue to remain significant, constituting 80-90% of overall application response time by some reports [1, 5].

1.3 Challenges in reducing web application latency

A key challenge in reducing web application latency is their complexity [16, 17]. Webpages comprised of tens to hundreds of static and dynamic objects (images, style-sheets (CSS), Javascript (*JS*) files, etc.) served from multiple domains including CDNs. However, the benefits with large-scale edge-caching employed by today's CDNs is limited by the fact that some objects are more critical to web-page latencies than others (e.g., *JS* that fetch further objects might be more important for an initial rendering than *JS* that support subsequent user requests). Yet, CDN placement and caching algorithms are agnostic of the criticality of objects to the page-load process, which could impact overall page latencies. The issues are particularly significant for the vast majority of pages beyond the most popular few hundreds, since the objects of these pages may not be sufficiently popular to be cached naturally at the edge.

The interactive nature of these web applications pose stringent requirements on the consistency and availability of content (including the application state) stored by the applications. Hence, an important requirement on the datastores used by these applications, is the need to support consistent updates on distributed replicas while ensuring both low write and

read latencies across geographically distributed users. Tailoring datastores to application workloads is especially challenging given the scale of applications (potentially hundreds of thousands of data items), workload diversity across individual data items (e.g. celebrities and normal users in Twitter have very different workload patterns), and workload dynamics (e.g. due to user mobility, changes in social graph etc).

Further, meeting these stringent SLA requirements is a challenge given the outages in cloud DCs [18, 19], and the high variability in the performance of cloud services [20–22]. This variability arises from a variety of factors such as the sharing of cloud services across a large number of tenants, and limitations in virtualization techniques [20]. For example, [21] showed that the $95\%ile$ latencies of cloud storage services such as tables and queues is 100% more than the median values for four different public cloud offerings.

1.4 Thesis contributions

The primary goal of this thesis is to develop frameworks that reduce the end-to-end latency of geo-distributed web applications. We present solutions that can simultaneously tackle both the front-end and back-end latencies described earlier.

•To reduce the front-end latency, we present a framework that allows CDNs to map objects more important for page latencies to the faster CDN cache layers. We consider a family of schemes for determining object priorities including a strategy based on content type, a strategy that prioritizes objects needed for an initial rendering of the page, and a scheme that explicitly takes the dependencies across objects of the page into account. We present adaptations of CDN cache placement and replacement algorithms that take object priorities into account, while still considering object popularity. In order to avoid staleness related misses, We present a family of schemes for proactive refreshing that differ in terms of which objects are refreshed. We consider an approach where, to keep bandwidth overheads small, only objects that has the most impact on page-load latency are proactively refreshed.

•We reduce the back-end latency by developing frameworks that can tailor the replication policies of geo-distributed datastores to the application workloads. We develop analytical

models automatically determine how best to customize replication configuration including the number and location of the replicas, as well as the underlying consistency parameters (e.g., quorum sizes in a quorum based system) to meet the desired application objectives. While our initial focus is on quorum-based systems given their wide usage in production [7,9], and the rich body of theoretical work they are based on [23–26], our frameworks can be extended to other classes of cloud storage systems as well. Our models are distinguished from theoretical quorum protocols in that we consider various practical aspects that arise in the context of datastores like impact of DC failures on latency, latency percentiles, asymmetry in read-write traffic and focus on realistic workloads in wide-area settings.

•Finally, we reduce the end-to-end latency by making multi-tiered applications resilient to the inherent performance variability in today's multi-tenant commercial cloud environments. To this end, we developed a system called *Dealer*, which dynamically re-routes requests across different application component replicas. *Dealer* abstracts application structure as a component graph, with nodes being application components and edges capturing inter-component communication patterns. To predict which combination of replicas can result in the best performance, *Dealer* continually monitors the performance of individual component replicas and communication latencies between replica pairs. Modern web applications consist of many components, not all of which are represent in each DC, and the costs are extremely high to over-provision each component in every DC to be able to handle all the traffic from another DC. *Dealer* is able to redistribute work away from poorly performing components by utilizing the capacity of all component replicas that can usefully contribute to reducing the latency of requests.

1.5 Research Methodology

We adopt a systematic approach for solving the above challenges, by beginning with real-world measurement studies on commercial cloud platforms to understand the impact and scope of the problem. Leveraging on the insights gained from the initial study, we develop solutions grounded by sound theoretical principles, and adapt them to the specific

problem being solved. Finally, we evaluate our ideas under realistic conditions to show their benefits – using real applications on commercial cloud deployments, or through extensive real-world application traces.

Importance of page-aware content-prioritization in reducing user perceived latency: To understand the importance of priority-based caching and delivery of content from the CDN, we conduct end-to-end measurements by downloading and analyzing 100 real-world pages selected randomly across all ranges of popularity from the Alexa Top pages. Using CDN specific pragmas which are set along with each request header when loading the pages, we derive insights on how the various objects in a page are served from within the CDN hierarchy. Through controlled experiments on these pages, we show the potential to achieve hundreds of milliseconds reduction in latency by prioritization of content at the CDNs. Through trace-based analysis of real CDN deployments, we also show the feasibility of priority based caching to reduce the miss rate for critical content while incurring only modest increases in the overall miss rates.

Importance of customizing datastore replication to application workloads: We demonstrate the benefits of our replica configuration framework using real-world traces of three popular applications: Twitter, Wikipedia and Gowalla, and through experiments with a multi-region Cassandra cluster [9] spanning all 8 EC2 geographic regions. While latencies with Cassandra vary widely across different replication configurations, our framework generates replica configurations which perform very close to predicted optimal on our multi-region EC2 setup. Further, our schemes that explicitly optimize latency under failure are able to out-perform failure-agnostic schemes under the failure of a DC by more than 40% while incurring only modest penalties under normal operation. Our results show the importance of adapting and customizing replica configurations to the heterogeneity in workloads.

Importance of application-aware fine grained request re-routing: We show the benefits of our system, *Dealer*, by integrating it with two real-world applications – *Thumbnail* and *StockTrader*. We deployed these applications on commercial cloud infrastructures (including Amazon AWS, and Windows Azure), and show that *Dealer* was able to reduce the 90th percentile application response times by a factor of 3 compared to a system that

used traditional DNS based redirection techniques. Further, *Dealer* ensures low latency, and significantly out-performs application level redirection mechanisms under a range of controlled experiments.

1.6 Thesis Organization

This thesis is further organized as follows. Chapter 2 presents our work on reducing the front-end latency of web-applications through page-aware content prioritization at CDNs. Chapter 3 describes our performance-aware replication configuration frameworks for geo-distributed datastores. Chapter 4 presents *Dealer*, our system that performs application-aware fine grained request redirection.

2. REDUCING LATENCY THROUGH PAGE-AWARE MANAGEMENT OF WEB OBJECTS BY CONTENT DELIVERY NETWORKS

2.1 Introduction

Reducing the latency of web pages is critical for electronic commerce as it directly impacts user engagement and revenue [2,4,27]. Amazon, e.g., found that $100ms$ of latency costs 1% in sales [27], while Google Search found that a 400 millisecond delay resulted in a 0.59% reduction in searches per user [5].

The quest to reduce web-page latencies has triggered much effort in the networking community among both researchers and practitioners. On the one hand, we have seen the large-scale adoption of widely distributed Content Delivery Networks (CDNs), that involve placing caches at thousands of Internet vantage points, close to end users. On the other hand, we have seen the recent emergence of new protocols such as SPDY [15] that significantly influenced the HTTP 2.0 standard. Despite these efforts, web-page latencies remain significant, constituting 80-90% of overall application response time by some reports [1,5].

A key challenge in reducing the latencies of web-pages (the time to get an acceptable initial rendering of the page, formally defined in §2.3.1) is their complexity [16, 17]. Web pages are comprised of tens to hundreds of static and dynamic objects such as images, style-sheets (CSS), and JavaScript (*JS*) files, which may be served from multiple domains. Web-page download process has complex dependencies [28, 29], where some objects may have more impact on web-page latencies than others. The first objects fetched during a download (e.g., HTML, CSS, and *JS*) may need to be parsed or executed to decide which objects to fetch subsequently. Objects needed for an initial rendering of the page (e.g., to trigger a browser load event) may be more critical to the user experience than those that refine the initial rendering.

The need to accommodate the varying impact of individual objects on overall latencies has begun to receive attention from the community [15, 30]. Specifically, SPDY allows servers to transmit objects out of order to reflect their priority in the page load process. While useful in single server settings, most web pages today are served from multiple domains, and make extensive use of CDNs. Simply enabling SPDY between clients and CDN servers addresses only part of the problem. It is also necessary to reduce the CDN retrieval time of critical objects, especially since CDNs are typically organized as a hierarchy of caches [31] with different capacities and latencies at each layer.

Our motivation arises in part from the results of a study we conducted in which we collected end-to-end measurements of clients downloading pages from a number of web sites. The data shows that (i) objects appearing on the same web page are often served from multiple layers of the CDN cache hierarchy; (ii) critical objects are not always served from the fastest caches; and (iii) delays in serving a small number of critical objects can disproportionately impact overall latency.

Motivated by these findings, we present a framework that allows CDNs to map objects more important for page latencies to faster cache layers. Our framework is enabled by the increasing shift of popular web-sites to CDNs for full-site delivery (e.g., for 89% of the pages in our study above, the main HTML document was served by the CDN). We consider a family of schemes for determining object priorities including a strategy based on content type, a strategy that prioritizes objects needed for an initial rendering of the page, and a scheme that explicitly takes the dependencies across objects of the page into account. We show how CDN cache placement and replacement algorithms may be redesigned to take object priorities into account, while still considering object popularity. We consider an approach where, to keep bandwidth overheads small, only objects most critical for latency are proactively refreshed to avoid staleness related misses. We present a family of schemes for proactive refreshing that differ in terms of which objects are refreshed.

We present an extensive evaluation study of our schemes using a combination of controlled experiments that emulate real web pages in hierarchical CDN settings, as well as trace data from a real CDN deployment. Our evaluations seek to understand the benefits

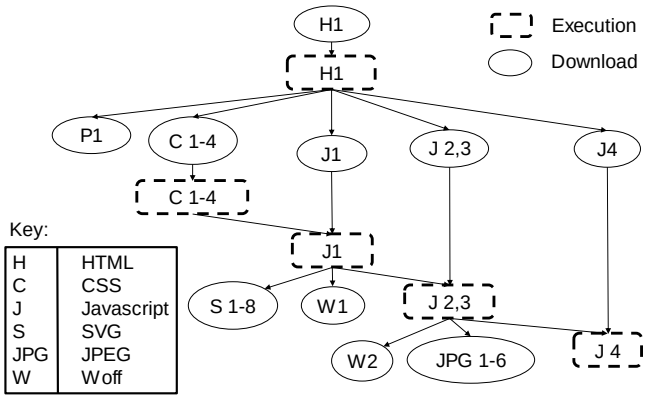


Fig. 2.1. Dependency graph for a single load of www.apple.com. Each node shows download or execution of an object, and the directed arrow shows the dependency between them.

of prioritization in CDN placement and refresh schemes, the relative benefits of different schemes for prioritization, and the sensitivity of our results to page popularity and composition.

Our evaluations with 83 real-world pages show that 30% of the most popular pages and 59% of the other pages show latency reduction larger than $100ms$, with some pages showing latency reductions as high as $500ms$. Both placement and proactive refreshing are important in achieving the benefits. For the vast majority of pages, considering content type in both placement and proactive refreshing provides most of the benefit. However, the additional benefits with other prioritization schemes can be significant in lower hit rate regimes, and when the penalty of going to the origin is higher. Finally, using trace driven simulations, we show the feasibility of the priority-based caching approach for reducing miss rates of page-critical objects in CDNs by 60% with modest increases (less than 2%) in the overall byte miss rates. We also highlight the opportunity of minimizing stale misses for objects critical for latency by as much as 60% while incurring additional bandwidth costs of less than 0.02%.

2.2 Motivating measurement study

Web pages consist of tens to hundreds of objects of multiple content types (HTML, CSS, *JS*, images). A typical page load process involves significant dependencies across objects [28, 29]. An initially downloaded HTML, CSS or *JS* (henceforth referred to as *HCJ*) object (which often embeds pointers to other objects) must be parsed (and executed) to identify further objects to download. Browser policies may dictate dependencies – e.g., execution of a *JS* must wait for a prior CSS to complete execution. Figure 2.1 shows an example dependency graph obtained using wprof [28]. Clearly, not all objects have the same impact on page latencies – e.g., $C1$ is much more important than $W1$. Further, content type need not necessarily reflect object importance – e.g., some *HCJ* objects may not be required for an initial rendering of a page most important for user experience, while Non-*HCJ* objects such as images may in fact be required. Moreover, other objects may be dependent on Non-*HCJ* objects – e.g., a *JS* execution may wait on the arrival of a sprited image.

CDNs consist of a hierarchy of caches [31], typically consisting of clusters of servers deployed in multiple edge locations, and in parent locations. A user request that arrives at a server in an edge cluster (*First* server) could "hit" either at the memory or disk layer of that server. On a cache miss at the *First* server, requests could be directed to other servers in the CDN hierarchy (*Second* server), which could be a peer server in the same cluster or a server in a parent cluster. The latency of CDN served objects may vary widely depending on whether the object hits at the CDN and the layer that serves it.

To understand opportunities for reducing page latencies with CDNs by better mapping more important objects to faster CDN caches, we analyzed a prominent CDN that extensively provides edge caching (which we refer to as *CDN*). In the rest of this section, we discuss our measurement approach, and our findings.

2.2.1 Measurement methodology

We conduct end-to-end experiments by downloading real web pages from web clients and for each page measure the fraction of objects served from the different *CDN* layers. To determine the layer in the CDN hierarchy from which an object is served, we leverage HTTP pragma headers supported by CDNs for debugging purposes. Specifically, *CDN* supports the following pragma headers – *CDN*-x-cache-on, *CDN*-x-remote-cache-on and *CDN*-x-get-request-id. We set these pragma headers on all HTTP requests issued from the client. If the object is served by *CDN*, then the first contacted *CDN* server appends an X-Cache header in the HTTP response, and if a second *CDN* server is involved it appends an X-Cache-Remote header. The response also contains an X-*CDN*-Request-ID header with a dot-separated list of request IDs appended by each of the contacted *CDN* servers.

The X-Cache and X-Cache-Remote response headers contain values such as TCP MEM HIT, TCP HIT, TCP MISS, TCP REFRESH MISS, which respectively indicate a hit in the memory of that server, a disk hit, a server miss and a TTL expiry of a cached object with a new version fetched from the origin. We also count the number of request IDs in the X-*CDN*-Request-ID header to obtain the total number of *CDN* servers contacted. We use the values in these three headers to determine the layer from which the object was served. For instance, a TCP MEM HIT in the X-Cache header with one ID in the X-*CDN*-Request-ID header implies the object was served from the memory of the first *CDN* server. Like-wise a TCP MISS in the X-Cache header with a TCP MEM HIT in the X-Cache-Remote with two IDs implies the object was served from the second *CDN* server, while a MISS in both the headers means the object was fetched from origin. Note that, if we see a TCP MISS in both the headers with more than two IDs, due to limitations of the pragma headers, it is not possible to precisely tell if the object was served from the origin or another *CDN* server. But in our real runs we find these cases to be insignificant. For 90% of the pages fewer than 6 requests had more than two IDs in the X-*CDN*-Request-ID header.

We chose 100 Web pages for our measurement study across a wide range of popularity (Alexa US top sites [32]). Our measurement set has 40 pages in the Alexa rank top 1–1000

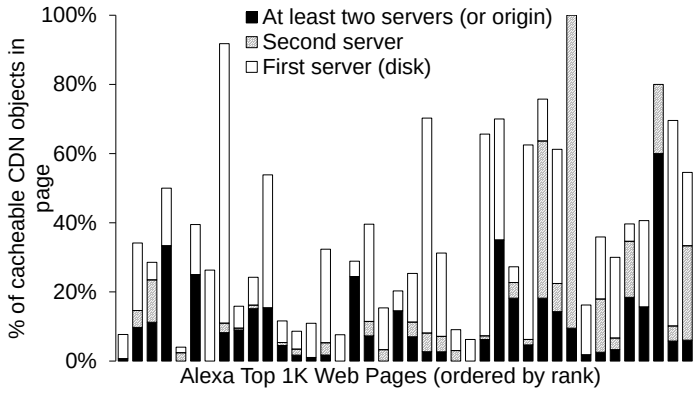


Fig. 2.2. Breakdown of the different caching layers from which CDN-served objects of a web page are received. Fractions not shown, hit at the memory layer of the first server.

and 60 pages beyond rank 1000 which we refer to as *Top1K* and *Beyond1K* respectively in the rest of this chapter. These pages were selected based on whether they had a good fraction of their objects served from *CDN*. Across all pages, at least 38% of the objects were served from *CDN* and for 25% of the pages more than 68% of the objects were served from *CDN*. Further, at least 92% of *CDN* served objects were cacheable for 90% of the pages. We also find that the main HTML for 89% of these pages were served from *CDN*.

Back-to-back downloads of the same page may artificially inflate the hit rates in subsequent runs owing to objects cached by the CDN from the earlier runs. To ensure our measurements themselves do not impact the hit rates, our entire set of measurements were spaced out across several weeks with consecutive downloads of the same page separated by 3 days – an analysis of the TTLs of objects in the pages indicated most objects would expire by this time.

2.2.2 Key findings

We now present key observations from our study.

• *Objects of a Web page may be served from different CDN caching layers incurring very different latencies:* Figure 2.2 presents the fraction of cacheable *CDN* objects served from each layer in the CDN hierarchy for a given run of the Alexa Top1K websites. Each stacked

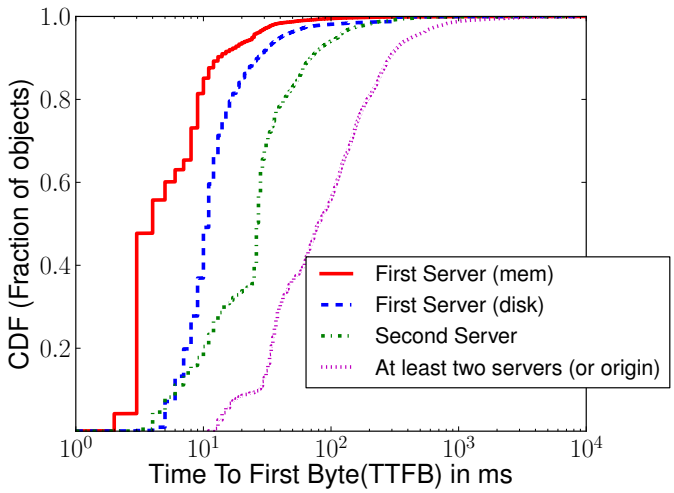


Fig. 2.3. TTFB of objects served at different CDN layers

bar corresponds to a web page, and the segments in the bar show the breakdown – e.g., for the second most popular page (second stacked bar from the left), going from top to bottom, 19% of the cacheable *CDN* objects are served from the disk of the first server, 5% from the second server, 10% from at least 2 *CDN* servers (or origin) and the rest (66% – not shown) are served from the memory of the first server.

Figure 2.3 shows a distribution of the *Time To First Byte (TTFB)* of objects across all pages categorized by the layer from which it was served. The TTFB for an object is the time elapsed from when the request was sent from the client until the first byte of the response was received at the client, including the network time and retrieval time at the cache (or server). As expected, the TTFB observed across the different CDN layers vary substantially.

Figure 2.4 shows the fraction of cacheable CDN objects that are served from beyond the first contacted *CDN* server for the Top1K and Beyond1K classes for three different days. The figure shows that a significant fraction of objects are served from beyond the first *CDN* server even for the Top1K pages – e.g., the $50th(90th)$ %ile of objects served from beyond the first server were more than $11\%(34\%)$. Further, for the Beyond1K pages more objects are served from the farther layers in the CDN hierarchy – e.g., the $50th(90th)$ %ile of objects served from beyond the first server were more than $37\%(74\%)$. Moreover across

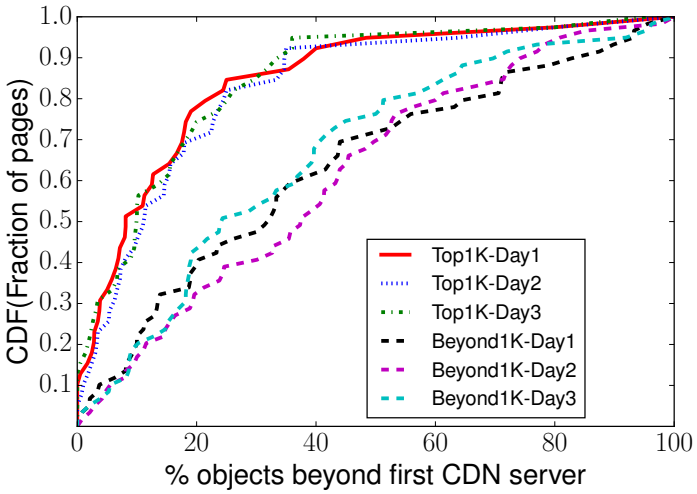


Fig. 2.4. Fraction of cacheable CDN objects that are served from beyond the first CDN server across pages of two popularity classes for 3 different days

multiple days, the hit rates remain similar for both the classes indicating the trends are consistent across many days and the hit rates are representative of the page popularity. We performed similar analysis across multiple days from different geographical locations and we find the trends to be similar. For instance, the median percentage of objects served from beyond the first server across the Top1K pages from another US location for three days were 15%, 12% and 10%, while for the Beyond1K pages they were 31%, 24% and 32%, with the $90\%ile$ being higher than 38% and 71% across the days for the Top1K and Beyond1K respectively.

• *Critical objects are not always served from the fastest CDN layers:* Figure 2.5 shows a stacked bar graph with the number of objects served from each level in the dependency graph (as described earlier in this section) of *www.weather.com*. Each bar corresponds to a level in the graph and the stacks in the bar show the number of objects served from the corresponding CDN layer. The figure shows that many critical objects (generally the internal nodes of the dependency graph) are served from beyond the first server indicating the potential to reduce page latencies by considering object priorities in CDN mechanisms.

Figure 2.6 shows a breakdown of the CDN layers from which the *HCJ* objects are served for the Alexa Top1K pages. While *HCJ* objects do not exactly correspond with

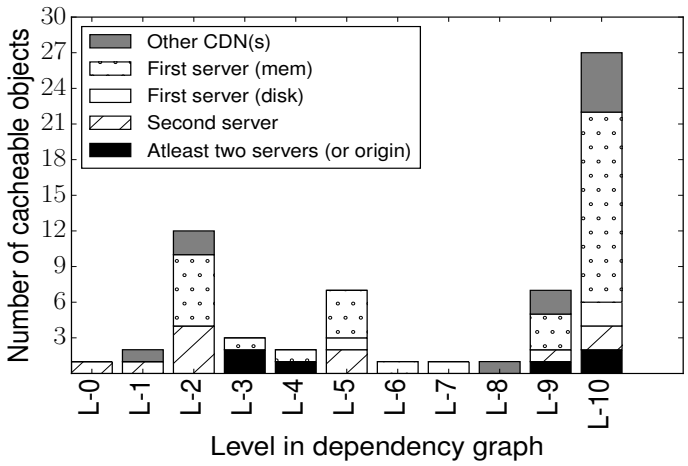


Fig. 2.5. Number of objects from different caching layer at each level of the dependency graph for *www.weather.com*

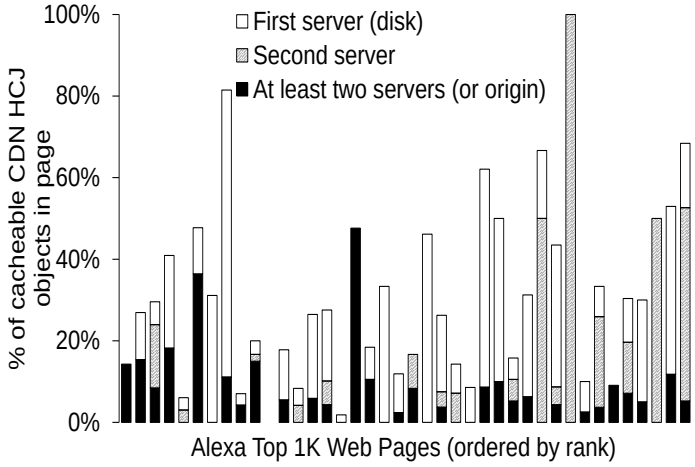


Fig. 2.6. Breakdown of the different caching layers from which CDN-served *HCJ* objects (HTML, CSS, JS) are received.

objects important for a page load, we consider them here for simplicity. Note that a significant fraction of the *CDN* served cacheable objects are *HCJ* objects – e.g., 42% of *CDN* served cacheable objects are *HCJ* for 50% of the pages. The figure shows that in general across all pages a significant fraction of *HCJ* objects are served from different layers of the CDN hierarchy incurring vastly different latencies – e.g., for 50% of the Top1K pages more than 10% of the *CDN* served cacheable *HCJ* objects are served from higher layers with this fraction being greater than 48% for 10% of the pages.

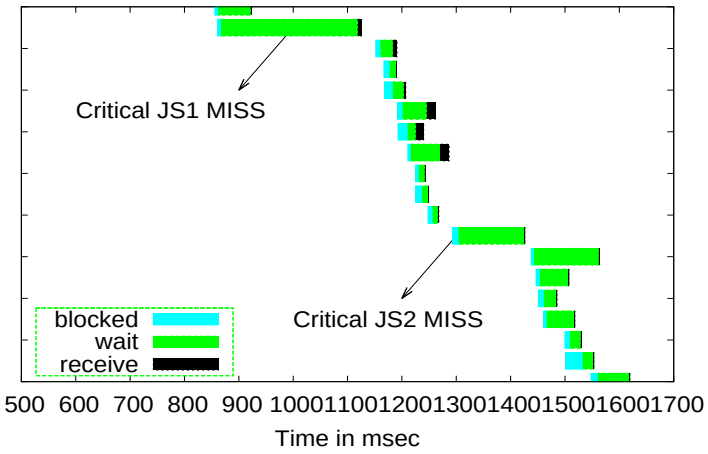


Fig. 2.7. Serving delays in *HCJ* objects disproportionately impacts latency

• *Delays in serving a few critical objects can disproportionately impact page latency:* Figure 2.7 shows a section of the waterfall diagram which depicts how the objects arrive at a client during the download of an actual web page. The X-Axis is the time since the start of the download (only the relevant time segment is shown). Each bar corresponds to an object, and extends from when a request to that object was made, to when the object was fully downloaded at the client. Further, each bar shows the breakdown of the time spent waiting for a connection to the server (blocked), time spent waiting for the first byte of the response (wait) and time spent in receiving the object (receive). From the figure, we see that many objects were delayed because they depended on two *JS* objects, which got delayed ($267ms$, $135ms$ respectively with the time dominated by wait time). Further investigation showed both *JS* objects were cacheable and served from *CDN*, but were served beyond the first two servers in the hierarchy. Interestingly, many of the dependent objects hit in the CDN. Avoiding delays of these two *JS* objects would potentially reduce page load times by over $400ms$ (a 19% reduction).

• *Both true misses and stale misses contribute to objects being served from beyond the first* CDN *server:* We conducted a deeper analysis on the causes for the first *CDN* server misses. We found that even though true misses contribute greatly to the first server misses, we also find significant fractions of staleness related misses – e.g., the fraction of first server misses

that were staleness related misses is more than 29% and 45% at the median and $75\%ile$ respectively, with the rest being true misses.

