Ambry: LinkedIn's Scalable Geo-Distributed Object Store

Shadi A. Noghabi[1], Sriram Subramanian[2], Priyesh Narayanan[2]
Sivabalan Narayanan[2], Gopalakrishna Holla[2], Mammad Zadeh[2], Tianwei Li[2]
Indranil Gupta[1], Roy H. Campbell[1]
[1]University of Illinois at Urbana-Champaign, [2]LinkedIn Corp.
{abdolla2, indy, rhc}@illinois.edu
{srsubramanian, pnarayanan, snarayanan, gholla, mzadeh, tili}@linkedin.com

ABSTRACT

The infrastructure beneath a worldwide social network has to continually serve billions of variable-sized media objects such as photos, videos, and audio clips. These objects must be stored and served with low latency and high throughput by a system that is geo-distributed, highly scalable, and load-balanced. Existing file systems and object stores face several challenges when serving such large objects. We present Ambry, a production-quality system for storing large immutable data (called blobs). Ambry is designed in a decentralized way and leverages techniques such as logical blob grouping, asynchronous replication, rebalancing mechanisms, zero-cost failure detection, and OS caching. Ambry has been running in LinkedIn's production environment for the past 2 years, serving up to 10K requests per second across more than 400 million users. Our experimental evaluation reveals that Ambry offers high efficiency (utilizing up to 88% of the network bandwidth), low latency (less than 50 ms latency for a 1 MB object), and load balancing (improving imbalance of request rate among disks by 8x-10x).

Keywords

Object Store, Geographically Distributed, Scalable, Load Balancing

1. INTRODUCTION

During the past decade, social networks have become popular communication channels worldwide. Hundreds of millions of users continually upload and view billions of diverse massive media objects, from photos and videos to documents. These large media objects, called *blobs*, are uploaded once, frequently accessed from all around the world, never modified, and rarely deleted. LinkedIn, as a global large-scale social network company, has faced the need for a geographically distributed system that stores and retrieves these read-heavy blobs in an efficient and scalable manner.

Handling blobs poses a number of unique challenges. First, due to diversity in media types, blob sizes vary significantly from tens of KBs (e.g., profile pictures) to a few GBs (e.g., videos). The system needs to store both massive blobs and a large number of small blobs efficiently. Second, there is an ever-growing number of blobs that need to be stored and served. Currently, LinkedIn serves more than 800 million put and get operations per day (over 120 TB in size). In the past 12 months, the request rate has almost doubled, from 5k requests/s to 9.5k requests/s. This rapid growth in requests magnifies the necessity for a linearly scalable system (with low overhead). Third, the variability in workload and cluster expansions can create unbalanced load, degrading the latency and throughput of the system. This creates a need for load-balancing. Finally, users expect the uploading process to be fast, durable, and highly available. When a user uploads a blob, all his/her friends from all around the globe should be able to see the blob with very low latency, even if parts of the internal infrastructure fail. To provide these properties, data has to be reliably replicated across the globe in multiple datacenters, while maintaining low latency for each request.

LinkedIn had its own home-grown solution called Media Server, built using network attached storage filers (for file storage), Oracle database (for metadata), and Solaris boxes. Media Server had multiple drawbacks. It faced CPU and IO spikes caused by numerous metadata operations for small objects, was not horizontally scalable, and was very expensive. Given that LinkedIn was scaling rapidly and the future web content will be largely dominated by media, we needed to find a replacement.

Several systems have been designed for handling a large amount of data, but none of them satisfactorily meet the requirements and scale LinkedIn needs. There has been extensive research into distributed file systems [10, 16, 22, 24, 28]. These systems have a number of limitations when used for storing blobs, as pointed out by [3, 11]. For instance, the hierarchical directory structure and rich metadata are an overkill for a blob store and impose unnecessary additional overhead.

Many key value stores [2, 5, 8, 14] have also been designed for storing a large number of objects. Although these systems can handle many small objects, they are not optimized for storing large objects (tens of MBs to GBs). Further, they impose extra overhead for providing consistency guarantees while these are typically not needed for immutable data. Some examples of these overheads include using vector clocks, conflict resolution mechanism, logging, and central coordinators.

A few systems have been designed specifically for large immutable objects including Facebook's Haystack [3] along with f4 [18] and Twitter's Blob Store [27]. However, these systems do not resolve load imbalance, especially when cluster expansions occur.

