Volley: Automated Data Placement for Geo-Distributed Cloud Services

Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman
Microsoft Research, {sagarwal, jdunagan, navendu, ssaroiu, alecw}@microsoft.com

Harbinder Bhogan
University of Toronto, hbhogan@cs.toronto.edu

Abstract: *As cloud services grow to span more and more globally distributed datacenters, there is an increasingly urgent need for automated mechanisms to place application data across these datacenters. This placement must deal with business constraints such as WAN bandwidth costs and datacenter capacity limits, while also minimizing user-perceived latency. The task of placement is further complicated by the issues of shared data, data inter-dependencies, application changes and user mobility. We document these challenges by analyzing month-long traces from Microsoft's Live Messenger and Live Mesh, two large-scale commercial cloud services.*

We present Volley, a system that addresses these challenges. Cloud services make use of Volley by submitting logs of datacenter requests. Volley analyzes the logs using an iterative optimization algorithm based on data access patterns and client locations, and outputs migration recommendations back to the cloud service.

To scale to the data volumes of cloud service logs, Volley is designed to work in SCOPE [5], a scalable MapReduce-style platform; this allows Volley to perform over 400 machine-hours worth of computation in less than a day. We evaluate Volley on the month-long Live Mesh trace, and we find that, compared to a state-of-the-art heuristic that places data closest to the primary IP address that accesses it, Volley simultaneously reduces datacenter capacity skew by over 2×, reduces inter-datacenter traffic by over 1.8× and reduces 75th percentile user-latency by over 30%.

1 Introduction

Cloud services continue to grow rapidly, with ever more functionality and ever more users around the globe. Because of this growth, major cloud service providers now use tens of geographically dispersed datacenters, and they continue to build more [10]. A major unmet challenge in leveraging these datacenters is automatically placing user data and other dynamic application data, so that a single cloud application can serve each of its users from the best datacenter for that user.

At first glance, the problem may sound simple: determine the user's location, and migrate user data to the closest datacenter. However, this simple heuristic ignores two major sources of cost to datacenter operators: WAN bandwidth between datacenters, and over-provisioning datacenter capacity to tolerate highly skewed datacenter utilization. In this paper, we show that a more sophisticated approach can both dramatically reduce these costs and still further reduce user latency. The more sophisticated approach is motivated by the following trends in modern cloud services:

Shared Data: Communication and collaboration are increasingly important to modern applications. This trend is evident in new business productivity software, such as Google Docs [16] and Microsoft Office Online [32], as well as social networking applications such as Facebook [12], LinkedIn [26], and Twitter [43]. These applications have in common that many reads and writes are made to shared data, such as a user's Facebook wall, and the user experience is degraded if updates to shared data are not quickly reflected to other clients. These reads and writes are made by groups of users who need to collaborate but who may be scattered worldwide, making it challenging to place and migrate the data for good performance.

Data Inter-dependencies: The task of placing shared data is made significantly harder by inter-dependencies between data. For example, updating the wall for a Facebook user may trigger updating the data items that hold the RSS feeds of multiple other Facebook users. These connections between data items form a communication graph that represents increasingly rich applications. However, the connections fundamentally transform the problem's mathematics: in addition to connections between clients and their data, there are connections in the communication graph in-between data items. This motivates algorithms that can operate on these more general graph structures.

Application Changes: Cloud service providers want to release new versions of their applications with ever greater frequency [35]. These new application features

can significantly change the patterns of data sharing and data inter-dependencies, as when Facebook released its instant messaging feature.

Reaching Datacenter Capacity Limits: The rush in industry to build additional datacenters is motivated in part by reaching the capacity constraints of individual datacenters as new users are added [10]. This in turn requires automatic mechanisms to rapidly migrate application data to new datacenters to take advantage of their capacity.

User Mobility: Users travel more than ever today [15]. To provide the same rapid response regardless of a user's location, cloud services should quickly migrate data when the migration cost is sufficiently inexpensive.

In this paper we present Volley, a system for automatic data placement across geo-distributed datacenters. Volley incorporates an iterative optimization algorithm based on weighted spherical means that handles the complexities of shared data and data inter-dependencies, and Volley can be re-run with sufficient speed that it handles application changes, reaching datacenter capacity limits and user mobility. Datacenter applications make use of Volley by submitting request logs (similar to Pinpoint [7] or X-Trace [14]) to a distributed storage system. These request logs include the client IP addresses, GUIDs identifying the data items accessed by the client requests, and the structure of the request "call tree", such as a client request updating Facebook wall 1, which triggers requests to data items 2 and 3 handling Facebook user RSS feeds.

Volley continuously analyzes these request logs to determine how application data should be migrated between datacenters. To scale to these data sets, Volley is designed to work in SCOPE [5], a system similar to Map-Reduce [11]. By leveraging SCOPE, Volley performs more than 400 machines hours worth of computation in less then a day. When migration is found to be worthwhile, Volley triggers application-specific data migration mechanisms. While prior work has studied placing static content across CDNs, Volley is the first research system to address placement of user data and other dynamic application data across geographically distributed datacenters.

Datacenter service administrators make use of Volley by specifying three inputs. First, administrators define the datacenter locations and a cost and capacity model (e.g., the cost of bandwidth between datacenters and the maximum amount of data per datacenter). Second, they choose the desired trade-off between upfront migration cost and ongoing better performance, where ongoing performance includes both minimizing user-perceived latency and reducing the costs of inter-datacenter communication. Third, they specify data replication levels and other constraints (e.g., three replicas in three different datacenters all located within Europe). This allows administrators to use Volley while respecting other external factors, such as contractual agreements and legislation.

In the rest of this paper, we first quantify the prevalence of trends such as user mobility in modern cloud services by analyzing month-long traces from Live Mesh and Live Messenger, two large-scale commercial datacenter services. We then present the design and implementation of the Volley system for computing data placement across geo-distributed datacenters. Next, we evaluate Volley analytically using the month-long Live Mesh trace, and we evaluate Volley on a live testbed consisting of 20 VMs located in 12 commercial datacenters distributed around the world. Previewing our results, we find that compared to a state-of-the-art heuristic, Volley can reduce skew in datacenter load by over $2\times$, decrease inter-datacenter traffic by over $1.8\times$, and reduce 75th percentile latency by over 30%. Finally, we survey related work and conclude.

2 Analysis of Commercial Cloud-Service Traces

We begin by analyzing workload traces collected by two large datacenter applications, Live Mesh [28] and Live Messenger [29]. Live Mesh provides a number of communication and collaboration features, such as file sharing and synchronization, as well as remote access to devices running the Live Mesh client. Live Messenger is an instant messaging application. In our presentation, we also use Facebook as a source for examples due to its ubiquity.

The Live Mesh and Live Messenger traces were collected during June 2009, and they cover all users and devices that accessed these services over this entire month. The Live Mesh trace contains a log entry for every modification to hard state (such as changes to a file in the Live Mesh synchronization service) and user-visible soft state (such as device connectivity information stored on a pool of in-memory servers [1]). The Live Messenger trace contains all login and logoff events, all IM conversations and the participants in each conversation, and the total number of messages in each conversation. The Live Messenger trace does not specify the sender or the size of individual messages, and so for simplicity, we model each participant in an IM conversation as having an equal likelihood of sending each message, and we divide the total message bytes in this conversation equally among all messages. A prior measurement study describes many aspects of user behavior in the Live Messenger system [24]. In both traces, clients are identified by application-level unique identifiers.

To estimate client location, we use a standard commercial geo-location database [34] as in prior work [36].

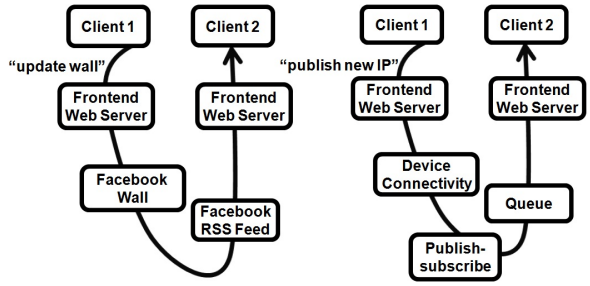

Figure 1. *Simplified data inter-dependencies in Facebook (left) and Live Mesh (right). In Facebook, an "updated wall" request arrives at a Facebook wall data item, and this data item sends the request to an RSS feed data item, which then sends it to the other client. In Live Mesh, a "publish new IP" request arrives at a Device Connectivity data item, which forwards it to a Publish-subscribe data item. From there, it is sent to a Queue data item, which finally sends it on to the other client. These pieces of data may be in different datacenters, and if they are, communication between data items incurs expensive inter-datacenter traffic.*

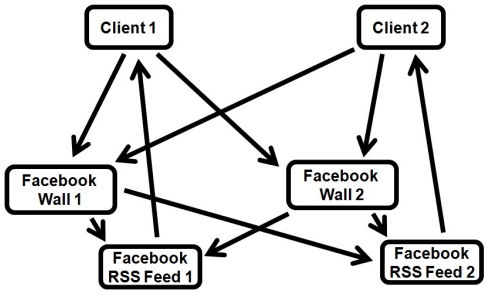

Figure 2. *Two clients, their four data items and the communication between them in the simplified Facebook example. Data placement requires appropriately mapping the four data items to datacenters so as to simultaneously achieve low inter-datacenter traffic, low datacenter capacity skew, and low latency.*

The database snapshot is from June 30th 2009, the very end of our trace period.

