Algorithmic Nuggets in Content Delivery

Bruce M. Maggs
Duke and Akamai
bmm@cs.duke.edu

Ramesh K. Sitaraman
UMass, Amherst and Akamai
ramesh@cs.umass.edu

This article is an editorial note submitted to CCR. It has NOT been peer reviewed.
The authors take full responsibility for this article's technical content. Comments can be posted through CCR Online.

ABSTRACT

This paper "peeks under the covers" at the subsystems that provide the basic functionality of a leading content delivery network. Based on our experiences in building one of the largest distributed systems in the world, we illustrate how sophisticated algorithmic research has been adapted to balance the load between and within server clusters, manage the caches on servers, select paths through an overlay routing network, and elect leaders in various contexts. In each instance, we first explain the theory underlying the algorithms, then introduce practical considerations not captured by the theoretical models, and finally describe what is implemented in practice. Through these examples, we highlight the role of algorithmic research in the design of complex networked systems. The paper also illustrates the close synergy that exists between research and industry where research ideas cross over into products and product requirements drive future research.

1. INTRODUCTION

The top-three objectives for the designers and operators of a content delivery network (CDN) are high reliability, fast and consistent performance, and low operating cost. While many techniques must be employed to achieve these objectives, this paper focuses on technically interesting algorithms that are invoked at crucial junctures to provide provable guarantees on solution quality, computation time, and robustness to failures. In particular, the paper walks through the steps that take place from the instant that a browser or other application makes a request for content until that content is delivered, stopping along the way to examine some of the most important algorithms that are employed by a leading CDN.

One of our aims, as we survey the various algorithms, is to demonstrate that algorithm design does not end when the last theorem is proved. Indeed, in order to develop fast, scalable, and cost-effective implementations, significant intellectual creativity is often required to address practical concerns and messy details that are not easily captured by the theoretical models or that were not anticipated by the original algorithm designers. Hence, much of this paper focuses on the translation of algorithms that are the fruits of research into industrial practice. In several instances, we demonstrate the benefits that these algorithms provide by describing experiments conducted on the CDN.

A typical request for content begins with a DNS query issued by a client to its resolving name server (cf. Figure 1). The resolving name server then forwards the request to the

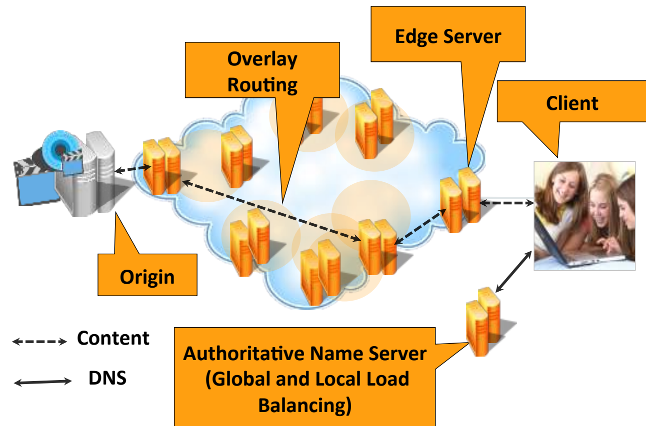

Figure 1: A CDN serves content in response to a client's request.

CDN's authoritative name server. The authoritative name server examines the network address of the resolving name server, or, in some cases, the edns-client-subnet provided by the resolving name server [9], and, based primarily on this address, makes a decision about which of the CDN's clusters to serve the content from. A variant of the stable marriage algorithm makes this decision, with the aim of providing good performance to clients while balancing load across all clusters and keeping costs low. This algorithm is described in Section 2.

But DNS resolution does not end here. The task of indicating which particular web server or servers within the cluster will serve the content is delegated to a second set of name servers. Within the cluster, load is managed using a consistent hashing algorithm, as described in Section 3. The web server address or addresses are returned through the resolving name server to the client so that the client's application, such as a browser, can issue the request to the web server. The web servers that serve content to clients are called *edge servers* as they are located proximal to clients at the "edges" of the Internet. As such, Akamai's CDN currently has over 170,000 edge servers located in over 1300 networks in 102 countries and serves 15-30% of all Web traffic.

When an edge server receives an HTTP request, it checks to see if the requested object is already present in the server's cache. If not, the server begins to query other servers in

the following order. First it asks the other servers in the same cluster, who share a LAN. If none of these servers have the object, it may ask a "parent" cluster. If all else fails, the server may issue a request for the object to an *origin server* operated by the content provider, who is a customer of the CDN. As new objects are brought into the CDN's edge server, it may be necessary to evict older items from the cache. Section 4 explains how algorithms based on Bloom filters are used to decide which objects to cache and which not to.

Often, CDNs deliver content that is not cacheable. Examples include dynamic web sites, web applications, and live video streaming. In this scenario, the CDN's edge server acts as a "front end" for a web site, but must forward requests for uncacheable content to the content provider's origin. Using the CDN servers as relays in an "overlay routing network", however, can actually reduce download time and increase availability. The idea is that exploring multiple overlay paths between the origin and the edge servers and picking the best ones can provide paths with lower latency and higher throughput. Further, the multiple paths provide redundancy in case of path failures, enhancing availability. In Section 5, we explain how multi-commodity flow provides an algorithmic framework for constructing good overlay routing networks.

A CDN is a large distributed system where individual servers and networks can fail at any time. The system has to be built to be resilient to such failures and should continue to function with only minimal degradation in performance. A CDN has dozens of software components that collect data, analyze it, and make key decisions based on it. Two examples described above are global and local load-balancing decisions, which are computed using real-time data feeds in time scales of minutes or seconds. A common strategy employed by the CDN for fault tolerance is to compute the solution independently on several machines, and then to apply a leader-election algorithm to determine which machine should distribute its solution to other CDN components that depend on it. Because different machines typically compute solutions at different times and with slightly different inputs, there may be slight differences in the solutions. Section 6 describes several variants of leader election algorithms. There are multiple algorithms because different responses to network partitions are desired in different contexts.

In each section, an effort is made to describe an abstract version of the problem to be solved along with one or more algorithms for solving the problem. Typically, however, there are practical considerations not captured in the abstract model. Hence in each section we also present the most important practical considerations and explain how they impact the actual implementations. Our focus is primarily on the algorithms employed by a CDN. For a broader look at the systems architecture of Akamai's CDN, the reader is referred to [10] and [24]. For deeper discussion of Akamai's overlay networks, the reader is referred to [30].

2. STABLE ALLOCATIONS

Global load balancing is the process of mapping clients to the server clusters of the CDN. Rather than making load balancing decisions individually for the billions of clients around the world, cluster assignments are made at the granularity of *map units*. Each map unit can be viewed as the tuple $\langle$IP address prefix, traffic class$\rangle$, where the first element of the tuple is a set of client IP addresses and the second element describes the type of traffic accessed by those clients. The IP addresses of a specific map unit are intended to be "proximal" to each other in the Internet and can be viewed as a single unit from the perspective of load balancing, i.e., they are likely to experience similar performance when assigned to the same CDN cluster. The second tuple of a map unit is the traffic class that the clients of the map unit access. A traffic class indicates the type of content service accessed by the users. There are tens of traffic classes such as video, web content, applications, software downloads, etc., each of which has distinctive properties. As a simple example, the map unit $\langle 1.2.3.4/24, \text{video}\rangle$ corresponds to the set of clients in a specific /24 prefix who are accessing videos. Note that the same set of users accessing web content form a different map unit $\langle 1.2.3.4/24, \text{web}\rangle$. If the block of IP addresses is too coarse, then assigning the map unit to a single cluster may produce suboptimal performance for some of those addresses. But if the granularity is too fine, there will be many map units, leading to scalability issues. Dividing the clients of the global Internet into prefixes at the right level of granularity is a complex problem in its own right, but beyond the scope of this paper. In all, it is customary to have tens of millions of map units to capture all of the clients and traffic classes of the trillions of requests per day served by Akamai.

Global Load Balancing. The goal of global load balancing is to assign each map unit $M_i, 1 \leq i \leq M$, to a server cluster C_j, $1 \leq j \leq N$, such that the clients represented by each map unit can download the requested content from their assigned cluster with high availability and performance. For each map unit, the candidate clusters are ordered in descending order of preference where a higher preference indicates better predicted performance for the clients of that map unit. For example, a candidate cluster with a higher preference might be one with lower latency, lower packet loss, and higher throughput to the clients in the map unit. Likewise, each server cluster C_j has preferences regarding which map units it would like to serve. A number of factors dictate a cluster's preferences for map units, including the contract terms under which the cluster was deployed, whether the IP addresses are local to the autonomous system (AS) in which the cluster is deployed, etc. For example, a cluster deployed in an upstream provider ISP may prefer to serve clients in the ISP's downstream customer ISPs.