In this paper we present Ambry, a production-quality system designed specifically for diverse large and small im-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-3531-7/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2882903.2903738

mutable data with read-heavy traffic, where data is written once, and read many times (>95% read traffic). Ambry is designed with four main goals in mind:

1) Low Latency and High Throughput: The system needs to serve a large number of requests per second in a timely fashion, while working on cheap commodity hardware (e.g., HDDs). In order to reach this goal, Ambry utilizes a number of techniques including exploiting the OS cache, using zero copy when reading data from disk to network, chunking data along with retrieving/storing chunks in parallel from multiple nodes, providing configurable polices for the number of replicas to write and read, and zero-cost failure detection mechanisms (Sections 2.3, 4.2, and 4.3).

2) Geo-Distributed Operation: Blobs have to be replicated in other geographically distributed datacenters for high durability and availability, even in the presence of failures. To achieve low latency and high throughput in this geo-distributed setting, Ambry is designed as a decentralized multi-master system where data can be written to or read from any of the replicas. Additionally, it uses asynchronous writes that write data to the closest datacenter and asynchronously replicate to other datacenter(s). Also, for higher availability, it uses proxy requests that forward requests to other datacenters when the data is not replicated in the current datacenter yet (Sections 2.3 and 4.2).

3) Scalability: With the ever-growing amount of data, the system has to scale out efficiently with low overhead. To achieve this goal, Ambry makes three main design choices. First, Ambry separates the logical placement of blobs from their physical placement, allowing it to change the physical placement transparently from the logical placement. Second, Ambry is designed as a completely decentralized system, with no manager/master. Third, Ambry uses on-disk segmented indexing along with Bloom filters and an in-memory cache of the latest segment, allowing for scalable and efficient indexing of blobs. (Section 4.3).

4) Load Balancing: The system has to stay balanced in spite of growth. Ambry uses chunking of large blobs along with a random selection approach to remain balanced in a static cluster, and a re-balancing mechanism to return to a balanced state whenever cluster expansion occurs (Section 3).

Ambry has successfully been in production for the last 24 months, across four datacenters, serving more than 400 million users. Our experimental results show that Ambry reaches high throughput (reaching up to 88% of the network bandwidth) and low latency (serving 1 MB blobs in less than 50 ms), works efficiently across multiple geo-distributed datacenters, and improves the imbalance among disks by a factor of 8x-10x while moving minimal data.

2. SYSTEM OVERVIEW

In this section we discuss the overall design of Ambry including the high-level architecture of the system (Section 2.1), the notion of partition (Section 2.2), and supported operations (Section 2.3).

2.1 Architecture

Ambry is designed as a completely decentralized multi-tenant system across geographically distributed data centers. The overall architecture of Ambry is shown in Figure 1. The system is composed of three main components: *Frontends* that receive and route requests, *Datanodes* that store the actual data, and *Cluster Managers* that maintain the state of the cluster. Each datacenter owns and runs its own set of these components in a decentralized fashion. The Frontends and Datanodes are completely independent of one another, and the Cluster Managers are synchronized using

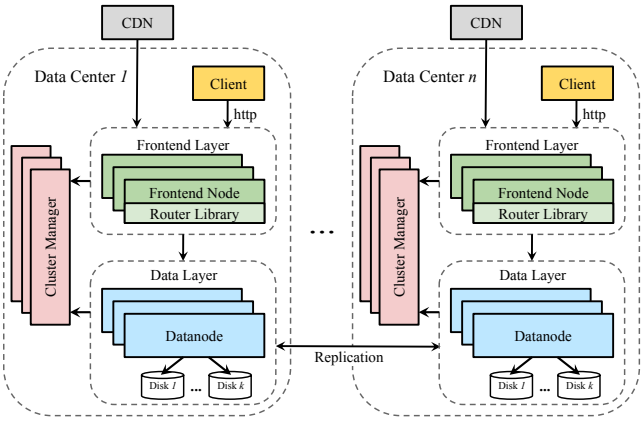


Figure 1: Architecture of Ambry.