We use the traces to study three of the trends motivating Volley: shared data, data inter-dependencies, and user mobility. The other motivating trends for Volley, rapid application changes and reaching datacenter capacity limits, are documented in other data sources, such as developers describing how they build cloud services and how often they have to release updates [1, 35]. To provide some background on how data inter-dependencies arise in commercial cloud services, Figure 1 shows simplified examples from Facebook and Live Mesh. In the Facebook example, Client 1 updates its Facebook wall, which is then published to Client 2; in Facebook, this allows users to learn of each other's activities. In the Live Mesh example, Client 1 publishes its new IP address,

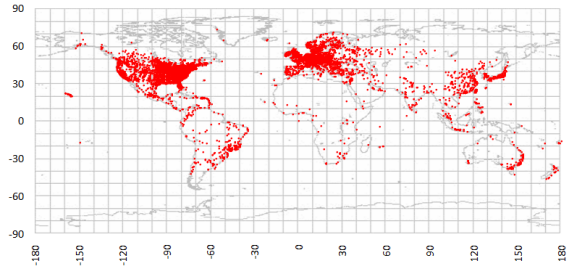

Figure 3. *Distribution of clients in the Mesh trace.*

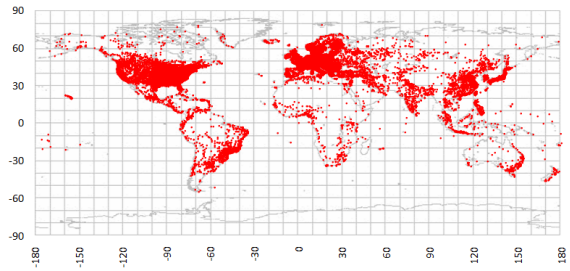

Figure 4. *Distribution of clients in the Messenger trace.*

which is routed to Client 2, enabling Client 2 to connect directly to Client 1; in Live Mesh, this is referred to as a notification session, and it enables both efficient file sharing and remote device access. The Figure caption provides additional details, as do other publications [1]. In both cases, the client operations involve multiple datacenter items; inter-datacenter traffic is minimized by co-locating these items, while the latency of this particular request is minimized by placing the data items as close as possible to the two clients.

Figure 2 attempts to convey some intuition for why data sharing and inter-dependencies make data placement challenging. The figure shows the web of connections between just two clients in the simplified Facebook example; these inter-connections determine whether a mapping of data items to datacenters achieves low inter-datacenter traffic, low datacenter capacity skew, and low latency. Actual cloud services face this problem with hundreds of millions of clients. Each client may access many data items, and these data items may need to communicate with each other to deliver results to clients. Furthermore, the clients may access the data items from a variety of devices at different locations. This leads to a large, complicated graph.

In order to understand the potential for this kind of inter-connection to occur between clients that are quite distant, we begin by characterizing the geographic diversity of clients in the traces.

Client Geographic Diversity: We first study the traces to understand the geographic diversity of these services' client populations. Figures 3 and 4 show the distribution of clients in the two traces on a map of the world. The figures show that both traces contain a geographi-

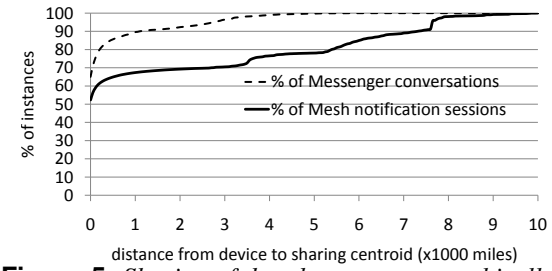

Figure 5. *Sharing of data between geographically distributed clients in the Messenger and Mesh traces. Large amounts of sharing occur between distant clients.*

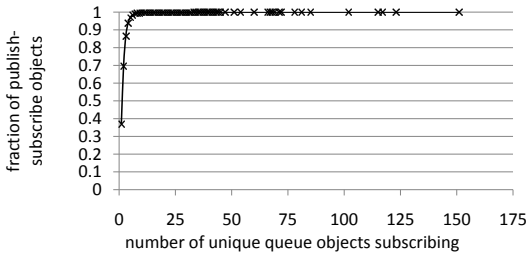

Figure 6. *Data inter-dependencies in Live Mesh between Publish-subscribe objects and Queue objects. A user that updates a hard state data item, such as a document stored in Live Mesh, will cause an update message to be generated at the Publish-subscribe object for that document, and all Queue objects that subscribe to it will receive a copy of the message. Each user or device that is sharing that document will have a unique Queue. Many Publish-subscribe objects are subscribed to by a single Queue, but there is a long tail of popular objects that are subscribed to by many Queues.*

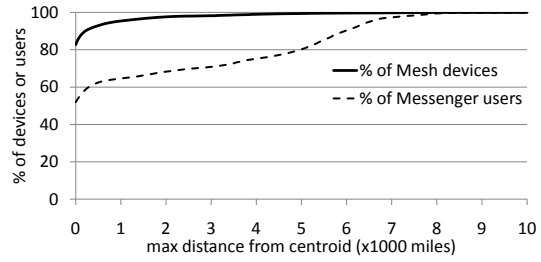

Figure 7. *Mobility of clients in the Messenger and Mesh traces. Most clients do not travel. However, a significant fraction do travel quite far.*

cally diverse set of clients, and thus these service's performance may significantly benefit from intelligent data placement.

Geographically Distant Data Sharing: We next study the traces to understand whether there is significant data sharing among distant users. For each particular data item, we compute its centroid (centroid on a sphere is computed using the weighted spherical mean methodology, which we describe in detail in Section 3). Figure 5 shows a CDF for the distance over which clients access data placed according to its centroid; data that is not shared has an access distance of 0, as does data shared by users whose IP addresses map to the same geographic location. Given the amount of collaboration across nations both within corporations and between them, it is perhaps not surprising that large amounts of sharing happens between very distant clients. This data suggests that even for static clients, there can be significant benefits to placing data closest to those who use it most heavily, rather than just placing it close to some particular client that accesses the data.

Data Inter-dependencies: We proceed to study the traces to understand the prevalence of data inter-dependencies. Our analysis focuses on Live Mesh because data inter-dependencies in Live Messenger have been documented in detail in prior work [24]. Figure 6 shows the number of Queue objects subscribing to receive notifications from each Publish-subscribe object; each such subscription creates a data inter-dependency where the Publish-subscribe object sends messages to the Queue object. We see that some Publish-subscribe objects send out notifications to only a single Queue object, but there is a long tail of popular Publish-subscribe objects. The presence of such data inter-dependencies motivates the need to incorporate them in Volley.

Client Mobility: We finally study the traces to understand the amount of client mobility in these services' client populations. Figure 7 shows a CDF characterizing client mobility over the month of the trace. To compute this CDF, we first computed the location of each client at each point in time that it contacted the Live Mesh or Live Messenger application using the previously described methodology, and we then compute the client's centroid. Next, we compute the maximum distance between each client and its centroid. As expected, we observe that most clients do not move. However, a significant fraction do move (more in the Messenger trace than the Mesh trace), and these movements can be quite dramatic – for comparison purposes, antipodal points on the earth are slightly more than 12,000 miles apart.

From these traces, we cannot characterize the reason for the movement. For example, it could be travel, or it could be that the clients are connecting in through a VPN to a remote office, causing their connection to the public Internet to suddenly emerge in a dramatically different location. For Volley's goal of reducing client latency, there is no need to distinguish between these different causes; even though the client did not physically move in the VPN case, client latency is still minimized by moving data closer to the location of the client's new connection to the public Internet. The long tail of client mobility suggests that for some fraction of clients, the ideal data placement changes significantly during this month.