Map units originate demand while the clusters have capacity to serve that demand. With each map unit M_i a demand d_i is associated, which represents the amount of content accesses that clients belonging to that map unit are likely to generate. Demands can vary rapidly in real-time as clients access more (or less) content. "Flash crowds," where the demands from certain map units sharply increase are not uncommon. Each cluster C_j also has a notion of capacity c_j that represents the maximum demand that the cluster can serve. Capacity is dictated by how much processing power, memory and disk capacity, and network bandwidth the servers in the cluster possess. The goal of global load balancing is to assign map units to clusters such that preferences are accounted for and capacity constraints are met.

Stable Allocations. Stable allocations [14] is a classical algorithmic paradigm that forms the basis for assigning map units to clusters in global load balancing. The simplest

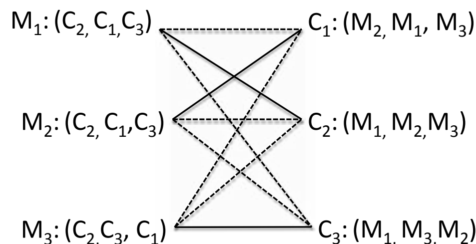

Figure 2: Each map unit has a preference order of clusters from where to download content, while each cluster has a preference order of which map units to serve content. A stable marriage (marked in bold) is a matching of map units to clusters such that no unmatched pair prefer each other over their matched partners.

example of stable allocations is the *stable marriage problem,* which was first studied by Gale and Shapley in 1962 in the provocatively titled paper "College admissions and the stability of marriage" [12]. Since the original work, stable marriage and related stable allocation problems have been used to match students to schools in New York city [2], medical interns to hospitals [27], and organ donors to patients [28]. Notably, Roth and Shapley won the Nobel Prize in Economics in 2012 for their work on stable allocation.

In our exposition, we start with the simplest form of stable allocation (Section 2.1), extend these results to more complex forms that appear in the CDN context (Section 2.2), and finally describe the real-world complexities of an operational system that go beyond the algorithmic models (Section 2.3).

2.1 The Stable Marriage Problem

The stable marriage problem is often described as the task of matching men and women for marriage. There is an equal number of men and women, each seeking a partner of the opposite sex. The stable marriage problem can model a simple case of global load balancing where each map unit is a man with unit demand for content, each cluster is a woman with unit capacity to server content, and the number of map units, M, equals the number of clusters, N. An illustrative example is shown in Figure 2. Each map unit orders the clusters by preference, e.g., M_1 prefers cluster C_2, followed by C_1, and then C_3. Likewise, each cluster has a preferential list of map units, e.g., cluster C_1 prefers map unit M_2, followed by M_1, and then M_3. As in real-world match-making, not everyone can have their top choice! It is easy to see in our example that not every map unit can obtain the top-rated cluster C_2. Nor can every cluster get its top choice, since two clusters rate M_1 highest.

A stable marriage is a matching of men to women where there is no pair of participants that both prefer each other to the partners that they are matched to. Given an assignment of map units to clusters, an unmatched pair M_i and C_j is said to be *blocking* if

1. M_i prefers C_j over its current partner $C_{j'}$, and
2. C_j prefers M_i over its current partner $M_{i'}$.

Note that the elements of a blocking pair have an incentive to switch away from their current partners to marry each other. An assignment that matches all map units with clusters such that there are no blocking pairs is said to be *stable.* As can be seen from the example, the assignment of M_1 to C_2, M_2 to C_1, and M_3 to C_3 is stable.

The notion of stability is a natural way to account for the preferences of all participants. Finding a stable marriage is achieved by a simple and distributed algorithm called the Gale-Shapley algorithm [12]. The algorithm works in rounds where men propose and women (provisionally) accept, forming an "engagement." Initially, all men and women are "free." As the rounds progress they become engaged, although engagements are sometimes broken. In each round, each free man proposes to the most preferred woman that he has not already proposed to, whether or not she is already engaged. Among the proposals received in a round, a free woman accepts the one from the man she most prefers. If a woman is already engaged, she accepts a new proposal, and breaks her previous engagement, only if she receives a proposal from a man who she prefers over her current fiancee. If there are multiple such proposals, she accepts the one from the most preferred man. The algorithm terminates when everyone is engaged. The fundamental theorem of stable marriage follows.

THEOREM 2.1 ([14]). *The Gale-Shapley algorithm terminates with a matching that is stable after no more than N^2 proposals, where N is number of men (and women). All possible executions of the algorithm with men as the proposers yield the same matching where each man has the most preferred partner that he can have in any stable marriage.*

The stable marriage produced by Gale-Shapley is often called "man-optimal" since each man is married to the most preferred woman that he could be married to in any stable marriage. It may also be termed "woman-pessimal" because each woman is married to the least preferred man that she could be married to in any stable marriage. Our load balancing algorithm produces a "map-unit-optimal" stable marriage as map units play the role of the proposers. Thus, the solution provides each map unit with the most preferred cluster possible in any stable marriage, a property that fits well with the CDN's mission of maximizing performance for clients.

2.2 Algorithmic Extensions

Global load balancing uses a generalized form of the Gale-Shapley algorithm described in Section 2.1. A list of of generalizations needed to model our load balancing problem more closely is provided below.

1) Unequal number of map units and clusters. In our problem, there are vastly more map units than clusters, i.e., tens of millions of map units versus thousands of clusters.

2) Partial preference lists. Given that there are tens of millions of map units and thousands of clusters, it is unnecessarily expensive to measure and rank *every* cluster for each map unit. It suffices to rank only those clusters that

are expected to provide a minimum level of performance. In fact, it may suffice to measure and rank for each map unit the top dozen or so clusters that are likely to provide the best performance. Likewise, the clusters need only express preferences for the subset of the map units that are likely candidates for assignment. For instance, a map unit in Boston may never need to rank a cluster in Tokyo and vice versa, since they are unlikely to be paired up.

3) Modeling integral demands and capacities. While the canonical stable marriage problem considers unit value demands and capacity, it can be extended to the case in which capacity and demand are expressed as arbitrary integers. For each map unit, we estimate the content access traffic generated by its clients and represent that as a demand value. Likewise, for each cluster, we can estimate its capacity, which is the amount of demand it can serve.

The above variants can be solved by generalizing the Gale-Shapley algorithm in straightforward ways. A survey of these common variants with further references can be found in [15]. The extension of stable marriage to a many-to-many problem where a man can marry many wives (polygamous) and a woman can marry many husbands (polyandrous) is also relevant [6].

2.2.1 Resource Trees

With the above extensions, our model for the global load balancing problem is better, but it still inadequate in an important regard. A server cluster cannot be accurately modeled as a single resource whose capacity can be represented as a single number. Instead, a server cluster has a number of different resources and interrelated resource constraints. These resources that include processing power, memory and disk capacity, and network bandwidth. Further, the capabilities of the servers may restrict what traffic classes they may serve. For example, the amount of video traffic that a cluster can serve may be separately constrained from the amount of web traffic that it might serve.

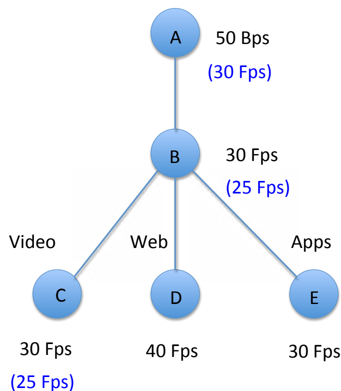

Figure 3: An example of a resource tree with capacities shown in black. The residual capacities after assigning a video map unit with 20 units of demand requiring 20 Bps and 5 Fps is shown in blue.

We illustrate the resource constraints of a cluster using a simple example. Suppose a server cluster has two resources, each with a separate capacity. First is a network resource that constrains the rate at which data can be sent out of the cluster modeled by a capacity expressed in the units of bytes per second (Bps). The second models all other resources of the server using a single number expressed in the units of "flytes" per second (Fps). The flyte capacity captures all non-network server resources such as the processor, memory, and disk. Besides raw resource constraints, the cluster may have limits on how much of each traffic class it can serve. As an example, assume that the cluster can serve three traffic classes: video, web, and applications, each with its own capacity limitations. The different resource constraints can be expressed as a tree as shown in Figure 3. The leaves of the tree bound the maximum number of Fps that can be used for each traffic class. The parent node B bounds the maximum number of Fps that can be served from the cluster *across* all traffic classes. The root node A bounds the total network traffic (in Bps) that can be sent from the cluster.

The resource tree allows us to express a hierarchy of constraints on the mix of traffic that a cluster is able to serve. A map unit is assigned to a leaf of the tree and it consumes resources on each node in its leaf-to-root path. For instance, a unit of demand from a map unit accessing video exhausts f Fps and b Bps at each node in the path from the leaf C to the root A, where f and b are *scale factors*. Each traffic class has its own scale factor. For instance, a unit of demand from a video map unit will consume a lot of Bps but not a lot of Fps, since video traffic is network intensive. In contrast, a banking application served by the cluster will exhaust a lot of Fps but not many Bps, since such traffic is processor intensive but the traffic sent out is small.

