Location, Location, Location! Modeling Data Proximity in the Cloud

Birjodh Tiwana
tiwana@eecs.umich.edu
University of Michigan
Ann Arbor, MI

Mahesh Balakrishnan
maheshba@microsoft.com
Microsoft Research
Mountain View, CA

Marcos K. Aguilera
aguilera@microsoft.com
Microsoft Research
Mountain View, CA

Hitesh Ballani
hiballan@microsoft.com
Microsoft Research
Cambridge, UK

Z. Morley Mao
zmao@eecs.umich.edu
University of Michigan
Ann Arbor, MI

ABSTRACT

Cloud applications have increasingly come to rely on distributed storage systems that hide the complexity of handling network and node failures behind simple, data-centric interfaces (such as PUTs and GETs on key-value pairs). While these interfaces are very easy to use, the application is completely oblivious to the location of its data in the network; as a result, it has no way to optimize the placement of data or computation. In this paper, we propose exposing the network location of data to applications. The primary challenge is that data does not usually exist at a single point in the network; it can be striped, replicated, cached and coded across different locations, in arbitrary ways that vary across storage systems. For example, an item that is synchronously mirrored in both Seattle and London will appear equally far from both locations for writes, but equally close to both locations for reads. Accordingly, we describe Contour, a system that allows applications to query and manipulate the location of data without requiring them to be aware of the physical machines storing the data, the replication protocols used or the underlying network topology.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network Topology; C.4 [**Performance of Systems**]: Modeling Techniques

General Terms

Design, Measurement, Performance

Hotnets '10, October 20–21, 2010, Monterey, CA, USA.

Keywords

Cloud Computing, Location, Network Topology, Key-Value Stores

1. INTRODUCTION

Existing cloud platforms offer developers storage services with simple, data-centric interfaces to store and retrieve application data (for example, Microsoft Azure's Blob Store and Amazon's S3 allow PUTs and GETs on key-value pairs). Behind such simple interfaces, these services use complex machinery to ensure that data is available and persistent in the face of network and node failures. As a result, developers can focus on application functionality without having to reason about complex failure scenarios.

Unfortunately, this simplicity comes at a cost; applications have little or no information regarding the location of their data in the network. Without this information, applications cannot optimize their execution by moving computation closer to data, data closer to users, or related data closer to each other. These kinds of optimizations can be crucial for applications executing across different data centers (where network latencies can be very high), as well as within hierarchical data center networks (where bandwidth can be limited).

The current state-of-the-art solution for this problem involves guesswork: the cloud determines data placement by predicting the future access patterns of the application based on past history, while treating the application as a black box. This approach can be expensive and counter-productive, since the application typically has more accurate information than the cloud about its own future behavior. In addition, without input from the application, the cloud can optimize only simple aggregates of low-level metrics such as bandwidth usage or access latency. Requesting information from the appli-

cation – i.e., its future access patterns or the high-level metrics of interest to it – is a possible solution; however applications can have arbitrary optimization criteria that are difficult to express to the cloud without complicating the storage interface.

In this paper, we examine a different approach: exposing the location of data to applications and allowing them to optimize their own execution. We want applications to be able to estimate the time taken to update or retrieve data from different network locations. We also want to enable cloud interfaces that allow applications to move computation closer to data (and vice versa), as well as request new computational resources near existing data. Importantly, we want to do so without breaking the abstraction of data-centric storage; applications must not be aware of physical storage servers or the underlying network topology.

The primary challenge is that *data does not exist at a single location in the network.* Data can be striped, mirrored, cached and coded across different points in the network, using protocols with widely varying semantics. Consider an example in which a machine in Los Angeles accesses data synchronously mirrored at data centers in Seattle and London. For reads, the data will appear to reside in Seattle, since only the local data center needs to be contacted. For writes, the data will appear to reside in London, since both data centers are contacted in parallel for a successful write operation. Note that different protocols – where the client machine waits for a response from the first mirror before updating the second, or lets one mirror directly update the other – lead to different access latencies to the data.

To capture such protocol-specific behavior, we propose the idea of *replication topologies*. These are simple representations of the interactions between different servers triggered by reads or writes to a storage service. Importantly, replication topologies are not meant to be complete descriptions of protocols; instead, they capture only the network pathways taken by the protocol in failure-free operation. We find that most protocols used in practice – such as synchronous and asynchronous mirroring, erasure codes, chain replication and different types of quorums – can be modeled with very simple replication topologies.