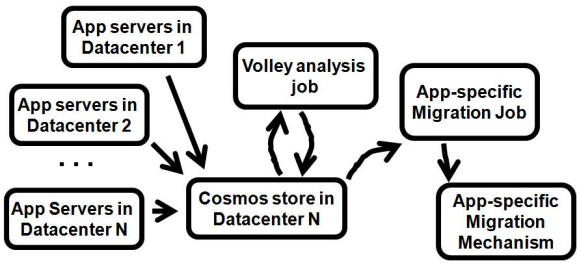

Figure 8. *Dataflow for an application using Volley.*

This data does leave open the possibility that some fraction of the observed clients are bots that do not correspond to an actual user (i.e., they are modified clients driven by a program). The current analysis does filter out the automated clients that the service itself uses for doing performance measurement from various locations. Prior work has looked at identifying bots automatically [45], and Volley might benefit from leveraging such techniques.

3 System Design and Implementation

The overall flow of data in the system is shown in Figure 8. Applications make use of Volley by logging data to the Cosmos [5] distributed storage system. The administrator must also supply some inputs, such as a cost and capacity model for the datacenters. The Volley system frequently runs new analysis jobs over these logs, and computes migration decisions. Application-specific jobs then feed these migration decisions into application-specific data migration mechanisms. We now describe these steps in greater detail.

3.1 Logging Requests

To utilize Volley, applications have to log information on the requests they process. These logs must enable correlating requests into "call trees" or "runtime paths" that capture the logical flow of control across components, as in Pinpoint [7] or X-Trace [14]. If the source or destination of a request is movable (i.e., because it is a data item under the control of the cloud service), we log a GUID identifier rather than its IP address; IP addresses are only used for endpoints that are not movable by Volley, such as the location that a user request came from. Because Volley is responsible for placing all the data named by GUIDs, it already knows their current locations in the steady state. It is sometimes possible for both the source and destination of a request to be referred to by GUIDs—this would happen, for example, in Figure 1, where the GUIDs would refer to Client 1's Facebook wall and Client 2's Facebook RSS feed. The exact fields in the Volley request logs are shown in Table 1. In total, each record requires only 100 bytes.

There has been substantial prior work modifying applications to log this kind of information, and many commercial applications (such as the Live Mesh and Live Messenger services analyzed in Section 2) already log a superset of this data. For such applications, Volley can incorporate simple filters to extract out the relevant subset of the logs.

For the Live Mesh and Live Messenger commercial cloud services, the data volumes from generating Volley logs are much less than the data volumes from processing user requests. For example, recording Volley logs for all the requests for Live Messenger, an IM service with hundreds of millions of users, only requires hundreds of GB per day, which leads to an average bandwidth demand in the tens of Mbps [24]. Though we cannot reveal the exact bandwidth consumption of the Live Mesh and Live Messenger services due to confidentiality concerns, we can state that tens of Mbps is a small fraction of the total bandwidth demands of the services themselves. Based on this calculation, we centralize all the logs in a single datacenter; this then allows Volley to run over the logs multiple times as part of computing a recommended set of migrations.

3.2 Additional Inputs

In addition to the request logs, Volley requires four inputs that change on slower time scales. Because they change on slower time scales, they do not noticeably contribute to the bandwidth required by Volley. These additional inputs are (1) the requirements on RAM, disk, and CPU per transaction for each type of data handled by Volley (e.g., a Facebook wall), (2) a capacity and cost model for all the datacenters, (3) a model of latency between datacenters and between datacenters and clients, and (4) optionally, additional constraints on data placement (e.g., legal constraints). Volley also requires the current location of every data item in order to know whether a computed placement keeps an item in place or requires migration. In the steady state, these locations are simply remembered from previous iterations of Volley.

In the applications we have analyzed thus far, the administrator only needs to estimate the average requirements on RAM, disk and CPU per data item; the administrator can then rely on statistical multiplexing to smooth out the differences between data items that consume more or fewer resources than average. Because of this, resource requirements can be estimated by looking at OS-provided performance counters and calculating the average resource usage for each piece of application data hosted on a given server.

The capacity and cost models for each datacenter specify the RAM, disk and CPU provisioned for the service in that datacenter, the available network bandwidth for both egress and ingress, and the charging model for service use of network bandwidth. While energy usage is a significant cost for datacenter owners, in our expe-

Request Log Record Format

Field	Meaning
Timestamp	Time in seconds when request was received (4B)
Source-Entity	A GUID if the source is another data item, an IP address if it is a client (40B)
Request-Size	Bytes in request (8B)
Destination-Entity	Like Source-Entity, either a GUID or an IP address (40B)
Transaction-Id	Used to group related requests (8B)

Table 1. *To use Volley, the application logs a record with these fields for every request. The meaning and size in bytes of each field are also shown.*

Migration Proposal Record Format

Field	Meaning
Entity	The GUID naming the entity (40B)
Datacenter	The GUID naming the new datacenter for this entity (40B)
Latency-Change	The average change in latency per request to this object (4B)
Ongoing-Bandwidth-Change	The change in egress and ingress bandwidth per day (4B)
Migration-Bandwidth	The one-time bandwidth required to migrate (4B)

Table 2. *Volley constructs a set of proposed migrations described using the records above. Volley then selects the final set of migrations according to the administrator-defined trade-off between performance and cost.*

rience this is incorporated as a fixed cost per server that is factored in at the long timescale of server provisioning. Although datacenter owners may be charged based on peak bandwidth usage on individual peering links, the unpredictability of any given service's contribution to a datacenter-wide peak leads datacenter owners to charge services based on total bandwidth usage, as in Amazon's EC2 [2]. Accordingly, Volley helps services minimize their total bandwidth usage. We expect the capacity and cost models to be stable at the timescale of migration. For fluid provisioning models where additional datacenter capacity can be added dynamically as needed for a service, Volley can be trivially modified to ignore provisioned capacity limits.

Volley needs a latency model to make placement decisions that reduce user perceived latency. It allows different static or dynamic models to be plugged in. Volley migrates state at large timescales (measured in days) and hence it should use a latency model that is stable at that timescale. Based on the large body of work demonstrating the effectiveness of network coordinate systems, we designed Volley to treat latencies between IPs as distances in some n-dimensional space specified by the model. For the purposes of evaluation in the paper, we rely on a static latency model because it is stable over these large timescales. This model is based on a linear regression of great-circle distance between geographic coordinates; it was developed in prior work [36], where it was compared to measured round trip times across millions of clients and shown to be reasonably accurate. This latency model requires translating client IP addresses to geographic coordinates, and for this purpose we rely on the geo-location database mentioned in Section 2. This geo-location database is updated every two weeks. In this work, we focus on improving latency to users and not bandwidth to users. Incorporating bandwidth would require both specifying a desired latency bandwidth tradeoff and a model for bandwidth between arbitrary points in the Internet.

Constraints on data placement can come in many forms. They may reflect legal constraints that data be hosted only in a certain jurisdiction, or they may reflect operational considerations requiring two replicas to be physically located in distant datacenters. Volley models such replicas as two distinct data items that may have a large amount of inter-item communication, along with the constraint that they be located in different datacenters. Although the commercial cloud service operators we spoke with emphasized the need to accommodate such constraints, the commercial applications we study in this paper do not currently face constraints of this form, and so although Volley can incorporate them, we did not explore this in our evaluation.

3.3 Volley Algorithm

Once the data is in Cosmos, Volley periodically analyzes it for migration opportunities. To perform the analysis, Volley relies on the SCOPE [5] distributed execution infrastructure, which at a high level resembles MapReduce [11] with a SQL-like query language. In our current implementation, Volley takes approximately 14 hours to run through one month's worth of log files; we analyze the demands Volley places on SCOPE in more detail in Section 4.4.

Volley's SCOPE jobs are structured into three phases. The search for a solution happens in Phase 2. Prior work [36] has demonstrated that starting this search in a good location improves convergence time, and hence

Recursive Step:

$$wsm\left(\{w_i, \vec{x}_i\}_{i=1}^{N}\right) = interp\left(\frac{w_N}{\sum w_i}, \vec{x}_N, wsm(\{w_i, \vec{x}_i\}_{i=1}^{N-1}\right)$$

Base Case:

$$interp(w, \vec{x}_A, \vec{x}_B) = (\phi_C, \lambda_C) = \vec{x}_C$$

$$\begin{aligned}
d &= \cos^{-1}\left[\cos(\phi_A)\cos(\phi_B) + \sin(\phi_A)\sin(\phi_B)\cos(\lambda_B - \lambda_A)\right] \\
\gamma &= \tan^{-1}\left[\frac{\sin(\phi_B)\sin(\phi_A)\sin(\lambda_B - \lambda_A)}{\cos(\phi_A) - \cos(d)\cos(\phi_B)}\right] \\
\beta &= \tan^{-1}\left[\frac{\sin(\phi_B)\sin(wd)\sin(\gamma)}{\cos(wd) - \cos(\phi_A)\cos(\phi_B)}\right] \\
\phi_C &= \cos^{-1}\left[\cos(wd)\cos(\phi_B) + \sin(wd)\sin(\phi_B)\cos(\gamma)\right] \\
\lambda_C &= \lambda_B - \beta
\end{aligned}$$