2.3 Enabling page-awareness in CDNs

In §2.2, we have shown that there is opportunity to reducing web-page latency by mapping objects in a page most critical for latency to the fastest caches in the CDN hierarchy. In this section, we revisit CDN design to exploit this opportunity. In doing so, a number of issues must be tackled including (i) determining which objects to prioritize; (ii) reconciling the need to prioritize objects more critical for latency with the traditional CDN goals of placing more popular objects at the edge to save bandwidth, by appropriately tailoring cache placement and replacement policies; and (iii) avoiding staleness misses for important objects in addition to capacity misses. We discuss our schemes for each of these issues in the following sections.

2.3.1 Schemes for prioritization

We consider a range of schemes for assigning priorities to objects, which involve different trade-offs between the complexity of the priority-marking scheme, and the potential latency benefits as we discuss below:

• **Prioritizing objects based on content type: (*Type*)** Our first scheme is content-type based priority assignment, where objects are accorded priorities based on content type – specifically, HTML objects receive higher priority, followed by JS and CSS, and finally images and others. This, in fact, conforms to the best-practices for prioritization with the SPDY protocol, and is implemented by Chrome today [33].

• **Prioritizing objects needed for initial page rendering (*OLType*):** Content type may not accurately reflect the importance of an object to page latency. Objects needed for an initial acceptable rendering of a page are more critical to user experience than other objects. While images may be important for such an initial rendering, some *HCJ* objects may not be required. A commonly used indicator to identify such an initial version of the page is an

Onload event triggered by the browser. The time to generate a browser Onload event, which we refer to as *Onload Time (OLT)*, is a commonly used metric to measure web page load performance [34]. This motivates our *OLType* strategy which prioritizes objects prior to the Onload event, and among such objects, prioritizes objects based on content type. More generically, this strategy could be refined to consider other indicators of an initial page load such as "above the fold" content, content related to most critical visual progress [35], or content with the highest utility to users [36].

• **Prioritization based on page dependency graph (*OLDep*):** While *Type* and *OLType* are fairly coarse-grained strategies, a more fine grained prioritization scheme is to consider the actual dependency graph associated with the page, and assign prioritizes based on the graph – e.g., in Figure 2.1, J1 would be accorded higher priority than J4. This motivates the *OLDep* algorithm. Like *OLType*, *OLDep* prioritizes objects needed to trigger the Onload event over other objects. However, among objects needed for the Onload event, it prioritizes objects based on their depth in the dependency graph, with *HCJ* objects preferred among those at the same depth. Likewise, objects after the Onload event are also prioritized based first on their depth in the dependency graph, and then their content-type.

2.3.2 Balancing popularity and priority in cache placement and replacement

CDNs have two potentially conflicting goals that they must consider in deciding whether to cache objects at a given edge location: (i) cost savings; and (ii) minimizing user latency. For cost savings, it is desirable to cache the most popular objects at the edge. However, for user latency savings, it is desirable to cache high priority objects at the edge. Since more popular objects might not be the highest priority and vice versa (e.g., page logos vs product related images), it is important to carefully reconcile these considerations.

A naive approach to tackling these issues is to use multiple LRU queues with one queue per priority level. When an eviction is required, incoming objects evict the least recently used objects in lower priority queues, before evicting the least recently used objects in the queue having the same priority as the incoming object. A key limitation of this approach

is that objects with higher priority tend to remain in the cache even if they are no longer accessed, creating cache starvation for the popular, but low priority objects.

Instead, our approach is inspired by the notable Greedy-Dual-Size algorithm [37] which considers how to balance locality of access patterns with object size and the variable costs associated with fetching objects. We adapt this algorithm to balance object priority and popularity by assigning a utility to each object based on its priority (P_i), and implement our cache as a priority queue ordered by the utility of the objects. When an eviction is required, objects with the lowest utility value (which are located at the tail of the queue) are evicted first. To prevent high priority objects from residing permanently at the head of the queue, we gradually decrement the utility value of the objects in the queue that are no longer accessed. This may be achieved in a computationally efficient manner by maintaining a monotonically increasing global *clock* for the cache, which is added to the utility value of the object ($U(i)$) as follows:

$$U(i) = clock + 1 + (R-1) * \frac{P_{min} - P_i}{P_{min} - 1} \tag{2.1}$$

Here, P_{min} is the lowest assignable priority and is higher than P_{max}, the highest assignable priority. For simplicity, we fix $P_{max} = 1$ in our formulation and hence P_i varies between 1 and P_{min}. The parameter R is the ratio of the lowest and highest assignable priorities (P_{min}/P_{max}). A linear interpolation is used to assign the initial utility to objects with any priority. R is a knob that the CDN could tune to decide how much to favor hits to higher priority objects over lower priority objects, and we evaluate the impact of R with real traces in §2.6.1. The utility value of the object is updated using the above equation when the object is accessed from the cache. The monotonicity of the clock is maintained by incrementing the clock on an eviction to the utility value of the evicted object. Therefore, objects that are accessed more frequently will have a higher utility value than objects in the cache that do not see any further accesses. This ensures that high priority objects that are no longer accessed, eventually get evicted from the cache. Finally, note that an item is placed in the cache only if its utility exceeds the utility of the lowest utility object in the queue.

2.3.3 Priority based proactive refreshing

Staleness related misses could be avoided by proactively refreshing objects that are currently in the cache, but are about to expire in the near future, at the cost of some bandwidth related to unnecessary refreshes. Given the trade-off between reducing staleness misses and the bandwidth penalty, it is desirable to only proactively refresh those objects most important for page latency. We consider a family of strategies which primarily differ in terms of which objects are proactively refreshed:

• ***HCJ*,** which only proactively refreshes *HCJ* objects. The primary advantage of the scheme is the simplicity in identifying objects to refresh.

• ***BO*,** which only proactively refreshes all objects required for the page Onload event. While the strategy has the potential to better mirror objects most important for latency, it is more involved to identify these objects.

• ***HCJ BO*,** which only proactively refreshes *HCJ* objects needed for the page load event. This strategy has the potential to reduce the bandwidth overheads compared to *HCJ* while matching its latency benefits.

For all schemes, a refresh is triggered only when a request for the object is received and the following conditions are satisfied: (i) the object is unlikely to receive further accesses until it expires; and (ii) the estimated number of accesses in its lifetime is sufficiently high to warrant a proactive refresh. Specifically, we require that

$$A_i * e_i \leq T_{P_i} \tag{2.2}$$

$$A_i * l_i \geq K_{P_i} \tag{2.3}$$

Here, A_i, l_i, e_i, and P_i are respectively the average request rate, lifetime, time left to expiry and priority of the object. A_i is computed by tracking the number of accesses seen by the object since it entered the cache, and l_i and e_i are obtained from the cache-TTL or expiry-time of the object. T_{P_i} is ideally kept smaller (close to 1) to trigger just-in-time refreshes. Note that larger T_{P_i} and smaller K_{P_i} support more aggressive refreshing. We evaluate the impact of these parameters with real traces in §2.6.2. These thresholds may be

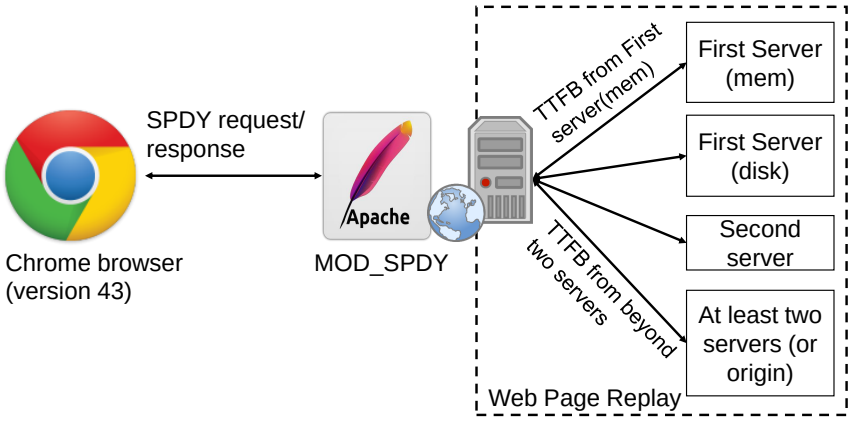


Fig. 2.8. Experimental setup for evaluating latency benefits.

set differently across priority classes to support more aggressive or conservative refreshing for each class.

2.4 Evaluation Methodology

Our evaluations have two primary goals. First, we seek to understand the potential latency benefits of our various schemes for priority-based placement and proactive refreshing, as well as their sensitivity to factors such as page popularity, CDN hit rates, page composition, and the relative latencies of various CDN cache layers. Second, we also seek to understand the impact of prioritization on CDN cache hit rates, and the bandwidth costs associated with the proactive refresh schemes.

We achieve the first goal by conducting a detailed emulation study of our schemes for the real-world pages analyzed in §2.2). Our emulations allow us to compare schemes in a fair manner while capturing the heterogeneity in latency (and object fetch times) across the CDN hierarchy, and realistic factors such as client execution times. We tackle the second goal by conducting a detailed analysis of traces from a real CDN. We present our experimental setup for latency comparisons in the rest of the section, latency benefits results in §2.5.1, and our trace-driven analysis in §2.6.

Table 2.1
Placement and refresh schemes studied in our evaluation.

Placement schemes	Proactive refresh schemes
OBS	None
Type	*HCJ*
OLType	*BO*
OLDep	*HCJ BO*
	All

2.4.1 Methodology for latency comparisons

We use *Onload Time (OLT)* (defined in §2.3.1) to quantify the page load latency. We focus on OLT since it is objective, easy to determine, and widely used as a measure of page latency, while other indicators [35] are inherently more subjective. We do not consider the time to download the last byte for the page since many of the pages we evaluated tend to request objects indefinitely, and the time for an initial rendering is more important in practice.

We next discuss factors impacting our comparisons. A first factor is the CDN hit rates in terms of how many objects are served from each layer of the hierarchy. Of particular importance is the edge hit rate (*EHR,*) which we define as the fraction of objects served from the first CDN server (edge). A second factor is the composition of pages. Specifically, the fraction of *HCJ* and *BO* objects as well as the complexity of the dependency graph, can impact our comparisons. We compare our schemes with 83 real-world web pages analyzed in §2.2, which exhibit a wide range of diversity in terms of popularity and page composition. We highlight the characteristics of a page which impact the relative performance of schemes when appropriate. A third factor is the relative ratio of latency to the various layers of the CDN hierarchy, which we vary based on real measurements.

2.4.2 Schemes compared

Our schemes (Table 2.1) include:

Baseline for comparison (*OBS*): The *OBS* scheme corresponds to the placement observed when the page was loaded in the real-world measurements (§ 2.2). All objects served by *CDN* are fixed to the same layer from which it was served during the real page-load. All cacheable objects not served through *CDN* are split across the caching layers according to the hit-rates for cacheable objects observed in the real page-load.

Placement schemes: Our CDN placement schemes differ in their algorithms for assigning objects to the cache layers according to the fractions described above. We consider the *Type*, *OLType* and *OLDep* schemes are as described in § 2.3.1.

Proactive refresh strategies: This includes the *HCJ*, *BO* and *HCJ BO* schemes described in §2.3.3 which primarily differ in terms of which objects are proactively refreshed. We also consider the *None* and *All* strategies as baselines for comparison which indicate none or all of the objects are proactively refreshed.

To ensure fair comparisons with *OBS*, for all schemes, all non-cacheable objects were always pinned to the farthest layer (origin). All objects that observed a staleness miss with *OBS* were served from the same layer as they were with *OBS* if they were not proactively refreshed. For example, with no proactive refresh all objects that saw refresh misses in *OBS* were pinned, while with the *HCJ* strategy non-*HCJ* objects with refresh misses were pinned. All remaining objects were assigned to the CDN cache layers as per the placement strategy. In doing so, the total number of objects served from each CDN layer was ensured to be (i) the same as *OBS* in the absence of proactive refreshing; and (ii) the same as *OBS* augmented with that proactive refreshing strategy otherwise.

If only a subset of objects of a given priority class (e.g., a subset of *HCJ* objects with *Type*) can be accommodated at a given CDN layer to satisfy the constraints on the number of objects that could be served from each layer, our schemes pick objects from within the priority class randomly. To ensure a robust comparison, we generate 50 different placement configurations with each scheme. We load each web-page with each scheme for each of its

50 configurations, alternating across schemes. We clear the browser cache between runs to eliminate the impact of local browser caching. We usually summarize the performance of a scheme for a given page by presenting the median OLT across the 50 configurations, but we also report on higher percentiles.

Many of our schemes require knowledge of objects needed for Onload and their dependencies. Rather than detailed activity dependence graphs [28] that may vary across runs, we obtain more static object level dependencies [29, 38]. Through multiple controlled experiments that each delay an object in a page, we determine the objects needed for Onload based on whether the Onload event is delayed. Likewise, dependent objects may be determined based on delays observed in their download times. We determine object importance based on its depth in the dependency graph rather than consider critical paths which may vary across runs.

2.4.3 Experimental setup

Figure 2.8 presents our experimental setup. Web pages are hosted on a web server (corresponds to an edge server in a CDN cluster), where TTFBs to the different caching layers are emulated and the page-load latency from an actual web browser is measured.

Web pages exhibit significant variability in the number of objects and aggregate download size for a given web page, even over short intervals of time. To ensure fair comparisons, we used an open source tool called web-page-replay [39]. Entire web pages including all constituent objects were first recorded through downloads from the actual web server(s). Then, the same recording was replayed for all schemes in later experiments. Some web pages still showed variability as they had *JS* that requested different URLs (e.g., using a random number or date) over different runs. We modified the web-page-replay code to replace such occurrences with constant values to ensure the same objects were requested for all schemes.

We focus our evaluations on settings where SPDY is enabled between the client and the edge server, in order to highlight that our benefits are complementary to SPDY. We also

note that our schemes show similar benefits in the presence of traditional HTTP as well. We use apache mod_spdy server co-located with WPR, and Chrome browser (version 43.0) running SPDY (version 3.0) as the client in all our experiments. The client uses SPDY to forward object requests to the mod_spdy server, which in turn proxies the requests (and responses) to (and from) WPR. We modify the local DNS resolver configuration file in Linux to resolve all domains to localhost so that the requests are issued to the apache proxy during the replay experiments and no requests are served over the Internet. In all our experiments, the client uses the default priorities set by the SPDY implementation of Chrome when issuing requests to the server.

In order to minimize the impact of browser variability on page-load times, we disable all extensions, background and sync activities in the browser using Chrome command line flags [40]. We set the browser cache and user profile directories to RAMDisk [41] to minimize the impact of disk read/write variability on page-load times. We also clear the RAM disk across runs to ensure clean-slate page-loads where all objects are fetched from the emulated CDN layer.

2.5 Results

We begin by evaluating the potential benefits of our placement and proactive refresh schemes, focusing on content type prioritization (§2.5.1). We next compare all our placement strategies in the absence of proactive refresh (§2.5.2), and all our proactive refresh schemes in the absence of priority-based placements (§2.5.3), with a view to understanding the potential benefits of prioritizing objects based on factors other than content type. Finally, §2.5.4 evaluates the benefits when such richer prioritization is used as part of both placement and proactive refresh.

To emulate heterogeneous latencies associated with CDN cache hierarchies, by default, we use the median TTFB observed across all objects fetched from each layer in our measurement study (Figure 2.3), but we present a sensitivity study to our latency settings in §2.5.4.

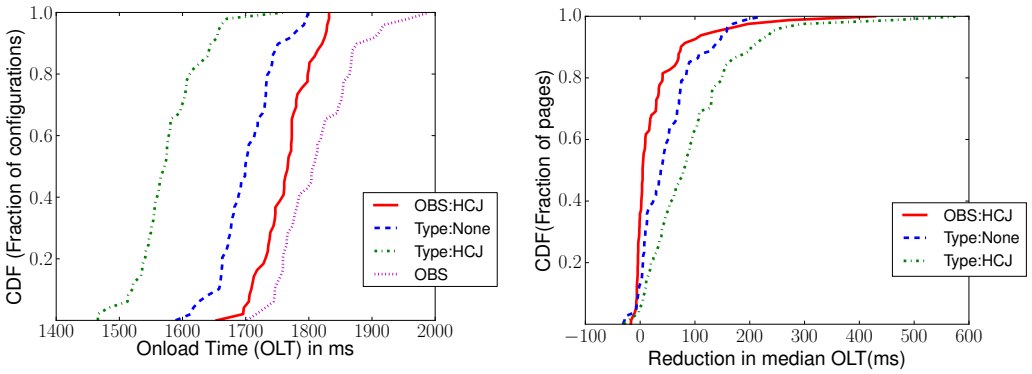
We compare the performance of the schemes with 83 real-world web pages (out of the 100 pages analyzed in §2.2). The remaining 17 pages either did not consistently trigger an Onload, or had 100% of its objects being served from the edge server(mem), in which case all placement schemes are equivalent. In comparisons where both placement and proactive refresh strategies vary, we use names such as *Type*:*HCJ* (indicating placement using the *Type* strategy and proactive refreshing of *HCJ* objects). When schemes compared vary in only in their placement (or refresh) strategy, we abbreviate by only using names of the placement (or refresh) schemes.

2.5.1 Latency benefits of prioritization

In this section, we evaluate the potential benefits of prioritized placement and proactive refresh, in isolation and in combination. We focus on schemes that primarily distinguish objects based on their content type (*Type* and *HCJ*).

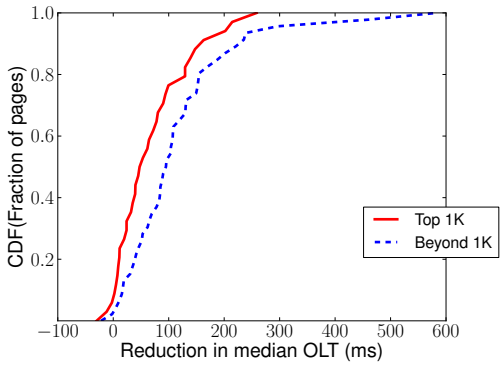
Figure 2.9(a) shows a CDF of the OLTs observed (across 50 placement configurations as described in §2.4) with *OBS*, *OBS*:*HCJ* indicating *OBS* placement with *HCJ* refresh strategy, *Type*:*None*, and *Type*:*HCJ* for a popular web-page *www.mercurynews.com* (Alexa Rank:1245). The breakdown of objects served from different layers in the real download (*OBS*) for this page was 54% at the edge server (34%mem, 20%disk), 26%remote, and 20%origin. The figure shows that both priority based placement and proactive refresh independently help reduce the OLT when compared to *OBS*. However, the combination of the two provides significantly higher benefits, reducing the OLT by more than 200ms.

Figure 2.9(b) shows the reduction in median OLT achieved by the three schemes relative to *OBS* for all pages. The figure shows that *Type*:*HCJ* provides significant reductions in OLT over *OBS*, despite the high EHRs for some of these pages. For instance, we see a median OLT reduction of more than 100ms for 40% of the pages and more than 200ms for 10% of the pages. Our results also show the importance of priority based placement only (*Type*:*None*), which reduces the median OLT by more than 50ms for 30% of the pages. Interestingly, we also find *OBS*:*HCJ* strategy providing significant latency reductions when



(a) OLT for *www.mercurynews.com* across schemes

(b) Median OLT reduction relative to *OBS* across all pages



(c) Median OLT reduction with *Type*:*HCJ* over *OBS* for Alexa Top1K and Beyond1K

Fig. 2.9. Latency benefits of prioritized placement and proactive refresh strategies in isolation and combination.

compared to *OBS* by avoiding refresh misses for some *HCJ* objects. For example, for 15% of the pages *OBS*:*HCJ* provides a latency reduction of more than 68ms. These correspond to pages where $> 17\%$ of all objects were *HCJ* and saw refresh misses. We also found similar trends for the reduction in $90\%ile$ latency for all the schemes, though the benefits were marginally higher.

To understand the impact of prioritization for pages with different popularities, in Figure 2.9(c) we show the OLT reduction with *Type*:*HCJ* over *OBS* split by the Alexa Top1K and Beyond1K classes (§ 2.2). The figure shows that though the benefits are more pronounced for the Beyond1K pages, prioritization provides significant reduction in Median

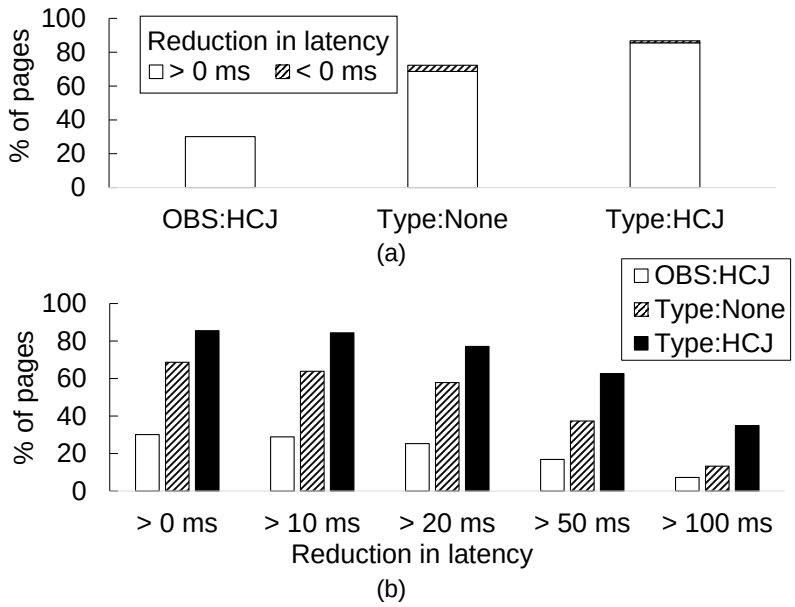
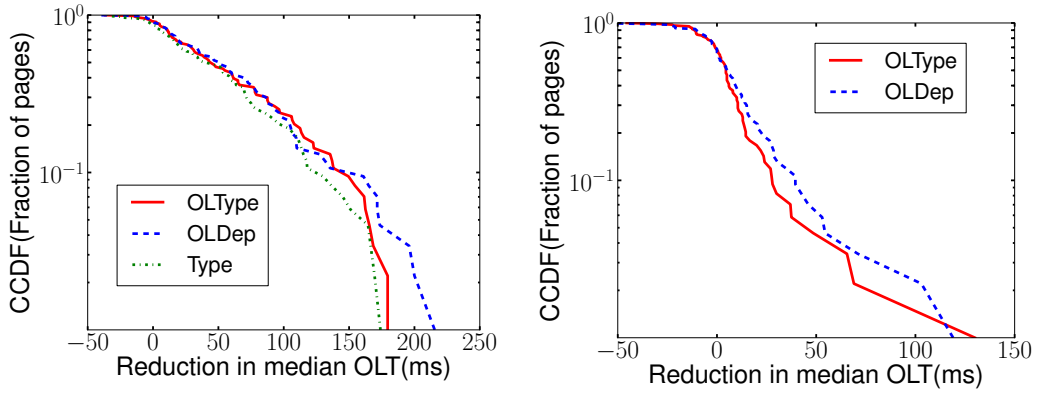


Fig. 2.10. Comparing the reduction in the median OLT for our schemes over *OBS*, validated with Mann-Whitney-Wilcoxon hypothesis testing [42] at a significance level of $p < 0.05$.

OLT even for the Top1K pages. For example, though *Type*:*HCJ* provides 93ms (225ms) reduction in median OLT for $50\%(10\%)$ of the Beyond1K pages, we see benefits of over 157ms for 10% of the Top1K pages also. Overall, our results clearly emphasize the importance of prioritization through better placement and proactive refresh strategies.

To ensure statistical significance of our results, we conducted a Mann-Whitney-Wilcoxon Test (MWW) [42, 43], a non-parametric hypothesis test to compare two populations. The null hypothesis is that the OLT distributions observed with the two schemes being compared are identical. We use a significance level of 0.05 (two-tailed p-value) and reject the null hypothesis when $p < 0.05$.

Figure 2.10 (a) shows a stacked bar graph, indicating the percentage of pages where the null hypothesis can be rejected and there is a reduction (increase) in median OLT. The remaining pages (not shown, adding upto 100% in each bar) correspond to pages which have similar performance across the schemes. The figure shows that the null hypothesis can be rejected for 86%, 69% and 30% of the pages respectively for the *Type*:*HCJ*, *Type*:None and *OBS*:*HCJ* schemes. The remaining pages typically correspond to pages in Figure 2.9(b)



(a) Reduction relative to *OBS*.

(b) Reduction relative to *Type*.

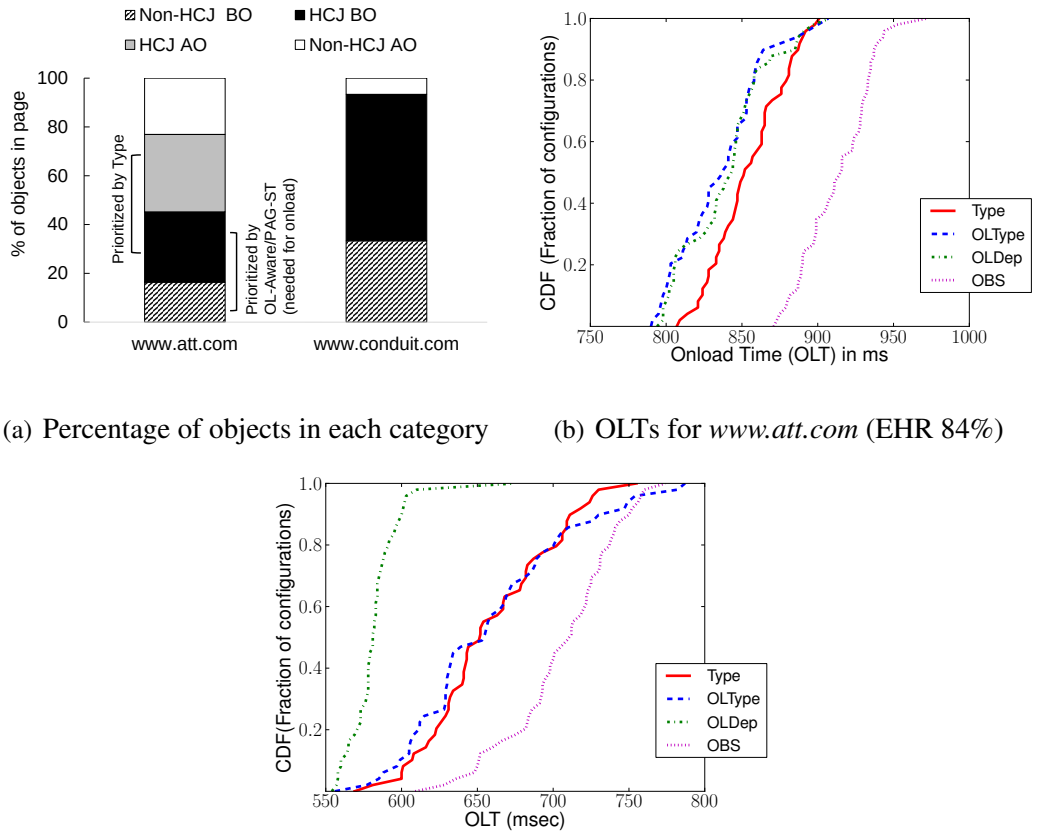
Fig. 2.11. CCDF of median OLT reduction with *Type*, *OLType* and *OLDep* for all pages with Y-Axis in log-scale. Proactive refresh is disabled for all schemes.

where the median OLT differences between the schemes is small (under 10 ms). Note that the *OBS*:*HCJ* performs comparably to *OBS* for many pages because these pages had relatively few staleness misses for *HCJ* objects. Figure 2.10 (b) corroborates this by showing the fraction of pages for which the null hypothesis can be rejected, and the reduction in median OLT exceeds a certain threshold for all schemes and a range of thresholds. For instance, the figure shows that for the *Type*:*HCJ* scheme, the reduction exceeds $50ms$ for 63% of the pages, and $100ms$ for 35% of the pages. Finally, we found exactly two pages where *Type* increases the OLT over *OBS*. Further investigation showed these were cases where a strictly type-based prioritization was inadequate, and where *OBS* was already serving important Non-*HCJ* objects from the edge as discussed in §2.5.1.