To continue our example of Figure 3, suppose 20 units of demand from a video map unit are assigned to the cluster, where the scale factor dictates that each unit of video demand requires 0.25 Fps and 1 Bps. The residual capacity of the cluster can now be calculated by following the leaf-to-root path and decrementing the consumed values, as shown in the figure. More map units can be assigned to the cluster as long as there is sufficient residual capacity left in all nodes in the corresponding leaf-to-root path.

The Gale-Shapley algorithm can be generalized to the more general situation where each cluster has a resource tree with capacity associated with each node of the tree [13]. To illustrate how the generalized algorithm works using the example in Figure 3, suppose that a second map unit with 26 units of application traffic is assigned to the cluster, where each unit of demand requires 1 Fps and 0.2 Bps. Adding the new map unit to the cluster requires an additional 26 Fps and $26 \times 0.25 = 6.5$Bps in its leaf-to-root path. This addition will cause a capacity violation at node B of the tree, but no capacity violation at node A or node E. Suppose that the cluster has a higher preference for map units with application traffic, rather than video traffic. The algorithm looks at the first node in the leaf-to-root path where there is a capacity violation and evicts a lower preference map unit in favor of the higher preference one. In our case, 4 units of video demand are evicted to create 1 Fps of capacity at node B to accommodate the additional 26 units of application traffic demand. By iteratively performing such reassignments the algorithm converges to a stable allocation of map units.

2.3 Implementation Challenges

While algorithms provide a conceptual framework for a problem solution, numerous architectural and system chal-

lenges also need to be addressed, some of which we list below.

1) Complexity and scale. A key challenge is the complexity and scale of global load balancing across tens of millions of map units and thousands of clusters for over a dozen traffic classes with more complex resource trees than the simple example we provide in Figure 3.

2) Time to solve. The map unit assignment must be recomputed every 10 to 30 seconds, as network performance and client demand change on short time scales. Network links can become congested or fail in seconds, changing the preference order of map units. Server capacities can vary at a similar granularity as demand rises and falls. Further, global and local events can cause flash crowds where demand for some map units shoots up, requiring quick reassignment to other clusters. The generalized Gale-Shapley algorithm has a strong advantage in that its run time is nearly-linear in the product of the number of map units and the maximum length of the map unit preference lists. It is also amenable to a distributed implementation where map unit assignments can proceed in parallel.

3) Demand and capacity estimation. The demand of a map unit is an estimate of the downloading activity that we are likely to see from its clients. Further, the actual server resources consumed by a unit of demand, i.e., the scale factors, are often difficult to estimate and only become measurable *after* the demand is assigned to a cluster, thus requiring a tight feedback loop to estimate demand and capacity based on past history.

4) Incremental and persistent allocation. While our algorithmic model assumes that a stable allocation is performed from scratch in each cycle, it is undesirable to do so for two reasons. First, typically only a small fraction of map units need to be reassigned, e.g., map units that have experienced significant changes in demand or preference since the last cycle. Second, recomputing the entire stable allocation can result in a very different solution than the previous one, even if the demands and preferences have not changed in major ways. A map unit reassigned from (say) cluster A to cluster B for which it has similar preference causes unnecessary havoc, because cluster A has already cached the content requested by the map unit clients, while cluster B may not have cached different content, causing cache misses when clients are switched to cluster B. This phenomenon calls for a “sticky” solution that changes assignments only when the preference differences are sufficiently large. A particularly harmful form of impersistence is *oscillation*, in which the global load balancer moves cyclically between different solutions as a result of small changes in input values.

The above challenges require additional innovations, pushing us into realms where the clean, provable algorithmic models no longer hold. Nevertheless, global load balancing is a good example of a problem in which algorithmic and architectural innovations reinforce each other and neither can exist without the other.

3. CONSISTENT HASHING

Consistent hashing [17, 22] is used by the CDN to balance the load within a single cluster of servers. Consistent hashing was the first algorithmic innovation in Akamai’s CDN and the patents on consistent hashing (e.g., [18]) formed an important component of the start-up company’s initial portfolio of licensed intellectual property.

In a traditional hash table, objects from a universe $\mathcal{U}$ are mapped to a set of buckets $\mathcal{B}$. For example, if $\mathcal{U}$ is the set of positive integers, and $\mathcal{B}$ is the set of integers $[0, n-1]$, then the mapping $\mathcal{U} \rightarrow \mathcal{B}$ might be achieved using a simple hash function h such as $h(x) = ((ax + b) \mod P) \mod n$, where P is a prime number such that $P \geq n$, a is chosen at random from $[1, P-1]$, and b is chosen at random from $[0, P-1]$. This hash function maps elements of $\mathcal{U}$ uniformly (over the random choices of a and b) to the buckets $\mathcal{B}$ and also provides pairwise independence, i.e., for any $x_1, x_2 \in \mathcal{U}$,

$$\Pr[h(x_1) = y_1 \& h(x_2) = y_2] = \Pr[h(x_1) = y_1] \Pr[h(x_2) = y_2].$$

When a hash table is used in a sequential computer program, an arbitrary subset $S \subset \mathcal{U}$ of $O(n)$ objects of interest are inserted into the buckets of B, i.e., each element $x \in S$ is inserted into bucket $h(x)$. Using an array of linked lists to represent the buckets, operations such as inserting an object into a bucket, removing an object from a bucket, or testing whether an object is already in a bucket can be implemented in a straightforward manner so that the expected time per operation is constant. Alternative data structures that support these operations, such as balanced search trees, typically require $\Theta(\log n)$ time per operation (but may support additional operations).

In the CDN setting, an object is a file such as a JPEG image or HTML page, and a bucket is the cache of a distinct web server. A new complication arises in this context: a server may fail, in which case it can no longer fulfill its role as a bucket. Unfortunately, the class of hash functions like $h(x)$ above, which map elements of $\mathcal{U}$ directly to buckets $\mathcal{B}$, provides no efficient mechanism for dealing with the loss of a bucket. Simply remapping all of the objects in the lost bucket to another bucket is not ideal because then one bucket stores double the expected load. On the other hand, balancing the load by renumbering the existing buckets and rehashing the elements using a new hash function has the disadvantage that many objects will have to be transferred between buckets.

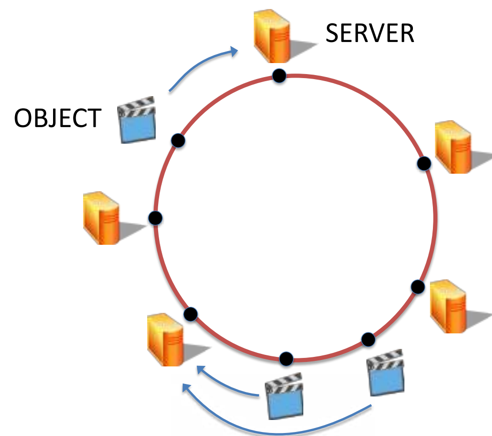

Figure 4: Consistent hashing first maps both objects and buckets (servers) to the unit circle. An object is then mapped to the next server that appears on the circle in clockwise order.

Consistent hashing solves this problem in a clever way. In-

stead of mapping objects directly to buckets, both objects *and* buckets are first mapped to the unit circle. As shown in Figure 4, each object is then mapped to the next bucket that appears in clockwise order on the unit circle. This approach provides a more graceful mechanism for dealing with the loss of a bucket: if a bucket fails then the objects that were formerly mapped to the bucket are instead mapped to the next bucket that appears in clockwise order on the unit circle. Similarly, if a new bucket is added, e.g., the failed server comes back on-line, the only objects that have to be moved are those that are mapped to the new bucket.

A simple enhancement to consistent hashing that improves the balance of objects among the buckets is to map each bucket to multiple locations on the unit circle. For example, each of the n buckets might be mapped to $\log n$ instances. The expected load for each instance is $1/\log n$ the load for a bucket, but the expected load for a bucket is the same as if there was only one instance. The distribution of load, however, is much more tightly centered around the expectation. As before, an object is mapped to the next instance of a bucket that appears in clockwise order on the unit circle. If a server fails, all of the corresponding bucket's instances are removed from the unit circle. The objects that were in the bucket are now remapped across a set of buckets rather than to a single bucket.

3.1 Popular Objects

An important practical consideration not addressed in the original consistent hashing work, but described in [16], is that some web objects may be far more popular than others. In some circumstances, such as a flash-crowd event, a single object may be so popular that it is not possible for a single server within a cluster to satisfy all of the requests for the object. Instead, the object must be served by more than one server. A straightforward extension to consistent hashing would be to map a popular object to the next k servers that appear in clockwise order on the unit circle, where k is a function of the popularity of the object. One disadvantage of this approach, however, is that if two very popular objects happen to hash to nearby positions on the unit circle, the buckets that they map to will overlap, and may have difficulty handling the requests for both objects simultaneously.

The approach taken by the CDN is to use a *separate mapping of the buckets to the unit circle for each object.* This modification to consistent hashing preserves the property that when a bucket is removed, the only objects that must be moved to different buckets are the ones that were formerly in the removed bucket, and when a bucket is added, the only objects that must be moved are the ones that are now mapped to the new bucket.