We describe the design of Contour, a system that uses replication topologies to provide data-centric location functionality. Contour provides estimates of the latency to retrieve or update data in a storage service from any node in the network. It also supports higher-level functionality such as closest-node discovery (e.g., finding a node that is closest to a key/value pair for reads) and constraint satisfaction (e.g., finding a node that can update a particular key/value pair within 100 ms and read another within 10 ms). These interfaces can be used directly by applications to optimize performance. They can also be used by cloud subsystems to support new application-facing interfaces for moving data closer to given network locations, or for requesting new resources near existing data.

To work with Contour, a storage service has to provide it with the replication topology used to access data. Contour combines this information with link-level network topology information – such as RTTs, bandwidth and loss rates – to estimate data access latencies from any other location in the cloud. This link-level information is collected continuously in the background, amortizing the cost of measurement across multiple applications. Synthesizing accurate estimates in this manner is a significant challenge; however, we believe that cloud platforms have enough monitoring infrastructure in place on their internal networks to make this approach viable. Additionally, high-level functions such as closest-node discovery do not require very accurate latency estimates; it is sufficient if the estimate for the closest node is lower than for any other node.

In this paper, our target applications are Internet services catering to a geo-distributed user base. Contour is equally relevant for different applications, such as MapReduce jobs running within bandwidth-constrained data centers, or enterprise applications split across private and public clouds. We omit discussing these applications in detail for lack of space.

2. LOCATION IN THE CLOUD

To understand why location is important in the cloud, we examine the anatomy of a typical application. Cloud applications are composed of three distinct types of entities:

- **Clients** are machines or devices accessing the service from the public Internet. When a user types in the URL of a service into her browser, the request is redirected (typically through DNS-based load-balancing) to a data center hosting the service.
- **Compute Nodes** are the work-horses of the cloud, executing application logic for services (known as 'worker roles' in Azure and 'instances' on EC2). Compute nodes usually do not store persistent state, though they can store soft session state and be 'sticky' with respect to individual user sessions. Within each data center, incoming requests are directed to compute nodes by a layer of web-servers that accept and manage HTTP connections from users.
- **Storage Services** store all application data and are accessed by compute nodes via simple, data-centric interfaces. Under the hood, these are distributed storage systems running complex protocols to ensure that data is always available and

durable, even when machines, disks and networks fail. Each cloud provider offers a range of such services, with different interfaces (such as key-value stores, queues, or even linear address spaces) and persistence guarantees.

Consider the example of *FaceTracker*, a hypothetical face recognition application for mobile phones: when the user of the phone points its camera towards some person, the image is matched against a library of her friends' photos. In a cloud-based version of this application, the client would be the phone itself, and the library of photos would be stored in a cloud-based storage service (say a key-value store). Images are uploaded from the phone to a compute node, which runs face recognition algorithms on them, fetching photos of the user's friends from the key-value store with GET calls. Successfully matched images are occasionally added to the library with PUT calls to improve accuracy.

Of the three types of entities described, the application obviously has no control over the location of clients, while it can move data around with some cost in terms of bandwidth and time. Computation is the easiest to move, since compute nodes have no persistent state. Compute nodes typically need to be close to clients to minimize interaction latencies; for *FaceTracker*, it is reasonable to assume that a user's compute node is always in the data center closest to her location. In practice, deployed systems do a good job of redirecting clients to compute nodes in close-by data centers; consequently, we do not focus on the location of compute nodes relative to clients.

Against this backdrop, we examine application scenarios where the location of data matters with respect to compute nodes, using *FaceTracker* as a running example.

Obtaining new resources: An application may need new compute nodes near existing data items. Alternatively, it may want to place a newly created data item near an existing compute node. *When Alice uses FaceTracker for the first time, her request is directed to a existing compute node, which creates her photo library close to itself.*

Load balancing and failure recovery: When a compute node processing some data fails or gets overloaded, the application may need to shift computation to a different compute node near the same data. *When the compute node matching Alice's incoming images fails, FaceTracker restarts the task on a different compute node close to her photo library.*

Dispersed data: When a task accesses a set of different data items, the application may want to locate it on a compute node equally close to all these data items, or closer to some of them than the rest. *Alice wants to match her camera's feed against all her friends' libraries; accordingly, FaceTracker locates her matching task on a compute node optimally placed with respect to all their libraries.*

Moving data closer to computation: When users move or change their access patterns, the application may want to relocate data to be closer to the new compute nodes handling those users. *When Alice moves from New York City to San Francisco, FaceTracker moves her photo library closer to the new compute node in California now matching her images.*