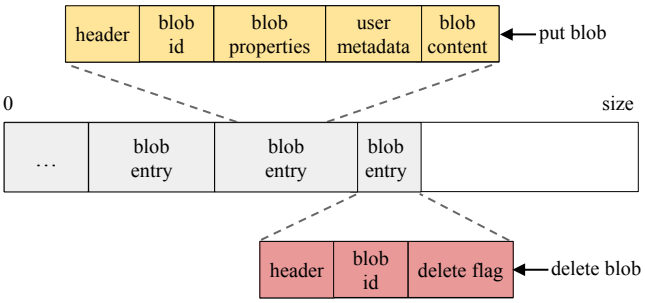


Figure 2: Partition and Blob layout.

Zookeeper [12]. We provide an overview of each component below (details in Section 4).

Cluster Manager: Ambry organizes its data in virtual units called *partitions* (Section 2.2). A partition is a logical grouping of a number of blobs, implemented as a large replicated file. On creation, partitions are read-write, i.e., immutable blobs are read and new blobs can be added. When a logical partition reaches its capacity, it turns read-only. The Cluster Manager keeps track of the state (read-write/read-only) and location of each partition replica, along with the physical layout of the cluster (nodes and disk placement).

Frontend: The Frontends are in charge of receiving and routing requests in a multi-tenant environment. The system serves three request types: put, get, and delete. Popular data is handled by a Content Delivery Network (CDN) layer above Ambry. Frontends receive requests directly from clients or through the CDN (if the data is cached). The Frontends forward a request to the corresponding Datanode(s) and return the response to the client/CDN originating the request.

Datanode: Datanodes store and retrieve the actual data. Each Datanode manages a number of disks. For better performance, Datanodes maintain a number of additional data structures including: indexing of blobs, journals and Bloom filters (Section 4.3).

2.2 Partition

Instead of directly mapping blobs to physical machines, e.g., Chord [26] and CRUSH [29], Ambry randomly groups blobs together into virtual units called *partitions*. The physical placement of partitions on machines is done in a separate procedure. This decoupling of the logical and physical placement enables transparent data movement (necessary for re-balancing) and avoids immediate rehashing of data during cluster expansion.

A partition is implemented as an append-only log in a pre-allocated large file. Currently, partitions are fixed-size during the life-time of the system [1]. The partition size should be large enough that the overhead of partitions, i.e., the additional data structures maintained per partition such as indexing, journals, and Bloom filters (Section 4.3), are negligible. On the other hand, the failure recovery and rebuild time should be small. We use 100 GB partitions in our clusters. Since rebuilding is done in parallel from multiple replicas, we found that even 100 GB partitions can be rebuilt in a few minutes.

Blobs are sequentially written to partitions as put and delete entries (Figure 2). Both entries contain a header (storing the offsets of fields in the entry) and a *blob id*. The blob id is a unique identifier, generated by the Frontend during a put operation, and used during get/delete operations for locating the blob. This id consists of the partition id in which the blob is placed (8 Bytes), followed by a 32 Byte universally unique id (UUID) for the blob. Collisions in blob ids are possible, but very unlikely (the probability is $< 2^{-320}$). For a collision to occur, two put operations have to generate equal UUIDs and chose similar partitions for the blob. Collisions are handled at the Datanodes by failing the late put request.

Put entries also include predefined properties including: blob size, time-to-live, creation time, and content type. Also, there is an optional map of user defined properties followed by the blob.

In order to offer high availability and fault-tolerance, each partition is replicated on multiple Datanodes. For replica placement, Ambry uses a greedy approach based on disk spaces. This algorithm chooses the disk with the most unallocated space while ensuring constraints such as: 1) not having more than one replica per Datanode and 2) having replicas in multiple data centers. Currently, the number of replicas per partition is configurable by the system administrator. As part of future work, we plan to adaptively change the number of replicas based on the popularity of the partition, and use erasure coding for cold data to even further reduce the replication factor.

On creation, partitions are read-write, serving all operations (put, get and delete). When the partition hits its upper threshold on size (capacity threshold) it becomes read-only, thereafter serving only get and delete operations.

The capacity threshold should be slightly less than the max capacity (80-90%) of the partition for two reasons. First, after becoming read-only, replicas might not be completely in-sync and need free space to catch-up later (because of asynchronous writes). Second, delete requests still append delete entries.