Suppose that there are two popular objects, each of which is to be mapped to k buckets. Regardless of whether the buckets are mapped to the unit circle using the same hash function for both objects or different hash functions, the expected number of buckets containing both objects is $\Theta(k^2/n)$. But the distributions are very different. For example, if the same hash function is used, the probability that at least $k/2$ buckets receive both popular objects is $\Theta(k/n)$. On the other hand, if different hash functions are used, then each of the k instances of the second object is independently mapped to a bucket containing the first object with probability at most k/n. Thus, the probability that at least $k/2$ of these instances map to buckets containing the first object is at most $\binom{k}{k/2}(k/n)^{k/2}$, which is at most $(2ek/n)^{k/2}$ (sometimes a loose upper bound), which is typically less than $\Theta(k/n)$, as long as $k < n/2e$.

3.2 Consistent Hashing in Practice

Although an object is a single file to be served by a web server, the CDN does not hash each object independently. Instead, when a content provider signs up as a customer of the CDN, they are granted one or more *serial numbers.* The content provider's objects are grouped together by serial number, and all objects with the same serial number are hashed to the same bucket (or same set of buckets, if popular). The advantage of grouping a content provider's objects together arises when multiple objects must be fetched by the browser in order to render a single web page. If these objects were all independently hashed to different servers, the browser would have to open a separate HTTP connection to each server. By mapping the objects to the same set of servers, the browser can make a persistent HTTP connection to just one randomly-chosen server in the set, and then fetch all of the objects from that server. The overhead for establishing and ramping up a new TCP connection is thus paid only once, rather than once per object.

We are now in a position to provide a concrete example of how the CDN employs consistent hashing in practice. Suppose that the serial numbers assigned to content providers range from 1 to 2048 and that the CDN has assigned the serial number 212 to a customer that operates the web site www.example.com. In order to use the CDN to deliver the images that appear on its web pages, the customer creates a new domain name images.example.com, and then uses the DNS CNAME mechanism to indicate that images.example.com is an alias for the canonical domain name a212.g.akamai.net. (The "g" in this domain name stands for "general," a domain name used by the CDN for delivering static objects.) In this example, the customer embeds the URL http://images.example.com/logo.jpg in the HTML for the web page http://www.example.com so that the browser will fetch the web site's logo from the CDN.

When an authoritative name server operated by the CDN receives a request to resolve the name a212.g.akamai.net from a resolving name server, it first determines which cluster should serve the ensuing HTTP requests from browsers sharing this resolving name server. The cluster is chosen using the algorithm described in Section 2. Once the cluster has been selected, consistent hashing is used to determine which server or servers within the cluster should serve objects under serial number 212. In this example the four servers in the chosen cluster have been assigned the IP addresses `192.168.1.1` through `192.168.1.4`. (Although these addresses are private, in practice the CDN's servers almost always use public addresses.) For each serial number there is a random map of the buckets to the unit circle. The map can be represented as an ordered list of servers, e.g., for serial number 212 the map might be 212: 192.168.1.3 192.168.1.1 192.168.1.4 192.168.1.2. (Since each serial number has its own random mapping of buckets to the unit circle, we can assume that the serial number itself is mapped to position 0.) Depending on the current popularity of the content being served under serial number 212, the authoritative name server for the CDN returns the first k addresses on this list, skipping any servers that are not operational.

The k servers in the DNS response are listed in random order to help spread the load among them.

4. BLOOM FILTERS

Bloom filters are used to approximately represent dynamically evolving sets in a space efficient manner. Bloom filters were first introduced by Burton Bloom [7] and have found numerous elegant applications in networking [8]. Bloom filters are useful in content delivery in two different contexts: content summarization and content filtering. We illustrate both uses and focus on a simple example of content filtering that offers significant benefits in the form of better disk resource usage and performance.

4.1 The Basics

Suppose that we want to store a set $S = \{e_1, e_2, \cdots, e_n\}$ in a manner that allows us to efficiently insert new elements into the set and answer membership queries of the form "Is element e in set S?". A simple data structure is a hash function h that maps elements to a hash table $T[1 \cdots m]$, where each table entry stores a binary value that is initialized to 0. When an element e is inserted into set S, we simply set the entry with index $h(e)$ to 1, i.e., we set $T[h(e)]$ to 1. To check if $e \in S$, we simply need to check if $T[h(e)]$ is 1. Note that "false positives" are possible with this solution when an element $e' \notin S$ "collides" with an element $e \in S$, i.e., when $h(e) = h(e')$. In this case, the membership query for e' will return the incorrect answer, i.e., we will conclude that $e' \in S$ when it is not. Note that a"false negative" cannot occur since when we conclude that an element is not in S we are always correct.

Bloom filters are a generalization of the simple hash table solution described above. Rather than use one hash function, a Bloom filter uses multiple hash functions $h_1, h_2, \cdots, h_k$ to reduce the probability of a false positive. As before, we have a table $T[1 \cdots m]$ where each $T[i]$ can store a binary value and all entries are initialized to 0. When an element e is inserted into set S the bits $T[h_i(e)], 1 \leq i \leq k$, are set to 1. To check if $e \in S$, we need to check if $T[h_i(e)] = 1$, for all $1 \leq i \leq k$. False positives are still possible with this solution, but the likelihood is smaller since an element $e' \notin S$ will be mistakenly believed to belong to S only if other elements have set the bits $T[h_i(e')]$ to 1 for *all* k hash functions h_i.

It is instructive to quantify the tradeoff between the false positive probability p, the number of elements n, the number of bits m, and the number of hash functions k. For simplicity, assume that for any element e and hash function h_i, $h_i(e)$ is distributed independently and uniformly over the range of values 1 to m (although in practice weaker hash functions might be used). For any $1 \leq l \leq m$, we evaluate the probability that $T[l] = 0$, after inserting all n elements of S into the table. For each $e \in S$ and $1 \leq i \leq k$, the probability that $h_i(e) \neq l$ is $(1 - 1/m)$. The probability that $T[l] = 0$ after inserting all n elements with k probes each is simply $(1 - 1/m)^{kn}$. Thus, the probability that an entry $T[l] = 1$ after n insertions is simply $1 - (1 - 1/m)^{kn}$. A false positive occurs when $e' \notin S$ but $T[h_i(e')] = 1$, for all $1 \leq i \leq k$. Although not strictly true, we assume that the probabilities that the different entries $T[l]$ are set to 1 are independent. Thus, the false positive probability p can be approximated as follows.

$$p = \left(1 - (1 - 1/m)^{kn}\right)^k \approx (1 - e^{-kn/m})^k. \quad (1)$$

(The reader is referred to a standard reference such as [23] for a more rigorous derivation of the relationship between p, m, n, and k.) For a given n and m, one can find the k that minimizes the probability p by differentiating the right-hand side of Equation 1 with respect to k and setting it to zero. Solving for k provides the following.

$$k \approx (\ln 2) \cdot (m/n). \quad (2)$$

That is, k should be chosen to be roughly proportional to the ratio of the size of the hash table and the number of elements that you expect to be placed in the Bloom filter. Plugging in the value of k from Equation 2 into Equation 1, we get the following relation.

$$p \approx 2^{-k} \approx 2^{-\frac{m \ln 2}{n}}. \quad (3)$$

As we will see, the above equation is key for sizing the Bloom filter for specific applications.

4.2 Content Summarization and Filtering

There are two key applications of Bloom filters to CDNs. The first application is *cache summarization* where a Bloom filter is used to succinctly store the set of objects stored in a CDN server's cache. Note that using a Bloom filter for this purpose is much more space-efficient than storing a list of URLS associated with the objects in cache. Cache summarization can be used to locate which server cache has which objects. For instance, if the CDN's servers periodically exchange cache summaries, a server that does not have a requested object in its cache can find it on other servers using their cache summaries. Cache summarization has been well-studied in the academic literature [11]. It is also implemented in popular web caching proxies like Squid, which create cache summaries called "digests" [29]. For reasons of efficiency, Squid uses a Bloom filter with $k = 4$ where a single 128-bit MD5 hash of the object's identifier is partitioned into four 32-bit chunks and treated as four separate hashes.

Note that cache summarization requires the Bloom filter to support deletion of elements in addition to insertions. In particular, when an object is evicted from cache, it has to removed from the Bloom filter as well. This requires a variant called *counting Bloom filters* where the table values are no longer binary but are counters instead [11]. To insert an element e into the set the entries $T[h_i(e)], 1 \leq i \leq k$ are incremented. To delete an element, the k locations are decremented. Testing for membership of a particular element checks if the k locations are non-zero.

The second application for Bloom filters is what we call *cache filtering*, in which a Bloom filter is used to determine what objects to cache in the first place. To motivate the need for cache filtering, we collected access logs from a single cluster of Akamai's CDN consisting of 45 servers that cache and serve Web traffic. Using the logs, we studied the relative popularity of the over 400 million distinct web objects that clients accessed from the servers over a period of two days. As shown in Figure 5, nearly three-quarters of the objects were accessed only once and 90% of the objects were accessed fewer than four times during the two days.