Shared data: When a data item is being concurrently accessed by multiple compute nodes, the application may want to place the data item equally close to all the compute nodes, or prioritize some over others. *Alice is in New York and wants to match her camera's feed against Bob's photo library, who lives in Seattle. Since moving her matching task to Seattle will increase her latency and cost to upload images, FaceTracker moves Bob's library closer to her compute node.*

3. REPLICATION TOPOLOGIES

Thus far, we have established the need for applications to know – and change – the location of data. This is relatively easy to achieve if the cloud stores each data item on a single machine in the network; the location of data is simply the location of the machine storing it. In this scenario, existing node-centric models for network location (such as network coordinates or tree-based models) can be used to estimate access latencies for the data. While such models usually provide low-level path properties between nodes (such as RTT and bandwidth), it is possible to convert these metrics into estimates of data transfer times; for example, the time taken to retrieve 1 MB of data via TCP/IP.

Unfortunately, data in the cloud does not usually reside on a single machine, making node-centric network models ineffective in this context. Storage services typically replicate data over multiple nodes, using a wide range of different protocols. We use 'replication' as a catch-all term, spanning techniques such as caching, mirroring, erasure coding and striping. When a read or write operation is issued on an item in a storage service, multiple storage servers communicate with each other to ensure the right semantics for the operation. The exact patterns of communication – which servers talk to each other, and whether they wait for responses from each other before replying back to the node issuing the request – depend on the replication protocol used.

The key insight in this paper is that the critical path of interactions between storage servers for any replication protocol can be captured using simple representations called *replication topologies*. Replication topologies can be drawn as DAGs, where a directed edge between two nodes represents a message going from one

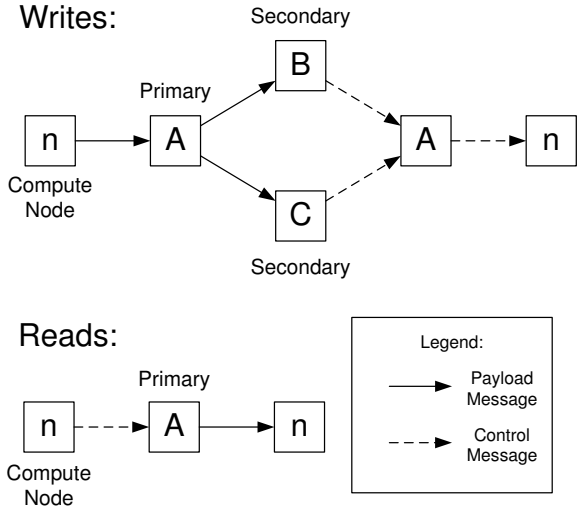

Figure 1: Read/Write replication topologies for a synchronous mirroring protocol with one primary (A) and two secondaries (B, C).

node to the other. Edges with solid lines correspond to messages that have the actual payload being read or written, whereas edges with dashed lines correspond to small control messages. To express message dependencies precisely, we allow a single machine to be represented by multiple nodes in the graph.

To understand replication topologies better, consider a simple synchronous mirroring protocol consisting of a primary replica and two secondary replicas. Write operations are first sent to the primary, which synchronously stores them on the secondary replicas before responding back to the compute node. All read operations are satisfied directly by the primary; the secondaries are read from only in the case of primary failure.

We show the corresponding replication topology for this protocol in Figure 1, where a compute node n is shown accessing an item synchronously mirrored on a primary A and two secondaries B and C. For reads, the replication topology simply consists of a single outgoing edge from an n node to a A node, and then back to an n node. For writes, we have an outgoing edge from n to A, which then has two edges outgoing to B and C, representing the mirroring messages. B and C then respond back to A, which in turn responds back to n.

Given this replication topology and the locations of n, A, B and C in the network, it is possible to estimate the time taken to write or retrieve a value. For example, if the value is of size 5 MB, the total time taken by n to write it can be computed as follows. First, we compute the time taken to transfer 5 MB from n to A. Since A then contacts B and C in parallel, we then take the max of the time taken along those two paths. Each path involves sending 5 MB from A to B or C, respectively, and then receiving a short acknowledgment message back. Lastly, we add the time taken to send an acknowledgment from A to n. In other words, the latency to write a value is a simple function over inter-node data transfer latencies, using sum and max operators.

In addition to providing estimates of data access latencies, replication topologies also allow us to understand how these latencies are impacted by the location of each replica in the network. In the example above, moving n closer to B or C does not necessarily improve performance for writes; what matters is the proximity of n to A, and of A to B and C. Similarly, the location of B or C has no impact on read performance from n. Such knowledge of the replication topology can be used to implement functionality such as closest-node discovery more efficiently.