2.5.2 Comparing placement strategies

We next evaluate the benefits of *OLType* and *OLDep* which consider factors besides content type in placement decisions. Since our focus is on placement schemes, our comparisons are done without proactive refresh.

Figure 2.11(a) presents a CCDF of the median OLT reduction relative to *OBS* for all the placement schemes, and all the pages in our experiment set. While all schemes show



(a) Percentage of objects in each category

(b) OLTs for *www.att.com* (EHR 84%)

(c) OLTs for *www.conduit.com* (EHR 20%)

Fig. 2.12. Impact of page composition on the relative performance of schemes

significant reductions compared to *OBS*, *OLType* and *OLDep* provide only slightly higher benefits than *Type*. The benefits are marginal for most pages, however somewhat more significant at the tail. Note that the CCDF is shown with Y-Axis in log scale since the schemes show more prominent difference in the tail. Figure 2.11(b) shows the median OLT reduction of *OLType* and *OLDep* relative to *Type*. Across all pages, *OLDep* achieves a median OLT reduction higher than 35ms for 12% of the pages, while for 6% of the tail pages both schemes achieve median OLT reduction higher than 50ms relative to *Type*. Though we find higher benefits for the Beyond1K pages, we also see latency savings of more than 35ms for 14% of the Top1K pages as well, with as much as 71ms for the tail page. We have also verified the statistical significance of our results using the MWW test.

To better understand these results, and when different schemes are most helpful, we analyze the page composition into objects in 4 categories as follows – (i)*HCJ* and Non-*HCJ* objects; and (ii) required for Onload (*BO*) or not (*AO*). Figure 2.12(a) shows the composition of objects in these 4 categories for two example pages where *OLDep* and *OLType* show benefits over *Type*. In general, the benefits with these schemes depend both on the EHR, and the composition of the page, among other factors.

When do *OLType* and *OLDep* perform better than *Type*? Both *OLType* and *OLDep* prioritize objects before Onload. For *www.att.com*, the EHR is sufficiently high that all *BO* objects may be placed in the edge, hence both *OLType* and *OLDep* prioritize all *BO* objects to the edge. However, *Type* prioritizes *HCJ* objects (agnostic of whether they were required for Onload) over Non-*HCJ BO* objects, and is unable to accommodate all Non-*HCJ BO* objects at the edge. Indeed, Figure 2.12(b) confirms that for this page, *OLType* and *OLDep* perform better than *Type*. Note that the two schemes themselves perform comparably - this makes sense due to the high EHR both schemes are able to place all *BO* objects in the edge.

Interestingly, we observed many other pages where *Type* performs comparably to *OLType* and *OLDep* even though it is not able to place all Non-*HCJ BO* objects in the edge. On further analysis, we found that the Non-*HCJ* objects were leaves in the dependency graph for these pages – consequently, the performance with *Type* was relatively unaffected even though these objects were not placed in the edge. In contrast, for pages like *www.att.com* some of the Non-*HCJ* objects were internal nodes in the dependency graph, in the sense that a lot of object fetches were dependent on these objects (delaying the internal Non-*HCJ* objects delays a lot of other objects being fetched). For example, in *www.att.com* some of the Non-*HCJ* internal nodes were *sprited images* and the execution of *JS* code waits on these images. As a result, delaying these images delays all the objects to be fetched after executing the *JS* code. Thus careful placement of the internal Non-*HCJ* objects was particularly important to reduce the page latencies.

We now analyze the statistical significance of the above results using Mann-Whitney-Wilcoxon Test (MWW) [42, 43] as described earlier. Figure 2.13 (a) shows the stacked bar graph indicating the percentage of pages where the null hypothesis is rejected, and

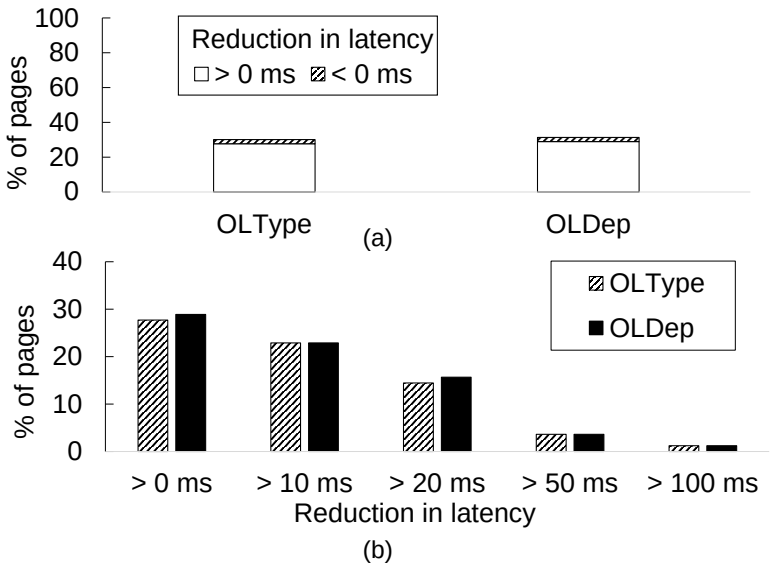


Fig. 2.13. Comparing the reduction in the median OLT with *OLType* and *OLDep* over *Type*, validated with Mann-Whitney-Wilcoxon hypothesis testing [42] at a significance level of $p < 0.05$. Proactive refreshing is disabled for all schemes in this experiment.

there is a reduction (increase) in median OLT with *OLType* and *OLDep* relative to *Type*. The figure shows that the null hypothesis can be rejected for 29% and 28% of the pages respectively for the *OLDep* and *OLType* schemes. Figure 2.13 (b) shows that the schemes reduce median OLT by $> 20ms$ for about 15% of the pages, and median OLT by $> 50ms$ for 4% of pages. This is consistent with our observations in Figure 2.11(b) that *OLType* and *OLDep* primarily provide benefits over *Type* at the tail.

When does *OLDep* perform better than *OLType*? For *www.conduit.com*, 93% of the objects are required before Onload with more *HCJ BO* objects (60%) than can fit in the edge. While both *OLDep* and *OLType* prioritize *HCJ BO* objects, *OLDep* makes finer-grained distinctions, and prioritizes objects at the highest levels of the dependency graph, which are more critical for latency. Indeed, Figure 2.12(c) confirms that *OLDep* performs better than *OLType* for this page. *OLType* and *Type* perform similar - this is because there are only a few *HCJ* AO objects and hence *OLType* and *Type* are choosing from relatively the same set of objects to place in the edge. More generally, *OLDep* also provides benefits

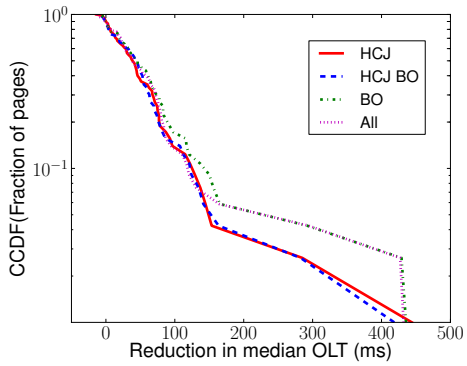
over *OLType* for pages where all *HCJ BO* but not all *BO* objects fit in the edge, and some Non-*HCJ BO* objects are internal nodes in the dependency graph.

Finally, we found rare cases where *OLDep* performs worse than *Type*, when objects deeper in the dependency graph have more impact on the OLT than the objects closer to the root in the graph. For example, in *www.comcast.com*, a *JS* object (appearing as a leaf in the dependency graph) had a larger impact on the OLT owing to its higher execution times than other internal Non-*HCJ* objects, while *Type* placed all *HCJ* objects at the edge. While prioritizing objects occurring consistently on the critical path across runs (and clients) may help, determining such objects is not trivial, and we did not explore such a technique in depth given the small number of occurrences.

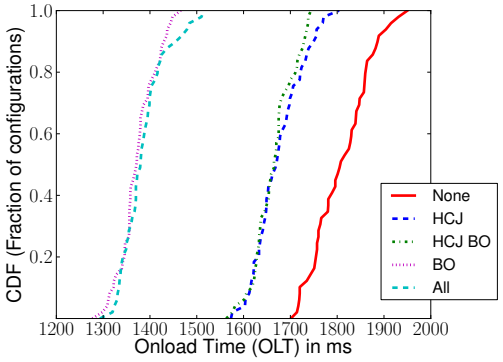
2.5.3 Comparing proactive refresh strategies

In this section, we fix the placement scheme as *OBS*, and compare the performance of various proactive refresh strategies described in §2.4. Since proactive refresh strategies differ only in their handling of refresh misses, we confine this study to only those pages(61 pages) that saw at least one stale access in the real page-load. Figure 2.14(a) shows that all strategies give significant latency reductions relative to *OBS*. For the vast majority of pages, the schemes perform similarly, though there are a small number of pages where *BO* and *All* perform better. Interestingly, we also find that the *HCJ BO* scheme performs similar to *HCJ* while incurring lesser bandwidth costs.

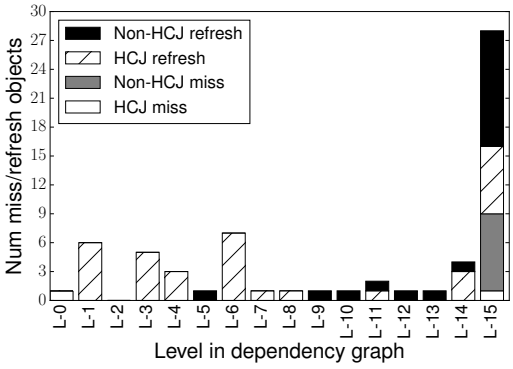
When does *BO* perform better than *HCJ*? We illustrate this using one of the pages *www.mercurynews.com*, where *BO* performs better than *HCJ*. Figure 2.14(b) shows the CDF of OLTs observed with each of the proactive refresh strategies. Clearly, all refresh schemes perform better than None, while *BO* performs even better than *HCJ* (and *HCJ BO*). We note that the page observed multiple refresh misses for Non-*HCJ BO* objects in the higher levels (L 9-14) of the dependency graph as shown in Figure 2.14(c). These objects have further dependent objects (in L15), and impacts the critical path of the page-



(a) CCDF of the median reduction in OLT relative to *OBS* with Y-Axis in log scale.



(b) OLTs for *www.mercurynews.com* with different refresh strategies.



(c) Composition of misses at each level of the dependency graph for *www.mercurynews.com*

Fig. 2.14. Latency reduction with proactive refresh. All schemes use the *OBS* placement.

load. Therefore, the *BO* scheme, which proactively refreshes all objects needed for Onload (including Non-*HCJ BO* objects), provides significant latency reductions.

Overall our results show that (i) *HCJ* suffices in most cases, though *BO* can provide further latency reduction for some pages; (ii) *BO* itself gives all the benefits of *ALL* with lower bandwidth costs; and (iii) *HCJ BO* provides comparable benefits to *HCJ* with lower bandwidth costs.

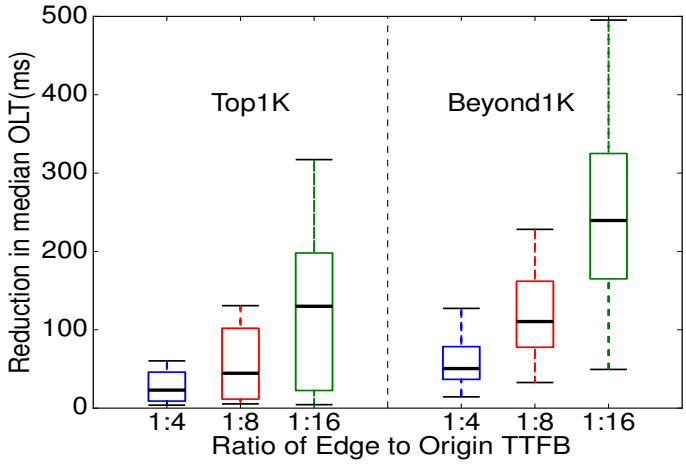


Fig. 2.15. Median OLT reduction with *OLDep*:*BO* over *OBS* with three different edge to origin TTFB ratios split by Alexa Top1K and Beyond1K. The boxes show the 25th, 50th and 75th and the whiskers showing the 10th and 90th percentile pages.

2.5.4 Sensitivity to origin TTFB

In this section, we achieve two goals. First, we evaluate the benefits achievable by considering factors other than content type in both placement and proactive refresh strategies together. Second, we study the sensitivity of the observed latency reductions with our schemes to heterogeneity in TTFBs for fetching objects. We focus on the TTFB to origin servers since they show the highest variability and have the highest impact on the page-load latency. Therefore, we conduct the sensitivity study by varying the TTFB to the highest layer retaining the same values for the other cache layers (used in §2.5). We compare *OLDep*:*BO* with *OBS* for three different ratios of *CDN* edge to origin server TTFBs viz. $1:4$, $1:8$ and $1:16$ (rounded-off) representing the $25th$, median and $75th$ percentiles respectively from the real downloads (see §2.2). Note that the ratio used in all our prior experiments (§2.5) is $1:8$.

Figure 2.15 shows the reduction in median OLT with *OLDep*:*BO* compared to *OBS* split by pages in Alexa Top1K and Beyond1K classes. Clearly, *OLDep*:*BO* which combines both placement and proactive refresh provides significant benefits – with $1:8$, the median

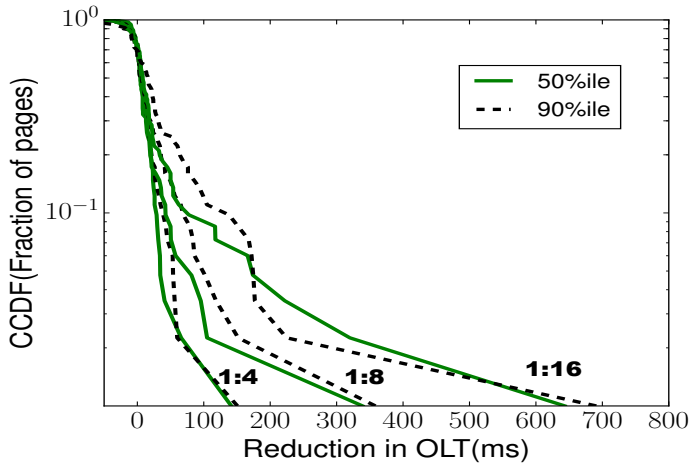


Fig. 2.16. CCDF of the reduction in $50\%ile$ and $90\%ile$ OLTs for all pages with *OLDep*:*BO* over *Type*:*HCJ* for the three edge to origin TTFB ratios

latency reduction is $> 100ms$ for 30% of the Top 1K pages and $> 100ms$ for 59% of the Beyond 1K pages. As expected, the benefits with prioritization increases with larger origin TTFB. Interestingly, the benefits are higher for both the Top1K and Beyond1K pages when the origin TTFB is high (ratio $1:16$). For instance, the median OLT reduction for 50% of the Beyond1K pages is about $2X$ higher with $1:16$ than $1:8$, while for 50% of the Top1K pages we see almost $3X$ higher reduction with $1:16$ than $1:8$. Though not shown, the reduction in median OLT with *OLDep*:*BO* was as much as $> 1s$ for $1:16$ (and $586ms$ for $1:8$) at the tail.

We now study the relative benefits of *OLDep*:*BO* over *Type*:*HCJ*, with the various edge to origin ratios. Figure 2.16 shows a CCDF of the reduction in median and $90\%ile$ OLTs with *OLDep*:*BO* over *Type*:*HCJ* for all pages. The figure shows that the latency benefits of *OLDep*:*BO* over *Type*:*HCJ* increases with higher ratios. The trends hold for reductions in both the median and $90\%ile$ OLTs, with reductions in the median OLTs as high as $700ms$ for *www.mercurynews.com* which had many Non-*HCJ* refresh misses. Overall our results show that *OLDep*:*BO* gives higher benefits over *Type*:*HCJ* for pages at the tail, especially when the TTFB to origin is high.

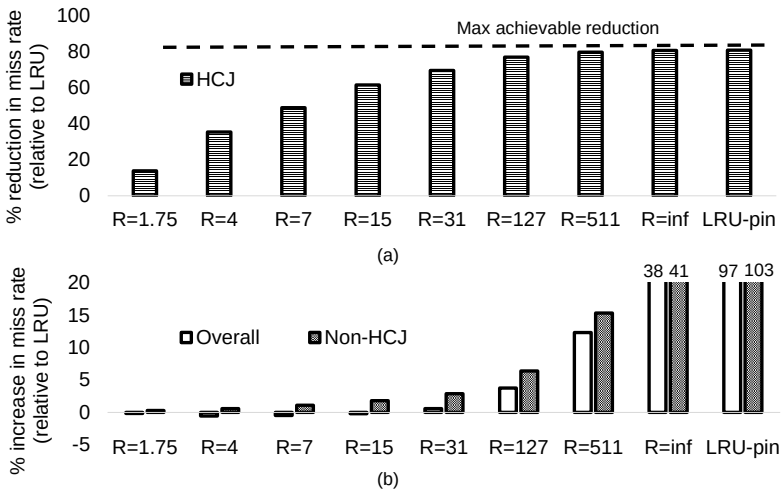


Fig. 2.17. Miss rates of priority based caching schemes when compared to LRU(Size), and LRU-Pin.

2.6 Trace-driven evaluation

In this section, we conduct trace-driven simulations for evaluating the feasibility of priority-based caching and proactive refresh in CDNs. The request traces for this study were obtained from the edge cluster of a real CDN deployment, which serves a wide class of web traffic, and consists of 162 million requests for about 13.5 million distinct objects. The week long trace is non-sampled and consists of all client requests observed at each of the 18 servers in the edge cluster. For all simulations in this section, we set the cache capacity to those seen in the real deployment. Since the page structure (dependencies) and *Onload* information is not deducible from the trace, we use content-type based prioritization, and focus on miss-rate reduction (and bandwidth overhead) for all experiments in this section.

2.6.1 Feasibility of priority based caching policy

We first show the feasibility of our approach in reducing the miss rate for the critical objects without significantly affecting the overall hit rates of the caches. We emulate the cache using our week-long trace for two caching algorithms - our priority-based caching described in §2.3.2, and (ii) LRU(size) - LRU with a size threshold, that is commonly

employed by CDNs today. We evaluate our algorithm by varying the relative importance of the *HCJ* and Non-*HCJ* objects, which is captured by the parameter R, the ratio of the priority of *HCJ* objects to the priority of Non-*HCJ* objects. We also compare our algorithm with a variant of LRU which preferentially pins the *HCJ* objects to the cache and ensures that they are never evicted by a Non-*HCJ* object. However, *HCJ* objects may evict Non-*HCJ* objects and other *HCJ* objects similar to LRU. We use the same cache size and object-size threshold used by the LRU across all the schemes.

Figure 2.17(a) shows the reduction in miss rates for the *HCJ* objects for the different schemes relative to the miss rates observed with the LRU(size). The horizontal line at the top of the graph shows the maximum achievable reduction in miss rates, where the rest of the misses are compulsory misses in our trace. Figure 2.17(b) shows the corresponding increase in the overall miss rate as well as in the miss rate for Non-*HCJ* objects. From the figure, we see that as R increases, the reduction in miss rate for *HCJ* objects ramps up quickly for smaller R (flattens out at higher R), while the increase in miss rate for Non-*HCJ* objects increases gradually for smaller R and more rapidly for higher R. This shows the opportunity for the CDN to tune R, such that, it reduces the miss rate for *HCJ* objects without increasing the overall miss rate of the cache. In our trace, this occurs around $R = 7$ and $R = 15$, which reduces the miss rate for *HCJ* objects by 49% and 61% respectively, without significant impact on the overall miss rate. We also see that while LRU-pin performs the best for *HCJ* objects, it drastically affects the overall miss rates.

Figure 2.18(a) shows the reduction in byte miss rates (the ratio of number of bytes for objects that were missed from the cache over the total number of bytes requested) for the *HCJ* objects for the different schemes relative to the byte miss rates observed with the LRU(size). Figure 2.18(b) shows the corresponding increase in the overall byte miss rate as well as in the byte miss rate for Non-*HCJ* objects. From the figures, we see that our schemes are able to reduce byte miss rates of *HCJ* objects by 52% and 65%, with modest increases of 0.7% and 1.8% in the overall byte miss rates for $R = 7$ and $R = 15$ respectively. We note that increase in overall byte miss rates are higher than the increase in overall cache miss rates (Figure 2.17) for any given value of R. This is because Non-*HCJ*

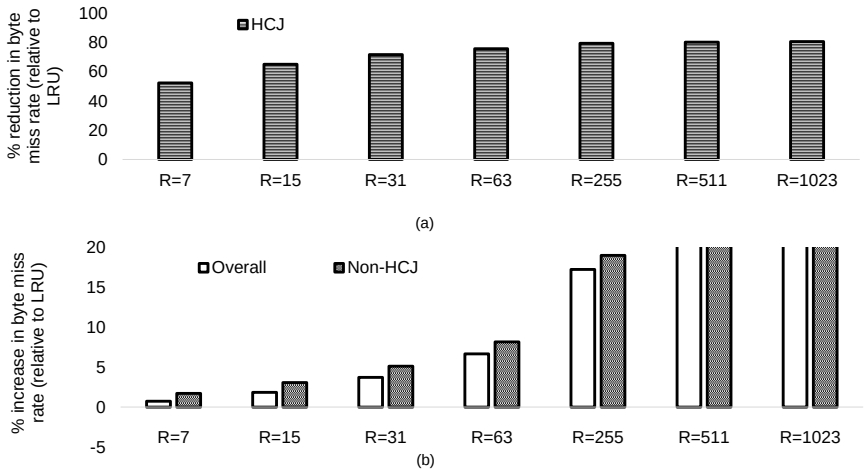


Fig. 2.18. Byte miss rates of priority based caching schemes when compared to LRU(Size)

objects typically tend to be larger in size compared to *HCJ* objects, which results in higher number of bytes missed when *HCJ* objects are prioritized over Non-*HCJ* objects.

Overall, our results show that our priority based caching scheme is able to significantly reduce the miss rates (and byte miss rates) for *HCJ* objects, while incurring only a modest increase in the overall miss rates (and byte miss rates).

2.6.2 Bandwidth impact of proactive refresh schemes

We now show the benefits of prioritization in reducing the additional bandwidth costs associated with proactive refreshing. Since our traces do not have *Onload* information for objects, we focus our evaluation in this section on the *HCJ* and *All* proactive refresh schemes. We augment our priority-based caching algorithm with proactive refreshing as described in §2.3.3 and emulate the cache with our week long traces. In all our experiments, we set the threshold $T = 2$ (for just-in-time refreshes), but vary the parameter K for the *HCJ* objects to illustrate the bandwidth-cost and performance trade-off with conservative and aggressive proactive refreshing.

Figure 2.19 compares the percentage reduction in stale accesses for *HCJ* objects, and the corresponding increase in bandwidth incurred with both the schemes. Note that the bandwidth costs estimated here are an upper bound since entire objects need not be fetched

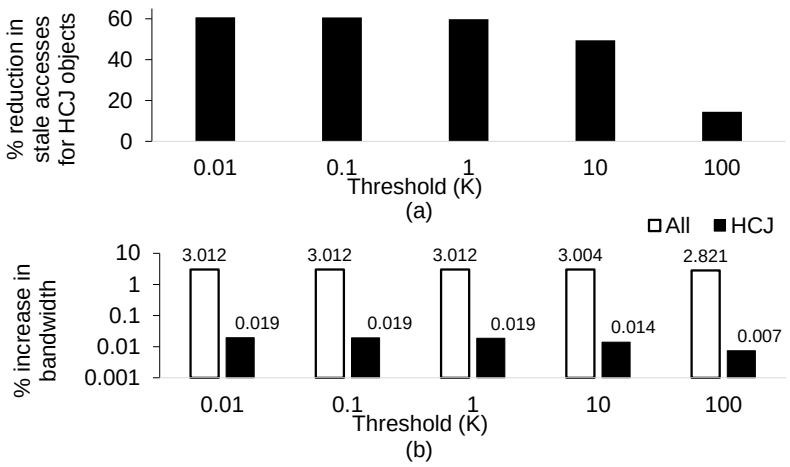


Fig. 2.19. Impact of the *HCJ* and All proactive refresh strategies. Note that both schemes reduce stale accesses for HCJ objects by identical amounts.

again if they are not modified at the origin, but that information is not available to us in the trace. The figure shows that both All and *HCJ* schemes reduce stale accesses for *HCJ* objects by 60%, while incurring an overall bandwidth increase of 3% and 0.02% respectively. Note that a smaller K (aggressive refreshing) results in fewer stale accesses, while a larger K (conservative refreshing) lowers the bandwidth costs of proactive refreshing. Overall, our results highlight the opportunity for priority-based proactive refresh in significantly reducing staleness for *HCJ* objects, while incurring only modest bandwidth penalties.

2.7 Related work

While SPDY [15] allows resource prioritization, it supports only priority based processing (and transmission) of objects from the server to a client [44]. Recent work [30] looks at re-prioritizing delivery of objects in a web page when they are pushed from a server to a mobile client. In contrast, our focus is on an orthogonal problem – enabling priority awareness within the CDN infrastructure. All our experiments including the *OBS* baseline are run with SPDY enabled, and our benefits are complementary to SPDY. Recent research has shown that SPDY is not always beneficial [38,45]. Our proposals in this chapter do not rely on SPDY - incorporating priority awareness in CDNs has benefits even with HTTP.

Our work builds on the rich literature on caching algorithms for web caches, proxy caches and CDNs (e.g., [37, 46–51]). We adapt the well known Greedy-Dual-Size algorithm [37] which considers how to balance locality of access patterns with object size and the variable costs associated with fetching objects on a miss, given some network paths could be more expensive than others. Others have extended the algorithm to more explicitly bias it towards more popular objects [46, 47]). In contrast to all these works, our focus is on determining the importance of an object within a page for lowering page latency, and balancing object popularity and priority.

Prefetching to reduce web latencies has been extensively studied since the earliest days of the web (e.g., [52]). Many of the early works focused on client-side prefetching (e.g., [52]) in which clients initiate prefetching guided by predictions on which files are likely to be accessed soon (e.g., based on models that indicate which hyper-links a client is likely to click when on a given page [52]). Others [53–56] have investigated prefetching in CDNs and proxy servers by using global access patterns to identify which objects should be proactively replicated to caches. While we leverage these techniques, we consider the more limited goal of avoiding refresh misses on objects already in the cache by proactively refreshing them. Further, we seek to proactively refresh objects that are more important for reducing page latencies, given refresh misses are a key component of overall miss rates for popular pages.

Researchers have explored how objects must be placed in a hierarchical caching system [57–60] so that the average latencies are minimized given constraints on cache capacities [58] or bandwidth [60]. [59] propose mechanisms to improve end user response times by tracking the data location and minimizing the number of hops on hits and misses within the CDN hierarchy. In contrast, our focus is on placement of objects taking priority into account - specifically, objects that are not as popular may be placed lower in the hierarchy since they may be critical for page-load.