In normal operation without cache filtering, a CDN's server caches each object that it serves by storing a copy of the ob-

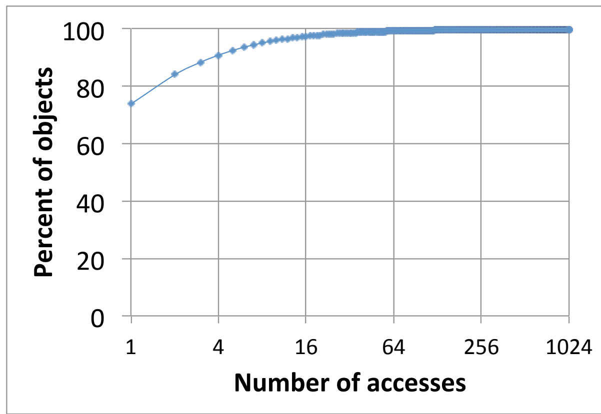

Figure 5: On a typical CDN server cluster serving web traffic over two days, 74% of the roughly 400 million objects in cache were accessed only once and 90% were accessed less than four times.

ject in its disk cache. When the disk cache fills up, the objects are evicted using a cache replacement policy. But there is no reason to cache objects that are likely to be accessed only once and never accessed again. Such objects are jocularly referred to as *"one-hit-wonders"*. Our popularity analysis suggests that over three-quarters of the objects are one-hit wonders and a significant amount of disk space would be saved if they were not cached. In addition, the number writes to disk would also be reduced significantly if these objects were not cached.

Cache-on-second-hit rule. A simple cache filtering rule that avoids caching one-hit-wonders is to cache an object only when it is accessed for a second time within a specific time period. The rule can be implemented by storing the set of objects that have been accessed in a Bloom filter. When an object is requested by an client, the server first checks to see if the object has been accessed before by examining the Bloom filter. If not, the object is fetched and served to the client, but it is not cached. If the object has been accessed before, it is fetched, served, and also stored in the server's disk cache.

The astute reader may have observed that as more objects are added to a Bloom filter and it starts to fill up, the probability of false positives increases. A simple approach to circumventing this issue is to have two Bloom filters, a primary and a secondary. All new objects are inserted into the primary filter, until it is reaches a threshold for maximum number of objects. At that point, the primary becomes the secondary filter and a new filter with all entries initialized to zero becomes the primary. As the old secondary filter is discarded, objects that have not been accessed recently are forgotten. When checking if an object has been accessed in the recent past, both primary and secondary Bloom filters need to be queried.

4.3 Empirical Benefits

To illustrate the benefits of the cache-on-second-hit rule using Bloom filters, we describe a simple experiment conducted by Ming Dong Feng using a cluster of about 47 production servers serving live traffic on the field. Each server has eight hard disks. The workload served by these servers have a "cold footprint" consisting of web and videos from social networking sites. Bloom filters that implement the cache-on-second-hit rule were turned on on March 14th and turned off on April 24th to create before, during, and after scenarios for analysis. Measurements were made every few tens of seconds and averaged across all machines every six hours. Thus, our figures show multiple measurements per day, enabling us to see both intra- and inter-day variations.

Figure 6 shows the average byte hit rate of the servers, where byte hit rate is simply the percent of bytes served to clients that were found in the server cache. For instance, a byte hit rate of 75% would mean that for every 100 bytes served to the client, 75 bytes were found in cache and 25 bytes had to be fetched from a peer, parent, or origin server. Note that during the period when Bloom filtering was turned

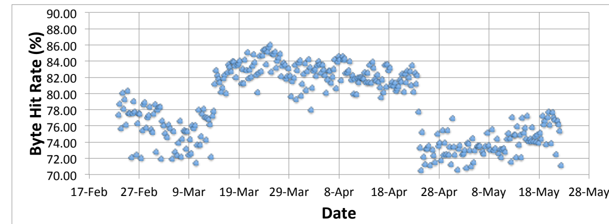

Figure 6: Byte hit rates increased when cache filtering was turned on between March 14th and April 24th because not caching objects that are accessed only once leaves more disk space to store more popular objects.

on, the byte hit rate increases from around 74% to 83%. Most CDNs implement cache replacement algorithms such as LRU, which evict less popular objects such as one-hit-wonders when the cache is full. Filtering out the less popular objects and not placing them in cache at all provides additional disk space for more popular objects and increases the byte hit rate.

In Figure 7, we show the impact of Bloom filters on the number of disk writes performed by the servers. The disk related metrics we report are measured by a utility similar to Linux's iostat [1] which runs continually on production servers and reports metrics relevant to disk performance. Not having to store the one-hit-wonders in cache reduces the aggregate rate of disk writes by nearly one-half. To be more precise, the rate of disk writes drops from an average of 10209 writes per second to 5738 writes per second, a decrease of 44%. One consequence of fewer disk writes is that the

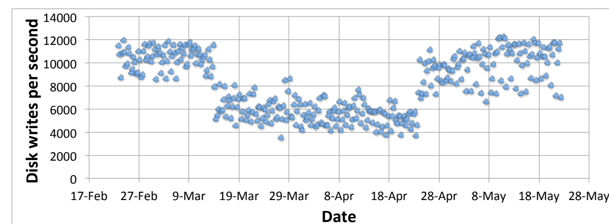

Figure 7: Turning on cache filtering decreases the rate of disk writes by nearly one half because objects accessed only once are not written to disk.

latency of accessing an object from disk decreases. As shown in Figure 8, the average disk latency drops from an average

of 15.6 milliseconds to 11.9 milliseconds, a decrease of about 24%.

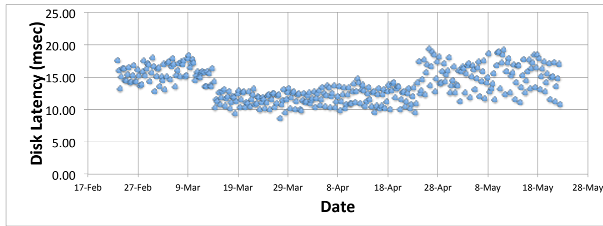

Figure 8: The latency of reading content from disk decreases when cache filtering is turned on, since there are fewer competing disk writes.

4.4 Implementation Challenges

Thus far we have described cache filtering in its simplest form as an illustrative example. Now we consider some real-world issues in implementing cache filtering in a production CDN.

1. Speeding up the Bloom filter. The Bloom filter implementation needs to be extremely fast because Bloom filter operations are on the critical path for client-perceived CDN performance, i.e., the time taken to perform these operations adds to the download time perceived by the client. Rather than evaluating k different hashes, the approach of using a single suitably-sized MD5 hash and partitioning the bits into k hashes is often used in practice.

2. Sizing the Bloom filter. The Bloom filter implementation should be space efficient. In fact, space efficiency is the main reason why Bloom filters are used in place of simpler hash table based solutions. Note that server memory is a valuable resource and is used to hold a "hot-object" cache of the most popular objects. The Bloom filter must be sized correctly since a filter that is too large wastes memory and a filter that is too small yield a large false positive probability.

In practice, the sizing of the Bloom filter is done as follows. First, we decide how many objects n we are likely to add to the Bloom filter while it is in use. Next, we decide what false positive probability p we can tolerate, i.e., what is our tolerance to mis-classifying objects due to false positives. Knowing n and p, the size of the Bloom filter m can be evaluated using Equation 3. Further, knowing n and m, the number of hash functions k can be evaluated using Equation 2. To take a typical example, suppose that a single server is likely to see $n = 40$ million objects and we are willing to tolerate a false positive probability of 0.1%, then solving Equation 3 we get a Bloom filter table with 575.1 million bits, or about 68.6 MBytes in size. This is a reasonably small size and would fit easily in server memory. Then, using Equation 2, we would use $k = 10$ hash functions.

3. Complex cache filtering. Thus far, we have considered only the simplest form of cache filtering, i.e., the cache-on-second-hit rule. In addition to this rule, more complex rules may need to be implemented in practice that take into consideration other object popularity metrics and other characteristics such as size. Further, cache filtering applies not just to the disk cache, but also to the hot object cache in memory. For instance, an object needs to have a much higher level of popularity and also be small enough for it to be eligible to be placed in memory. In general, there is content filtering occurring at every level of the hierarchy of caches within a CDN server, including memory, solid-state disks, and hard disks. Implementing a hierarchy of complex cache filtering rules with variants of Bloom filters is a key challenge.

5. OVERLAY ROUTING

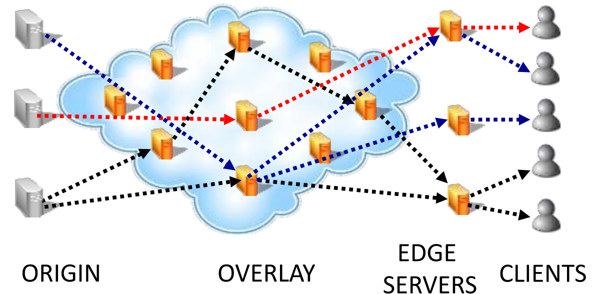

Figure 9: An overlay routing network enables fast and reliable communication between origins and edge servers. Different content (i.e., commodities) represented by different colors are routed simultaneously from their respective origins to edge servers who requested them. On receiving the content, the edge servers serve it to their proximal clients.