Our representation includes two more operators to indicate that a node should contact or wait for the closest subset of its neighbors. Figure 2 shows the replication topology for an erasure coding protocol; an item is stored as six coded pieces on six different machines (A to F), of which any four pieces are sufficient to reconstruct the original item. We show a variant of this protocol where n contacts all six machines in parallel, but waits for only the first four machines that respond. To implement this, we introduce the *first-k* operator on the DAG, indicating that the node waits for only the first k incoming messages. A different variant of this protocol might have n contact only the four closest machines, than all six; to model this, we use a *closest-t* operator on the outgoing edges of a node to indicate that it contacts only its t closest neighbors.

We believe that replication topologies are general enough to model a wide range of replication protocols. For instance, quorum-based protocols are similar to erasure coding in behavior, in that only a subset of nodes needs to be contacted (or waited for). One challenging aspect of some protocols is that their behavior can change over time; for example, an asynchronous primary-backup protocol may allow reads from the secondary except when the primary has just been updated. We can model these protocols as exhibiting different replication topologies at different points in time.

4. THE CONTOUR SYSTEM

In this section, we describe the design of the Contour system, and how it interacts with applications and storage services.

4.1 Contour and the Storage Service

Contour expects the storage service to implement a simple interface that returns the read or write replication topology for a passed-in key. For example, if the storage service implements synchronous mirroring, the

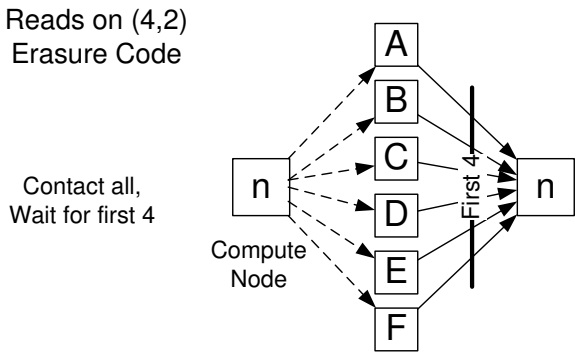

Figure 2: Read replication topology for an erasure coding protocol where any four coded blocks are sufficient to read the whole item.

returned replication topology will be identical to the diagram in Figure 1, with actual IP addresses substituted for abstract node identifiers (e.g., 192.168.0.1 instead of A).

As described in the previous section, computing the access latency from this replication topology involves a simple function over inter-node data transfer latencies (for example, the time taken to transfer 5 MB from 192.168.0.1 to 192.168.0.2), involving sum, min and max operators. Contour determines this function from the replication topology and evaluates it using estimates for inter-node data transfer latencies obtained from its own link-level measurements.

A secondary interface that the storage service needs to implement is a call that returns the size of the value corresponding to a passed-in key. This is used by Contour to compute access latencies for reads given the replication topology.

4.2 Contour and the Application

The basic functionality provided by Contour is data access latency estimation: applications (or cloud subsystems) can estimate the time taken to read or write data from any compute node in the network. Over this basic primitive, Contour builds more specific functionality; for example, it allows applications to find the node from a set closest to a particular unit of data. It also supports finding nodes that satisfy access latency constraints with respect to multiple data items. This is useful if the application wants to choose an existing compute node to run a particular task based on the data it accesses. It is also useful for cloud allocation or scheduling components that want to satisfy application-specified requirements for new compute nodes.

All these calls identify individual data units with an opaque *key* parameter, which depends on the underlying storage service involved; it can be a simple key, a block number in a linear address space, or a (row, column) pair. Contour does not actively operate on the key in any way; it merely uses it as a parameter to retrieve the replication topology from the storage service.

On their own, these interfaces naturally support any application scenario that involves moving computation closer to data. With the involvement of the storage service, they can also support scenarios that involve moving data closer to specific network locations. In these cases, we expect the storage service to support an application-facing interface that allows data to be moved closer to some node. The storage service can then use Contour's interfaces to estimate the distance of different replication topologies from the target node to find one that fits the access latency constraint.

4.3 Design Considerations

The simplest way to implement Contour is as a centralized service, accessed by applications via local libraries. Every time an application wishes to estimate access latencies to a key, it can first issue a query to a close-by Contour server, which correspondingly retrieves the replication topology from the storage service. While simple to implement, this purely pull-based approach can result in high query latencies. Caching responses – both at the Contour server and the application machine – can reduce latencies, but introduces the possibility of staleness.

An alternative involves introducing push-based mechanisms. Applications could register their interest in specific keys to their local Contour server, which in turn registers its interest in those keys to the storage service. When the Contour server is notified by the storage service of a change in the replication topology for a key – or the underlying network topology changes – it can notify the compute node interested in that key.