2.8 Conclusions

In this chapter, we have made two contributions. First, we have shown that there is significant potential to reduce web-page latencies through page-structure-aware strategies for placing objects in CDN cache hierarchies. Second, we have presented several strategies to this end which differ in their degree of page-awareness, and conducted a detailed evaluation study of their benefits. Our evaluations with more than 80 real-world web pages show that for popular pages, more than 30% of pages see median OLT reductions higher than $100ms$, while for less popular pages, the median OLT reduction is more than $100ms$ for more than 59% of the pages, with some pages showing latency reductions as high as $500ms$. Both placement and proactive refreshing are important in achieving the benefits, though each can help in isolation. For the vast majority of pages, the *Type*:*HCJ* scheme provides most of the benefit. However, *OLDep*:*BO* can provide significant additional benefits for some pages, especially in lower hit rate regimes, when there are Non-*HCJ* internal nodes in the dependency graph, and when the penalty of going to the origin is higher. Finally, using trace driven simulations, we show the feasibility of priority-based caching approach to reduce miss rates of page-critical objects in CDNs by 60% with modest increases (less than 2%) in overall byte miss rates. We also highlight the opportunity of minimizing staleness related misses for objects critical for latency by as much as 60% while incurring additional bandwidth costs of less than 0.02%.

3. PERFORMANCE SENSITIVE REPLICATION IN GEO-DISTRIBUTED DATASTORES

3.1 Introduction

Interactive web applications face stringent requirements on latency, and availability. Service level agreements (SLAs) often require bounds on the $90th$ (and higher) percentile latencies [7], which must be met while scaling to hundreds of thousands of geographically dispersed users. Applications require 5 9's of availability or higher, and must often be operational despite downtime of an entire DC. Failures of entire DCs may occur due to planned maintenance (e.g. upgrade of power, cooling and network systems), and unplanned failure (e.g. power outages, and natural disasters) [7, 8, 14, 61] (Figure 3.1). Application latencies and downtime directly impact business revenue [3].

In response to these challenges, a number of systems that replicate data across geographically distributed data-centers (DCs) have emerged in recent years [7–14]. An important requirement on these systems is the need to support consistent updates on distributed replicas, and ensure both low write and read latencies. This is necessitated given datastores target interactive web applications that involve reads and writes by geographically distributed users (e.g. Facebook timelines, collaborative editing). Consequently, a distinguishing aspect of cloud datastores is the use of algorithms (e.g., quorum protocols [7, 9], Paxos [8, 13, 14]) to maintain consistency across distributed replicas.

Achieving low read and write latencies with cloud datastores while meeting the consistency requirements is a challenge. Meeting these goals requires developers to carefully choose the number of replicas maintained, which DCs contain what data, as well as the underlying consistency parameters (e.g., quorum sizes in a quorum based system). Replica placement techniques in traditional Content Delivery Networks (CDNs) (e.g., [62]) do not apply because consistency has to be maintained with distributed writes while maintaining

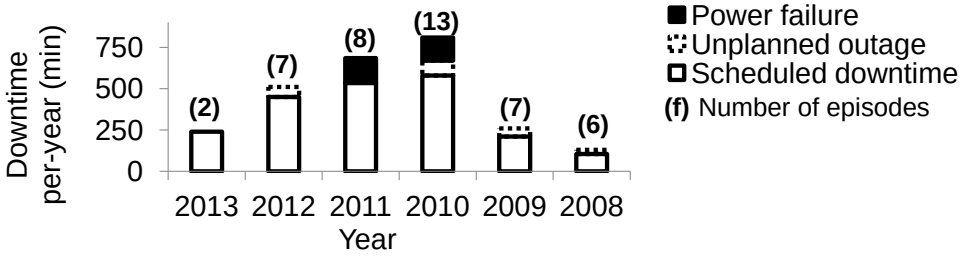


Fig. 3.1. Downtime and number of failure episodes (aggregated per year) of the Google App Engine data store obtained from [65].

low latencies. Tailoring cloud datastores to application workloads is especially challenging given the scale of applications (potentially hundreds of thousands of data items), workload diversity across individual data items (e.g. celebrities and normal users in Twitter have very different workload patterns), and workload dynamics (e.g. due to user mobility, changes in social graph etc.)

The problem of customizing replication policies in cloud datastores to application workloads has received limited systematic attention. Some datastores like [7, 9] are based on consistent hashing, which limits their flexibility in placing replicas. Other datastores like [12, 63] assume that all data is replicated everywhere, which may be prohibitively expensive for large applications. While a few datastores can support flexible replication policies [8, 64], they require these replication decisions to be configured manually which is a daunting task.

In this chapter, we present frameworks that can automatically determine how best to customize the replication configuration of geo-distributed datastores to meet desired application objectives. We focus our work on systems such as Amazon's Dynamo [7], and Cassandra [9] that employ quorum protocols. We focus on quorum-based systems given their wide usage in production [7, 9], the rich body of theoretical work they are based on [23–26], and given the availability of an open-source quorum system [9]. However, we believe our frameworks can be extended to other classes of cloud storage systems as well.

We focus on optimization frameworks to obtain insights into the fundamental limits on application latency achievable for a given workload while meeting the consistency require-

ment. Our models are distinguished from quorum protocols in the theoretical distributed systems community [23–26], in that we focus on new aspects that arise in the context of geo-distributed cloud datastores. In particular, our models consider the impact of DC failures on datastore latency, and guide designers towards replica placements that ensure good latencies even under failures. Further, we optimize latency percentiles, allow different priorities on read and write traffic, and focus on realistic application workloads in wide-area settings.

We validate our models using traces of three popular applications: *Twitter*, *Wikipedia* and *Gowalla*, and through experiments with a multi-region Cassandra cluster [9] spanning all 8 EC2 geographic regions. While latencies with Cassandra vary widely across different replication configurations, our framework generates configurations which perform very close to predicted optimal on our multi-region EC2 setup. Further, our schemes that explicitly optimize latency under failure are able to out-perform failure-agnostic schemes by as much as 55% under the failure of a DC while incurring only modest penalties under normal operation. Our results also show the importance of choosing configurations differently across data items of a single application given the heterogeneity in workloads. For instance, our *Twitter* trace required 1985 distinct replica configurations across all items, with optimal configurations for some items often performing poorly for other items. Overall the results confirm the importance and effectiveness of our frameworks in customizing geo-distributed datastores to meet the unique requirements of cloud applications.

3.2 Replication in geo-distributed datastores

A commonly used scheme for geo-replicating data is to use a master-slave system, with master and slave replicas located in different DCs, and data asynchronously copied to the slave [61, 66]. However, slaves may not be completely synchronized with the master when a failure occurs. The system might serve stale data during the failure, and application-level reconciliation may be required once the master recovers [14, 61]. On the other hand, synchronized master-slave systems ensure consistency but face higher write latencies.

To address these limitations with master-slave systems, many geo-distributed cloud storage systems [8, 10, 12, 13, 13, 14, 63, 64, 67, 68] have been developed in the recent years. A distinguishing aspect of cloud datastores is the use of algorithms to maintain consistency across distributed replicas, though they differ in their consistency semantics and algorithms used. Systems like Spanner [8] provide database-like transaction support while other systems like EIGER [12] and COPS [63] offer weaker guarantees, primarily with the goal of achieving lower latency.

Quorum-based datastores: Quorum protocols have been extensively used in the distributed systems community for managing replicated data [23]. Under quorum replication, the datastore writes a data item by sending it to a set of replicas (called a write quorum) and reads a data item by fetching it from a possibly different set of replicas (called a read quorum). While classical quorum protocols [23] guarantee strong consistency, many geo-distributed datastores such as Dynamo [7], and Cassandra [9] employ adapted versions of the quorum protocol, and sacrifice stronger consistency for greater availability [7]. In these systems, reads (or writes) are sent to all replicas, and the read (or write) is deemed successful if acknowledgments are received from a quorum. In case the replicas do not agree on the value of the item on a read, typically, the most recent value is returned to the user [7, 9], and a background process is used to propagate this value to other replicas. Replication in these systems can be configured so as to satisfy the strict quorum property:

$$R + W > N \tag{3.1}$$

where N is the number of replicas, R and W are the read and write quorum sizes respectively. This ensures that any read and write quorum of a data item intersect. Configuring replication with the strict quorum property in Cassandra and Dynamo guarantees read-your-writes consistency [69]. Further, any read to a data item sees no version older than the last complete successful write for that item (though it may see any later write that is unsuccessful or is partially complete). Finally, note that Dynamo and Cassandra can be explicitly configured with weaker quorum requirements leading to even weaker consistency guarantees [70].

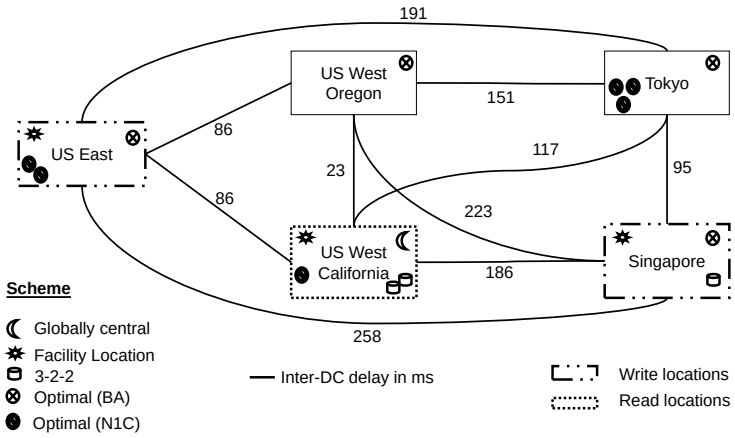


Fig. 3.2. Replica configuration across schemes for a set of *Twitter* data items. Reads/writes are mapped to the nearest Amazon EC2 DC. While all 8 EC2 regions (and 21 Availability Zones) were used to compute the configurations for all schemes, only DCs that appear in at least one solution are shown. For clarity, placement with *N-1C* is not shown.

3.3 Motivating example

In using cloud storage systems, application developers must judiciously choose several parameters such as the number of replicas (N), their location, and read(R) and write(W) quorum sizes. In this section, we illustrate the complexity in the problem using a real example, and highlight the need for a systematic framework to guide these choices. The example is from a real *Twitter* trace (Section 3.7.1), and represents a set of users in the West Coast who seldom tweet but actively follow friends in Asia and the East Coast.

Figure 3.2 depicts the placement with multiple replica configuration schemes. The DC locations and inter-DC delays were based on Amazon EC2, and we required that at most one replica may be placed in any EC2 Availability Zone (AZ). Table 3.1 summarizes the performance of the schemes. Our primary performance metric is the quorum latency, which for the purpose of this example is the maximum of the read and write latency from any DC. The read (write) latency in a quorum datastore is the time to get responses from as many replicas as the read (write) quorum size. Our frameworks are more general and can generate configurations optimized for different priorities on read and write latencies. We discuss possible schemes:

User centric: This scheme is representative of traditional CDN approaches and aims to place replicas as close to users as possible with no regard to quorum requirements. In the limit, replicas are placed at all DCs from which accesses to the data item arrive (USW-1, APS-1, and USE-1 in our example). It may be verified that for this choice of replicas, the best quorum latency achievable is 186 msec, obtained with read and write quorum sizes of 2. Note that this placement would also be generated by the classical *Facility Location* problem when facilities may be opened with zero cost.

Globally central: This scheme seeks to place replicas at a DC which is centrally located with respect to all users by minimizing the maximum latency from all DCs with read/write requests. In our example, this scheme places a replica at USW-1. Note that for resiliency, replicas could be placed in additional availability zones of the US West region, but the quorum latency would still remain 186 msec.

***Basic Availability*:** This is our model (Section 3.6), which optimizes quorum latencies under normal conditions (all DCs are operational) while ensuring the system is functional under the failure of a single DC. This scheme chooses 4 replicas, one at each of the DCs, as shown in Figure 3.2, with $R = 3$ and $W = 2$. This configuration has a quorum latency of 117msec - a gain of 69 msec over other schemes. Intuitively, the benefit comes from our scheme's ability to exploit the asymmetry in read and write locations, increasing the number of replicas and appropriately tuning the quorum sizes.

***N-1 Contingency*:** While the Basic Availability scheme guarantees operations under any single DC failure, latencies could be poor. For e.g., on the failure of APN-1, the write latency from USE-1 increases to 258msec. Our *N-1 Contingency* scheme (Section 3.6) suggests configurations that guarantee optimal performance even under the failure of an entire DC. In our example, the *N-1 Contingency* scheme configures 6 replicas (3 in APN-1, 2 in USE-1 and 1 in USW-1) with $R = 5$ and $W = 2$. This configuration ensures the quorum latency remains 117 msec even under any single DC failure. Note that this configuration has the same performance as the *BA* scheme under normal conditions as well.

Overall, these results indicate the need and benefits for a systematic approach to configure replication policies in cloud datastores. Further, while our example only considers a

Table 3.1
Comparing performance of schemes

Scheme	Quorum latency (msec)		N,R,W
	Normal	Failure	
Globally central	186	186	$3, 2, 2$
User centric	186	258	$3, 2, 2$
Basic Availability	117	191	$4, 3, 2$
N-1 Contingency	117	117	$6, 5, 2$

subset of items, applications may contain tens of thousands of groups of items with different workload characteristics. Manually making decisions at this scale is not feasible.

3.4 System Overview

Figure 3.3 shows the overview of our system. The datastore is deployed in multiple geographically distributed DCs (or availability zones), with each data item replicated in a subset of these DCs. Since our focus is on geo-replication, we consider scenarios where each DC hosts exactly one replica of each item, though our work may be easily extended to allow multiple replicas.

Applications consist of front-end application servers and back-end storage servers. To read/write data items, an application server contacts a ”coordinator” node in the storage layer which is typically co-located in the same DC. The coordinator determines where the item is replicated (e.g. using consistent hashing or explicit directories), fetches/updates the item using a quorum protocol, and responds to the application server.

We use the term “requests” to denote read/write accesses from application servers to the storage service, and we consider the request to “originate” from the DC where the application server is located. We model “request latency” as the time taken from when an application server issues a read/write request to when it gets a response from the storage service. It is possible that the application issues a single API call to the storage service that

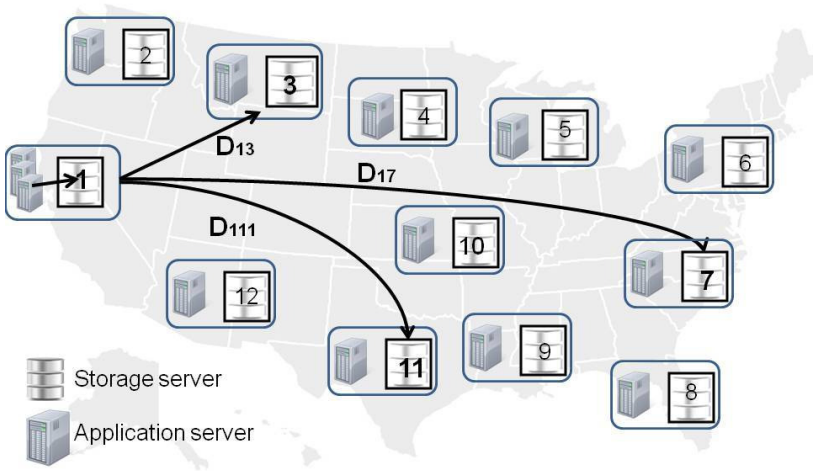


Fig. 3.3. System overview

accesses multiple data items. (e.g. a multi-get call in Cassandra with multiple keys). We treat such a call as separate requests to each data item.

Users are mapped to application servers in DCs nearest to them through traditional DNS redirection mechanisms [71]. While application servers typically contact a coordinator in the same DC, a coordinator in a nearby DC may be contacted if a DC level storage service failure occurs (Section 3.6).

3.5 Latency optimized replication

In this section, we present a model that can help application developers optimize the latency seen by their applications with a quorum-based datastore. Our overall goal is to determine the replication parameters for each group of related data items. These include (i) the number, and location of DCs in which the data items must be replicated; and (ii) the read and write quorum sizes.

We expect our formulations to be applied over classes of items that see similar access patterns. For e.g., while access patterns for *Wikipedia* vary across languages, documents within a language see accesses from the same geographic regions, and could be grouped together. Systems like Spanner [8] require applications to bucket items into “directories”,

Table 3.2
Parameters and inputs to the model

Term	Meaning
M	Number of available DCs.
D_{ij}	Access latency between DCs i and j.
C_i	Cost of outgoing traffic at DC i.
N_i^l	Number of reads/writes from DC i.
T^l	Read/Write Latency Threshold.
p^l	Fraction of requests to be satisfied within T^l.
x_i	Whether DC i hosts a replica.
q_{ij}^l	Whether i's requests use replica in j to meet quorum.
Q^l	Quorum size.
Y_i^l	Whether requests from i are satisfied within T^l.
Y_{ik}^l	Whether requests from i are satisfied within T^l
	on failure of replica in k.
n_{ij}	Whether reads from i fetch the full data item from j.
l	$l \in r, w$ indicates if term refers to reads/writes.

and items in the bucket see the same replica configuration. Our formulations would be applied at the granularity of directories.

In this section, we focus on latency under normal operation. In Sections 3.6 and 3.6.2, we show how our models may be extended to consider latency under failure, and incorporate communication costs.

3.5.1 Meeting SLA targets under normal operation

We consider settings where the datastore is deployed in up to M geographically distributed DCs. D_{ij} denotes the time to transfer a data item from DC j to DC i. For the applications we consider, the size of objects is typically small (e.g., tweets, meta-data,

small text files etc.), and hence data transmission times are typically dominated by propagation delays rather than the bandwidth between the DCs. Therefore, the D_{ij} parameter in our formulations (and evaluation) are based on the round trip times between the DCs. For applications dealing with large data objects, the measured D_{ij} values would capture the impact of data size and bandwidth as well.

Our focus is on regimes where the load on the storage node is moderate, and the primary component of the access latency is the network delay. Hence, we do not model the processing delays at the datastore node which are not as critical in the context of geo-replication.

We do not model details specific to implementation – e.g., on a read operation, the Cassandra system retrieves the full item from only the closest replica, and digests from the others. If a replica besides the closest has a more recent value, additional latency is incurred to fetch the actual item from that replica. We do not model this additional latency since the probability that a digest has the latest value is difficult to estimate and small in practice. Our experimental results in Section 3.8 demonstrate that, despite this assumption, our models work well in practice.

Let x_i be a binary indicator variable which is 1 iff DC i holds a replica of the data item. Let Q^r and Q^w be the read and write quorum sizes, and T^r and T^w respectively denote the latency thresholds within which all read and write accesses to the data item must successfully complete. Let q^r_{ij} and q^w_{ij} respectively be indicator variables that are 1 if read and write accesses originating from DC i use a replica in location j to meet their quorum requirements.

Typical SLAs require bounds on the delays seen by a pre-specified percentage of requests. Let p^r and p^w denote the fraction of read and write requests respectively that must have latencies within the desired thresholds. A key observation is that, given the replica locations, all read and, similarly all write requests, that originate from a given DC encounter the same delay. Thus, it suffices that the model chooses a set of DCs so that the read (resp. write) requests originating at these DCs experience a latency no more than T^r (resp. T^w) and these DCs account for a fraction p^r (resp. p^w) of read (resp. write) requests. Let N^r_i (resp. N^w_i) denote the number of read (write) requests originating from DC i. Let Y^r_i (resp.

Y_i^w) denote indicator variables which are 1 iff reads (resp. writes) from DC i meet the delay thresholds. Then, we have :

$$q_{ij}^l \leq x_j \; \forall i, j \; l \in \{r, w\} \tag{3.2}$$

$$D_{ij} q_{ij}^l \leq T^l \; \forall i, j \; l \in \{r, w\} \tag{3.3}$$

$$\sum_j q_{ij}^l \geq Q^l Y_i^l \quad \forall i; \; l \in \{r, w\} \tag{3.4}$$

$$\sum_i N_i^l Y_i^l \geq p^l \sum_i N_i^l \quad \forall i; \; l \in \{r, w\} \tag{3.5}$$

Equations (3.2) and (3.3) require that DC i can use a replica in DC j to meet its quorum only if (i) there exists a replica in DC j; and (ii) DC j is within the desired latency threshold from DC i. Equation (3.4) ensures that, within i's quorum set, there are sufficiently many replicas that meet the above feasibility constraints for the selected DCs. Equation (3.5) ensures the selected DCs account for the desired percentage of requests.

To determine the lowest latency threshold for which a feasible placement exists, we treat T^r and T^w as variables of optimization, and minimize the maximum of the two variables. We allow weights a^r and a^w on read and write delay thresholds to enable an application designer to prioritize reads over writes (or vice-versa). In summary, we have the ***Latency Only*(LAT)** model:

(LAT)

$$\begin{aligned}
&\min && \text{T} \\
&\text{subject to} && T \geq a^l T^l, \quad l \in \{r, w\} \\
&&& Q^r + Q^w = \textstyle\sum_j x_j + 1 \\
&&& \text{Quorum constraints (3.2), (3.3), (3.4)} \\
&&& \text{Percentile constraints (3.5)} \\
&&& Q^l \in \mathbb{Z}, \quad l \in \{r, w\} \\
&&& q_{ij}^l, x_j, Y_i^l \in \{0, 1\}, \quad \forall i, j; \, l \in \{r, w\}
\end{aligned}$$

Note that the constraint on quorum sizes captures the strict quorum requirement (Section 3.2) that each read sees the action of the last write. Also, when $p^r = p^l = 1$, (LAT) minimizes the delay of all requests and we refer to this special case as (LATM). Finally,

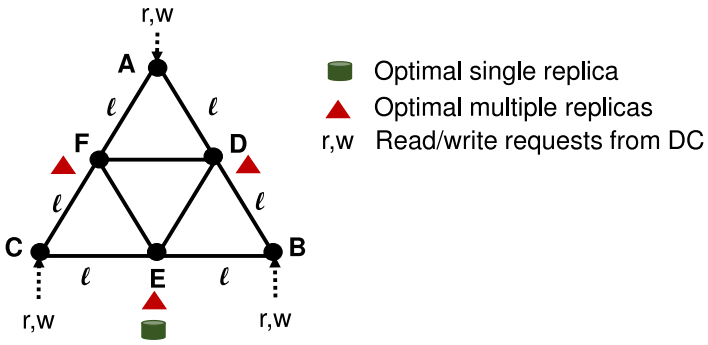


Fig. 3.4. An optimal multi replica solution with $Q^r = 2$, $Q^w = 2$ ensures a latency threshold of l, while an optimal single replica solution increases it to $\sqrt{3}l$

while (3.4) is not linear, it may be easily linearized as we show in [72]. Hence, our model can be solved using ILP solvers like CPLEX [73].

3.5.2 How much can replication lower latency?

Given the consistency requirement of quorum datastores, can replication lower latency, and, if so, by how much? In this section, we present examples to show that replication can lower latency, and provide bounds on the *replication benefit* (ratio of optimal latency without and with replication). In assessing the benefits of replication, two key factors are (i) *symmetric/asymmetric spread:* whether read and write requests originate from an identical or different set of DCs; and (ii) *symmetric/asymmetric weights:* whether the weights attached to read and write latency thresholds (a^r, a^w) are identical or different.

Figure 3.4 shows an example where spread and weights are symmetric and the *replication benefit* is $\sqrt{3} \approx 1.732$. When replicas can be placed arbitrarily on a Euclidean plane, it can be shown via an application of Helly's theorem [74] that the *replication benefit* is bounded by $\frac{2}{\sqrt{3}} \approx 1.155$. The setup of Figure 3.4 shows that this is a tight bound since replication achieves this benefit over single placement at the centroid of the triangle. Replication benefit can be even higher with asymmetric weights as seen in the observation below.

Observation 3.5.1 *With asymmetric spreads and metric delays, the replication benefit for (LATM) and (LAT) is at most* $4\frac{\max(a^r,a^w)}{\min(a^r,a^w)}$.

The proof can be found in our technical report [72].

3.6 Achieving latency SLAs despite failures

So far, we have focused on replication strategies that can optimize latency under normal conditions. In this section we discuss failures that may impact entire DCs, and present strategies resilient to such failures.

3.6.1 Failure resilient replication strategies

While several techniques exist to protect against individual failures in a DC [75], geo-distributed DCs are primarily motivated by failures that impact entire DCs. While failures within a DC have been studied [75, 76], there are few studies on failures across DCs to the best of our knowledge. Discussions with practitioners suggests that while DC level failures are not uncommon (Figure 3.1), correlated failures of multiple geographically distributed DCs are relatively rare (though feasible). Operators strive to minimize simultaneous downtime of multiple DCs through careful scheduling of maintenance periods and gradual roll-out of software upgrades.

While a sufficiently replicated geo-distributed cloud datastore may be available despite a DC failure, the latency are likely negatively impacted. We present replication strategies that are resilient to such failures. Pragmatically, we first focus on the common case scenario of single DC failures. Then, in Section 3.6.2, we show how our models easily extend to more complex failure modes. Our models are:

Basic Availability Model (BA): This model simply optimizes latency using (LAT) with the additional constraints that the read and write quorum sizes are at least 2 (and hence the number of replicas is at least 3). Clearly, read and write requests can still achieve quorum when one DC is down and basic availability is maintained. This model does not explicitly consider latency under failure and our evaluations in Section 3.8 indicate that the

scheme may perform poorly under failures – for e.g., the 90^{th} percentile request latency for English *Wikipedia* documents increased from 200msec to 280msec when one replica was unavailable.

N-1 Contingency Model (N-1C): This model minimizes the maximum latency across a pre-specified percentile of reads and writes allowing at most one DC to be unavailable at any given time. The model is motivated by contingency analysis techniques commonly employed in power transmission systems [77] to assess the ability of a grid to withstand a single component failure. Although this model is similar in structure to (LAT), there are two important distinctions. First, the quorum requirements must be met not just under normal conditions, but under all possible single DC failures. Second, the desired fraction of requests serviced within a latency threshold, could be met by considering requests from different DCs under different failure scenarios.