Deletes are similar to put operations, but on an existing blob. By default, deletes result in appending a delete entry (with the delete flag set) for the blob (soft delete). Deleted blobs are periodically cleaned up using an in-place compaction mechanism. After compaction, read-only partitions can become read-write if enough space is freed-up. In the rest of the paper we mainly focus on puts, due to the similarity of delete and put operations.

2.3 Operations

Ambry has a lightweight API supporting only 3 operations: put, get, and delete. The request handling procedure is shown in Figure 3. On receiving a request, the Frontend optionally conducts some security checks on the request. Then, using the Router Library (that contains the core logic of operation handling) it chooses a partition, communicates with the Datanode(s) in charge, and serves the request. In the put operation, the partition is chosen randomly (for data balancing purposes), and in the get/delete operation the partition is extracted from the blob id.

[1]As part of future work we plan to investigate potential improvements by using variable-size partitions.

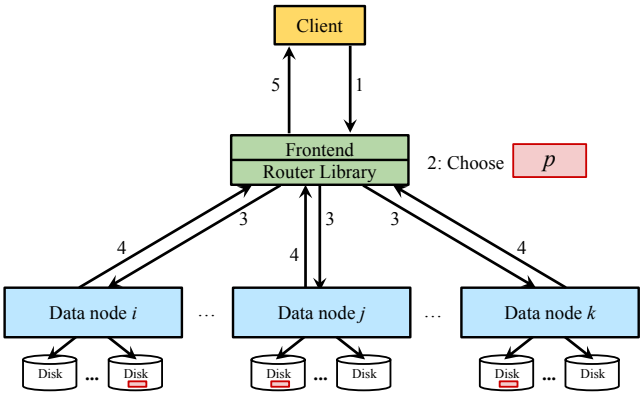


Figure 3: Steps in processing an operation.

Operations are handled in a multi-master design where operations can be served by any of the replicas. The decision of how many replicas to contact is based on user-defined policies. These policies are similar to consistency levels in Cassandra [14], where they control how many (one, k, majority, all) replicas to involve in an operation. For puts (or deletes), the request is forwarded to all replicas, and policies define the number of acknowledgments needed for a success (trade-off between durability and latency). For gets, policies determine how many randomly selected replicas to contact for the operation (trade-off between resources usage and latency). In practice, we found that for all operations the $k = 2$ replica policy gives us the balance we desire. Stricter polices (involving more replicas) can be used to provide stronger consistency guarantees.

Additionally, performing write operations to all replicas placed in multiple geo-distributed datacenters in a synchronous fashion can affect the latency and throughput. In order to alleviate this issue, Ambry uses asynchronous writes where puts are performed synchronously only in the local datacenter, i.e., the datacenter in which the Frontend receiving the request is located. The request is counted as successfully finished at this point. Later on, the blob is replicated to other datacenters using a lightweight replication algorithm (Section 5).

In order to provide read-after-write consistency in a datacenter which a blob has not been replicated yet (e.g., writing to one datacenter and reading from another), Ambry uses proxy requests. If the Frontend cannot retrieve a blob from its local datacenter, it proxies the request to another datacenter and returns the result from there. Although a proxy request is expensive, in practice we found that proxy requests happen infrequently (less than 0.001 % of the time).

3. LOAD BALANCING

Skewed workloads, massively large blobs, and cluster expansions create load imbalance and impact the throughput and latency of the system. Ambry achieves load balancing (in terms of disk usage and request rates) in both static and dynamic (scale-out) clusters.

Static Cluster: Splitting large blobs into multiple small chunks (Section 4.2.1) as well as routing put operations to random partitions, achieves balance in partition sizes. Additionally, using fairly large partition sizes along with relying on CDNs to handle very popular data significantly decrease the likelihood of hot partitions. Using these techniques the

load imbalance of request rates and partition sizes in production gets to as low as 5% amongst Datanodes.

Dynamic Cluster: In practice, read-write partitions receive all the write traffic and also the majority of the read traffic (due to popularity). Since partitions grow in a semi-balanced manner, the number of read-write partitions becomes the main factor of load imbalance. After cluster expansion, new Datanodes contain only read-write partitions, while older Datanodes contain mostly read-only partitions. This skewed distribution of read-write partitions creates a large imbalance in the system. In our initial version, the average request rates of new Datanodes were up to 100x higher than old Datanodes and 10x higher than the average-aged ones.