Figure 9. *Weighted spherical mean calculation. The weighted spherical mean (wsm()) is defined recursively as a weighted interpolation (interp()) between pairs of points. Here, w_i is the weight assigned to $\vec{x}_i$, and $\vec{x}_i$ (the coordinates for node i) consists of ϕ_i, the latitudinal distance in radians between node i and the North Pole, and λ_i, the longitude in radians of node i. The new (interpolated) node C consists of w parts node A and $1 - w$ parts node B; d is the current distance in radians between A and B; γ is the angle from the North Pole to B to A (which stays the same as A moves); β is the angle from B to the North Pole to A's new location. These are used to compute $\vec{x}_C$, the result of the interp(). For simplicity of presentation, we omit describing the special case for antipodal nodes.*

Phase 1 computes a reasonable initial placement of data items based on client IP addresses. Phase 2 iteratively improves the placement of data items by moving them freely over the surface of the earth—this phase requires the bulk of the computational time and the algorithm code. Phase 3 does the needed fix up to map the data items to datacenters and to satisfy datacenter capacity constraints. The output of the jobs is a set of potential migration actions with the format described in Table 2. Many adaptive systems must incorporate explicit elements to prevent oscillations. Volley does not incorporate an explicit mechanism for oscillation damping. Oscillations would occur only if *user behavior* changed in response to Volley migration in such a way that Volley needed to move that user's state back to a previous location.

Phase 1: Compute Initial Placement. We first map each client to a set of geographic coordinates using the commercial geo-location database mentioned earlier. This IP-to-location mapping may be updated between Volley jobs, but it is not updated within a single Volley job. We then map each data item that is directly accessed by a client to the weighted average of the geographic coordinates for the client IPs that access it. This is done using the weighted spherical mean calculation shown in Figure 9. The weights are given by the amount of communication between the client nodes and the data item whose initial location we are calculating. The weighted spherical mean calculation can be thought of as drawing an arc on the earth between two points, and then finding the point on the arc that interpolates between the two initial points in proportion to their weight. This operation is then repeated to average in additional points. The recursive definition of weighted spherical mean in Figure 9 is conceptually similar to defining the more familiar weighted mean recursively, e.g.,

$$\text{weighted-mean}(\{3, x_k\}, \{2, x_j\}, \{1, x_i\}) = \left(\frac{3}{6} \cdot x_k + \frac{3}{6} \cdot \text{weighted-mean}(\{2, x_j\}, \{1, x_i\})\right)$$

Compared to weighted mean, weighted spherical mean has the subtlety that the rule for averaging two individual points has to use spherical coordinates.

Figure 10 shows an example of this calculation using data from the Live Mesh trace: five different devices access a single shared object from a total of eight different IP addresses; device D accesses the shared object far more than the other devices, and this leads to the weighted spherical mean (labeled "centroid" in the figure) being placed very close to device D.

Finally, for each data item that is never accessed directly by clients (e.g., the Publish-subscribe data item in the Live Mesh example of Figure 1), we map it to the weighted spherical mean of the data items that communicate with it using the positions these other items were already assigned.

Phase 2: Iteratively Move Data to Reduce Latency. Volley iteratively moves data items closer to both clients and to the other data items that they communicate with. This iterative update step incorporates two earlier ideas: a weighted spring model as in Vivaldi [9] and spherical coordinates as in Htrae [36]. Spherical coordinates define the locations of clients and data items in a way that is more conducive to incorporating a latency model for geographic locations. The latency distance between two nodes and the amount of communication between them increase the spring force that is pulling them together. However, unlike a network coordinate system, nodes in

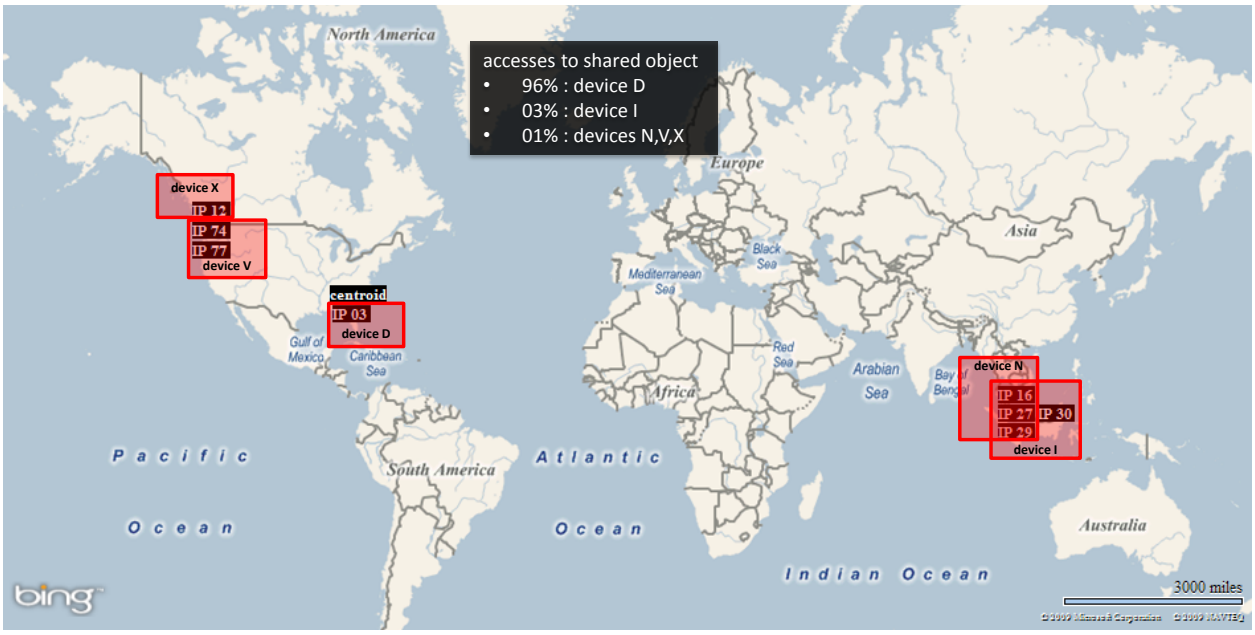

Figure 10. *An example of a shared object being placed at its weighted spherical mean (labeled "centroid" in the Figure). This particular object, the locations of the clients that access it, and their access ratios are drawn from the Live Mesh trace. Because device D is responsible for almost all of the accesses, the weighted spherical mean placement for the object is very close to device D's location.*

$$w = \frac{1}{1 + \kappa \cdot d \cdot l_{AB}}$$
$$\vec{x}_A^{new} = interp(w, \vec{x}_A^{current}, \vec{x}_B^{current})$$

Figure 11. *Update rule applied to iteratively move nodes with more communication closer together. Here, w is a fractional weight that determines how much node A is moved towards node B, l_{AB} is the amount of communication between the two nodes, d is the distance between nodes A and B, $\vec{x}_A^{current}$ and $\vec{x}_B^{current}$ are the current locations of node A and B, $\vec{x}_A^{new}$ is the location of A after the update, and κ is an algorithmic constant.*

Volley only experience contracting forces; the only factor preventing them from collapsing to a single location is the fixed nature of client locations. This yields the update rule shown in Figure 11. In our current implementation, we simply run a fixed number of iterations of this update rule; we show in Section 4 that this suffices for good convergence.

Intuitively, Volley's spring model attempts to bring data items closer to users and to other data items that they communicate with regularly. Thus it is plausible that Volley's spring model will simultaneously reduce latency and reduce inter-datacenter traffic; we show in Section 4 that this is indeed the case for the commercial cloud services that we study.

Phase 3: Iteratively Collapse Data to Datacenters. After computing a nearly ideal placement of the data items on the surface of the earth, we have to modify this placement so that the data items are located in datacenters, and the set of items in each datacenter satisfies its capacity constraints. Like Phase 2, this is done iteratively: initially, every data item is mapped to its closest datacenter. For datacenters that are over their capacity, Volley identifies the items that experience the fewest accesses, and moves all of them to the next closest datacenter. Because this may still exceed the total capacity of some datacenter due to new additions, Volley repeats the process until no datacenter is over capacity. Assuming that the system has enough capacity to successfully host all items, this algorithm always terminates in at most as many iterations as there are datacenters in the system.

For each data item that has moved, Volley outputs a migration proposal containing the new datacenter location, the new values for latency and ongoing inter-datacenter bandwidth, and the one-time bandwidth required for this migration. This is a straightforward calculation using the old data locations, the new data locations, and the inputs supplied by the datacenter service administrator, such as the cost model and the latency model. These migration proposals are then consumed by application-specific migration mechanisms.