Most web sites have some content that is either not cacheable or cacheable for only short periods of time. Take, for example, a banking portal. A significant part of the portal is personalized for individual users and cannot be cached, though some page elements such as CSS, JavaScript, and images may be common across users and can be cached. Besides dynamic web content, CDNs also host applications that users can interact with in real-time where much of the interactions cannot be cached. Further, in the world of media, live streaming and teleconferencing are delivered by CDNs in real-time and are uncacheable. Finally, even when content can be cached, there are often time-to-live (TTL) parameters that require the CDN to periodically refresh the content. An example is a stock chart that changes continually and must be updated every few minutes.

A common framework that can capture all of the above situations is shown in Figure 9. There are (i) origins that create the content, (ii) edge servers that clients access to consume the content, and (iii) an overlay network that is responsible for transporting the content from the origins to the edges. Clients request the content from a proximal edge server, and the edge server in turn downloads the requested content from the origin via the overlay network.

In the case of a web site, the origin is a collection of application servers, databases, and web servers deployed by the content provider in one or more data centers on the Internet. In the case of a live stream, the origin denotes the servers that receive the stream in real-time from the encoders capturing the event. While the origin is usually operated by the content provider, CDNs often deploy a middlebox called a

reverse proxy near the origin to provide more efficient end-to-end transport between the origin and the edges. The overlay network is a large collection[1] of servers operated by CDN that can be used as intermediate nodes for routing content from the origin to the edge servers. The edge servers are operated by the CDN and are deployed in more than a thousand data centers around the world, so as to be proximal to clients.

5.1 The Overlay Construction Problem

The key problem in overlay routing is how to construct an overlay to provide efficient communication between origins and edge servers. An overlay construction algorithm takes as input (i) client demands, which dictate which origins need to send their content to which edge servers, and (ii) real-time network measurements of latency, loss, and available bandwidth for the paths through the Internet between origins, overlay servers, and edge servers. For each origin that needs to communicate with a specific edge server, the overlay construction algorithm returns a set of one or more overlay paths that have high availability and performance. An overlay path starts at an origin, passes through one[2] or more intermediate servers in the overlay network, and ends at an edge server as shown in Figure 9. Content is sent from the origin to the edge server along one of more of the constructed overlay paths. Note that overlay construction must be performed frequently as both client demand and network characteristics change rapidly. Therefore, algorithms for overlay construction must be very efficient.

5.2 Multicommodity Flow

A natural algorithmic framework for performing overlay routing is to formulate it as a multicommodity flow problem. As shown in Figure 9, the origins, the overlay servers, and the edge servers form the nodes of a flow network. The links of the flow network are all possible pairs of nodes that can communicate with each other, with each link having a *cost* and a *capacity*. Each origin is the source of a distinct commodity that needs to be routed to a set of edge servers. Note that a commodity represents content hosted at an origin, i.e., a commodity could be the contents of a web site or a stream of packets from a live video. The goal of overlay routing is to create routing paths for each commodity from its origin to the edge servers, while obeying capacity constraints and minimizing cost. Note that performance requirements are often codified as cost such that minimizing cost maximizes performance.

We now list key aspects of overlay construction that must be modeled in our multicommodity flow formulation. Many of these aspects make our problem different from the classical multicommodity flow studied in the literature, requiring new algorithmic innovations.

1. Multipath transport. Overlays use multiple paths between each origin and edge to enhance reliability and performance. There are two different modes in which these paths can be used, depending on the application context. In one mode that is common for live video, the paths are used simultaneously with content sent across all the paths. Content packets are encoded across these paths so that packets lost on one path can be recovered at the edge server using packets from other paths. A second mode that is more common in web delivery is to consider the multiple paths as candidates and have the edge server choose the best performing path among the candidates in real-time. To select the best path, the edge server conducts periodic "races" between all the candidate paths and chooses the one that provides the fastest download time.

2. Role of overlay servers. Overlay servers may receive content and send it in a replicated fashion to multiple receivers. They may also perform coding and decoding operations if network coding is used in the overlay. As a result there is no conservation of flow that classical multicommodity flow problems possess.

3. Cost function. Each link has an associated cost and capacity. The cost function primarily encodes link performance, though one could also incorporate the bandwidth costs incurred in using that link. Link performance is characterized differently for different application contexts. For dynamic web content, the primary consideration is link latency, which impacts the download time perceived by the user. In this case, lower latency links have lower cost. In the case of video streaming, link throughput is a more important metric. A link with higher available bandwidth and lower packet loss that is capable of supporting higher throughputs is given a lower cost.

4. Capacity. Both node and link capacities are used to reflect the amount of traffic demand that can be routed through the node and link respectively. Capacity incorporates both server and network resources as well as contractual requirements that bound the amount of traffic that can be sent.

5. Optimized transport protocols. CDNs often use optimized transport protocols between nodes that they own and operate. Thus, the communication between overlay servers and edge servers are optimized. Often a reverse proxy is present at the origin or proximal to it, allowing for origin traffic to be optimized as well. The link performance model that is incorporated as a cost reflects the functioning of these protocols.

5.3 Algorithmic Solutions

The considerations listed above lead to different variants of the overlay construction problem for different application contexts. As a result, multiple algorithms are used in practice to solve these variants. We list two examples below.

1. *Dynamic web content.* Overlays that are more latency sensitive such as those for routing dynamic web content use customized algorithms for the well-studied all-pairs shortest-path (APSP) problem. Note that overlay construction involves both cost and capacity constraints. A first step to constructing an APSP instance is to perform Lagrangian relaxation (i.e., penalizing violations of constraints rather than forbidding them) [3] to encode the capacity constraints as a cost that can be added to existing link-performance-dependent cost terms. Customized algorithms for APSP can then be applied to compute the least cost paths between each origin-edge pair, i.e., the algorithm yields paths that have the best performance without capacity violations.

2. *Live videos.* For throughput sensitive traffic such as

[1] While overlay servers and edge servers are conceptually distinct as shown here, a typical physical server may simultaneously play both roles in a CDN.

[2] A path from origin to the edge server that does not pass through any intermediate overlay servers is called the *direct* path. Overlays always use the direct path when no overlay path is superior to it.

live videos, a different approach may be employed. In particular, given the large amounts of data transported across the overlay, particular attention must be paid to the various capacity constraints in the overlay and the bandwidth bottlenecks in the Internet. One approach is to formulate a mixed integer program (MIP) that captures all the performance, capacity, and bandwidth constraints. The MIP often cannot be solved efficiently as it is NP-hard to so. To overcome this problem, we can "relax" the integral variables in the MIP so that they can take on real values, resulting in a linear program (LP). The LP can then be solved efficiently to yield a fractional solution that can then be "rounded" to get an integral solution. The solution produced in this fashion is of course not optimal. But, in some cases, the solution can be shown to be near-optimal.

For a more in-depth treatment of an algorithmic solution for constructing overlays for live streaming, the reader is referred to [5, 4], while the system architecture of a live streaming network is described in [19]. The algorithm produces an overlay using LP relaxation that is rounded to an integral solution using GAP rounding. The overlay produced is guaranteed to be within a logarithmic factor of optimal.

5.4 Empirical Benefits

The empirical benefits of an overlay network as described above are two-fold. First, on a *consistent* basis, overlay routing provides better user-perceived performance, such as faster download times or fewer video stream rebuffers. The improvement in performance is due to many factors that favor overlay paths from origins to edge servers to the direct paths provided by the Internet. The overlay paths may have lower latency and/or higher throughput depending on the application context. The CDN can save the overhead of establishing TCP connections between different nodes within the overlay by holding them persistently. The CDN may also use optimized transport protocols between their nodes. Our empirical results, however, do not separate out the benefits of each effect.

A second benefit is what one could term as a *catastrophe insurance*. Catastrophic events such as a cable cut are not rare on the Internet. During such an event, overlays provide alternate paths when the direct path from origin to edge is impacted, leading to outsized improvements in both availability and performance.

There has been much work in quantifying the benefits of overlays. A large-scale empirical study of Akamai's routing overlay is presented in [26]. The empirical observations reported here are based on [30], which describes many different overlay technologies at Akamai and their relative benefits.

5.4.1 Consistent benefits

The routing overlay provides significant and consistent performance benefits by discovering and using better performing overlay paths for communication, even when there are no significant catastrophic network events. Figure 10, which was reported in [30], shows the performance benefit of an overlay for dynamic web content. During the operation of the overlay measured in this experiment, each edge server wanting to download content from an origin chooses the best overlay path from a set of candidate overlay paths provided by the overlay construction algorithm. In this experiment, a web site hosted in an origin in Dallas was accessed by

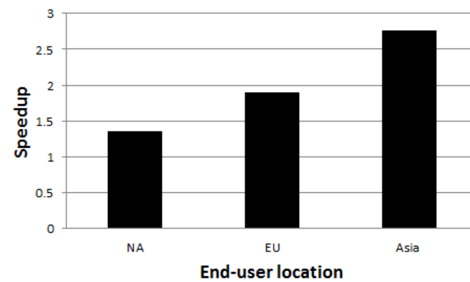

Figure 10: A routing overlay provides significant speedups by choosing better performing paths from the origin to the client. Key: North America (NA), Europe (EU), Asia.