Formally, let p_f^r (resp. p_f^w) be the fraction of reads (resp. writes) that must meet the delay thresholds when a replica in any DC is unavailable. Note that the SLA requirement on failures may be more relaxed, possibly requiring a smaller fraction of requests to meet a delay threshold. Let Y_{ik}^r (resp. Y_{ik}^w) be indicator variables that are 1 if read (resp. write) requests from DC i are served within the latency threshold when the replica in DC k is unavailable. Then, we replace (3.5) and (3.4) with the following:

$$\sum_i Q_i^l Y_{ik}^l \geq p_f^l \sum_i N_i^l \ \forall i \forall k \tag{3.6}$$

$$\sum_{j,j\neq k} q_{ij}^l \geq Q^l Y_{ik}^l \ \forall i,k \ \ l \in \{r,w\} \tag{3.7}$$

The first constraint ensures that sufficient requests are serviced within the latency threshold no matter which DC fails. The index k for the Y variables allows the set of requests satisfied within the latency threshold to depend on the DC that fails. The second constraint ensures that the quorum requirements are met when DC k fails with the caveat that DC k cannot

be used to meet quorum requirements. We remark that (3.7) may be linearized in a manner similar to (3.4). Putting everything together, we have:

(N-1C)

$$\begin{aligned} &\min && T_f \\ &\text{subject to} && T_f \geq a^l T^l, \quad l \in \{r, w\} \\ &&& Q^r + Q^w = \textstyle\sum_j x_j + 1 \\ &&& \text{Quorum constraints (3.2), (3.3), (3.7)} \\ &&& \text{Percentile constraints (3.6)} \\ &&& Q^l \in \mathbb{Z}, \quad l \in \{r, w\} \\ &&& q^l_{ij}, x_j, Y^l_{ik} \in \{0, 1\}, \forall i, j, k; l \in \{r, w\}. \end{aligned}$$

3.6.2 Model Enhancements

We discuss enhancements to the *N-1 Contingency* model:

Cost-sensitive replication: When datastores are deployed on public clouds, it is important to consider dollar costs in addition to latency and availability. We focus on wide-area communication costs since (i) this is known to be a dominant component of costs in geo-replicated settings [78]; (ii) best practices involve storing data in local instance storage with periodic backups to persistent storage [79] - the costs of such backups are independent of our replication policy decision; and (iii) instance costs are comparable to a single DC deployment with the same number of replicas. Most cloud providers today charge for out-bound bandwidth transfers at a flat rate per byte (in-bound transfers are typically free), though the rate itself depends on the location of the DC. Let C_i be the cost per byte of out-bound bandwidth transfer from DC i. Consider an operation that originates in DC i and involves writing a data item whose size is S bytes. Then, the total cost associated with all write operations is $\sum_i N_i^w SC_i \sum_j X_j$. However, read operations in Cassandra retrieve the full data item only from its nearest neighbor but receives digest from everyone. Let n_{ij} denote an indicator variable, which is 1 if the full data item is fetched from DC j. The size of the digest is assumed negligibly small. The total cost associated with all read operations is: $\sum_i \sum_j N_i^r n_{ij} SC_j$. It is now straight-forward to modify (N-1C) to optimize costs subject

to a delay constraint. This may be done by making threshold (T) a fixed parameter rather than a variable of optimization and adding additional constraints on n_{ij}.

Jointly considering normal operation and failures: Formulation (N-1C) finds replication strategies that reduce latency under failure. In practice, a designer prefers strategies that work well in normal conditions as well as under failure. This is achieved by combining the constraints in (LAT) and (N-1C), with an objective function that is a weighted sum of latency under normal conditions T and under failures T_f. The weights are chosen to capture the desired preferences.

Failures of multiple DCs: While we expect simultaneous failures of multiple DCs to be relatively uncommon, it is easy to extend our formulations to consider such scenarios. Let K be a set whose each element is a set of indices of DCs which may fail simultaneously and we are interested in guarding the performance against such a failure. We then employ (N-1C) but with k iterating over elements of K instead of the set of DCs. A naive approach may exhaustively enumerate all possible combination of DC failures, could be computationally expensive, and may result in schemes optimized for unlikely events at the expense of more typical occurrences. A more practical approach would involve explicit operator specifications of correlated failure scenarios of interest. For e.g., DCs that share the same network PoP are more likely to fail together, and thus of practical interest to operators.

Network partitions: In general, it is impossible to guarantee availability with network partition tolerance given the strict quorum requirement [80]. For more common network outages that partition one DC from others, our *N-1C* model ensures that requests from all other DCs can still be served with low latency. To handle more complex network partitions, an interesting future direction is to consider weaker quorum requirements subject to bounds on data staleness [70].

3.7 Evaluation Methodology

[1]Aggregating all articles per language (e.g. 4 million articles in English *Wikipedia* are aggregated.)

Table 3.3
Trace characteristics

Application	# of keys/classes	Span
Twitter [81]	3,000,000	2006-2011
Wikipedia [82]	196[1]	2009-2012
Gowalla [83]	196,591	Feb 2009-Oct 2010

We evaluate our replication strategies *Latency Only* (LAT), *Basic Availability* (*BA*), and *N-1 Contingency* (*N-1C*) with a view to exploring several aspects such as:

• Accuracy of our model in predicting performance

• Limits on latency achievable given consistency constraints

• Benefits and costs of optimizing latency under failures

• Importance of employing heterogeneous configurations for different groups of data items within an application

• Robustness to variations in network delays and workloads

We explore these questions using experiments on a real wide-area Cassandra cluster deployed across all the 8 regions (and 21 availability zones) of Amazon EC2 and using trace-driven simulations from three real-world applications: *Twitter*, *Wikipedia* and *Gowalla*. Our EC2 experiments enable us to validate our models, and to evaluate the benefits of our approach in practice. Simulation studies enable us to evaluate our strategies on a larger scale (hundreds of thousands of data items), and to explore the impact of workload characteristics and model parameters on performance. We use GAMS [84] (a modeling system for optimization problems) and solve the models using the CPLEX optimizer.

3.7.1 Application workloads

The applications we choose are widely used, have geographically dispersed users who edit and read data, and fit naturally into a key-value model. We note that both *Twitter* and

Gowalla are already known to use Cassandra [85]. We discuss details of the traces below (see table 3.3 for summary):

Twitter: We obtained *Twitter* traces [81] which included a user friendship graph, a list of user locations, and public tweets sent by users (along with timestamp) over a 5 year period. We analyzed Twissandra, an open-source twitter-like application, and found three types of data items: users, tweets and timelines. We focus our evaluations on timeline objects which are pre-materialized views that map each user to a list of tweets sent by the user and her friends. Writes to a timeline occur when the associated user or her friends post a tweet, and can be obtained directly from the trace. Since the traces do not include reads, we model reads by assuming each user reads her own timeline periodically (every 10 min), and reads her friend's timeline with some probability (0.1) each time the friend posts a tweet.

Wikipedia: We obtained statistics regarding *Wikipedia* usage from [82], which lists the total as well as the breakdown of the number of views and edits by geographic region for each language and collaborative project. The data spans a 3 year period with trends shown on quarterly basis. Our model for the *Wikipedia* application consists of article objects with the document id as a key and the content along with its meta data (timestamps, version information, etc). Article page views are modeled as reads while page edits are modeled as writes. Since per article access data is not available, we model all articles of the same language and project as seeing similar access patterns since access patterns are likely dominated by the location of native speakers of the language.

Gowalla: *Gowalla* is a (now disabled) geo-social networking application where users "check-in" at various locations they visit and friends receive all their check-in messages. The traces [86] contained user friendship relationships, and a list of all check-ins sent over a two year period. Since the application workflows are similar, we model *Gowalla* in a similar fashion to *Twitter*. Check-ins represent writes to user timelines from the location of the check-in, and reads to timelines were modeled like with *Twitter*.

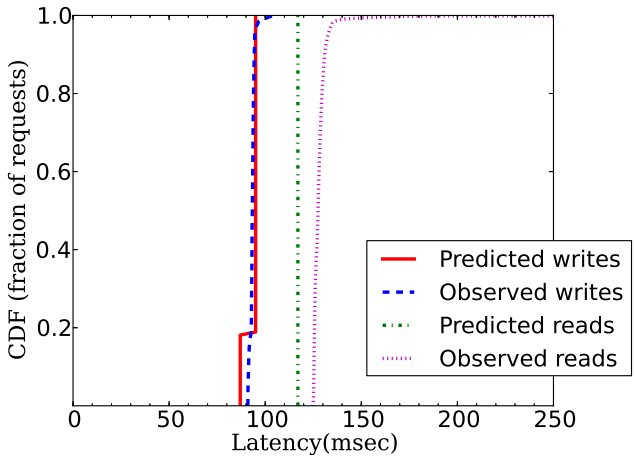


Fig. 3.5. Validating the accuracy of models.

3.8 Experimental Validation

In this section, we present results from our experiments using Cassandra deployed on Amazon EC2.

3.8.1 Implementation

Off-the-shelf, Cassandra employs a random partitioner that implements consistent hashing to distribute load across multiple storage nodes in the cluster. The output range of a hash function is treated as a fixed circular space and each data item is assigned to a node by hashing its key to yield its position on the ring. Nodes assume responsibility for the region in the ring between itself and its predecessor, with immediately adjacent nodes in the ring hosting replicas of the data item. Cassandra allows applications to express replication policies at the granularity of keyspaces (partitions of data). We modified the applications to treat groups of data items as separate keyspaces and configure distinct replication policy for each keyspace. Keyspace creation is a one-time process and does not affect the application performance. The mapping from data object to the keyspace is maintained in a separate directory service. We implemented the directory service as an independent Cassandra clus-

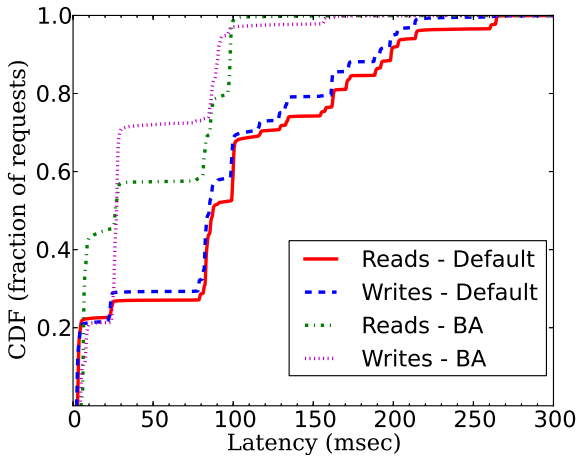


Fig. 3.6. Comparing the performance of *BA* scheme with Cassandra's default random partitioner.

ter deployed in each of the DCs and configured its replication such that lookups(reads) are served locally within a DC (e.g. $R = 1, W = N$).

3.8.2 Experimental platform on EC2

We performed our experiments and model validations using Cassandra deployed on medium size instances on Amazon EC2. Our datastore cluster comprises of nodes deployed in each of the 21 distinct availability zones (AZ) across all the 8 regions of EC2 (9 in US, 3 in Europe, 5 in Asia, 2 in South America and 2 in Australia). We treat availability zones (AZs) as distinct DC in all our experiments. The inter-DC delays ($21 * 21$ pairs) were simultaneously measured for a period of 24 hours using medium instances deployed on all the 21 AZs and the median delays values (MED) were used as input to our models. We mapped users from their locations to the nearest DC. Since the locations are free-text fields in our traces, we make use of geocoding services [87] to obtain the user's geographical co-ordinates.

3.8.3 Accuracy and model validation

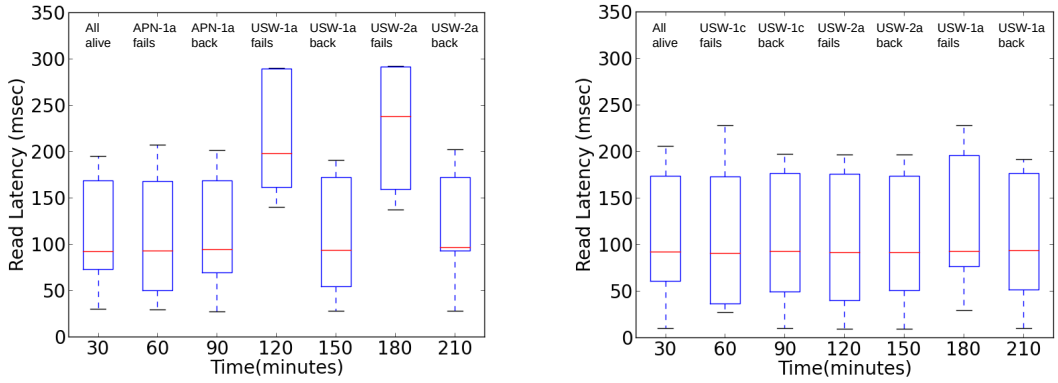
We validate the accuracy of our models with experiments on our EC2 Cassandra cluster described above. We use the example from our *Twitter* trace (Figure 3.2) for this experiment. Replica configurations were generated with the MED delay values measured earlier and read/write requests to Cassandra cluster were generated from application servers deployed at the corresponding DCs as per the trace data. The duration of the entire experiment was about 6 hours.

Figure 3.5 shows the CDFs of the *observed* and *predicted* latencies for read and write requests for the *BA* configuration. The CDFs almost overlap for write requests, while we observe a delay of approximately 9 msec evenly for all read requests. This constant delay difference in the reads can be attributed to the processing overhead of read requests in Cassandra which includes reconciling the response of multiple replicas to ensure consistency of the read data. Overall, our results validate the accuracy of our models. They also show that our solutions are fairly robust to the natural delay variations present in real cloud platforms.

3.8.4 Benefits of performance sensitive replication

We first evaluate the benefits of flexible replication policy over a fixed replication policy on the EC2 Cassandra cluster described above. For this experiment, we use a month long trace from *Twitter* consisting of $524,759$ objects corresponding to user timelines in *Twitter*. The replica configurations were generated for each timeline object using the *BA* model and the corresponding directory entries were created in all the regions. Reads and writes were initiated as per the traces from the Twissandra application servers deployed in each of the EC2 regions. While the duration of the entire experiment was scaled to 16 hours, care was taken to ensure that the fraction of requests to all objects from each DC was proportional to what was observed in the trace data.

Figure 6 shows the CDF comparing the read and write latency observed with our *BA* scheme and Cassandra's random partitioner. The Y-Axis shows the CDF of the fraction



(a) *Basic Availability* (b) *N-1 Contingency*

Fig. 3.7. Boxplot showing the distribution of read latency with *BA* and *N-1C* models for every half hour period. Whiskers show the 10th and 90th percentiles.

of all requests seen in the system (approx 6 million each for *BA* and Random) while the X-Axis shows the observed per request latency in msec. To ensure a fair comparison of schemes, the observed latency values for *BA* includes directory lookup latency as well. From the figure, we see that our flexible replication scheme is able to outperform the default replication scheme by 50 msec (factor of 3) at the $50th\%$ile and by 100msec at the $90th\%$ile (factor of 2). A keen observer might note that Random performs marginally better than (approx $3-8$msec) *BA* at the initial percentiles due to the latency overhead incurred for the directory lookup.

3.8.5 Availability and performance under failures

In this section, we study the performance of the *BA* and *N-1C* schemes under the failures of different DCs using our multi-region Cassandra cluster on EC2. We perform this study using the trace data from *Wikipedia* for the English wiki articles for which the accesses arrive from all the 8 EC2 regions including 50% from the US, 23% from Europe, 10% from Singapore, 5% from Sydney and the rest from South America and Tokyo. Failures were created by terminating the Cassandra process in a DC and redirecting requests from the

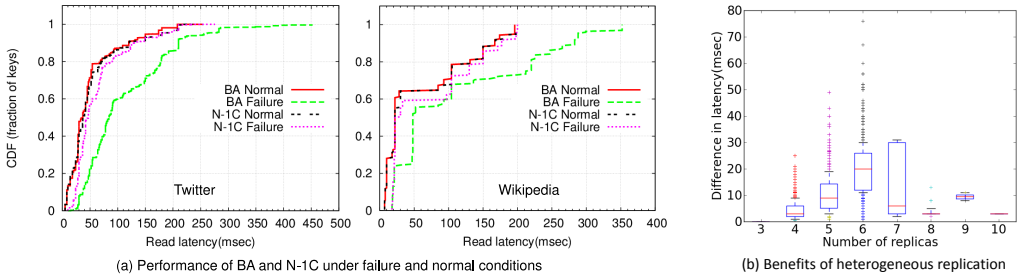
application to the Cassandra process in the closest DC. The duration of the experiment was approximately 9 hours.

For the English wiki articles, our *BA* scheme placed two replicas in the west coast (USW-1a and USW-2a) and the 3rd replica in Tokyo (APN-1a) with $R = 2$ and $W = 2$. This is reasonable since nodes in the US West are reasonably equidistant from Asia, Australia, Europe and US East while placing the 3rd replica in Asia also reduces the $90\%ile$ latency under normal operation. Figure 3.7(a) shows the performance of the *BA* scheme under failure of different DCs. The corresponding events for every half hour period is marked at the top of the plots. From the figure, we see that the $90\%ile$ latency increases significantly from 200msec (under normal operation) to 280msec when the west coast DCs fail (40% increase), while the failure of Tokyo DC (APN-1a) has only a marginal impact on the performance.

In contrast, the *N-1C* scheme explicitly optimizes for latency under a failure and places the 3rd replica in USW-1a instead of Tokyo. Figure 3.7(b) shows the performance of the *N-1C* scheme under failures of different DCs. The figure shows that our *N-1C* scheme performs similar to the *BA* scheme (median of 90msec and $90\%ile$ of $200ms$) during normal operation. However, unlike the *BA* configuration, the $90\%ile$ latency remains largely unaffected under all failures. Our results highlight the need to explicitly optimize for performance under failure and show the benefits of *N-1C* over the *BA* scheme. Further, the median and $90\%ile$ latencies from our experiments were found to be very close to our model predictions under normal and failure conditions for both the models, thereby validating our models.

3.9 Large scale evaluation

We adopt a trace driven simulation approach for our large scale evaluation on the three application traces, where we consider the datastore cluster to comprise of nodes from each of 27 distinct DCs world-wide, whose locations were obtained from AWS Global Infrastructure [88]. Inter-DC delays were measured between Planet-lab nodes close to each DC



(a) Performance of BA and N-1C under failure and normal conditions

(b) Benefits of heterogeneous replication

Fig. 3.8. Trace driven study with all keys in the application.

and delay measurements were collected simultaneously between all pairs of locations over a few hours and median delays were considered. Users were mapped to the closest DCs as in our EC2 experiments. We pick this extended set of DCs as the EC2 regions are limited in number. For example, EC2 has no regions in the Mid-west US, but AWS Global Infrastructure provides multiple DCs in those areas. Moreover, we expect these DCs to be expanded to offer more services in the future. Experiments in this section use traces of one month (Dec 2010) in *Twitter*, one month (Oct 2010) in *Gowalla* and one quarter (Q4 2011) in *Wikipedia*.

3.9.1 Performance of our optimal schemes

Figure 3.8(a) shows the CDF of the observed read latency across both schemes for all keys in *Twitter* and *Wikipedia* traces under normal and failure conditions. For each key, we plot the read latency under normal conditions (all replicas are alive) and when the most critical replica (replica whose failure results in the worst latency) for that key fails. From the figure, we see that the read latency observed by the *BA* scheme deteriorates drastically under failure for almost all keys in both the applications. For instance, more than 40% of the keys in *Twitter* observed an increase of $50+$ msec (more than 20% of the keys observed an increase of $100+$ msec in *Wikipedia*) under failure conditions. However, read latency for *N-1C* observed only a marginal variation under failure (most keys in *Twitter* observed less than 30msec increase in latency on its replica failures). Surprisingly, we find that the *N-1C* scheme incurs an almost negligible penalty in its latency under normal conditions despite

optimizing the replica configuration explicitly for the failure of a replica. Further, we found that *BA* was often able to optimize latency with two of the chosen replicas and the third choice did not significantly impact performance. In contrast, the *N-1C* scheme carefully selects the 3rd replica ensuring good performance even under failures. Overall, our results clearly show the benefit of explicitly optimizing the replication for failure conditions.

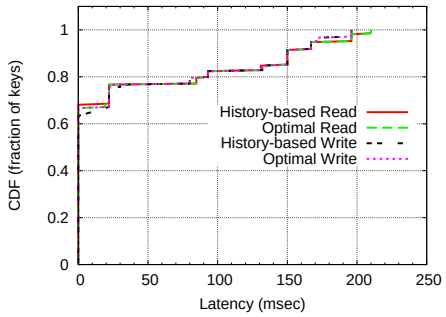
3.9.2 Need for heterogeneous configuration policy

In this section, we highlight the importance of allowing heterogeneous replica configurations in datastores and show why a uniform replication configuration for all data in the application can often have poor performance. We analyzed the configurations generated by *N-1C* for all keys in the *Twitter* trace. From our analysis we find that there were as many as 1985 distinct configurations (combination of replica location, N, R, W) that were used in the optimal solutions.

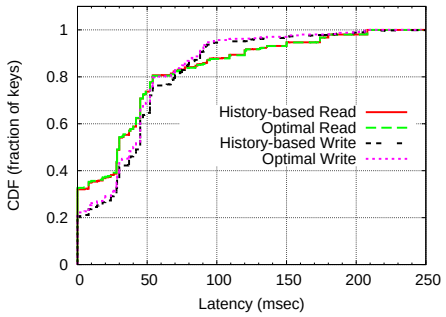
Interestingly, we find that the benefits are not only due to optimizing the location of replicas but also due to careful configuration of the replication parameters - N, R and W. To isolate such cases we consider a variant of our *N-1C* model that we call $3-2-2$ which has fixed replication parameters $N = 3$, $R = 2$ and $W = 2$, but allows flexibility in the location of the replicas. Figure 3.8(b) shows the difference in the access latency between the $3-2-2$ and *N-1C* schemes for *Twitter*. The X-axis has the various replication factors observed in the optimal solutions and each corresponding box plot shows the 25th, median and 75th percentiles (whiskers showing the 90th percentile) of the difference in access latency between the two schemes. Our results clearly show that a uniform configuration policy for all data in the application can be sub-optimal and allowing heterogeneity in replica configuration can greatly lower the latency (as much as 70msec in some cases).

3.9.3 History-based vs Optimal

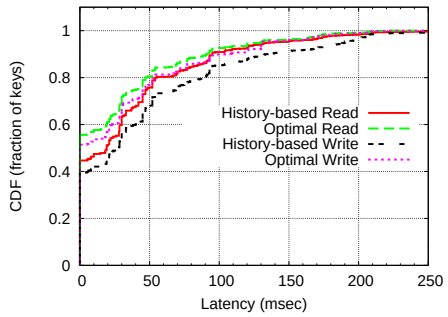
So far, we had assumed that the workloads for the applications are known. However, in practice, this may need to be obtained from historical data. In this section, we analyze this



(a) *Wikipedia*



(b) *Twitter*



(c) *Gowalla*

Fig. 3.9. Optimal performance vs performance using replica placements from the previous period.

gap by comparing the performance of our schemes using historical and actual workloads for all three applications.

Figure 3.9(a) shows the CDF comparing the performance of *Wikipedia* during the first quarter of 2012 when using the history-based and the optimal replication configuration. The curves labeled history-based correspond to the read and write latency observed when using the replica configuration predicted from the fourth quarter of 2011. The curves labeled optimal correspond to the read and write latency observed when using the optimal replica configuration for the first quarter of 2012. Figures 3.9(b) and 3.9(c) show similar graphs for *Twitter* and *Gowalla*. These figures show that history-based configuration performs close to optimal for *Wikipedia* and *Twitter*, while showing some deviation from optimal

performance for *Gowalla*. This is because users in *Gowalla* often move across geographical regions resulting in abrupt workload shifts. For such abrupt shifts, explicit hints from the user when she moves to a new location or automatically detecting change in the workload and rerunning the optimization are potential approaches for improving the performance.

3.9.4 Robustness to delay variations

Our experiments on EC2 (Section 3.8) show that our strategies are fairly robust to natural delay variations across cloud DCs. In this section, we extend our analysis over a larger set of keys. We compute about 1800 time snapshots of the entire 27*27 inter-DCs delays for our extended DC set. All delay values in the snapshot were measured approximately at the same time. Next, we computed the optimal replica configurations (using our *BA* and *N-1C* schemes) for 500 random keys from the *Twitter* trace for each of 1800 snapshots. We call these the *SNAP* configurations. Similarly, replica configurations are computed using the median delay values of the 1800 snapshots. We call these the *MED* configurations. We then compare the performance of the *MED* configuration using delays observed at each snapshot with the performance of the optimal *SNAP* configuration at the same snapshot.

Figure 3.10 shows the CDF of the difference in access latency between the *MED* and *SNAP* configurations. Each curve in the figure corresponds to a range of latencies observed using the *SNAP* configurations. For *SNAP* latencies less than 100msec, and for over 90% of snapshots, MED only incurs less than 5msec additional latency. Also, for almost 80% of all the *SNAPs*, the corresponding *MED* configuration was optimal. While the penalty is higher for *SNAP* latencies over 100 msec, we believe they are still acceptable (less than 15msec for 90% of the cases) given the relatively higher *SNAP* latencies. Overall, the results further confirm our EC2 results and show that delay variation impacts placement modestly.

3.9.5 Asymmetric read and write thresholds

Thus far, we assumed that read and write latencies are equally desirable to optimize. However, in practice, some applications may prioritize read latencies, and others might pri-

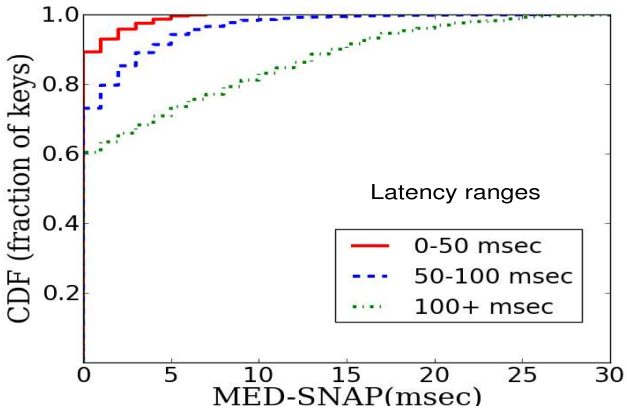


Fig. 3.10. Comparing SNAP and MED performance.

oritize writes. We have explored solutions generated by our approach when our models are modified to explicitly constrain the read and write thresholds. For *Twitter*, we found that a bound of 100msec on the write latency has no noticeable impact on the read latency, though the tail was more pronounced. Interestingly, we also found that the bound of 50msec increases the read latency by less than 20msec for 60% of the keys. We found that constraints on write latency resulted in configurations that had a significantly higher replication factor and higher read quorum sizes. This is expected because our models tries to minimize the latency by moving the replica closer to the write locations in order to meet the constraint. We omit results for lack of space.

3.10 Related Work

SPAR [89] presents a middle-ware for social networks which co-locates data related to each user within the same DC to minimize access latency. [89] achieves this by having a master-slave arrangement for each data item, creating enough slave replicas, and updating them in an eventually consistent fashion. However, master-slave solutions are susceptible to issues related to data loss, and temporary downtime (see Section 3.2). In contrast, we consider a strict quorum requirement, and allow updates on any replica.

Owing to consistency constraints, quorum placement is different from facility location (FL) problems, and known variants [62]. The classical version of FL seeks to pick a subset of facilities (DCs) that would minimize the distance costs (sum of distances from each demand point to its nearest facility), plus the opening costs of the facilities. Without opening cost or capacity constraints, FL is trivial (a replica is introduced at each demand point) – however quorum placement is still complex. For e.g., in Figure 3.4, the optimal FL solution places 3 replicas at the triangle vertices which is twice the quorum latency of our solution. Increasing the number of replicas can hurt quorum latencies owing to consistency requirements, but does not increase distance costs with FL.

Volley [90] addresses the problem of placing data considering both user locations and data inter-dependencies. However, [90] does not address replication in depth, simply treating replicas as different items that communicate frequently. [90] does not model consistency requirements, a key focus of our work. Also, unlike [90], our models automatically determine the number of replicas and quorum parameters while considering important practical aspects like latency percentiles and performance under failures.

While systems like Spanner [8] and Walter [64] support flexible replication policies, they require these policies to be manually configured by administrators. In contrast, our formulations enable quorum based datastores to make these replica configuration decisions in an automated and optimal fashion. Recent works like Vivace [91] suggest novel read/write algorithms that employ network prioritization which enable geo-replicated datastores adapt to network congestion. Unlike these systems, we focus on the more general and important problem of automatically configuring the replication parameters including the number of replicas, location of replicas and quorum sizes. SPANStore [92] focuses on placing replicas across multiple cloud providers with the primary aim of minimizing costs exploiting differential provider pricing. In contrast, we focus on supporting flexible replication policies at different granularities that can be tuned to a variety of objectives such as minimizing latencies under failure. Also, the quorum protocol implemented by SPANStore is different from the ones used in quorum based systems like Cassandra, and hence our model formulations are different. [93] proposes algorithms extending scalable

deferred update replication (SDUR) in the context of geographically replicated systems. In contrast, we focus on the orthogonal problem of configuring optimal replication policies for geo-distributed datastores.