3.4 Application-specific Migration

Volley is designed to be usable by many different cloud services. For Volley to compute a recommended placement, the only requirement it imposes on the cloud

service is that it logs the request data described in Table 1. Given these request logs as input, Volley outputs a set of migration proposals described in Table 2, and then leaves the actual migration of the data to the cloud service itself. If the cloud service also provides the initial location of data items, then each migration proposal will include the bandwidth required to migrate, and the expected change in latency and inter-datacenter bandwidth after migration.

Volley's decision to leave migration to application-specific migration mechanisms allows Volley to be more easily applied to a diverse set of datacenter applications. For example, some datacenter applications use migration mechanisms that follow the pattern of marking data read-only in the storage system at one location, copying the data to a new location, updating an application-specific name service to point to the new copy, marking the new copy as writeable, and then deleting the old copy. Other datacenter applications maintain multiple replicas in different datacenters, and migration may simple require designating a different replica as the primary. Independent of the migration mechanism, datacenter applications might desire to employ application-specific throttling policies, such as only migrating user state when an application-specific predictive model suggests the user is unlikely to access their state in the next hour. Because Volley does not attempt to migrate the data itself, it does not interfere with these techniques or any other migration technique that an application may wish to employ.

4 Evaluation

In our evaluation, we compare Volley to three heuristics for where to place data and show that Volley substantially outperforms all of them on the metrics of datacenter capacity skew, inter-datacenter traffic, and user-perceived latency. We focus exclusively on the month-long Live Mesh trace for conciseness. For both the heuristics and Volley, we first compute a data placement using a week of data from the Live Mesh trace, and then evaluate the quality of the resulting placement on the following three weeks of data. For all four placement methodologies, any data that appears in the three-week evaluation window but not in the one-week placement computation window is placed in a single datacenter located in the United States (in production, this new data will be handled the next time the placement methodology is run). Placing all previously unseen data in one datacenter penalizes the different methodologies equally for such data.

The first heuristic we consider is *commonIP* – place data as close as possible to the IP address that most commonly accesses it. The second heuristic is *oneDC* – put all data in one datacenter, a strategy still taken by many companies due to its simplicity. The third heuristic is *hash* – hash data to datacenters so as to optimize for load-balancing. These three heuristics represent reasonable approaches to optimizing for the three different metrics we consider—oneDC and hash optimize for inter-datacenter traffic and datacenter capacity skew respectively, while commonIP is a reasonably sophisticated proposal for optimizing latency.

Throughout our evaluation, we use 12 commercial datacenters as potential locations. These datacenters are distributed across multiple continents, but their exact locations are confidential. Confidentiality concerns also prevent us from revealing the exact amount of bandwidth consumed by our services. Thus, we present the inter-datacenter traffic from different placements using the metric "fraction of messages that are inter-datacenter." This allows an apples-to-apples comparison between the different heuristics and Volley without revealing the underlying bandwidth consumption. The bandwidth consumption from centralizing Volley logs, needed for Volley and commonIP, is so small compared to this inter-datacenter traffic that it does not affect graphs comparing this metric among the heuristics. We configure Volley with a datacenter capacity model such that no one of the 12 datacenters can host more than 10% of all data, a reasonably balanced use of capacity.

All latencies that we compute analytically use the latency model described in Section 3. This requires using the client's IP address in the trace to place them at a geographic location. In this Live Mesh application, client requests require sending a message to a first data item, which then sends a second message to a second data item; the second data item sends a reply, and then the first data item sends the client its reply. If the data items are in the same datacenter, latency is simply the round trip time between the client and the datacenter. If the data items are in separate datacenters, latency is the sum of four one-way delays: client to datacenter 1, datacenter 1 to datacenter 2, datacenter 2 back to datacenter 1, and datacenter 1 back to the client. These latency calculations leave out other potential protocol overheads, such as the need to initially establish a TCP connection or to authenticate; any such protocol overheads encountered in practice would magnify the importance of latency improvements by incurring the latency multiple times. For clarity of presentation, we consistently group latencies into 10 millisecond bins in our graphs. The graphs only present latency up to 250 milliseconds because the better placement methodologies all achieve latency well under this for almost all requests.

Our evaluation begins by comparing Volley and the three heuristics on the metrics of datacenter capacity skew and inter-datacenter traffic (Section 4.1). Next, we evaluate the impact of these placements on the latency of client requests, including evaluating Volley in the con-

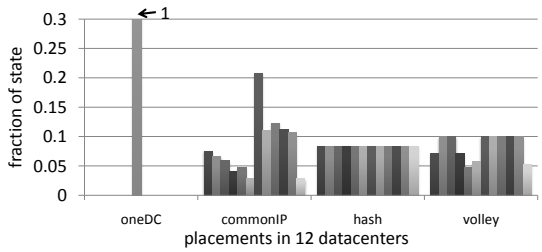

Figure 12. *Datacenter capacity required by three different placement heuristics and Volley.*

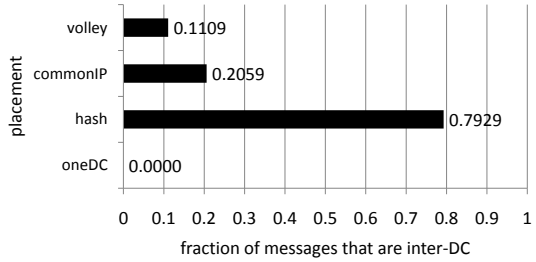

Figure 13. *Inter-datacenter traffic under three different placement heuristics and Volley.*

text of a simple, hypothetical example to understand this impact in detail (Section 4.2). We then evaluate the incremental benefit of Volley as a function of the number of Volley iterations (Section 4.3). Next, we evaluate the resource demands of running Volley on the SCOPE distributed execution infrastructure (Section 4.4). Finally, we evaluate the impact of running Volley more frequently or less frequently (Section 4.5).

4.1 Impact on Datacenter Capacity Skew and Inter-datacenter Traffic

We now compare Volley to the three heuristics for where to place data and show that Volley substantially outperforms all of them on the metrics of datacenter capacity skew and inter-datacenter traffic. Figures 12 and 13 show the results: hash has perfectly balanced use of capacity, but high inter-datacenter traffic; oneDC has zero inter-datacenter traffic (the ideal), but extremely unbalanced use of capacity; and commonIP has a modest amount of inter-datacenter traffic, and capacity skew where 1 datacenter has to support more than twice the load of the average datacenter. Volley is able to meet a reasonably balanced use of capacity while keeping inter-datacenter traffic at a very small fraction of the total number of messages. In particular, compared to commonIP, Volley reduces datacenter skew by over 2× and reduces inter-datacenter traffic by over 1.8×.

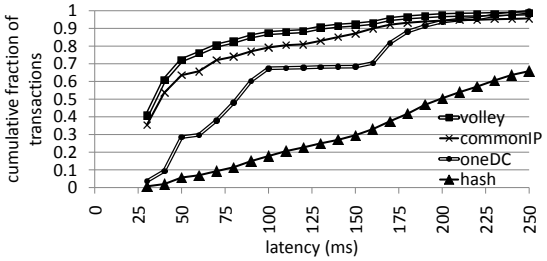

Figure 14. *Client request latency under three different placement heuristics and Volley.*

4.2 Impact on Latency of Client Requests

We now compare Volley to the three heuristics on the metric of user-perceived latency. Figure 14 shows the results: hash has high latency; oneDC has mediocre latency; and commonIP has the best latency among the three heuristics. Although commonIP performs better than oneDC and hash, Volley performs better still, particularly on the tail of users that experience high latency even under the commonIP placement strategy. Compared to commonIP, Volley reduces 75th percentile latency by over 30%.

4.2.1 Multiple Datacenter Testbed

Previously, we evaluated the impact of placement on user-perceived latency analytically use the latency model described in Section 3. In this section, we evaluate Volley's latency impact on a live system using a prototype cloud service. We use the prototype cloud service to emulate Live Mesh for the purpose of replaying a subset of the Live Mesh trace. We deployed the prototype cloud service across 20 virtual machines spread across the 12 geographically distributed datacenters, and we used one node at each of 109 Planetlab sites to act as clients of the system.

The prototype cloud service consists of four components: the frontend, the document service, the publish-subscribe service, and the message queue service. Each of these components run on every VM so as to have every service running in every datacenter. These components of our prototype map directly to the actual Live Mesh component services that run in production. The ways in which the production component services cooperate to provide features in the Live Mesh service is described in detail elsewhere [1], and we provide only a brief overview here.