5. DISCUSSION

Contour in different settings: We focused this paper on Internet services running across geographically distant data centers. Contour generalizes easily to other types of cloud applications with one alteration: the way that Contour computes data transfer latencies from low-level link metrics such as RTT, bandwidth and loss rates can change depending on the setting. For example, a different methodology may be required to compute data transfer latencies within a data center, as opposed to wide area links. This functionality can be abstracted away into a module responsible for generating estimates of data transfer latencies between two physical nodes.

Modeling accesses to storage media: One aspect of using Contour within a single data center is that the latency of accessing storage media (such as disk or flash) needs to be modeled as well, since it constitutes a much larger fraction of end-to-end latency in such settings. We can modify replication topologies to model

media access latencies by introducing new nodes and edges into the graph as appropriate. For example, if A is a storage server in a replication topology, it could be rewritten as two nodes instead, A_1 and A_2, with a directed edge going from one to the other. All incoming edges into A would now go into A_1, and all outgoing edges would now leave from A_2.

Does the cloud care about its own privacy?: An intriguing aspect of Contour's approach is that the cloud could end up revealing information about its composition. In a geo-distributed setting, this could result in applications learning the number of data centers or their location; within a data center, they could possibly learn the type of topology used or the number of machines. While this possibility exists even in the absence of Contour, the ability to collect large numbers of latency estimates without actively transferring data provides applications an inexpensive way to infer the cloud's internal details. This is an open problem for Contour; one possibility involves adding jitter to estimates to offer the cloud some measure of privacy.

Other data-centric metrics: Contour provides applications with the estimated latency to retrieve or update data from a storage service. A different metric of interest could be cost in terms of dollars; for example, if an access results in traffic between data centers, the application may be charged for it by the cloud. Accordingly, Contour could report to the application the cost of a data access. Another possible data-centric metric is availability; Contour could provide the probability that an access succeeds, based on the failure rates of the network paths involved in the replication topology.

6. RELATED WORK

Contour is inspired by existing work on network models for predicting path properties such as latency between Internet end-hosts; for example, network coordinate systems such as Vivaldi [3] attempt to embed inter-node latencies in a coordinate space. Other work in this space includes systems that offer different models [5], narrow location-centric functionality [6] or general query interfaces [4]. Contour can be viewed as an attempt to extend this class of work to provide a data-centric notion of network location.

Volley [1] does automated data placement for geo-distributed applications based on user request patterns. A key difference from our work is that Volley ignores write performance and uses a simple, fixed replication strategy; accordingly, it equates the location of data to the location of its closest replica. In addition, Volley does not seek to retain a data-centric application interface. Lastly, Volley represents a design where optimization is handled by infrastructure instead of the applications themselves; as mentioned earlier, it is difficult for such designs to handle arbitrary application priorities.

Another related system is PADS [2], which provides an architecture for building distributed storage systems by defining policies for locating and updating replicas. Like Contour, PADS comes up with a general representation for different replication policies; however, its goal is more ambitious since it tries to completely define the protocol. In contrast, Contour's replication topologies are meant to only model the end-to-end latencies exhibited by the protocol.

7. CONCLUSION

Modern cloud platforms make it easy for developers to write applications by abstracting away node-level details under data-centric interfaces. However, doing so robs developers of the ability to understand and optimize application performance. We present a system called Contour that allows nodes to reason about the location of data in the network without breaking the abstraction of data-centric storage. At the core of Contour are replication topologies, abstractions that express the critical server interactions that occur on data accesses.

Acknowledgments

We'd like to thank Vijayan Prabhakaran, Venugopalan Ramasubramanian and Patrick Stuedi for their feedback during the project. We would also like to thank the HotNets reviewers for their detailed comments on the paper.

8. REFERENCES

[1] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan. Volley: Automated Data Placement for Geo-Distributed Cloud Services. In *NSDI 2010*.

[2] N. Belaramani, J. Zheng, A. Nayate, R. Soulé, M. Dahlin, and R. Grimm. PADS: a policy architecture for distributed storage systems. In *NSDI 2009*.

[3] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *SIGCOMM 2004*.

[4] H. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *OSDI 2006*.

[5] V. Ramasubramanian, D. Malkhi, F. Kuhn, M. Balakrishnan, A. Gupta, and A. Akella. On the treeness of internet latency and bandwidth. In *SIGMETRICS 2009*.

[6] B. Wong, A. Slivkins, and E. Sirer. Meridian: A lightweight network location service without virtual coordinates. In *SIGCOMM 2005*.