To alleviate this issue, Ambry employs a rebalancing mechanism that returns the cluster to a semi-balanced state (in terms of disk usage and request rate) with minimal data movement. The rebalancing approach reduces request rate and disk usage imbalance by 6-10x and 9-10x respectively.

Ambry defines the ideal (load balanced) state as a triplet (idealRW, idealRO, idealUsed) representing the ideal number of read-write partitions, ideal number of read-only partitions and ideal disk usage each disk should have. This ideal state (idealRW, idealRO, idealUsed) is computed by dividing the total number of read-write/read-only partitions and total used disk space by the number of disks in the cluster, respectively. A disk is considered above (or below) ideal if it has more (or less) read-write/read-only partitions or disk usage than the ideal state.

The rebalancing algorithm attempts to reach this ideal state. This is done by moving partitions from disks above ideal to disks below ideal using a 2 phase approach, as shown in the pseudo-code below.

Algorithm 1 Rebalancing Algorithm

```
 1: // Compute ideal state.
 2: idealRW=totalNumRW / numDisks
 3: idealRO=totalNumRO / numDisks
 4: idealUsed=totalUsed / numDisks

 5: // Phase1: move extra partitions into a partition pool.
 6: partitionPool = {}
 7: for each disk d do
 8:     // Move extra read-write partitions.
 9:     while d.NumRW > idealRW do
10:         partitionPool += chooseMinimumUsedRW(d)
11:     // Move extra read-only partitions.
12:     while d.NumRO > idealRO & d.used > idealUsed do
13:         partitionPool += chooseRandomRO(d)

14: // Phase2: Move partitions to disks needing partitions.
15: placePartitions(read-write)
16: placePartitions(read-only)

17: function PLACEPARTITIONS(Type t)
18:     while partitionPool contains partitions type t do
19:         D=shuffleDisksBelowIdeal()
20:         for disk d in D and partition p in pool do
21:             d.addPartition(p)
22:             partitionPool.remove(p)
```

Phase1 - Move to Partition Pool: In this phase, Ambry moves partitions from disks above ideal into a pool, called *partitionPool* (Lines 6-13). At the end of this phase no disk should remain above ideal, unless removing any partition would cause it to fall below ideal.

Ambry starts from read-write partitions (which are the main factor), and moves extra ones solely based on idealRW threshold. The same process is repeated for read-only partitions, but with considering both idealRO and idealUsed when moving partitions. The strategy of choosing which partition to move is based on minimizing data movement. For read-write partitions, the one with the minimum used capacity is chosen, while for read-only partitions, a random one is chosen since all such partitions are full.

Datacenter	Datanode	Disk	Size	Status
DC 1	Datanode 1	disk 1 ... disk k	4 TB ... 4 TB	UP ... UP
DC 1	Datanode 2	disk 1 ... disk k'	4 TB ... 4 TB	DOWN ... UP
	...	...	...	...
DC n	Datanode j	disk 1 ... disk k''	1 TB ... 1 TB	DOWN ... UP

Table 1: Hardware layout in Cluster Manager.

Phase2 - Place Partitions on Disks: In this phase, Ambry places partitions from the partition pool on disks below ideal (Lines 14-16), starting from read-write partitions and then read-only ones. Partitions are placed using a random round-robin approach (Line 17-22). Ambry finds all disks below ideal, shuffles them, and assigns partitions to them in a round-robin fashion. This procedure is repeated until the pool becomes empty.

After finding the the new placement, replicas are seamlessly moved by: 1) creating a new replica in the destination, 2) syncing the new replica with old ones using the replication protocol while serving new writes in all replicas, and 3) deleting the old replica after syncing.

4. COMPONENTS IN DETAIL

In this section we further discuss the main components of Ambry. We describe the detailed state stored by the Cluster Manager (Section 4.1), extra responsibilities of Frontends including chunking and failure detection (Section 4.2), and additional structures maintained by the Datanodes (Section 4.3).

4.1 Cluster Manager

The Cluster Manager is in charge of maintaining the state of the cluster. Each datacenter has its local Cluster Manager instance(s) kept in-sync with others using Zookeeper. The state stored by the Cluster Manager is very small (less than a few MBs in total), consisting of a hardware and logical layout.