The prototype cloud service exposes a simple frontend that accepts client requests and routes them to the appropriate component in either its own or another datacenter. In this way, each client can connect directly to any datacenter, and requests that require an additional step (e.g., updating an item, and then sending the update to others) will be forwarded appropriately. This design

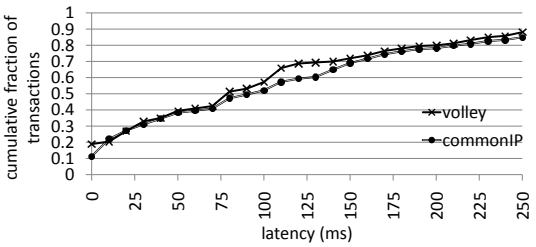

Figure 15. *Comparing Volley to the commonIP heuristic on a live system spanning 12 geographically distributed datacenters and accessed by Planetlab clients. In this Figure, we use a random sample of the Live Mesh trace. We see that Volley provides moderately better latency than the commonIP heuristic.*

allows clients to cache the location of the best datacenter to connect to for any given operation, but requests still succeed if a client request arrives at the wrong datacenter due to cache staleness.

We walk through an example of how two clients can rendezvous by using the document, publish-subscribe, and message queue services. The document service can store arbitrary data; in this case, the first client can store its current IP address, and a second client can then read that IP address from the document service and contact the first client directly. The publish-subscribe service is used to send out messages when data in the document service changes; for example, if the second client subscribes to updates for the first client's IP address, these updates will be pro-actively sent to the second client, instead of the second client having to poll the document service to see if there have been any changes. Finally, the message queue service buffers messages for clients from the publish-subscribe service. If the client goes offline and then reconnects, it can connect to the queue service and dequeue these messages.

To evaluate both Volley and the commonIP heuristic's latency on this live system, we used the same data placements computed on the first week of the Live Mesh trace. Because the actual Live Mesh service requires more than 20 VMs, we had to randomly sample requests from the trace before replaying it. We also mapped each client IP in the trace subset to the closest Planetlab node, and replayed the client requests from these nodes.

Figure 15 shows the measured latency on the sample of the Live Mesh trace; recall that we are grouping latencies into 10 millisecond bins for clarity of presentation. We see that Volley consistently provides better latency than the commonIP placement. These latency benefits are visible despite a relatively large number of external sources of noise, such as the difference between the actual client locations and the Planetlab locations, differences between typical client connectivity (that Volley's latency model relies on) and Planetlab connectivity, and occasional high load on the Planetlab nodes leading to high slice scheduling delays.

Timestamp	Source-Entity	Request-Size	Destination-Entity	Transaction-Id
T_0	PSS^a	100 B	Q^1	1
T_0	Q^1	100 B	IP^1	1
$T_0 + 1$	PSS^b	100 B	Q^1	2
$T_0 + 1$	Q^1	100 B	IP^2	2
$T_0 + 2$	PSS^b	100 B	Q^1	3
$T_0 + 2$	Q^1	100 B	IP^2	3
$T_0 + 5$	PSS^b	100 B	Q^2	4
$T_0 + 5$	Q^2	100 B	IP^1	4

Table 3. *Hypothetical application logs. In this example, IP^1 is located at geographic coordinates (10,110) and IP^2 at (10,10).*

Data	commonIP	Volley Phase 1	Volley Phase 2
PSS^a	(10,110)	(14.7,43.1)	(15.1,49.2)
PSS^b	(10,10)	(15.3,65.6)	(15.3,63.6)
Q^1	(10,10)	(14.7,43.1)	(15.1,50.5)
Q^2	(10,110)	(10,110)	(13.6,88.4)

Table 4. *CommonIP and Volley placements computed using Table 3, assuming a datacenter at every point on Earth and ignoring capacity constraints and inter-datacenter traffic.*

Transaction-Id	commonIP		Volley Phase 2	
	distance	latency	distance	latency
1	27,070 miles	396 ms	8,202 miles	142 ms
2	0 miles	31 ms	7,246 miles	129 ms
3	0 miles	31 ms	7,246 miles	129 ms
4	13,535 miles	182 ms	6,289 miles	116 ms

Table 5. *Distances traversed and latencies of user requests in Table 3 using commonIP and Volley Phase 2 placements in Table 4. Note that our latency model [36] includes an empirically-determined access penalty for all communication involving a client.*

Other than due to sampling of the request trace, the live experiment has no impact on the datacenter capacity skew and inter-datacenter traffic differences between the two placement methodologies. Thus, Volley offers an improvement over commonIP on every metric simultaneously, with the biggest benefits coming in reduced inter-datacenter traffic and reduced datacenter capacity skew.

4.2.2 Detailed Examination of Latency Impact

To examine in detail how placement decisions impact latencies experienced by user requests, we now consider a simple example. Table 3 lists four hypothetical Live Mesh transactions involving four data objects and clients behind two IP addresses. For the purposes of this simple example, we assume there is a datacenter at every point on Earth with infinite capacity and no inter-datacenter traffic costs. We pick the geographic coor-

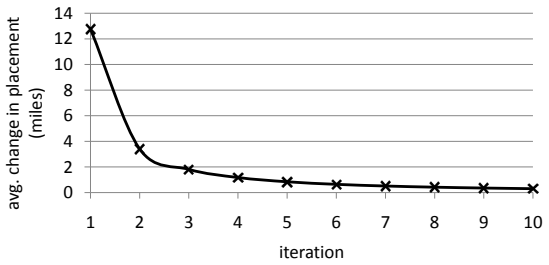

Figure 16. *Average distance traveled by each object during successive Volley Phase 2 iterations. The average incorporates some objects traveling quite far, while many travel very little.*

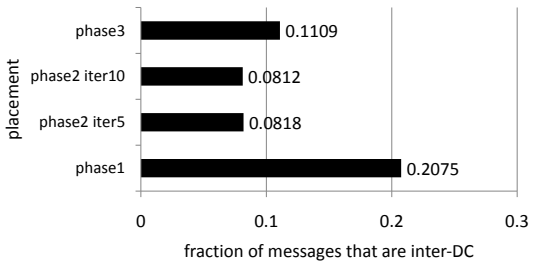

Figure 17. *Inter-datacenter traffic at each Volley iteration.*

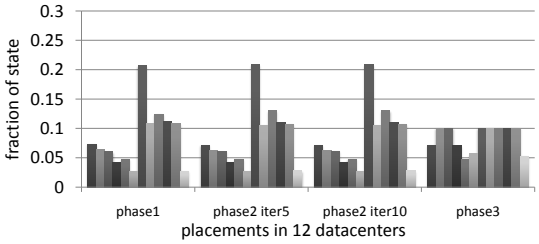

Figure 18. *Datacenter capacity at each Volley iteration.*

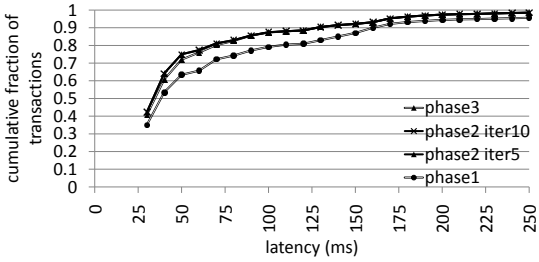

Figure 19. *Client request latency at each Volley iteration.*

dinates of (10,110) and (10,10) for ease of examining how far each object's placement is from the client IP addresses. Table 4 shows the placements calculated by commonIP and Volley in Phases 1 and 2. In Phase 1, Volley calculates the weighted spherical mean of the geographic coordinates for the client IPs that access each "Q" object. Hence Q^1 is placed roughly two-thirds along the great-circle segment from IP^1 to IP^2, while Q^2 is placed at IP^1. Phase 1 similarly calculates the placement of each "PSS" object using these coordinates for "Q" objects. Phase 2 then iteratively refines these coordinates.

We now consider the latency impact of these placements on the *same* set of user requests in Table 3. Table 5 shows for each user request, the physical distance traversed and the corresponding latency (round trip from PSS to Q and from Q to IP). CommonIP optimizes for client locations that are most frequently used, thereby driving down latency to the minimum for some user requests but at a significant expense to others. Volley considers all client locations when calculating placements, and in doing so drives down the worst cases by more than the amount it drives up the common case, leading to an overall better latency distribution. Note that in practice, user requests change over time after placement decisions have been made and our trace-based evaluation does use later sets of user requests to evaluate placements based on earlier requests.

4.3 Impact of Volley Iteration Count

We now show that Volley converges after a small number of iterations; this will allow us to establish in Section 4.5 that Volley runs quickly (i.e., less than a day), and thus can be re-run frequently. Figures 16, 17, 18 and 19 show the performance of Volley as the number of iterations varies. Figure 16 shows that the distance that Volley moves data significantly decreases with each Volley iteration, showing that Volley relatively quickly converges to its ideal placement of data items.