While there has been much theoretical analysis of quorum protocols, our work is distinguished by our focus on widely used quorum datastores, and issues unique to datastore settings. Prior work has considered communication delays with quorum protocols [24–26]. In particular, [24, 25] consider problems that minimize the maximum node delays. However, none of these works optimize latency percentiles, latency under failures, or consider different priorities for read and write traffic. To our knowledge, our framework is the first to consider these factors, all of which are key considerations for geo-distributed datastores. We also note that [24–26] are in the context of coteries [94], and do not immediately apply to cloud datastores which are adapted from weighted voting-based quorum protocols [23].

Several works have examined availability in quorum construction [95–99]. Most of these works do not consider the impact of failures on latency. Recent work [98] has considered how to dynamically adapt quorums to changes in network delays. Given that systems like Cassandra and Dynamo contact all replicas and not just the quorum, we focus on the orthogonal problem of replica selection so that failure of one DC does not impact latency. Several early works [95, 96] assume independent identically distributed (IID) failures, though non-IID failures are beginning to receive attention [97]. Instead, we focus on choosing replication strategies that are resilient and low-latency under failures of a single DC, or a small subset of DCs which are prone to correlated failures (Section 3.6.2).

3.11 Discussion and Implications

We discuss the implications of our findings:

Implications for datastore design: Our results in Section 3.9.2 show the importance of diverse replica configurations for the same application given heterogeneity in workloads for different groups of items – 1985 distinct replica configurations were required for *Twitter*. Many geo-replicated datastores are not explicitly designed with this requirement in mind

and may need to revisit their design decisions. For e.g., Eiger [12] replicates all data items in the same set of DCs. Cassandra [9] and Dynamo [7] use consistent hashing which makes it difficult to flexibly map replicas to desirable DCs (we effectively bypass consistent hashing with multiple keyspaces in Section 3.8). In contrast, Spanner [8] explicitly maintain directories that list locations of each group of items, and is thus better positioned to support heterogeneous replication policies.

Delay variation: Our multi-region EC2 evaluations (Section 3.8) and simulation results (Section 3.9.4) show that placements based on median delays observed over several hours of measurement are fairly robust to short-term delay variations. We believe delay variation impacts placement modestly since links with lower median delay also tend to see smaller variations. These results indicate that the benefits of explicitly modeling stochasticity in delay is likely small, and these benefits must be weighed against the fact that stochastic delay values are hard to quantify in practice especially when not independent. Further, we note that placements from our *N-1C* model can tolerate congestion close to any DC. Finally, more persistent variations in delay over longer time-scales are best handled by recomputing placements on a periodic basis or on a prolonged change in network delays.

Workload variation: Section 3.9.3 shows that for many applications, the optimal solution based on historical access patterns performs well compared to the solution obtained with perfect information of future access patterns. Consider the case where workloads exhibit seasonal patterns (for e.g. diurnal effects) and data-migration costs over short time-scales are large enough that one chooses to maintain same replicas across the seasons. Then, our models optimize placement assuming a percentage of total requests across seasons are satisfied within the specified latency. Instead, if one wants to have a certain service level for each season, our models may be extended by replicating the model for each season and imposing the constraint that placement decisions are season independent. Finally, we also evaluated our models with placement recomputations performed at different time granularities. We found that daily, weekly and monthly recomputations perform similarly, while hourly recomputation benefits a modest fraction(15%) of requests but incurs higher mi-

gration overheads. Hence, recomputation at coarser granularities seems to be the more appropriate choice.

Computational Complexity: Our optimization framework allows a systematic approach to analyzing replication strategies in cloud datastores, and delivers insights on the best latency achievable for a given workload with consistency constraints. With our prototype implementation LAT, *BA*, and *N-1C* models solve within 0.16, 0.17 and 0.41 seconds respectively using a single core on a 4 core, $3GHz$, $8GB$ RAM machines. While already promising, we note that (i) our implementation is not optimized. Many opportunities (heuristics, valid cuts, modeling interface) exist for better efficiency; (ii) systems like Spanner [8] require applications to bucket items, and computations would be performed at coarser bucket granularities; (iii) our per-bucket formulations are embarrassingly parallel; and (iv) our placements are stable over days (Sec 3.9.3) and placement recomputations are not frequent.

3.12 Conclusions

In this chapter, we make several contributions. First, we have developed a systematic framework for modeling geo-replicated quorum datastores in a manner that captures their latency, availability and consistency requirements. Our frameworks capture requirements on both read and write latencies, and their relative priority. Second, we have demonstrated the feasibility and importance of tailoring geo-distributed cloud datastores to meet the unique workloads of groups of items in individual applications, so latency SLA requirements (expressed in percentiles) can be met during normal operations and on the failure of a DC. Third, we explore the limits on latency achievable with geo-replicated storage systems for three real applications under strict quorum requirement. Our evaluations on a multi-region EC2 test-bed, and longitudinal workloads of three widely deployed applications validate our models, and confirm their importance.

4. MAKING MULTI-TIER APPLICAITONS RESILIENT TO PERFORMANCE VARIABILITY IN THE CLOUD

4.1 Introduction

Cloud computing promises to reduce the cost of IT organizations by allowing them to purchase as much resources as needed, only when needed, and through lower capital and operational expense stemming from the cloud's economies of scale. Further, moving to the cloud greatly facilitates the deployment of applications across multiple geographically distributed data-centers. Geo-distributing applications, in turn, facilitates service resilience and disaster recovery, and could enable better user experience by having customers directed to DCs close to them.

While these advantages of cloud computing are triggering much interest among developers and IT managers [100, 101], a key challenge is meeting the stringent *Service Level Agreement (SLA)* requirements on availability and response times for interactive applications (e.g. customer facing web applications, enterprise applications). Application latencies directly impact business revenue [2, 3]– e.g., Amazon found every $100ms$ of latency costs 1% in sales [3]. Further, the SLAs typically require bounds on the $90th$ (and higher) percentile latencies [102, 103].

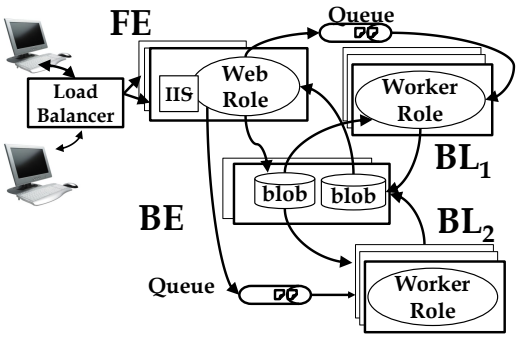
Multi-tier applications consist of potentially hundreds of components with complex inter-dependencies and hundreds of different transactions all involving different subsets of components [104]. Meeting such stringent SLA requirements for these applications is a challenge, given outages in cloud DCs [18, 19], and the high variability in the performance of cloud services [20–22]. This variability arises from a variety of factors such as the sharing of cloud services across a large number of tenants, and limitations in virtualization techniques [20]. For example, [21] showed that the $95\%ile$ latencies of cloud storage

services such as tables and queues is 100% more than the median values for four different public cloud offerings.

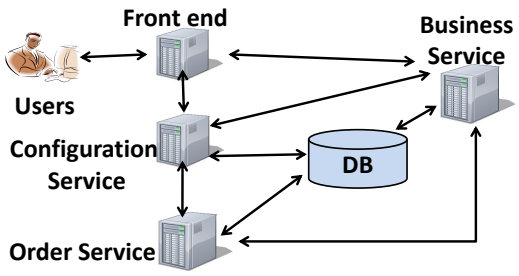
In this chapter, we make the following contributions. First, we show the importance of designing multi-tier applications to be intrinsically resilient to cloud performance variations. We deploy three real-world multi-tier applications on commercial cloud platforms like Windows Azure and Amazon AWS and perform extensive measurement study on the performance of these applications. Second, we present *Dealer*, a system that enables applications to meet their stringent SLA requirements on response times by finding the combination of replicas –potentially located across multiple DCs– that should be used to serve any given request. This is motivated by the fact that only a small number of application components of large multi-tier applications experience poor performance at any time.

Dealer abstracts application structure as a component graph, with nodes being application components and edges capturing inter-component communication patterns. To predict which combination of replicas can result in the best performance, *Dealer* continually monitors the performance of individual component replicas and communication latencies between replica pairs. Operating at a component-level granularity offers *Dealer* several advantages over conventional approaches that merely pick an appropriate DC to serve user requests [105–108]. Modern web applications consist of many components, not all of which are represent in each DC, and the costs are extremely high to over-provision each component in every DC to be able to handle all the traffic from another DC. *Dealer* is able to redistribute work away from poorly performing components by utilizing the capacity of all component replicas that can usefully contribute to reducing the latency of requests.

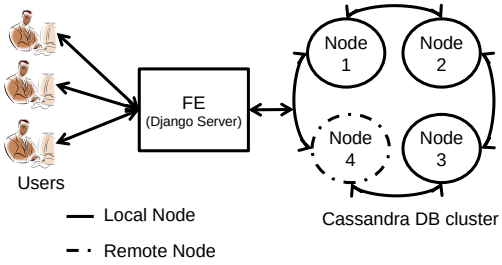
We evaluate *Dealer* on two stateful multi-tier applications on Azure cloud deployments. The first application is data-intensive, while the second application involves interactive transaction processing. Under natural cloud dynamics, using *Dealer* improves application performance by a factor of 3 for the 90^{th} and higher delay percentiles, compared to DNS-based DC-level redirection schemes which are agnostic of application structure. Overall, the results indicate the importance and feasibility of *Dealer*.



(a) Thumbnails application architecture and data-flow. The application is composed of a Front-End (FE), Back-End (BE), and two Business-Logic components BL_1 (creates thumbnail) and BL_2 (creates Gray-scale, rotating images).



(b) *StockTrader* architecture and data-flow. Components include a front-end (FE), Business Server (BS), Order Service (OS) (handles buys/sells), Database (DB), and a Config Service (CS) that binds all components. The precise data-flow depends on transaction type.



(c) *Twissandra* consisting of a FE (Django web server) and database (Cassandra cluster). The FE receives user requests and initiates a request with a local Cassandra node to retrieve data from the DB cluster.

Fig. 4.1. Applications Testbed.

4.2 Performance and Workload Variability

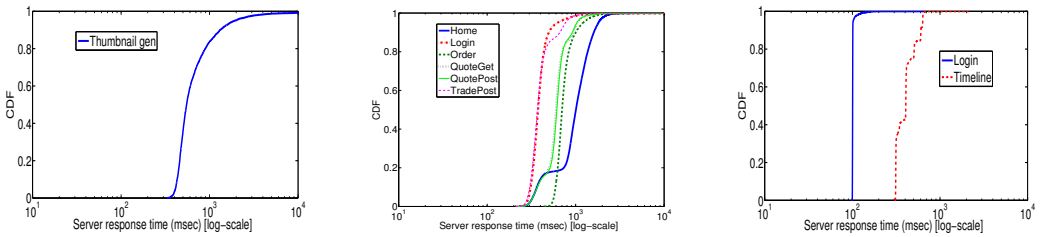
In this section, we present observations that motivate *Dealer*'s design. In §4.2.1, we characterize the extent and nature of the variability in performance that may be present in cloud DCs. Our characterization is based on our experiences running different multi-tier applications on the cloud.

We measure the performance variability with the following three applications. Figure 4.1(a), Figure 4.1(b) and Figure 4.1(c) respectively show the component architecture and data-flow for each application.

Thumbnail: *Thumbnail* is a three tier web application provided as a part of the Windows Azure SDK. The application is data-intensive involving users uploading an image to the server and receiving the thumbnail version of the image in turn. Figure 4.1(a) shows the architecture of *Thumbnail*. The application consists of a Front-End (FE), Back-End (BE), and Business-Logic (BL) components. It has a simple and linear data flow where users upload pictures to the FE (t0). The FE writes the image to the BE (t1-b) and notifies the BL(t1-a). The BL in turn creates a thumbnail, and stores it in BE (t3). The FE retrieves the thumbnail (t4) and sends it to back the user (t5). The FE components were deployed as Azure Web roles, BL as Azure worker roles and BE as an Azure Blob storage service.

StockTrader: *StockTrader* is a tiered enterprise web application that allows a user to buy/sell stocks, view her portfolio information, modify her profile and perform other tasks like viewing a stock quote or her recent transactions. *StockTrader* follows the Model-View-Controller (MVC) architecture and all the components are deployed as web services. Figure 4.1(b) shows the component architecture and data-flow for the application. The components include a user facing front-end (FE), a business logic server (BS) that handles computation associated with most requests, the Order Service (OS) that handles buy and sell operations, a Database (DB) and a Config Service (CS) that binds these components. While the components interact amongst themselves (sometimes multiple times) to serve a user request, the precise data-flow and the components involved depends on the transaction type. The FE, BS, OS and CS were hosted as Azure web roles (small instances) while the DB was hosted on SQL Azure. We used the version of *StockTrader* from Apache Stonehenge Interoperability Project [109] and re-wrote parts of it to make it compatible with Azure cloud.

Twissandra: Unlike the applications described above, *Twissandra* is a social networking application similar to Twitter, providing a simple but core set of Twitter features. Figure 4.1(c) shows the application architecture of *Twissandra* consisting of a FE (Django



(a) *Thumbnail* (b) *StockTrader* (c) *Twissandra*

Fig. 4.2. CDF of total response-time for all applications, dissected by transaction types.

web server) and database (Cassandra cluster). Cassandra [9] is distributed storage system that is designed to scale to a very large size across many commodity servers, with no single point of failure and provides a simple schema-optional data model designed to allow good performance at scale. The FE receives user requests and initiates a request with a local Cassandra node to retrieve data from the DB cluster. Our deployment of Cassandra cluster consists of four nodes hosted on Amazon EC2 instances, and spread across two data-centers (the recommended multi data-center configuration) for availability and performance.

The workload for *Thumbnail* consists of fixed size 1.4 MB images. The workloads for *StockTrader* were obtained from the associated DaCapo benchmarks [110], which consists of several user sessions, each involving a series of transactions like login, view home page, view quote(s), buy/sell quotes, etc. Finally, for *Twissandra*, we made use of the Twitter Streaming API [111] to obtain a real data stream (Spritzer stream) to drive our experiments.

4.2.1 Performance variability in the cloud

We ran each application simultaneously in two separate data-centers (DC1 and DC2), both located in the United States, and subjected them to the same workload simultaneously. We instrumented each application to measure the total response-time, as well as the delays contributing to total response-time. The contributing delays include processing delays encountered at individual application components, communication delay between components (internal DC communication delays), and the upload/download delays (Internet communication delays between users and each DC). We now present our key findings:

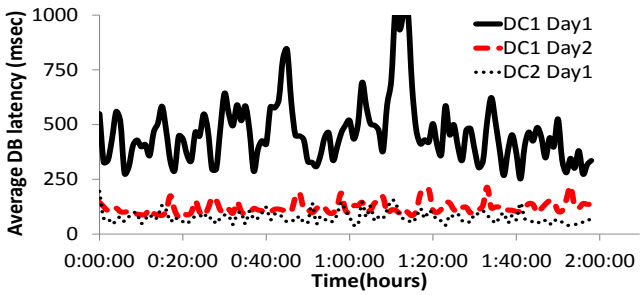


Fig. 4.3. Comparing the latency of DB transactions in DC1 and DC2 across two consecutive days. The curve for DC2 Day2 is very similar to DC2 Day1 and is therefore omitted.

User-perceived performance of applications in the cloud (total response-time): Figure 4.2 shows the CDF of the user-perceived performance for each of the three applications. Each graph shows the CDF of the total response-time, separated by different transaction types in the applications. From the figure, we observe that that there is significant variation in total response-time across all applications and transaction types. Further, the total response-time across all applications shows a significant tail highlighting the need for a system like *Dealer*.

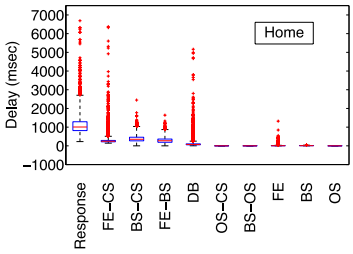
Performance of component replicas in multiple data centers is not correlated: Figure 4.3 shows a two hour snapshot from an experiment comparing the latency of database(DB) transactions for *StockTrader* across two consecutive days. The figure shows that the DB latency for DC1 on Day1 is significantly higher than on Day2, and has more prominent variation. The figure also shows that on Day1, the DB in DC2 performed significantly better than the DB in DC1 on the same day. This illustrates that the performance of similar components across multiple DCs is not correlated. Further investigation revealed that the performance variability was due to high load on the DC during a 9 day period [112]. Our interaction with the cloud providers indicated that during this period, different subsets of databases were impacted at different time snapshots.

Figure 4.4 shows the correlation coefficients across all pairs of elements, for *StockTrader*. In general, there is little correlation in performance across elements. In cases where there is some correlation, we found that the degradation in the performance of a downstream element in the application graph affected the upstream elements (e.g., degra-

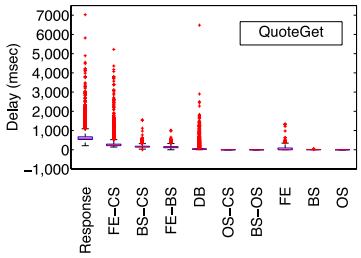
	FE	DB	BS	OS	FE-BS	FE-CS	BS-CS	BS-OS	OS-CS
FE	1	-0.08	-0.11	-0.04	-0.31	0.03	-0.32	-0.07	-0.04
DB		1	0.50	0.03	-0.01	-0.01	0.04	0.05	0.02
BS			1	0.14	0.08	-0.02	0.09	0.14	0.14
OS				1	-0.37	-0.03	-0.40	0.66	0.74
FE-BS					1	0.01	0.87	-0.31	-0.37
FE-CS						1	-0.01	-0.02	-0.03
BS-CS							1	-0.34	-0.41
BS-OS								1	0.71
OS-CS									1

(a) StockTrader

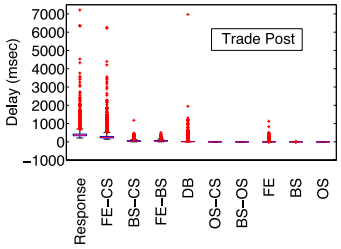
Fig. 4.4. Correlations across various *elements* of *StockTrader*. Values range between -1 (strongly anti-correlated) and +1 (strongly correlated).



(a) Home transactions



(b) Quote transactions



(c) Post transactions

Fig. 4.5. Boxplots showing total response-time and its constituent component and link delays (processing and communication delays) for 3 transaction types for *StockTrader*.

dation of BS-CS affected FE-BS in *StockTrader*). Similar trends were also observed for other applications.

All application components show performance variability: Figure 4.5 considers the *StockTrader* application and presents a box plot for the total response-time (first) and its per-component delays (component processing times and inter-component communication

latencies). The X-axis is annotated with the component or link whose delay is being measured. For example, FE-BS represents the delay between the Business-Logic server (BS) and the Front-End (FE) instances. The bottom and top of each box represent the 25^{th} and 75^{th} percentiles, and the line in the middle represents the median. The vertical line (whiskers) extends to the highest datum within 3*IQR of the upper quartile, where IQR is the inter-quartile range. Points larger than this value are considered outliers and shown separately. From the figure, we can see that while some components or links show more variation than others (e.g., FE-CS, DB in *StockTrader*), in general there is variability in all the components.

4.3 *Dealer* Design Rationale

In this section, we present the motivation behind *Dealer*'s design, and argue why traditional approaches don't suffice. *Dealer* is designed to enable applications meet their SLA requirements despite performance variations of cloud services. *Dealer* is motivated by two observations: (i) in any DC, only instances corresponding to a small number of application components see poor performance at any given time; and (ii) the latencies seen by instances of the same component located in different DCs are often uncorrelated.

Dealer's main goal is to dynamically identify a replica of each component that can best serve a given request. *Dealer* may choose instances located in different DCs for different components, offering a rich set of possible choices. In doing so, *Dealer* considers performance and loads of individual replicas, as well as intra- and inter-DC communication latencies.

Dealer is distinguished from DNS-based [105–107] and server-side [108] redirection mechanisms, which are widely used to map users to appropriate DCs. Such techniques focus on alleviating performance problems related to Internet congestion between users and DCs, or coarse-grained load-balancing at the granularity of DCs. *Dealer* is complementary and targets performance problems of individual cloud services inside a DC. There are several advantages associated with the *Dealer* approach:

• *Exploit heterogeneity in margins across different components:* In large multi-tier applications with potentially hundreds of components [104], only a few services might be temporarily impacted in any given DC. *Dealer* can reassign work related to these services to other replicas in remote DCs if they have sufficient margins. For instance, *Dealer* could tackle performance problems with storage elements (e.g., a blob) by using a replica in a remote DC, while leveraging compute instances locally. Complete request redirection, however, may not be feasible since instances of other components (e.g., business-logic servers) in the remote DC may not be over-provisioned adequately over their normal load to handle the redirected requests. In fact,

• *Utilize functional cloud services in each DC: Dealer* enables applications to utilize cloud services that are functioning satisfactorily in all DCs, while only avoiding services that are performing poorly. In contrast, techniques that redirect entire requests fail to utilize functional cloud services in a DC merely due to performance problems associated with a small number of other services. Further, the application may be charged for the unutilized services (for example, they may correspond to already pre-paid reserved compute instances [113]). While *Dealer* does incur additional inter DC communication cost, our evaluations in §4.5.4 indicate these costs are small.

• *Responsiveness:* Studies have shown that DNS-based redirection techniques may have latencies of over 2 hours and may not be well suited for applications which require quick response to link failures or performance degradations [114]. In contrast, *Dealer* targets adaptations over the time-scale of tens of seconds.

4.4 System Design

In this section, we present the design of *Dealer*. We begin by presenting an overview of the design, and then discuss its various components.

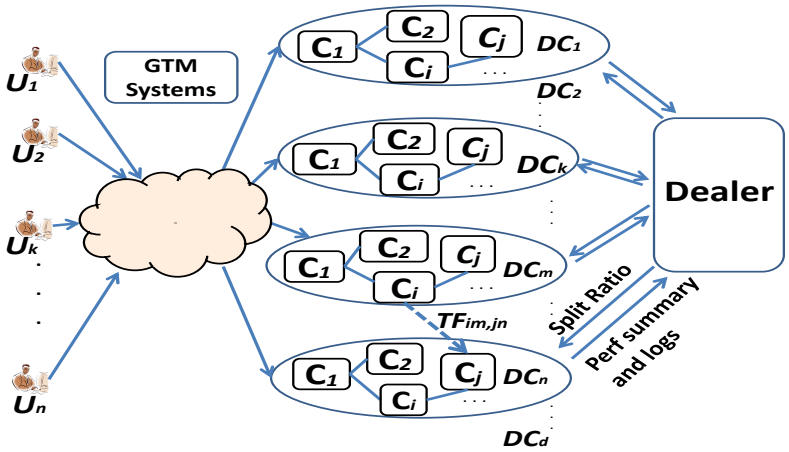


Fig. 4.6. System overview

4.4.1 System Overview

Consider an application with multiple components $\{C_1..C_l\}$. We consider a multi-cloud deployment where the application is deployed in d DCs, with instances corresponding to each component located in every one of the DCs. Note that there might be components like databases which are only present in one or a subset of DCs. We represent all replicas of component C_i in DC m as C_{im}.

Traffic from users is mapped to each DC using standard mapping services used today based on metrics such as geographical proximity or latencies [106]. Let U_k denote the set of users whose traffic is mapped to DC k. We refer to DC k as the primary DC for U_k, and to all other DCs as the secondary DCs. The excess capacity of each component replica is the additional load that can be served by that replica which is not being utilized for the primary traffic of that DC. Traffic corresponding to U_k can use the entire available capacity of all components in DC k, as well as the excess capacity of components in all other DCs.

For each user group U_k, *Dealer* seeks to determine how application transactions must be split in the multi-cloud deployment. In particular, the goal is to determine $TF_{im,jn}$, that is the number of user transactions that must be directed between component i in DC m to component j in DC n, for every pair of <component,DC > combinations. In doing so, the objective is to ensure the overall delay of transactions can be minimized. Further,

Dealer periodically recomputes how application transactions must be split given dynamics in behavior of cloud services.

Complex multi-tier applications may have hundreds of different transactions all involving different subsets of application components. Detailed knowledge of the components involved in every single type of transaction is hard to obtain. Instead, *Dealer* dynamically learns a model of the application that captures component interaction patterns. In particular, *Dealer* estimates the fraction of requests that involve communication between each pair of application components, and the average size of transactions between each component pair. In addition, *Dealer* estimates the processing delays of individual components replicas, and communication delays between components, as well as the available capacity of component replicas in each DC, (i.e., the load each replica can handle).

We now briefly discuss how this information is estimated and dynamically updated by *Dealer*.

4.4.2 Determining delays

There are three key components to the estimation algorithms used by *Dealer* when determining the processing delay of components and communication delays between them. These include: (i) passive monitoring of components and links over which application requests are routed; (ii) heuristics for smoothing and combining multiple estimates of delay for a link or component; and (iii) active probing of links and components which are not being utilized to estimate the delays that may be incurred if they were used. We describe each of these in turn:

Monitoring: Monitoring distributed applications is a well studied area [115–117], and we use *X-Trace* [116], since it can track application performance at the granularity of individual requests. To facilitate easy integration of *X-Trace* with the application, we automate a large part of the integration effort using *Aspect Oriented Programming (AOP)* techniques [118]. The measured latency at each component is reported periodically to a central monitor. A

smaller reporting time ensures greater agility of *Dealer*. We use reporting times of 10 seconds in our implementation, which we believe is reasonable.

Smoothing delay estimates: It is important to trade-off the agility in responding to performance dips in components or links with potential instability that might arise if the system is overly aggressive. To handle this, *Dealer* uses a weighted moving average (WMA) scheme. For each link and component, the average delay seen during the last W time windows of observation is considered. Briefly, the weight depends on the number of samples seen during a time window, and the recency of the estimate (i.e., recent windows are given a higher weight). Our empirical experience has shown choosing W values between 3 and 5 are most effective for good performance.

Probing: *Dealer* uses active probes to estimate the performance of components and links that are not currently being used. This enables *Dealer* to decide if it should switch transactions to a replica of a component in a different DC, and determine which replica must be chosen. Probe traffic is generated by test-clients using application workload generators (e.g., [119]). We restrict active probes to read-only requests that do not cause changes in persistent application state. While this may not accurately capture the delays for transactions involving writes, we have found the probing scheme to work well for the applications we experiment with. We also note that many applications tend to be read-heavy and it is often more critical to optimize latencies of transactions involving reads. To bound probes' overhead, we limit the probe rate to 10% of the application traffic rate. Also, *Dealer* probes 5% of the paths at random to ensure more choices can be explored.