4.1.1 Hardware Layout

The hardware layout includes information about the physical structure of the cluster, i.e., the arrangement of datacenters, Datanodes, and disks. It also maintains the raw capacity and status, i.e., healthy (UP) or failed (DOWN), for each disk. An example hardware layout is shown in Table 1. As shown, Ambry works in a heterogeneous environment with different hardware and configuration used inside and across different datacenters.

4.1.2 Logical Layout

The logical layout maintains the physical location of partition replicas, and the state (read-only/read-write) of each partition. In order to find the state of a partition, the Cluster Manager periodically contacts the Datanodes, and requests the state of their partitions. This layout is used for choosing a partition to write a new blob to (put operation), and locating the Datanode in charge of a given replica (all

Partition id	State	Placement
partition 1	read-write	DC 1: Datanode 1: disk 1 DC 1: Datanode 4: disk 5 ... DC 3: Datanode 7: disk 2
...	...	...
partition p	read-only	DC 1: Datanode 1: disk 1 ... DC 4: Datanode 5: disk 2

Table 2: Logical Layout in Cluster Manager.

operations). An example of this layout is shown in Table 2. As shown, replicas of a partition can be placed on multiple Datanodes in one datacenter, and/or in different datacenters. Additionally, one disk (e.g., DC 1: Datanode 1: disk 1) can contain replicas of distinct partitions, where some are read-only and some are read-write. Partitions are added by updating the logical layout stored in the Cluster Manager instances[2].

4.2 Frontend Layer

The Frontend is the entry point to Ambry for external requests. Each datacenter has its own set of Frontends. Frontends are decentralized involving no master or coordination, identically performing the same task, and stateless with all state stored in the Cluster Manager (which is periodically pulled). This design enhances scalability (new Frontends can be added without much performance penalty), fault-tolerance (requests can be forwarded to any Frontend), and failure recovery (failed Frontends can quickly be replaced) for Frontends. Frontends have three main responsibilities:

1. Request Handling: This involves receiving requests, routing them to the corresponding Datanode(s) using the Router Library (Section 4.2.1), and sending back the response.
2. Security Checks: Optionally performing security checks, such as virus scanning and authentication on requests.
3. Capturing Operations: Pushing events to a change capture system out of Ambry for further offline analysis, such as finding request patterns of the system. We use Kafka [13] as our change-capture system due to the high durability, high throughput, and low overhead it provides.

4.2.1 Router Library

The Router Library contains all the core logic of handling requests and communicating with Datanodes. Frontends simply embed and use this library. Clients can bypass Frontends by embedding this library and directly serving requests. This library includes four main procedures: 1) policy-based routing, 2) chunking large blobs, 3) failure detection, and 4) proxy requests.

Policy Based Routing: On receiving a request, the library decides which partition to involve (randomly chosen for puts and extracted from blob id for gets/deletes). Then, based on the policy used ({one, k, majority, all} discussed in Section 2.3), it communicates with the corresponding replica(s) until the request is served/failed.

Chunking: Extremely large blobs (e.g., videos) create load imbalance, block smaller blobs, and inherently have high latency. To mitigate these issues, Ambry splits large blobs into smaller equal-size units called chunks. A large chunk size does not fully resolve the large blob challenges and a small chunk size adds too much overhead. Based on our current large blob size distribution, we found the sweet spot for the chunk size to be in the range of 4 to 8 MB[3].

[2]Currently, the system administrator manually adds partitions in order to prevent unwanted and rapid cluster growths. However, this can easily be automated.

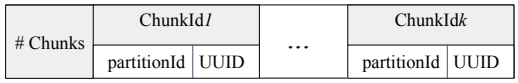


Figure 4: Content of the metadata blob used for chunked blobs.

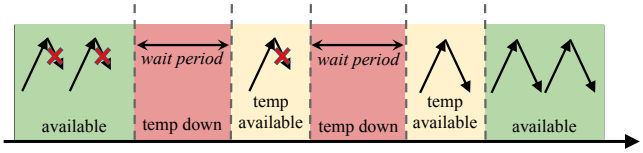


Figure 5: Failure detection algorithm with maximum tolerance of 2 consecutive failed responses.