Figures 17, 18 and 19 further break down the changes in Volley's performance in each iteration. Figure 17 shows that inter-datacenter traffic is reasonably good after the initial placement of Phase 1, and is quite similar to the commonIP heuristic. In contrast, recall that the hash heuristic led to almost 80% of messages crossing datacenter boundaries. Inter-datacenter traffic then decreases by over a factor of 2 during the first 5 Phase 2 iterations, decreases by a small amount more during the next 5 Phase 2 iterations, and finally goes back up slightly when Volley's Phase 3 balances the items across datacenters. Of course, the point of re-balancing is to avoid the kind of capacity skew seen in the commonIP heuristic, and in this regard a small increase in inter-datacenter traffic is acceptable.

Turning now to datacenter capacity, we see that Volley's placement is quite skewed (and by an approximately constant amount) until Phase 3, where it smooths out datacenter load according to its configured capacity

Volley Phase	Elapsed Time in Hours	SCOPE Stages	SCOPE Vertices	CPU Hours
1	1:22	39	20,668	89
2	14:15	625	255,228	386
3	0:10	16	200	0:07

Table 6. *Volley's demands on the SCOPE infrastructure to analyze 1 week's worth of traces.*

model (i.e., such that no one of the 12 datacenters hosts more than 10% of the data). Turning finally to latency, Figure 19 shows that latency has reached its minimum after only five Phase 2 iterations. In contrast to the impact on inter-datacenter traffic, there is almost no latency penalty from Phase 3's data movement to satisfy datacenter capacity.

4.4 Volley Resource Demands

Having established that Volley converges after a small number of iterations, we now analyze the resource requirements for this many iterations; this will allow us to conclude that Volley completes quickly and can be re-run frequently. The SCOPE cluster we use consists of well over 1,000 servers. Table 6 shows Volley's demands on the SCOPE infrastructure broken down by Volley's different phases. The elapsed time, SCOPE stages, SCOPE vertices and CPU hours are cumulative over each phase – Phase 1 has only one iteration to compute the initial placement, while Phase 2 has ten iterations to improve the placement, and Phase 3 has 12 iterations to balance out usage over the 12 datacenters. Each SCOPE stage in Table 6 corresponds approximately to a single map or reduce step in MapReduce [11]. There are 680 such stages overall, leading to lots of data shuffling; this is one reason why the total elapsed time is not simply CPU hours divided by the degree of possible parallelism. Every SCOPE vertex in Table 6 corresponds to a node in the computation graph that can be run on a single machine, and thus dividing the total number of vertices by the total number of stages yields the average degree of parallelism within Volley: the average stage parallelizes out to just over 406 machines (some run on substantially more). The SCOPE cluster is not dedicated for Volley but rather is a multi-purpose cluster used for several tasks. The operational cost of using the cluster for 16 hours every week is small compared to the operational savings in bandwidth consumption due to improved data placement. The data analyzed by Volley is measured in the terabytes. We cannot reveal the exact amount because it could be used to infer confidential request volumes since every Volley log record is 100 bytes.

4.5 Impact of Rapid Volley Re-Computation

Having established that Volley can be re-run frequently, we now show that Volley provides substantially

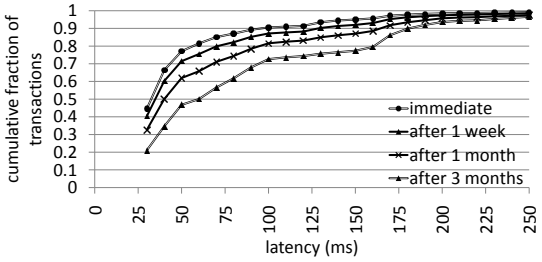

Figure 20. *Client request latency with stale Volley placements.*

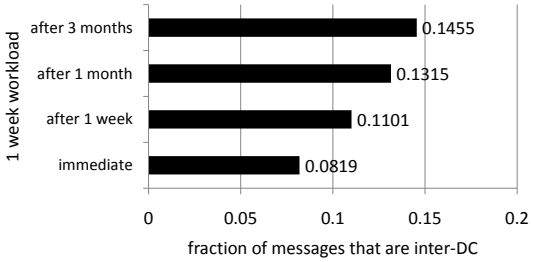

Figure 21. *Inter-datacenter traffic with stale Volley placements.*

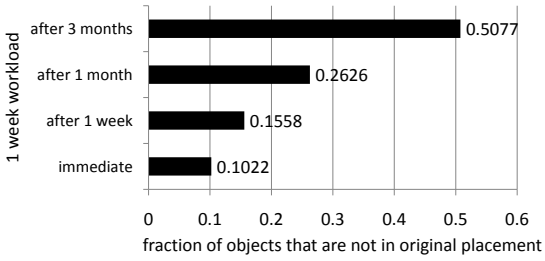

Figure 22. *Previously unseen objects over time.*

better performance by being re-run frequently. For these experiments, we use traces from the Live Mesh service extending from the beginning of June 2009 all the way to the beginning of September 2009. Figures 20, 21 and 22 show the impact of rapidly re-computing placements: Volley computes a data placement using the trace from the first week of June, and we evaluate the performance of this placement on a trace from the immediately following week, the week after the immediately following week, a week starting a month later, and a week starting three months after Volley computed the data placement. The better performance of the placement on the immediately following week demonstrates the significant benefits of running Volley frequently with respect to both latency and inter-datacenter traffic. Figure 20 shows that running Volley even every two weeks is noticeably worse than having just run Volley, and this latency penalty keeps increasing as the Volley placement

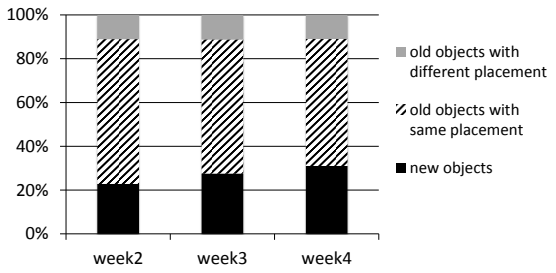

Figure 23. *Fraction of objects moved compared to first week.*

becomes increasingly stale. Figure 21 shows a similar progressively increasing penalty to inter-datacenter traffic; running Volley frequently results in significant inter-datacenter traffic savings.

Figure 22 provides some insight into why running Volley frequently is so helpful; the number of previously unseen objects increases rapidly with time. When run frequently, Volley detects accesses to an object sooner. Note that this inability to intelligently place previously unseen objects is shared by the commonIP heuristic, and so we do not separately evaluate the rate at which it degrades in performance.

In addition to new objects that are created and accessed, previously placed objects may experience significantly different access patterns over time. Running Volley periodically provides the added benefit of migrating these objects to locations that can better serve new access patterns. Figure 23 compares a Volley placement calculated from the first week of June to a placement calculated in the second week, then the first week to the third week, and finally the first week to the fourth week. About 10% of the objects in any week undergo migrations, either as a direct result of access pattern changes or due to more important objects displacing others in capacity-limited datacenters. The majority of objects retain their placement compared to the first week. Running Volley periodically has a third, but minor advantage. Some client requests come from IP addresses that are not present in geo-location databases. Objects that are accessed solely from such locations are not placed by Volley. If additional traces include accesses from other IP addresses that are present in geo-location databases, Volley can then place these objects based on these new accesses.

5 Related Work

The problem of automatic placement of application data re-surfaces with every new distributed computing environment, such as local area networks (LANs), mobile computing, sensor networks, and single cluster web sites. In characterizing related work, we first focus on the mechanisms and policies that were developed for these other distributed computing environments. We then describe prior work that focused on placing static content on CDNs; compared to this prior work, Volley is the first research system to address placement of dynamic application data across geographically distributed datacenters. We finally describe prior work on more theoretical approaches to determining an optimal data placement.

5.1 Placement Mechanisms

Systems such as Emerald [20], SOS [37], Globe [38], and Legion [25] focused on providing location-independent programming abstractions and migration mechanisms for moving data and computation between locations. Systems such as J-Orchestra [42] and Addistant [41] have examined distributed execution of Java applications through rewriting of byte code, but have left placement policy decisions to the user or developer. In contrast, Volley focuses on placement policy, not mechanism. Some prior work incorporated both placement mechanism and policy, e.g., Coign [18], and we characterize its differences with Volley's placement policy in the next subsection.

5.2 Placement Policies for Other Distributed Computing Environments

Prior work on automatic data placement can be broadly grouped by the distributed computing environment that it targeted. Placing data in a LAN was tackled by systems such as Coign [18], IDAP [22], ICOPS [31], CAGES [17], Abacus [3] and the system of Stewart et al [39]. Systems such as Spectra [13], Slingshot [40], MagnetOS [27], Pleaides [23] and Wishbone [33] explored data placement in a wireless context, either between mobile clients and more powerful servers, or in ad hoc and sensor networks. Hilda [44] and Doloto [30] explored splitting data between web clients and web servers, but neither assumed there were multiple geographic locations that could host the web server.