"agents" deployed around the world that simulate clients by downloading web content periodically using an embedded browser[3]. For the experiment, the agents repeatedly downloaded a single dynamic (i.e., uncacheable) 38KB web page. The agents downloaded the web content through Akamai's CDN, which uses the routing overlay, and also downloaded the content directly from origin. Speedup is the ratio of the download time of the file when downloaded directly from the origin to the download time of the same file when downloaded using the routing overlay. Thus, speedup measures the amount of benefit that the routing overlay is providing. The results were then aggregated by continent and the average speedup is shown in the figure. Note that the speedup is significant in all geographies. The speedup, however, increases as the client (i.e., agent) moves farther away from the origin. This is a natural consequence of the fact that when the content needs to travel a longer distance from origin to edge there are more opportunities for optimizing that communication using better overlay paths.

5.4.2 Catastrophe Insurance

Overlay routing provides substantially more benefit during catastrophic events. In [30], performance measurements collected during the occurrence of one such event are reported. In April 2010, a large-scale Internet outage occurred when a submarine communications cable system (called SEA-ME-WE 4), which links Europe with the Middle East and South Asia, was cut. The cable underwent repairs from 25 April to 29 April during which time several cable systems were affected, severely impacting Internet connectivity in many regions across the Middle East, Africa, and Asia.

Figure 11 shows the download time experienced by clients in Asia using a routing overlay during the outage period in comparison with the download time experienced by the same clients without the overlay. Specifically, times were measured for agents distributed across India, Malaysia, and Singapore to download a dynamic (i.e., uncacheable) web page

[3]Agents are often used for repeated "apples-to-apples" performance measurements of the same content over long time periods, something real users would find too boring to do!

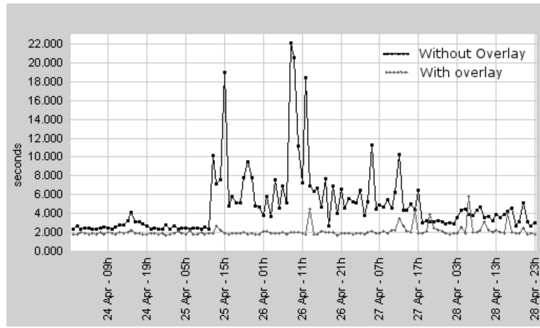

Figure 11: Performance of the routing overlay during a cable cut.

approximately 70KB in size hosted at an origin in Boston. The agents downloaded two versions of the web page - one directly from the origin in Boston, and another through the routing overlay. It can be seen that during the outage, the performance of the download directly from the Boston origin to Asia was very slow. Downloads using the routing overlay, however, did not experience much degradation on account of alternate overlay paths being used.

6. LEADER ELECTION AND CONSENSUS

Leader election algorithms are employed ubiquitously by the CDN in dozens of different distributed system components. Because these algorithms are tricky to design and implement correctly, the CDN has developed a leader election software library. The algorithms in this library can produce different leadership outcomes. For instance, some algorithms guarantee that at least one server will be elected as a leader, while others guarantee at most one. Leader election is an instance of the more general problem of consensus. There are numerous scenarios where general consensus protocols are key for the correct functioning of a CDN. For this purpose, more complex algorithms such Paxos [20] are used by multiple CDN components.

6.1 Leader Election

We start with the problem of leader election. Every decision-making software component of the CDN is executed in a replicated fashion on many servers that often reside in multiple data centers. Examples of components replicated in this fashion include the global and local load balancers described in the earlier sections. The key reason for replicating the decision-making components is to ensure that the CDN will continue to function even if some servers or even entire data centers fail. In fact, such failures are common in Internet-scale systems.

A server executing a decision-making component takes real-time inputs from several sources and produces outputs that are consumed by other components running on other servers. When such a component is replicated across multiple servers, each server independently collects the required data and computes the outputs. Even if the software running on these servers is identical, the outputs produced by each server could be different, since each server could be using slightly different or even partial sets of inputs. Therefore, it is necessary to select a server called the *leader* whose output is used by other system components that require it. The leader is determined in a distributed fashion using a leader election algorithm. We now describe some of the key assumptions and concepts that underlie leader election.

1) Failure model. The server failure model assumed by the algorithm designers is that a server may cease to function at any instant in time, and may likewise resume at any instant, but will not operate in a malicious or Byzantine [21] manner. (The consequences of server compromise go far beyond any impact on leader election algorithms!) The network may also fail at arbitrary times, but, by assumption, only by preventing pairs of servers from communicating or by introducing arbitrarily large packet delays. In particular, network failures may partition the servers into disjoint subsets, where the servers in each subset can communicate with each other, but servers in different subsets cannot.

2) Candidate set. The servers participating in a leader election form the candidate set. The servers are always configured to know the candidate set in advance, although there is never a guarantee that every candidate server can communicate with every other candidate server. Ideally, servers participating in a leader election algorithm should be configured with the same candidate set or problems may arise. For example, ties in voting for a leader are often broken by simply choosing the candidate with the smallest numbered IP address. If different participants have different candidate sets, there may not be an agreed-upon smallest address. However, periods where different servers have different candidate sets is unavoidable. When a new configuration is released on the network, it is inevitable that some servers have the latest candidate set while others have the older version. The leader election algorithm must be designed such that it does not produce adverse outcomes during such periods.

3) Health. It is important that the elected leader is "healthy" in terms of being able to perform its duties. A candidate may be disqualified from participating in the election due to "poor health." Health is a numerical score that is computed differently for different components. Generally, a candidate server is considered healthy if it has recent versions of all the data feeds needed for its computation, has the resources to execute the computation, and is able to communicate with other servers that require its outputs. For instance, a server performing global load balancing cannot be considered healthy if it does not have sufficiently recent values for the map unit demands and cluster capacities that are required to execute the stable marriage algorithm. A leader is often chosen to be healthiest candidate, breaking ties using their IP addresses. The aim is to prevent a server from becoming the leader if it has an invalid, outdated, or incomplete view of the data needed to make decisions.

4) Electoral process. Elections are triggered when there is no current leader. All candidates broadcast health values with a fixed (but configurable) periodicity to every one else. Note that each candidate not only broadcasts their own health but also the latest health values for others it has heard from. Each candidate chooses a leader using pre-determined rules based on its own health value and the health values it receives from others. The periodic broadcasts of health values serves a similar role as the heartbeats in the leader election protocol of Raft [25]. Specifically, if the heart beat from

the current leader is not heard for a specified timeout period, it is presumed “dead” and will trigger the election of a new leader. The timeout value is chosen to be larger so as not to trigger unnecessary leadership changes, e.g., heartbeat with a 5-second periodicity could have a 60-second timeout.

5. Outcome requirements. There are two outcome requirements that are common in leader election within the CDN. In an “at-most-one” election, at most one leader is desired across the entire CDN. In this case, leadership requires an absolute majority within the entire candidate set. In the case of a network partition, it is possible that no leader is chosen. Whereas in an “at-least-one” election, an absolute majority is not required, we allow multiple leaders, one in each partition. We now provide examples where these different outcomes are appropriate.

6.2 At-Least-One Leader Election

While the goal of a leader election algorithm is to select a single global leader, in many circumstances it may be safe, or even desirable, to allow each subset in a network partition to elect its own leader. One example comes from the system that uses consistent hashing to balance load within a cluster, as described in Section 3. Each server in a cluster is configured so that it has a list of the addresses of all servers in the cluster, whether or not it can communicate with them. Each server also has an identical copy of the mapping of servers in the cluster to the unit circle for each serial number. Periodically (on a time scale of seconds) the servers within a cluster try to contact each other to exchange information about how much traffic (e.g., megabits per second) they are serving for each serial number. If there is a partition within the cluster, then this information is exchanged within subsets.

The assignment of servers to serial numbers is computed as follows. Each server independently adds up the traffic for each serial number. Then, based on the total traffic, the server determines how many servers, k, to assign to the serial number, and creates a list of the first k servers within the subset that appear on the unit circle. The lists computed by different servers in a subset should be nearly identical, although there may be small differences based on the timings at which traffic measurements are taken or reported, or based on different views of which servers are currently on-line. Rather than use different solutions simultaneously within a subset, however, a leader election algorithm is applied, and the leader distributes its solution to the other members of the subset. The servers in the subset then use the leader’s solution to answer DNS queries for domain names such as a212.g.akamai.net.

In the event of a partition within a cluster, therefore, each subset has its own leader and operates independently without interfering with other subsets. This circumstance is not ideal, however, because each subset may not have enough storage capacity to cache all of the popular content that formerly was distributed across the entire cluster, so cache hit rates may suffer. On the other hand, the servers in a subset cannot direct clients to servers in other subsets, because the other subsets may be completely inoperable. Fortunately, partitions within a cluster are very rare, because the servers within a cluster are almost always housed in the same room.