While probing may add a non-negligible overhead on applications, we are investigating ways to restrict our use of active probing to only measuring inter-DC latency and bandwidth. The key insights behind our approach are to (i) use passive user-generated traffic to update component processing delays and inter-component link latencies [1]; and (ii) limit active probes to measuring inter-DC latency and bandwidth. These measurements can then be combined, along with passive measurements on transaction sizes observed between com-

[1]We expect each DC to continually receive some traffic which would ensure such passive observations are feasible.

ponents, to estimate the performance of any combination. Further, rather than having each application measure the bandwidth and latency between every pair of DCs, cloud providers could provide such services in the future, amortizing the overheads across all applications. We leave further exploration of this as future work.

4.4.3 Determining transaction split ratios

We now discuss how *Dealer* uses the processing delays of components and communication times of links to compute the split ratio matrix **TF**. Here, $TF_{im,jn}$ is the number of user transactions that must be directed between component i in DC m to component j in DC n, for every $<$component, DC $>$ pair. In determining the split ratio matrix, *Dealer* considers several factors including i) the total response-time; ii) stability of the overall system; and iii) capacity constraints of application components. A *combination* refers to an assignment of each component to exactly one DC. For e.g., in Figure 4.6, a mapping of C_1 to DC_1, C_2 to DC_k, C_i to DC_m and C_j to DC_m represents a combination. *Dealer* iteratively assigns a fraction of transactions to each combination. The split ratio matrix is easily computed once the fraction of transactions assigned to each combination is determined. The details of the assignment algorithm are available in our paper [120].

Considering total response-time: *Dealer* computes the mean delay for each possible combination like in [104]. It is the weighted sum of the processing delays of nodes and communication delay of links associated with that combination, where the weights are determined by the fraction of user transactions that traverse that node or link. The fractions may be determined by monitoring the application in its past window like in § 4.4.2. Once the delays of combinations are determined, *Dealer* sorts the combinations in ascending order of mean delay such that the best combinations get utilized the most, thereby ensuring a better performance.

Ensuring system stability: To ensure stability of the system and prevent oscillations, *Dealer* avoids abrupt changes in the split ratio matrix in response to minor performance changes. To achieve this, *Dealer* limits the maximum fraction of transactions that may be

assigned to a given combination. The limit (which we refer to as the damping ratio) is based on how well that combination has performed relative to others, and how much traffic was assigned to that combination in the recent past.

Honoring capacity constraints: In assigning transactions to a combination of application components, *Dealer* ensures the capacity constraints of each of the components is honored (details of the algorithm in [120]. Briefly, *Dealer* considers the combinations in ascending order of mean delay, and determines the maximum fraction of transactions that can be assigned to that combination without saturating any component. *Dealer* assigns this fraction of transactions to the combination, and updates the available capacities of each component to reflect this assignment. If the assignment of transactions is not completed at this point, the process is repeated with the next best combination.

4.4.4 Estimating capacity of components

We now discuss how *Dealer* determines the capacity of components in terms of the load each component can handle. Typically, application delays are not impacted by an increase in load up to a point which we term as the *threshold*. Beyond this, application delays increase gradually with load, until a breakdown region is entered where vastly degraded performance is seen. Ideally, *Dealer* must operate at the threshold to ensure the component is saturated while not resulting in degraded performance. The threshold is sensitive to transaction mix changes. Hence, *Dealer* dynamically estimates the threshold, and seeks to operate just above the threshold. If *Dealer* operated exactly at *thresh,* it would not be possible to know if *thresh* has increased, and hence discover if *Dealer* is operating too conservatively. The details of the capacity estimation algorithm are available in our paper [120]. Finally, while *Dealer* uses component delays to estimate if the component is saturated, one could also use other metrics such as CPU, memory utilization and queues sizes.

Algorithm 1 Integration with stateful applications.

Original code:

```
procedure SENDREQUEST(Component cmp, Request req)
    Replica replica ← cmp.Replica
    replica.Send(req)
end procedure
```

With *Dealer*:

```
procedure SENDREQUEST(Component cmp, Request req)
    Replica replica ← metaData[req.ID][cmp]
    if replica is null then                                  ▷ Not in meta-data.
        replica ← GetDealerReplica(cmp)                      ▷ Use Dealer suggestion.
        if cmp is stateful then ▷ Cmp is stateful but its information has not been propagated
yet in meta-data.
            metaData[req.ID][cmp] ← replica
        end if
    end if
    replica.Send(req)
end procedure
```

4.4.5 Integrating *Dealer* with applications

We integrated *Dealer* with both *Thumbnail* and *StockTrader*, and we found that the overall effort involved was small. Integrating *Dealer* with applications involves: i) adding logic to re-route requests to replicas of a downstream component across different DCs; and ii) maintaining consistent state in stateful applications.

Re-routing requests. To use *Dealer*, application developers need to make only a small change to the *connection logic* – the code segment inside a component responsible for directing requests to downstream components. *Dealer* provides both push and pull API's for retrieving the split ratios. Instead of forwarding all requests to a single service endpoint,

the connection logic now allocates requests to downstream replicas in proportion to the split ratios provided by *Dealer*.

Integration with stateful applications. While best practices emphasize that cloud applications should use stateless services whenever possible [121, 122], some applications may have stateful components. In such cases, the application needs to affinitize requests to component replicas so that each request goes to the replicas that hold the state for processing the request. Integrating *Dealer* with such applications does not change the consistency semantics of the application. *Dealer* does not try to understand the application's policy for allocating requests to components. Instead, it proposes the desired split ratios to the application, and the application uses its own logic to determine which replicas can handle a request.

In integrating *Dealer* with stateful applications, it is important to ensure that related requests get processed by the same set of stateful replicas due to data consistency constraints. For instance, the *StockTrader* application involves session state. To integrate *Dealer*, we made sure all requests belonging to the same user session use the same combination, and *Dealer*'s split-ratios only determine the combination taken by the first request of that session. *StockTrader* persists user session information (users logged in, session IDs, etc.) in a database. We modified the application so that it also stores the list of stateful replicas for each session. We also note that some web applications maintain the session state in the client side through session cookies. Such information could again be augmented to include the list of stateful replicas.

To guarantee all requests within the same session follow the same combination, the application must be modified to propagate *meta-data* (such as a unique session ID and the list of stateful replicas associated with it) along all requests between components. Many web applications (such as *StockTrader*) use SOAP and RESTful services that provide *Interceptors* which can be easily used to propagate meta-data with very minimal modifications. In the *StockTrader* application, we used *SOAP Extensions* [123] to propagate meta-data. In other cases where Interceptors cannot be used, endpoint interfaces can be changed or overloaded to propagate such data.

The propagated meta-data is used by components to guide the selection of downstream replicas. Algorithm 1 illustrates this. A component initiating a request must first check if the downstream component is stateful (by examining the meta-data), and if it is, it picks the replica specified in the meta-data. Otherwise, it picks the replica suggested by *Dealer*. If a downstream stateful component is visited for the first time, it picks the replica that *Dealer* suggests and saves this information into the meta-data which gets propagated along requests to the front-end.

While handling such state may require developer knowledge, we found this required only moderate effort from the developer in the applications we considered. As future work, we would like to integrate *Dealer* with a wider set of applications with different consistency requirements and gain more experience with the approach.

4.5 Experimental Evaluation

In this section, we evaluate the importance and effectiveness of *Dealer* in ensuring good performance of applications in the cloud. We begin by discussing our methodology in §4.5.1. We then evaluate the effectiveness of *Dealer* in responding to various events that occur naturally in a real cloud deployment (§4.5.2). These experiments both highlight the inherent performance variability in cloud environments, and evaluate the ability of *Dealer* to cope with them. We then evaluate *Dealer* using a series of controlled experiments which stress the system and gauge its effectiveness in coping with extreme scenarios such as sharp spikes in application load, failure of cloud components, and abrupt shifts in application transaction sizes.

4.5.1 Evaluation Methodology

We study and evaluate the design of *Dealer* by conducting experiments on *Thumbnail* and *StockTrader* (introduced in §4.2).

Cloud testbed and application workloads: All experiments were conducted on Microsoft Azure by deploying each application simultaneously in two DCs located geograph-

ically apart in the U.S. (North and South Central). In all experiments, application traffic to one of the DCs (referred to as DC_A) is controlled by *Dealer*, while traffic to the other one (DC_B) was run without *Dealer*. The objective was to not only study the effectiveness of *Dealer* in enhancing performance of traffic to DC_A, but also ensure that *Dealer* did not negatively impact performance of traffic to DC_B.

Application traffic to both DCs was generated using a Poisson arrival process when the focus of an experiment is primarily to study the impact of cloud performance variability. In *Thumbnail*, we set the transaction mix (fraction of requests to BL_1 and BL_2) according to the fraction of requests to Component1 and Component2 in the trace. Another key workload parameter that we did vary was the size of pictures uploaded by users. Requests in *Thumbnail* had an average upload size of 1.4 MB (in the form of an image) and around 3.2 (860) KB download size for BL_1 (BL_2) transactions. *StockTrader*, on the other hand, had a larger variety of transactions (buying/selling stocks, fetching quotes, etc.) with relatively smaller data size. To generate a realistic mix of transactions, we used the publicly available DaCapo benchmark [110], which contains a set of user sessions, with each session consisting of a series of requests (e.g., login, home, fetch quotes, sell stocks, and log out). A total of 66 PlanetLab users, spread across the U.S., were used to send requests to DC_A. Further, another set of users located inside a campus network were used to generate traffic to DC_B.

Application Deployments: Applications were deployed with enough instances of each component so that they could handle typical loads along with additional margins. We estimated the capacities of the components through a series of stress-tests. For instance, with an average load of $2\ \frac{req}{sec}$ and 100% margin (typical of real deployments as shown in §4.2), we found empirically that $2/5/16$ instances of FE/BL_1/BL_2 components were required. Likewise, for *StockTrader*, handling an average load of $1\ \frac{req}{sec}$ ($0.25\ \frac{session}{sec}$) required 1/2/1 instances of FE/BS/OS.

In *StockTrader*, we deployed the DB in both DCs and configured it in master-slave mode. We used SQL Azure Data Sync [124] for synchronization between the two databases. We note that *Dealer* can be integrated even if the application uses sharding or has weaker consistency requirements (§4.4.5) – the choice of master-slave is made for illustration pur-

poses. While reads can occur at either DB, writes are made only at the master DB (DC_B). Therefore, transactions involving writes (e.g., buy/sell) can only occur through the BS and OS instances in DC_B. Thus, the BS component would see a higher number of requests (by $\approx 20\%$) than the FE and therefore requires higher provisioning than FE. Further, each component can only connect to its local CS and DB to obtain communication credentials of other components. Finally, all requests belonging to a user session must use the same set of components given the stateful nature of the application.

Comparison with existing schemes: We evaluate *Dealer* against two prominent load-balancing and redirection techniques used today:

• *DNS-based redirection*: Azure provides *Windows Azure Traffic Manager (WATM)* [125] as its solution for DNS-based redirection. WATM provides *Failover*, *Round-Robin* and *Performance* distribution policies. Failover deals with total service failures and sends all traffic to the next available service upon failure. Round-robin routes traffic in a round-robin fashion. Finally, Performance forwards traffic to the closest DC in terms of network latency. In our experiments, we use the Performance policy because of its relevance to *Dealer*. In WATM, requests are directed to a single URL which gets resolved through DNS to the appropriate DC based on performance tables that measure the round trip time (RTT) of different IP addresses around the globe to each DC. We believe WATM is a good representative of DNS-based redirection schemes for global traffic management. However, its redirection is based solely on network latency and is agnostic to application performance. We therefore compare *Dealer* with another scheme that considers overall application performance.

• *Application-level Redirection*: We implemented a per-request load-balancer, that we call *Redirection*, which re-routes each request as a single unit, served completely by a single DC. *Redirection* re-routes requests based on the overall performance of the application, calculated as the weighted average of total response-time (excluding Internet delays) across all transactions. If it finds the local response time of requests higher than that of the remote DC, it redirects clients to the remote DC by sending a `302 HTTP` response message upon receiving a client request. It re-routes requests as long as the remote DC is performing bet-

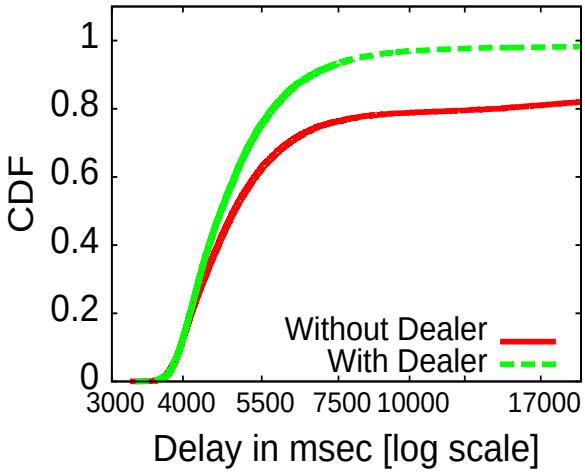


Fig. 4.7. CDF of total response-time under natural cloud dynamics.

ter, or until capacity limits are reached remotely (limited by the capacity of lowest margin component). Similar to *Dealer*, re-routing in *Redirection* does not depend on transaction types. We use the same monitoring and probing infrastructure described in §4.4.2.

4.5.2 *Dealer* under natural cloud dynamics

In this section, we evaluate the effectiveness of *Dealer* in responding to the natural dynamics of real cloud deployments. Our goal is to explore the inherent performance variability in cloud environments and evaluate the ability of *Dealer* to cope with such variability.

We experiment with *Thumbnail* and compare its performance with and without *Dealer*. Ideally it is desirable to compare the two schemes under identical conditions. Since this is not feasible on a real cloud, we ran a large number of experiments alternating between the two approaches. The experiment was 48 hours, with each hour split into two half-hour runs; one without activating *Dealer*, and another with it. Traffic was generated using a Poisson process with an average request rate of $2\ \frac{req}{sec}$ to each DC.

Figure 4.7 shows the CDF of the total response-time for the whole experiment. *Dealer* performs significantly better. The 50^{th}, 75^{th}, 90^{th}, and 99^{th} percentiles with *Dealer* are 4.6, 5.4, 6.6 and 12.7 seconds respectively. The corresponding values without *Dealer* are

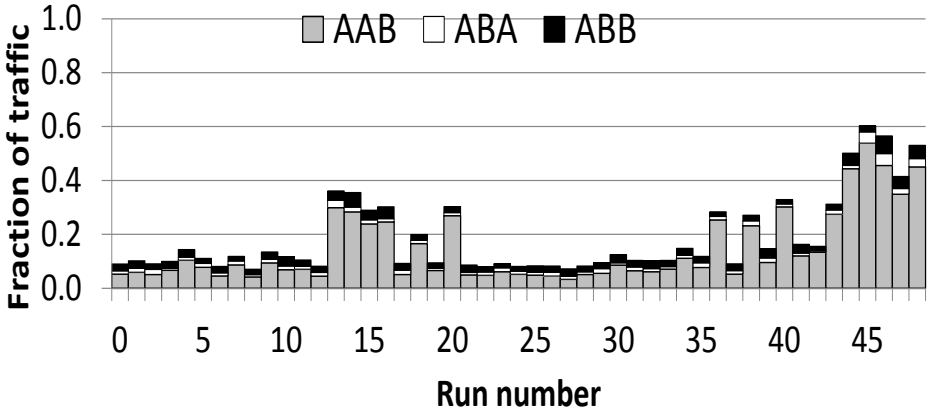


Fig. 4.8. Fraction of *Dealer* traffic sent from DC_A to DC_B.

4.9, 6.8, 43.2 and 90.9 seconds. The reduction is more than a factor of 6.5x for the top 10 percentiles.

Further investigation showed these high delays were caused by the BL instances in DC_A, which had lower capacity to absorb requests during those periods of high delay, and consequently experienced significant queuing. Such a sudden dip in capacity is an example of the kind of event that may occur in the cloud, and highlights the need for *Dealer*.

While *Dealer* too experienced the same performance problem with BL in DC_A, *Dealer* mitigated the problem by tapping into the margin available at DC_B. Figure 4.8 shows the fraction of requests directed to one or more components in DC_B by *Dealer*. Each bar corresponds to a run and is split according to the combination of components chosen by *Dealer*. Combinations are written as the location of FE, BE, BL_1 and BL_2 components[2] respectively, where A refers to DC_A and B to DC_B. For example, for run 0 around 9% of all requests handled by *Dealer* used one or more components from DC_B. Further, for this run, 5% of requests used the combination AAB, while 1% used ABA, and 3% used ABB. Further, most requests directed to DC_B during the problem take the path AAB, which indicates the BL component in DC_B is used.

Further, *Dealer*'s handles transient spikes in workload by directing transactions to the BL replica in DC_B. There were also some instances of congestion in the blob of DC_A which led *Dealer* to direct transactions to the blob of DC_B.

[2]Since all transactions in this experiment were of type BL_1, we drop the 4^{th} tuple.

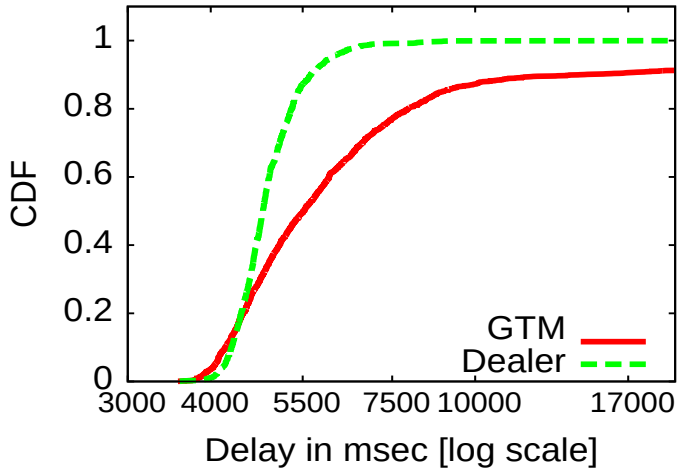


Fig. 4.9. CDF of total response-time for GTM vs. *Dealer* (*Thumbnail*).

4.5.3 *Dealer* vs. DNS-based redirection

Global Traffic Managers (GTM) are used to route user traffic across DCs to get better application performance and cope with failures. We conducted an experiment with the same setup mentioned in §4.5.2 to compare *Dealer* against WATM (§4.5.1). Figure 4.9 shows that *Dealer* achieves a reduction of at least 3x times in total response-time for the top 10 percentiles. Like before, we found the BL instances had lower capacity in some of the runs leading to a higher total response-time in GTM. Since the GTM approach only takes into account the network latency and not the application performance, it was unable to react to performance problems involving the BL instances.

4.5.4 *Dealer* vs. application-level redirection

In this section, we evaluate the effectiveness of *Dealer* in adapting to transient performance issues and compare its performance with application-level redirection described in §4.5.1.

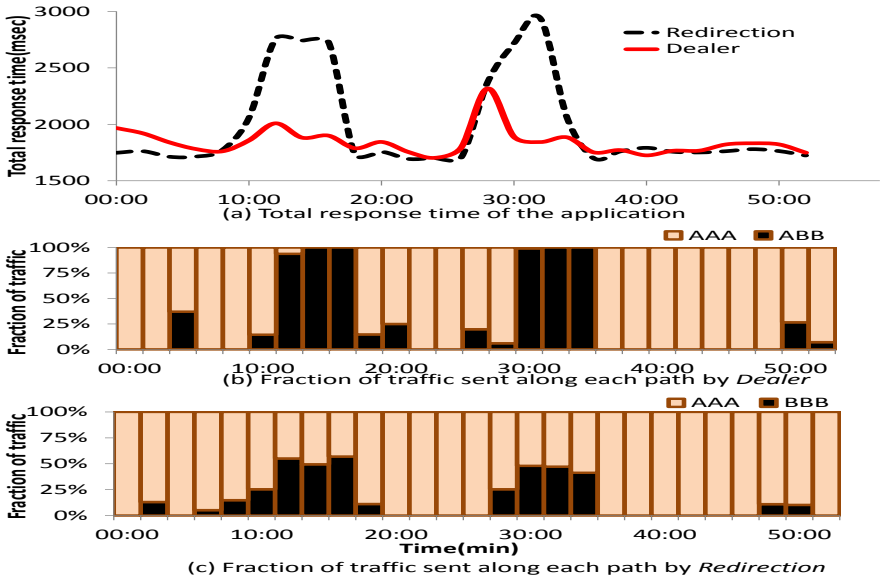


Fig. 4.10. Performance of *Dealer* vs. *Redirection* using traces collected during the DB performance issue. A combination (FE, BS, OS) is represented using the DC (DC_A or DC_B) to which each component belongs. 20% of transactions perform DB writes (combination ABB), hence we exclude them for better visualization.

Reaction to transient performance problems

We present our evaluation of *Dealer*'s response to performance variation in the cloud by deploying *StockTrader* at both DCs, using the master-slave mode as described in §4.5.1. We emulate a performance degradation in the database (DB) at DC_A using the traces we collected during the DB performance issue in §4.2.1 by taking a 10 minutes period with high DB latency and using the corresponding data points to induce delay at the DB.

Figure 4.10 shows that during the period of performance degradation at the DB (9-18th and 27-36th min), the average response time of *Dealer* is significantly better than that of *Redirection*. Figure 4.10(b) shows that *Dealer* takes ABB and switches requests over to the BS and OS at DC_B to avoid the high latency at DB. Similarly, Figure 4.10(c) shows the path (BBB) taken by *Redirection* and how this scheme switches a fraction of the requests entirely to the DC, DC_B. The fraction of traffic redirected to BBB in (c) is less than the fraction of traffic sent through ABB in (b). This is because *Dealer* is able to better utilize the margin available at the BS by switching a larger fraction of requests to the BS in DC_B.

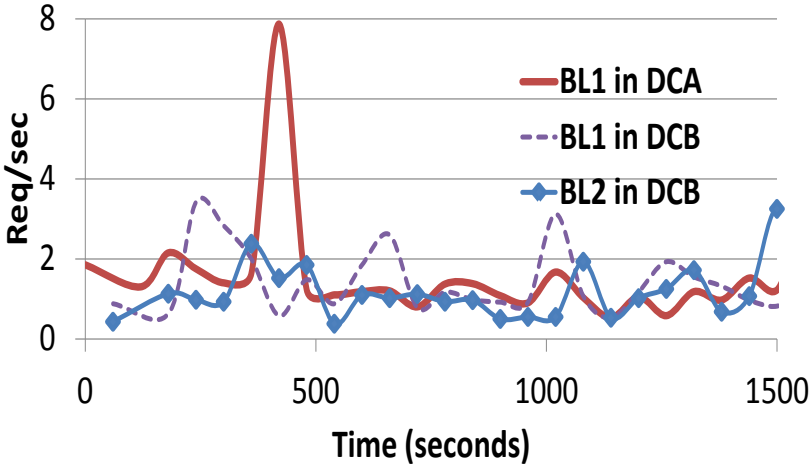


Fig. 4.11. Request rate for each component in both DCs. BL_2 in DC_A not shown for better visualization.

On the other hand, *Redirection* is constrained by the available capacity at the FE (DC_B) and hence is not able to completely utilize the margin available at the BS (DC_B).

Reaction to failures in the cloud

Applications in the cloud may see failures which reduce their margins, making them vulnerable to even modest workload spikes. Failures can happen due to actual physical outages or due to maintenance and upgrades. For example, Windows Azure's SLA states that a component has to have 2 or more instances to get 99.95% availability [126] as instances can be taken off for maintenance and upgrades at any time.

In Figure 4.12, we reproduced the case of a single fault-domain failure at time 300 affecting BL_2 instances in DC_B[3]. The combination AABA represents requests which were served by FE, BE, BL_2 at DC_A and BL_1 at DC_B. The increased response time is due to a surge in traffic at the DC_A. *Dealer* handled this surge by redirecting requests to the BL_1 replica in DC_B. *Redirection*, on the other hand, could not re-direct all excess traffic to DC_B since the other components did not have sufficient capacity in the remote DC to handle

[3]This involved bringing 4 BL_2 VM's offline since Azure deploys each component's VMs on 2 or more fault-domains.

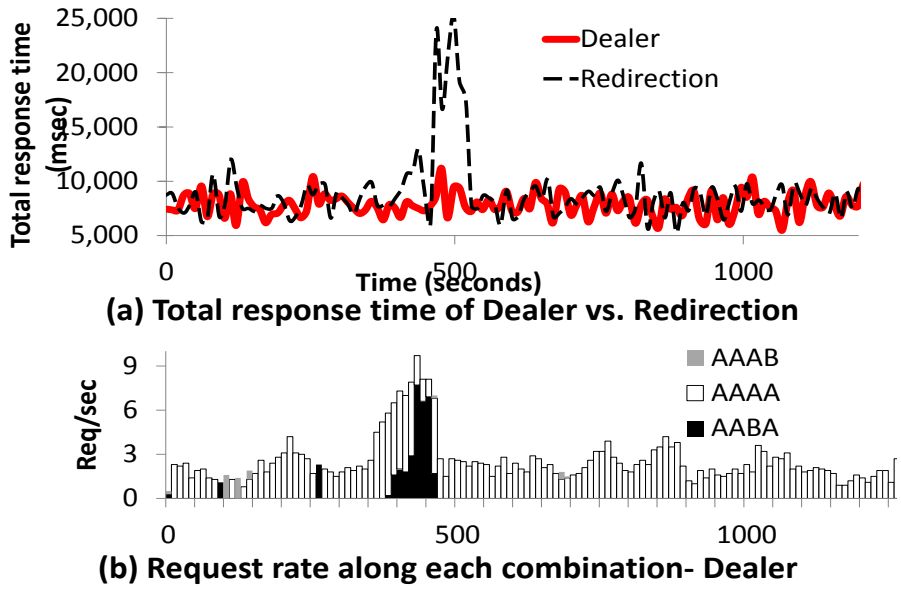


Fig. 4.12. Performance of *Dealer* vs. *Redirection* using real workload trace with cloud failures (*Thumbnail*).

all the load from DC_A. Therefore, *Dealer* maintained a significantly lower response time during the surge in workload (130% lower). The results show that *Dealer* is effective in handling failures in the cloud.

Inter DC bandwidth costs

A potential concern arises due to wide-area traffic that *Dealer* introduces in re-routing requests across DCs. In this section, we compute the cost percentage increase for *Thumbnail* and *StockTrader* based on the experiments described in §4.5.4.

We consider the bandwidth, storage and compute (small instances) costs based on Microsoft Azure tariffs in January, 2012. The bandwidth cost is based on all transactions exiting each DC (incoming transactions do not incur bandwidth costs in Azure). The average size of each request in *Thumbnail* (*StockTrader*) is 1.5MB (2 KB). *StockTrader* uses SQL Azure DB (Web Ed.) and *Thumbnail* uses Azure blobs for storage. We calculate the storage cost for *Thumbnail* based on the number of storage transactions and storage size

consumed. The cost of the DB and compute instances is normalized to the duration of the experiments.

The cost percentage increase for *Thumbnail* and *StockTrader* were found to be 1.94% and 0.06% respectively. This shows that the cost introduced due to inter DC bandwidth is minimal, even for data-intensive applications such as *Thumbnail*. We have repeated our calculations using the Amazon EC2 pricing scheme [113], and we have found similar results. Finally, we note that in our evaluations we assume compute instances in both DCs cost the same. However, in practice, application architects are likely to provision *reserved instances* in each DC [113] (i.e., instances contracted over a longer period for a lower rate). Under such scenarios, *Dealer* has the potential to incur lower costs than *Redirection* by leveraging reserved instances in each DC to the extent possible.

4.6 Related Work

Several researchers have pointed out the presence of performance problems with the cloud (e.g., [20–22]). In contrast, our focus is on designing systems to adapt to short-term variability in the cloud.