Volley differs from these prior systems in several ways. First, the scale of the data that Volley must process is significantly greater. This required designing the Volley algorithm to work in a scalable data analysis framework such as SCOPE [5] or MapReduce [11]. Second, Volley must place data across a large number of datacenters with widely varying latencies both between datacenters and clients, and between the datacenters themselves; this aspect of the problem is not addressed by the algorithms in prior work. Third, Volley must continuously update its measurements of the client workload, while some (though not all) of these prior approaches used an upfront profiling approach.

5.3 Placement Policies for Static Data

Data placement for Content Delivery Networks (CDNs) has been explored in many pieces of prior work [21, 19]. These systems have focused on static data – the HTTP caching header should be honored, but no other more elaborate synchronization between replicas is needed. Because of this, CDNs can easily employ decentralized algorithms e.g., each individual server or a small set of servers can independently make decisions about what data to cache. In contrast, Volley's need to deal with dynamic data would make a decentralized approach challenging; Volley instead opts to collect request data in a single datacenter and leverage the SCOPE distributed execution framework to analyze the request logs within this single datacenter.

5.4 Optimization Algorithms

Abstractly, Volley seeks to maps objects to locations so as to minimize a cost function. Although there are no known approximation algorithms for this general problem, the theory community has developed approximation algorithms for numerous more specialized settings, such as sparsest cut [4] and various flavors of facility location [6, 8]. To the best of our knowledge, the problem in Volley does not map to any of these previously studied specializations. For example, the problem in Volley differs from facility location in that there is a cost associated with placing two objects at different datacenters, not just costs between clients and objects. This motivates Volley's choice to use a heuristic approach and to experimentally validate the quality of the resulting data placement.

Although Volley offers a significant improvement over a state-of-the-art heuristic, we do not yet know how close it comes to an optimal placement; determining such an optimal placement is challenging because standard commercial optimization packages simply do not scale to the data sizes of large cloud services. This leaves open the tantalizing possibility that further improvements are possible beyond Volley.

6 Conclusion

Cloud services continue to grow to span large numbers of datacenters, making it increasingly urgent to develop automated techniques to place application data across these datacenters. Based on the analysis of month-long traces from two large-scale commercial cloud services, Microsoft's Live Messenger and Live Mesh, we built the Volley system to perform automatic data placement across geographically distributed datacenters. To scale to the large data volumes of cloud service logs, Volley is designed to work in the SCOPE [5] scalable data analysis framework.

We evaluate Volley analytically and on a live system consisting of a prototype cloud service running on a geographically distributed testbed of 12 datacenters. Our evaluation using one of the month-long traces shows that, compared to a state-of-the-art heuristic, Volley simultaneously reduces datacenter capacity skew by over $2\times$, reduces inter-datacenter traffic by over $1.8\times$, and reduces 75th percentile latency by over 30%. This shows the potential of Volley to simultaneously improve the user experience and significantly reduce datacenter costs.

While in this paper we have focused on using Volley to optimize data placement in existing datacenters, service operators could also use Volley to explore future sites for datacenters that would improve performance. By including candidate locations for datacenters in Volley's input, the operator can identify which combination of additional sites improve latency at modest costs in greater inter-datacenter traffic. We hope to explore this more in future work.

Acknowledgments

We greatly appreciate the support of Microsoft's ECN team for donating usage of their VMs, and the support of the Live Mesh and Live Messenger teams in sharing their data with us. We thank our shepherd, Dejan Kostic, and the anonymous reviewers for their detailed feedback and help in improving this paper.

References

[1] A. Adya, J. Dunagan, and A. Wolman. Centrifuge: Integrating Lease Management and Partitioning for Cloud Services. In *NSDI*, 2010.

[2] Amazon Web Services. http://aws.amazon.com.

[3] K. Amiri, D. Petrou, G. Ganger, and G. Gibson. Dynamic Function Placement for Data-intensive Cluster Computing. In *USENIX Annual Technical Conference*, 2000.

[4] S. Arora, S. Rao, and U. Vazirani. Expander Flows, Geometric Embeddings and Graph Partitioning. In *STOC*, 2004.

[5] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. In *VLDB*, 2008.

[6] M. Charikar and S. Guha. Improved Combinatorial Algorithms for the Facility Location and k-Median Problems. In *FOCS*, 1999.

[7] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Systems. In *DSN*, 2002.

[8] F. Chudak and D. Williamson. Improved approximation algorithms for capacitated facility location problems. *Mathematical Programming*, 102(2):207–222, 2005.

[9] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. In *SIGCOMM*, 2004.

[10] Data Center Global Expansion Trend.

http://www.datacenterknowledge.com/archives/2008/03/27/google-data-center-faq/.

[11] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *OSDI*, 2004.

[12] Facebook. http://www.facebook.com.

[13] J. Flinn, S. Park, and M. Satyanarayanan. Balancing Performance, Energy, and Quality in Pervasive Computing. In *ICDCS*, 2002.

[14] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. In *NSDI*, 2007.

[15] Global Air travel Trends. http://www.zinnov.com/presentation/Global_Aviation-Markets-An_Ananlysis.pdf.

[16] Google Apps. http://apps.google.com.

[17] G. Hamlin Jr and J. Foley. Configurable applications for graphics employing satellites (CAGES). *ACM SIGGRAPH Computer Graphics*, 9(1):9–19, 1975.

[18] G. Hunt and M. Scott. The Coign Automatic Distributed Partitioning System. In *OSDI*, 1999.

[19] S. Jamin, C. Jin, A. Kurc, D. Raz, and Y. Shavitt. Constrained Mirror Placement on the Internet. In *INFOCOM*, 2001.

[20] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, 1966.

[21] M. Karlsson and M. Mahalingam. Do We Need Replica Placement Algorithms in Content Delivery Networks? In *International Workshop on Web Content Caching and Distribution (WCW)*, 2002.

[22] D. Kimelman, V. Rajan, T. Roth, M. Wegman, B. Lindsey, H. Lindsey, and S. Thomas. Dynamic Application Partitioning in VisualAge Generator Version 3.0. *Lecture Notes In Computer Science; Vol. 1543*, pages 547–548, 1998.

[23] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan. Reliable and Efficient Programming Abstractions for Wireless Sensor Networks. In *PLDI*, 2007.

[24] J. Leskovec and E. Horvitz. Planetary-Scale Views on an Instant-Messaging Network. In *WWW*, 2008.

[25] M. Lewis and A. Grimshaw. The core Legion object model. In *HPDC*, 1996.

[26] LinkedIn. http://linkedin.com.

[27] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. Sirer. Design and Implementation of a Single System Image Operating System for Ad Hoc Networks. In *MobiSys*, 2005.

[28] Live Mesh. http://www.mesh.com.

[29] Live Messenger. http://messenger.live.com.

[30] B. Livshits and E. Kiciman. Doloto: Code Splitting for Network-bound Web 2.0 Applications. In *SIGSOFT FSE*, 2008.

[31] J. Michel and A. van Dam. Experience with distributed processing on a host/satellite graphics system. *ACM SIGGRAPH Computer Graphics*, 10(2):190–195, 1976.

[32] Microsoft Office Online. http://office.microsoft.com/en-us/office_live/default.aspx.

[33] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: Profile-based Partitioning for Sensornet Applications. In *NSDI*, 2009.

[34] Quova IP Geo-Location Database. http://www.quova.com.

[35] Rapid Release Cycles. http://www.ereleases.com/pr/20070716009.html.

[36] S. Agarwal and J. Lorch. Matchmaking for Online Games and Other Latency-Sensitive P2P Systems. In *SIGCOMM*, 2009.

[37] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, and M. Ruffin. SOS: An object-oriented operating system—assessment and perspectives. *Computing Systems*, 1989.

[38] M. Steen, P. Homburg, and A. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, pages 70–78, 1999.

[39] C. Stewart, K. Shen, S. Dwarkadas, M. Scott, and J. Yin. Profile-driven component placement for cluster-based online services. *IEEE Distributed Systems Online*, 5(10):1–1, 2004.

[40] Y. Su and J. Flinn. Slingshot: Deploying Stateful Services in Wireless Hotspots. In *MobiSys*, 2005.

[41] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of ”legacy” java software. In *ECOOP*, pages 236–255. Springer-Verlag, 2001.

[42] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning. In *ECOOP*, pages 178–204. Springer-Verlag, 2002.

[43] Twitter. http://www.twitter.com.

[44] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A High-Level Language for Data-DrivenWeb Applications. *ICDE*, 2006.

[45] Y. Zhao, Y. Xie, F. Yu, Q. Ke, Y. Yu, Y. Chen, and E. Gillum. Botgraph: Large Scale Spamming Botnet Detection. In *NSDI*, 2009.