6.3 At-Most-One Leader Election

Perhaps surprisingly, there are circumstances in which it is better for a leader election algorithm to fail to select any leader rather than select different leaders for different subsets in a network partition. (At-most-one-leader semantics is achieved by requiring a majority of the candidates to vote for the same leader.) Leader elections that produce at most one leader is suitable for software components that obey two criteria: (i) the periodicity with which the component executes and produces a new output is flexible, i.e., it is relatively safe for the other components of the CDN to operate based on a slightly older output; (ii) the use of two different versions of the output can lead to unpredictable or adverse consequences.

As an example, consider the software component (called “topology discovery” [9]), which explores the current structure of the Internet by performing network measurements and then outputs a map unit definition table, which lists all the map units, and assigns an identification number to each map unit. Over time, as the structure of the Internet and its users change, the map unit definitions also change. Several components may share a common map unit definition table to perform their functions. For instance, one component uses the map unit definitions to compute the demand from each map unit, while the global load balancer discussed in Section 2 uses the map unit definitions and their demands to route traffic to clusters. If there are two or more *different* map unit definition tables in the system, numerous problems may arise, such as data and decisions meant for one map unit accidentally being associated with or applied to another. As with any other component, topology discovery is performed in a replicated fashion on several servers. However, for this component, if leader election produces no leader and no output for a short period of time, it is safer for other components to operate with a last-known-good version of the map unit definition table than to operate with potentially conflicting versions of it.

6.4 Consensus

There are many contexts where full-blown consensus algorithms are required for the proper functioning of a CDN. Consider the case of a CDN delivering an e-commerce Web site. For better performance the user’s session state (such as the contents of his/her shopping cart) can be stored in the proximal edge server that is serving the user, rather than at a centralized origin site. But, what would happen if that server were to fail? It is not acceptable for the user to lose his or her session state! Thus, there is a need for the session state to be replicated across different servers and the copies to be kept consistent, so that server failures do not disrupt user transactions. There are several other such applications within a CDN that require a distributed, replicated, fault-tolerant store that supports atomic reads and writes. The Paxos algorithm [20] has been implemented to provide such functionality in the CDN, and the Raft [25] algorithm appears to be an attractive alternative.

7. CONCLUDING REMARKS

This paper explores the interaction between algorithmic research and commercial distributed systems based on our personal experiences in building one of the largest distributed systems in the world, Akamai’s content delivery network. While our examples are by no means comprehensive, we hope that they illustrate how research influenced the design of the CDN and how the system-building challenges inspired

more research.

Pasteur's quadrants [31] provide an intuitive classification of research based on whether its goal is a fundamental understanding (i.e., basic research), a particular use for society (i.e., applied research), or both (i.e., use-inspired basic research)[4]. Our examples illustrate the impact of research of all three types.

Basic research in stable allocations began decades before the first CDNs were built. Applying stable allocations to networked systems, however, was never the intention of the early researchers. Yet the framework provides an elegant way to formulate and solve the global load-balancing problem. Moreover, the impact is not a one-way street. The real-world implementation of the global load-balancing system required modeling multiple capacity constraints, which inspired further research, eventually leading to algorithms in which constraints are expressed as resource trees.

In contrast to stable allocations, the algorithmic work on consistent hashing is a classic example of use-inspired basic research. A primary motivator of the research was in fact load balancing within a CDN, although it has subsequently found applications in numerous other contexts.

Finally, in our examples, developing the algorithms is not the end, but the beginning. Much applied research is required to translate algorithmic ideas into practice. In some cases, the simple tractable models used for analysis do not adequately model the complexities of the real world. Therefore, theorems of optimality may not strictly apply in practice. In most cases, the algorithm that is actually implemented differs from the algorithms analyzed in the literature. In all examples, however, provably effective algorithms provided the conceptual basis for the system that was built. From this standpoint, an algorithm that proposes a conceptually new way of thinking about a problem may be as important as being the most efficient in its class.

8. ACKNOWLEDGEMENTS

The systems presented in this paper are the product of sixteen years of hard work by numerous Akamai engineers and researchers. While it is not feasible to list all of their names here, we have cited publications describing these systems where possible. The research of the broader community laid the foundations for building these systems, and we have cited the relevant work where appropriate. The authors would also like to thank specific colleagues for help in preparing this paper. Ming Dong Feng conducted the Bloom filter experiments described in Section 4. The empirical evaluation of overlay routing in Section 5 was first reported in [30]. We also thank Mangesh Kasbekar, Ming Dong Feng, Marcelo Torres, Brian Dean, and Michel Goemans for insightful discussions about the material. This work was supported in part by NSF grant CNS-1413998.

9. REFERENCES

[1] iostat - linux man page. `http://linux.die.net/man/1/iostat`.

[2] Atila Abdulkadiroğlu, Parag A. Pathak, and Alvin E. Roth. The New York City high school match. *American Economic Review*, pages 364–367, 2005.

[3] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[4] Konstantin Andreev, Bruce M. Maggs, Adam Meyerson, Jevan Saks, and Ramesh K. Sitaraman. Algorithms for constructing overlay networks for live streaming. *arXiv preprint arXiv:1109.4114*, 2011.

[5] Konstantin Andreev, Bruce M. Maggs, Adam Meyerson, and Ramesh K. Sitaraman. Designing overlay multicast networks for streaming. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 149–158. ACM, 2003.

[6] Mourad Baıou and Michel Balinski. Many-to-many matching: stable polyandrous polygamy (or polygamous polyandry). *Discrete Applied Mathematics*, 101(1):1–12, 2000.

[7] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[8] Andrei Broder and Michael Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.

[9] Fangfei Cheng, Ramesh K. Sitaraman, and Marcelo Torres. End-user mapping: Next generation request routing for content delivery. In *Proceedings of the 2015 ACM Conference on SIGCOMM*, SIGCOMM '15, 2015.

[10] John Dilley, Bruce Maggs, Jay Parikh, Harald Prokop, Ramesh Sitaraman, and Bill Weihl. Globally distributed content delivery. *Internet Computing, IEEE*, 6(5):50–58, 2002.

[11] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

[12] David Gale and Lloyd S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, pages 9–15, 1962.

[13] Michel Goemans. Load balancing in content delivery networks. *IMA Annual Program Year Workshop: Network Management and Design*, April 2003.

[14] Dan Gusfield and Robert W. Irving. *The Stable Marriage Problem: Structure and Algorithms*. MIT Press, Cambridge, MA, 1989.

[15] Kazuo Iwama and Shuichi Miyazaki. A survey of the stable marriage problem and its variants. In *Informatics Education and Research for Knowledge-Circulating Society, 2008. ICKS 2008. International Conference on*, pages 131–136. IEEE, 2008.

[16] D. Karger, A. Sherman, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. *Computer Networks*, 31(11):1203–1213, 1999.

[17] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[4]The scientists often cited as canonical examples in each quadrant are Niels Bohr (basic), Thomas Edison (applied), and Louis Pasteur (use-inspired basic).

[18] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. U.S. patent number 8,458,259: Method and apparatus for distributing requests among a plurality of resources, August 2002.

[19] Leonidas Kontothanassis, Ramesh Sitaraman, Joel Wein, Duke Hong, Robert Kleinberg, Brian Mancuso, David Shaw, and Daniel Stodolsky. A transport layer for live streaming in a content delivery network. *Proceedings of the IEEE*, 92(9):1408–1419, 2004.

[20] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[21] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[22] Daniel M. Lewin. Consistent hashing and random trees: Algorithms for caching in distributed networks. Master's thesis, Massachusetts Institute of Technology, 1998.

[23] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

[24] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. The Akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.

[25] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference*, pages 305–320, 2014.

[26] H. Rahul, M. Kasbekar, R. Sitaraman, and A. Berger. Towards realizing the performance and availability benefits of a global overlay network. In *Proceedings of the Passive and Active Measurement Conference*, 2006.

[27] Alvin E. Roth and Elliott Peranson. The redesign of the matching market for American physicians: Some engineering aspects of economic design. Technical report, National Bureau of Economic Research, 1999.

[28] Alvin E. Roth, Tayfun Sönmez, and M. Utku Ünver. Kidney exchange. *The Quarterly Journal of Economics*, pages 457–488, 2004.

[29] Alex Rousskov and Duane Wessels. Cache digests. *Computer Networks and ISDN Systems*, 30(22):2155–2168, 1998.

[30] Ramesh K. Sitaraman, Mangesh Kasbekar, Woody Lichtenstein, and Manish Jain. Overlay networks: An Akamai perspective. In Pathan, Sitaraman, and Robinson, editors, *Advanced Content Delivery, Streaming, and Cloud Services*. John Wiley & Sons, 2014.

[31] Donald E. Stokes. *Pasteur's Quadrant: Basic Science and Technological Innovation*. Brookings Institution Press, 1997.