The cloud industry already provides mechanisms to scale up or down the number of server instances in the cloud (e.g., [127, 128]). However, it takes tens of minutes to invoke new cloud instances in commercial cloud platforms today. Recent research has shown the feasibility of starting new VMs at faster time scales [129,130]. For instance, [129] presents a VM-fork abstraction which enables the cloning of a VM into multiple replicas on-the-fly. While such schemes are useful for handling variability in performance due to excess load on a component, they cannot handle all types of dynamics in the cloud (e.g., problems in blob storage, network congestion, etc.). Further, ensuring the servers are warmed up to serve requests after instantiation (e.g., by filling caches, running checks, copying state, etc.) demands additional time. In contrast, *Dealer* can enable faster adaptation at shorter time-scales, and is intended to complement solutions for dynamic resource invocation.

DNS-based techniques [105–107] and server-side redirection mechanisms [108] are widely used to map users to appropriate DCs. However, such techniques focus on alleviating performance problems related to Internet congestion between users and DCs, and load-balance user traffic coarsely at the granularity of DCs. In contrast, *Dealer* targets performance problems of individual cloud components inside a DC, and may choose components that span multiple DCs to service an individual user request. This offers several advantages in large multi-tier applications (with potentially hundreds of components [104]) where possibly only a small number of components are temporarily impacted. When entire user requests are redirected to a remote DC as in [105–108], not all components in the remote DC may be sufficiently over-provisioned to handle the redirected requests. Further, redirecting entire user requests does not utilize functional resources in the local DC that have already being paid for. For instance, the local DC may have underutilized *reserved* instances [113], while the remote DC might require the use of more expensive *on-demand* instances. The cost could be substantial over a large number of components. Finally, studies have shown that the use of DNS-based redirection techniques may lead to delays of more than 2 hours and thus may not be suitable for applications which require quick response to failures [114]. We note that [108] does mention doing the redirection at the level of the bottleneck component; however, *Dealer* is distinguished in that it makes no apriori assumption about which component is the bottleneck, and dynamically reacts to whichever component or link performs poorly at any given time.

Several works [131–133] study utility resource planning and provisioning for applications. [131] studies resource planning for compute batch tasks by building predictive models in shared computing utilities. Further, [132, 133] build analytic models for handling workload variability (changing transaction mix and load) in multi-tier applications. For example, [132] aims at handling peak workloads by provisioning resources at two levels; predictive provisioning that allocates capacity at the time-scale of hours or days, and reactive provisioning that operates at time scales of minutes. While such techniques are complementary to *Dealer*, their focus is not applications deployed in public clouds. *Dealer* not only deals with workload variability, but also handles all types of performance vari-

ability (e.g., due to service failures, network congestion, etc.) in geo-distributed multi-tier applications, deployed in commercial public clouds. *Dealer* provides ways to avoid components with poor performance and congested links via re-routing requests to replicas in other DCs at short time scales.

Other works [134, 135] study the performance of multi-tier applications. [134] tries to control the performance of such applications by preventing overload using self-tuning proportional integral (PI) controller for admission control. Such a technique can be integrated with *Dealer* to control the load directed to each component replica. Further, [135] combines performance modeling and profiling to create analytical models to accomplish SLA decomposition. While SLA decomposition is outside the scope of *Dealer*, component profiling may be incorporated with *Dealer* to capture component's performance as a function of allocated resources (e.g., CPU) to achieve performance prediction.

4.7 Conclusions

In this chapter, we have shown that it is important and feasible to architect latency-sensitive applications in a manner that is robust to the high variability in performance of cloud services. We have presented *Dealer*, a system that can enable applications to meet their SLA requirements by dynamically splitting transactions for each component among its replicas in different DCs. Under natural cloud dynamics, the $90th$ and higher percentiles of application response times were reduced by more than a factor of 3 compared to a system that used traditional DNS-based redirection. Further, *Dealer* not only ensures low latencies but also significantly out-performs application-level redirection mechanisms under a range of controlled experiments.

5. CONCLUSIONS AND FUTURE WORK

As commercial cloud deployments scale by building new datacenters, more applications and their content are expected move to the cloud for better performance and availability, which present many opportunities for improving user experience. While many techniques and systems like SPDY, DNS-based request redirection, and geo-replicated datastores have been proposed to improve application performance, these solutions often tend to improve the performance of a specific application component, and may have different impact on the user perceived latency. This thesis takes the first steps towards improving user experience in a holistic manner, by presenting solutions that helps reduce the front-end and back-end latency, while simultaneously ensuring reductions in the end-to-end user perceived latency.

5.1 Contributions

This thesis makes the following contributions. First, it reduces the user-facing latency by priority-aware organization of web content within the CDNs. Next, it reduces the back-end latency through optimal data-replication and placement at the storage layer. Finally, it improves the application performance by carefully rerouting requests across different application component replicas.

Improving application performance by page-aware prioritization of content within CDNs: The quest to reduce user-perceived web-application latencies has led to the large-scale adoption of widely distributed CDNs that involve placing caches at thousands of Internet vantage points, close to users. With more applications opting for all-content delivery through CDNs, there is an opportunity to improve the end-user experience by optimizing the delivery of entire web pages, rather than just individual objects. The key idea behind our approach is that the objects in a web page having the largest impact on page latency should be served out of the closest or fastest caches in the hierarchy. We presented a family

of prioritization schemes for identifying important objects for the page and develop mechanisms that serve them with higher priority from the CDN, while balancing traditional CDN concerns such as optimizing the delivery of popular objects and minimizing bandwidth costs. Through extensive experiments on 100 real world pages (across all popularities from Alexa Top pages), we showed that latency reductions of over 100ms can be obtained for more than 50% of the pages.

Balancing latency, availability and replication-cost in geo-distributed datastores: In response to the stringent latency and availability requirements of modern applications, several geo-distributed cloud datastores have emerged in recent years. Despite the presence of many tools and techniques that improve the performance of these datastores, meeting application SLAs is still challenging, given the scale of applications, and their diverse and dynamic workloads which are not entirely exposed to the datastores. Using application level data access patterns as a focal input, we developed data-replication models for quorum-based systems that optimize percentiles of response time under normal operation and under a datacenter (DC) failure. These models also consider various factors like the geographic spread of users, DC locations, consistency requirements and inter-DC communication costs when determining data placements. We showed the benefits of our approach using real-world traces of three geo-distributed applications: Twitter, Wikipedia and Gowalla deployed along with a Cassandra cluster across the 8 DCs in Amazon EC2.

Application-aware request splitting for multi-tier cloud applications: A key factor affecting the performance of applications deployed in the cloud is the high variability in performance of cloud services. To understand the impact of performance variability on user perceived performance, we conducted extensive measurement studies using three real-world applications deployed on commercial cloud platforms like Windows Azure and Amazon AWS. Leveraging on insights from our measurement study, we built Dealer a system that helps geo-distributed, multi-tier applications meet their stringent requirements on response time in the presence of these variabilities. Dealer is motivated by the fact that, at any time, only a small number of application components of large multi-tier applications experience poor performance. It abstracts application structure as a component graph, with

nodes being application components and edges capturing inter-component communication patterns. Dealer continually monitors the performance of individual component replicas and communication latencies between replica pairs, and seeks to minimize user response times by picking the best combination of replicas (potentially located across different DCs). By integrating Dealer with two multi-tier applications on real cloud deployments, we show that the 90th percentile of application response times could be reduced by a factor of 3 under natural cloud dynamics compared to the conventional, application-structure-agnostic redirection techniques.

5.2 Future directions

Mobile web application performance: Mobile applications present unique challenges due to their higher last-mile latencies and diversity in client conditions. The performance of mobile applications depends on the two key contributing factors - compute and network activities of the applications. Our priority-based schemes can be potentially extended to help schedule these activities in a priority aware manner to optimize the overall performance. The key idea here is to make both the client and server-side application (including third party components) aware of its impact on the user performance by explicitly modeling its deadlines and priorities. This allows the server to serve content to the client in a timely manner, while being aware of the client conditions and requirements. Such models can also easily incorporate various practical aspects like bandwidth cost, power and policies as constraints while constructing the schedules.

Priority-based push for improving end-user experience: Protocols like SPDY provide an important feature called server push that has the potential to reduce page-load latency, by pushing objects to the client without waiting for explicit client requests. However, a key challenge with enabling server push is the extensive personalization of web pages. Specifically, web pages are extensively customized to individual users, and identifying the objects for a given user typically involves executing JS code. In addition, even across back-to-back runs for a given user, there is significant variability in the fetched objects, and identifying

objects may require executing JS code. Thus, server push today is restricted to static objects (i.e., objects that are common to all users and across runs of a single user which limits its potential). However, recent efforts like PARCEL [136], that perform redundant web-page execution at the proxy, have opened up the opportunity for improving the accuracy in push schemes. The spectrum of prioritization and proactive refresh strategies proposed in this thesis can be extended to these proxies, which can significantly lower the page-load latency and improve the user experience.

LIST OF REFERENCES

LIST OF REFERENCES

[1] J. Hamilton, "The cost of latency." http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx, 2009.

[2] B. Forrest, "Bing and google agree: Slow pages lose users." http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html, 2009.

[3] "Latency - it costs you." http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it.

[4] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[5] S. Souders, "Velocity and the bottom line." http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html, 2009.

[6] "Powers of 10: Time Scales in User Experience." http://www.nngroup.com/articles/powers-of-10-time-scales-in-ux/.

[7] D. Hastorun *et al.*, "Dynamo: amazons highly available key-value store," in *Proc. SOSP*, 2007.

[8] J. C. Corbett *et al.*, "Spanner:google's globally-distributed database," in *Proceedings of the OSDI*, 2012.

[9] A. Lakshman and P. Malik, "Cassandra:a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, pp. 35–40, 2010.

[10] R. Escriva, B. Wong, and E. G. Sirer, "Hyperdex:a searchable distributed key-value store," in *Proc. SIGCOMM*, 2012.

[11] B. F. Cooper *et al.*, "Pnuts: Yahoo!'s hosted data serving platform," in *Proceedings of the VLDB*, 2008.

[12] W. Lloyd *et al.*, "Stronger semantics for low-latency geo-replicated storage," *NSDI 2013*.

[13] J. Baker *et al.*, "Megastore:providing scalable, highly available storage for interactive services," in *Proc. CIDR*, 2011.

[14] "More 9s please: Under the covers of the high replication datastore." http://www.google.com/events/io/2011/sessions/more-9s-please-under-the-covers-of-the-high-replication-datastore.html.

[15] Google, "SPDY: An experimental protocol for a faster web." http://goo.gl/vy63I4.

[16] S. Ihm and V. S. Pai, "Towards understanding modern web traffic," in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pp. 295–312, ACM, 2011.

[17] M. Butkiewicz, H. V. Madhyastha, and V. Sekar, "Understanding website complexity: measurements, metrics, and implications," in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pp. 313–328, ACM, 2011.

[18] "Amazon cloud outage." http://aws.amazon.com/message/2329B7/.

[19] "Microsoft Live outage due to DNS corruption." http://windowsteamblog.com/windows_live/b/windowslive/archive/2011/09/20/follow-up-on-the-sept-8-service-outage.aspx.

[20] G. Wang and T. S. E. Ng, "The impact of virtualization on network performance of Amazon EC2 data center," in *IEEE INFOCOM 2010*.

[21] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: comparing public cloud providers," in *IMC 2010*.

[22] S. Barker and P. Shenoy, "Empirical evaluation of latency-sensitive application performance in the cloud," in *MMSys 2010*.

[23] D. K. Gifford, "Weighted voting for replicated data," in *Proc. SOSP*, 1979.

[24] A. W. Fu, "Delay-optimal quorum consensus for distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8(1), 1997.

[25] T. Tsuchiya *et al.*, "Minimizing the maximum delay for reaching consensus in quorum-based mutual exclusion schemes," *IEEE TPDS*, 1999.

[26] F. Oprea and M. K. Reiter, "Minimizing response time for quorum-system protocols over wide-area networks," in *Proc. DSN*, 2007.

[27] T. Hoff, "Latency is everywhere and it costs you sales-how to crush it." http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it, 2009.

[28] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "Demystify page load performance with wprof," in *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[29] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. G. Greenberg, and Y.-M. Wang, "Webprophet: Automating performance prediction for web services.," in *NSDI*, vol. 10, pp. 10–10, 2010.

[30] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar, "Klotski: Reprioritizing web content to improve user experience on mobile devices," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, (Oakland, CA), USENIX Association, May 2015. To appear in.

[31] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell, "A hierarchical internet object cache," in *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, (Berkeley, CA, USA), pp. 13–13, USENIX Association, 1996.

[32] Alexa. Available at `http://www.alexa.com/topsites`.

[33] "SPDY best practices." `http://www.chromium.org/spdy/spdy-best-practices`.

[34] S. Souders, "Onload event and post-onload requests." `http://www.stevesouders.com/blog/2012/10/30/qa-nav-timing-and-post-onload-requests`.

[35] "Google Speed-index." `https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index`.

[36] M. Butkiewicz, Z. Wu, S. Li, P. Murali, V. Hristidis, H. V. Madhyastha, and V. Sekar, "Enabling the transition to the mobile web with websieve," in *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*, p. 14, ACM, 2013.

[37] P. Cao and S. Irani, "Cost-aware www proxy caching algorithms.," in *Usenix symposium on internet technologies and systems*, vol. 12, pp. 193–206, 1997.

[38] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "How speedy is spdy?," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, (Berkeley, CA, USA), pp. 387–399, USENIX Association, 2014.

[39] web-page replay, "Record and play back web pages with simulated network conditions." `https://www.code.google.com/p/web-page-replay/`.

[40] "Chrome command-line switches." `http://goo.gl/7t5nk5`.

[41] "RAMDisk for faster browsing." `http://goo.gl/qtbDTe`.

[42] H. B. Mann *et al.*, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, 1947.

[43] A. Hart, "Mann-whitney test is not just a test of medians: differences in spread can be important," *BMJ: British Medical Journal*, vol. 323, no. 7309, p. 391, 2001.

[44] "SPDY Protocol(draft3)-stream-priority." `http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3#TOC-2.3.3-Stream-priority`.

[45] J. Erman, V. Gopalakrishnan, R. Jana, and K. K. Ramakrishnan, "Towards a spdy'ier mobile web?," in *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, (New York, NY, USA), pp. 303–314, ACM, 2013.

[46] S. Jin and A. Bestavros, "Popularity-aware greedy dual-size web proxy caching algorithms," in *Distributed computing systems, 2000. Proceedings. 20th international conference on*, pp. 254–261, IEEE, 2000.

[47] M. Arlitt, R. Friedrich, and T. Jin, *Performance evaluation of web proxy cache replacement policies*. Springer, 1998.

[48] J. Wang, "A survey of web caching schemes for the internet," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 5, pp. 36–46, 1999.

[49] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, and E. A. Fox, "Caching proxies: Limitations and potentials," 1995.

[50] R. P. Wooster and M. Abrams, "Proxy caching that estimates page load delays," *Computer Networks and ISDN Systems*, vol. 29, no. 8, pp. 977–986, 1997.

[51] J.-C. Bolot and P. Hoschka, "Performance engineering of the world wide web: Application to dimensioning and cache design," *Computer Networks and ISDN Systems*, vol. 28, no. 7, pp. 1397–1405, 1996.

[52] V. N. Padmanabhan and J. C. Mogul, "Using predictive prefetching to improve world wide web latency," *ACM SIGCOMM Computer Communication Review*, vol. 26, no. 3, pp. 22–36, 1996.

[53] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin, "The potential costs and benefits of long-term prefetching for content distribution," *Computer Communications*, vol. 25, no. 4, pp. 367–375, 2002.

[54] R. Kokku, P. Yalagandula, A. Venkataramani, and M. Dahlin, "A non-interfering deployable web prefetching system," in *In Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems*, Citeseer, 2002.

[55] Y. Jiang, M.-Y. Wu, and W. Shu, "Web prefetching: Costs, benefits and performance," in *Proceedings of the 7th international workshop on web content caching and distribution (WCW2002). Boulder, Colorado*, Citeseer, 2002.

[56] B. Wu and A. D. Kshemkalyani, "Objective-optimal algorithms for long-term web prefetching," *Computers, IEEE Transactions on*, vol. 55, no. 1, pp. 2–17, 2006.

[57] H. Che, Y. Tung, and Z. Wang, "Hierarchical web caching systems: Modeling, design and experimental results," *Selected Areas in Communications, IEEE Journal on*, vol. 20, no. 7, pp. 1305–1314, 2002.

[58] M. R. Korupolu and M. Dahlin, "Coordinated placement and replacement for large-scale distributed caches," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 14, no. 6, pp. 1317–1329, 2002.

[59] R. Tewari, M. Dahlin, H. M. Vin, and J. S. Kay, "Design considerations for distributed caching on the internet," in *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*, pp. 273–284, IEEE, 1999.

[60] A. Venkataramani, P. Weidmann, and M. Dahlin, "Bandwidth constrained placement in a wan," in *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pp. 134–143, ACM, 2001.

[61] "Google app engine - transactions across datacenters." `http://www.google.com/events/io/2009/sessions/TransactionsAcrossDatacenters.html`.

[62] L. Qiu *et al.*, "On the placement of web server replicas," in *Proceedings of IEEE INFOCOM 2001*.

[63] W. Lloyd *et al.*, "Don't settle for eventual: Scalable causal consistency for wide-area storage with cops," in *Proc. SOSP*, 2011.

[64] Y. Sovran *et al.*, "Transactional storage for geo-replicated systems," in *Proc. of SOSP*, ACM, 2011.

[65] "Google groups for App Engine Downtime Notification." `https://groups.google.com/forum/?fromgroups=#!forum/google-appengine-downtime-notify`.

[66] "Facebook's master slave data storage." `http://www.facebook.com/note.php?note_id=23844338919`.

[67] N. Bronson *et al.*, "Tao: Facebook's distributed data store for the social graph," in *USENIX ATC*, 2013.

[68] T. Kraska *et al.*, "Mdcc: Multi-data center consistency," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ACM, 2013.

[69] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[70] P. Bailis *et al.*, "Probabilistically bounded staleness for practical partial quorums," in *Proc. VLDB*, 2012.

[71] A. Su *et al.*, "Drafting behind Akamai," *SIGCOMM 2006*.

[72] P. N. Shankaranarayanan *et al.*, "Balancing latency and availability in geo-distributed cloud data stores," *Purdue University ECE Technical Reports TR-ECE-13-03*, 2013.

[73] "IBM ILOG CPLEX." `http://www-01.ibm.com/software/integration/optimization/cplex/`.

[74] A. I. Barvinok, *A course in convexity*. American Mathematical Society, 2002.

[75] D. Ford *et al.*, "Availability in globally distributed storage systems," in *Proc. of OSDI*, 2010.

[76] P. Gill *et al.*, "Understanding network failures in data centers: Measurement, analysis, and implications," in *Proc. of SIGCOMM*, 2011.

[77] U. G. Knight, *Power Systems in Emergencies: From Contingency Planning to Crisis Management*. John Wiley & Sons, LTD, 2001.

[78] James Hamilton, "Inter-Datacenter Replication and Geo-Redundancy." `http://perspectives.mvdirona.com/2010/05/10/InterDatacenterReplicationGeoRedundancy.aspx`.

[79] DataStax, "Planning an Amazon EC2 cluster." `http://www.datastax.com/documentation/cassandra/1.2/webhelp/cassandra/architecture/architecturePlanningEC2_c.html`.

[80] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT Newsletter*, 2002.

[81] R. Li *et al.*, "Towards social user profiling: unified and discriminative influence model for inferring home locations," in *KDD*, 2012.

[82] "Wikimedia statistics." `http://stats.wikimedia.org/`.

[83] E. Cho *et al.*, "Friendship and mobility: user movement in location-based social networks," in *Proceedings of the SIGKDD*, 2011.

[84] J. Bisschop *et al.*, "On the development of a general algebraic modeling system in a strategic planning environment," *Applications*, 1982.

[85] "Twitter application uses key-value data store." `http://engineering.twitter.com/2010/07/cassandra-at-twitter-today.html`.

[86] "Stanford large network dataset collection." `http://snap.stanford.edu/data/loc-gowalla.html`.

[87] "Geocoding in ArcGIS." `http://geocode.arcgis.com/arcgis/index.html`.

[88] "Aws edge locations." `http://aws.amazon.com/about-aws/globalinfrastructure/`.

[89] J. M. Pujol *et al.*, "The little engine(s) that could: Scaling online social networks," in *Proc. SIGCOMM*, 2010.

[90] S. Agarwal *et al.*, "Volley:automated data placement for geo-distributed cloud services," in *Proc. NSDI*, 2010.

[91] B. Cho and M. K. Aguilera, "Surviving congestion in geo-distributed storage systems," in *Proc. of USENIX ATC*, 2012.

[92] Z. Wu *et al.*, "Cost-effective geo-replicated storage spanning multiple cloud services," in *Proc. of SOSP*, 2013.

[93] D. Sciascia and F. Pedone, "Geo-replicated storage with scalable deferred update replication," in *IEEE/IFIP DSN*, 2013.

[94] H. Garcia-molina and D. Barbara, "How to assign votes in a distributed system," *Journal of the Association for Computing Machinery*, 1985.

[95] D. Barbara and H. Garcia-Molina., "The reliability of voting mechanisms," *IEEE Transactions on Computers*, vol. 36(10), October 1987.

[96] Y. Amir and A. Wool, "Evaluating quorum systems over the internet," in *Proc. of FTCS*, pp. 26–35, 1996.

[97] F. Junqueira and K. Marzullo, "Coterie availability in sites," in *Proc. DISC*, 2005.

[98] M. M.G., F. Oprea, and M. Reiter, "When and how to change quorums on wide area networks," in *Proc. SRDS*, 2009.

[99] S. Y. Cheung *et al.*, "Optimizing vote and quorum assignments for reading and writing replicated data," in *Proc. of the ICDE*, 1989.

[100] Symantec, "2010 State of the Data Center Global Data." http://www.symantec.com/content/en/us/about/media/pdfs/Symantec_DataCenter10_Report_Global.pdf.

[101] M. Armbrust *et al.*, "Above the Clouds: A Berkeley View of Cloud Computing," tech. rep., EECS, University of California, Berkeley, 2009.

[102] D. Hastorun *et al.*, "Dynamo: amazons highly available key-value store," in *In Proc. SOSP*, 2007.

[103] "Response Time Metric for SLAs." http://testnscale.com/blog/performance/response-time-metric-for-slas.

[104] M. Hajjat *et al.*, "Cloudward bound: Planning for beneficial migration of enterprise applications to the cloud," *SIGCOMM 2010*.

[105] J. Dilley *et al.*, "Globally distributed content delivery," *Internet Computing, IEEE*, 2002.

[106] A. Su *et al.*, "Drafting behind Akamai," *SIGCOMM 2006*.

[107] P. Wendell *et al.*, "DONAR: decentralized server selection for cloud services," in *SIGCOMM 2010*.

[108] S. Ranjan, R. Karrer, and E. Knightly, "Wide area redirection of dynamic content by Internet data centers," in *INFOCOM 2004*.

[109] "Apache, Project Stonehenge." http://wiki.apache.org/incubator/StonehengeProposal.

[110] S. M. Blackburn and et al., "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA 2006*.

[111] Twitter Streaming APIs. https://dev.twitter.com/docs/streaming-apis.

[112] "SQL Azure Performance Issue." http://cloudfail.net/513962.

[113] "Amazon EC2 pricing." http://aws.amazon.com/ec2/pricing/.

[114] J. Pang *et al.*, "On the responsiveness of DNS-based network control," in *IMC 2004*.

[115] P. Barham *et al.*, "Magpie: Online modelling and performance-aware systems," in *HOTOS 2003*.

[116] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *NSDI 2007*.

[117] "Event Tracing for Windows (ETW)." http://msdn.microsoft.com/en-us/library/aa363668.aspx.

[118] "Aspect Oriented Programming." http://msdn.microsoft.com/en-us/library/aa288717%28v=vs.71%29.aspx.

[119] "Grinder Load Testing Framework." http://grinder.sourceforge.net/index.html.

[120] M. Hajjat, S. P. N, D. Maltz, S. Rao, and K. Sripanidkulchai, "Dealer: Application-aware Request Splitting for Interactive Cloud Applications," in *ACM CoNEXT 2012*.

[121] "Coding in the Cloud. Use a stateless design whenever possible.." http://www.rackspace.com/blog/coding-in-the-cloud-rule-3-use-a-stateless-design-whenever-possible/.

[122] "Architecting for the Cloud: Best Practices.." http://jineshvaria.s3.amazonaws.com/public/cloudbestpractices-jvaria.pdf.

[123] "Using SOAP Extensions in ASP.NET." http://msdn.microsoft.com/en-us/magazine/cc164007.aspx.

[124] "SQL Azure Data Sync." http://social.technet.microsoft.com/wiki/contents/articles/sql-azure-data-sync-overview.aspx.

[125] "Windows Azure Traffic Manager (WATM)." http://msdn.microsoft.com/en-us/gg197529.

[126] "Windows Azure SLA." http://www.microsoft.com/windowsazure/sla/.

[127] RightScale Inc., "Cloud computing management platform." http://www.rightscale.com.

[128] "Microsoft Windows Azure." http://www.microsoft.com/windowsazure/.

[129] Lagar-Cavilla *et al.*, "SnowFlock: rapid virtual machine cloning for cloud computing," in *ACM EuroSys, 2009*.

[130] M. Vrable *et al.*, "Scalability, fidelity, and containment in the potemkin virtual honeyfarm," in *ACM SOSP, 2005*.

[131] P. Shivam, S. Babu, and J. Chase, "Learning application models for utility resource planning," in *ICAC'06*.

[132] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal, "Dynamic provisioning of multi-tier internet applications," in *ICAC 2005*.

[133] Q. Zhang, L. Cherkasova, and E. Smirni, "A regression-based analytic model for dynamic resource provisioning of multi-tier applications," in *ICAC'07*.

[134] A. Kamra, V. Misra, and E. Nahum, "Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites," in *IWQOS 2004*.

[135] Y. Chen, S. Iyer, X. Liu, D. Milojicic, and A. Sahai, "SLA decomposition: Translating service level objectives to system level thresholds," in *ICAC'07*.

[136] A. Sivakumar, S. Puzhavakath Narayanan, V. Gopalakrishnan, S. Lee, S. Rao, and S. Sen, "Parcel: Proxy assisted browsing in cellular networks for energy and latency reduction," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pp. 325–336, ACM, 2014.

VITA

VITA

Shankaranarayanan is a PhD student at the School of Electrical and Computer Engineering at Purdue University, where he is working on his research, advised by Prof.Sanjay Rao. Broadly, his research interests are in the areas of Computer Networks, Distributed Systems and Cloud Computing. His work focuses on developing algorithms and building systems that improve the performance of geo-distributed, large-scale online applications. He has developed systems and techniques that can enhance the end-to-end user perceived performance of these applications in a holistic fashion across the different constituent layers including datastores, replicated application components, client-side entities, and content distribution networks. He has also collaborated with researchers from AT&T Labs on many challenging ideas including network auditing and management, performance of applications in the mobility world and coordination in distributed NFV environments.

Shankaranarayanan hails from the city of Coimbatore from the southern parts of India, where he completed his high school from Bharatiya Vidya Bhavan. His academic pursuit moved further with Bachelors Degree (B.E.) in Computer Science And Engineering at Coimbatore Institute Of Technology, one of the premier engineering colleges in India. Following his undergraduate studies, he worked as a software developer for three years with D. E. Shaw & Co., a New York based hedge fund in various technological initiatives within the firm.