During a put operation, a blob b is split into k chunks $\{c_1, c_2, ..., c_k\}$, each treated as an independent blob. Each chunk goes through the same steps as a normal put blob operation (Section 2.3), most likely being placed on a different partition. It is also assigned a unique chunk id with the same format as a blob id. In order to be able to retrieve b Ambry creates a metadata blob $b_{metadata}$ for b. $b_{metadata}$ stores the number of chunks and chunk ids in order, as shown in Figure 4. This metadata blob is then put into Ambry as a normal blob and the blob id of $b_{metadata}$ is returned to the user as the blob id of b. If the put fails before writing all chunks, the system will issue deletes for written chunks and the operation has to be redone.

During a get, the metadata blob is retrieved and chunk ids are extracted from it. Then, Ambry uses a sliding buffer of size s to retrieve the blob. Ambry queries the first s chunks of the blob independently and in parallel (since they are most likely written on unique partitions placed on separate Datanodes). When the first chunk in the buffer is retrieved, Ambry slides the buffer to the next chunk, and so on. The whole blob starts being returned to the user the moment the first chunk of the blob is retrieved.

Although an extra put/get is needed in this chunking mechanism (for the metadata blob), overall, our approach improves latency since multiple chunks are written and retrieved in parallel.

Zero-cost Failure Detection: Failures happen frequently in a large system. They range from unresponsiveness and connection timeouts, to disk I/O problems. Thus, Ambry needs a failure detection mechanism to discover unavailable Datanodes/disks and avoid forwarding requests to them.

Ambry employs a zero-cost failure detection mechanism involving no extra messages, such as heartbeats and pings, by leveraging request messages. In practice, we found our failure detection mechanism is effective, simple, and consumes very little bandwidth. This mechanism is shown in Figure 5. In this approach, Ambry keeps track of the number of consecutive failed requests for a particular Datanode (or disk) in the last *check_period* of time. If this number exceeds a MAX_FAIL threshold (in our example set to 2) the

[3]Chunk size is not fixed and can be adapted to follow the growth in blob sizes, improvements in network, etc.

Datanode is marked as *temporarily_down* for a *wait_period* of time. In this state all queued requests for this Datanode will eventually time out and need to be reattempted by the user. After the *wait_period* has passed, the Datanode becomes *temporarily_available*. When a Datanode is in the *temporarily_available* phase, if the next request sent to that Datanode fails, it will move to the *temporarily_down* phase again. Otherwise, it will be marked as *available*, working as normal again.

Proxy Requests: As described in Section 2.3, Ambry uses proxy requests to reach higher availability and read-after-write consistency in remote datacenters. When a blob has not been replicated in the local datacenter yet, requests for that blob are forwarded to other datacenters and served there (proxy requests). However, datacenter partitions can cause unavailability of unreplicated data until the partition is healed and replicas converge.

Proxy requests are handled by the Router Library, transparently from the user issuing the request. In practice, we found proxy requests occur less than 0.001% of the time, thus minimally affecting the user experience.

4.3 Datanode Layer

Datanodes are in charge of maintaining the actual data. Each Datanode manages a number of disks, and responds to requests for partition replicas placed on its disks. Puts are handled by writing to the end of the partition file. Gets can be more time-consuming, especially since the location of the blob in the partition is not known. To minimize both read and write latencies, Datanodes employ a few techniques:

- **Indexing blobs:** Ambry stores an index of blob offsets per partition replica to prevent sequential sweeps for finding blobs (Section 4.3.1).
- **Exploiting OS cache:** Ambry utilizes OS caching to serve most reads from the RAM, by limiting the RAM usage of other components (Section 4.3.2).
- **Batched writes, with a single disk seek:** Ambry batches writes for a particular partition together and periodically flushes the writes to disk. Thus, it incurs at most one disk seek for a batch of sequential writes. The flush period is configurable and trades off latency for durability. Although, batching can introduce overheads of flushing, dirty buffers, and tuning, the benefits outweigh these overheads.
- **Keeping all file handles open:** Since partitions are typically very large (100 GB in our setting), the number of partition replicas placed on a Datanode is small (a few hundred). Thus, Ambry keeps all file handles open at all times.
- **Zero copy gets:** When reading a blob, Ambry utilizes a zero copy [25] mechanism, i.e., the kernel directly copies data from disk to the network buffer without going through the application. This is feasible since the Datanodes do not perform any computation on the data at get operations.

4.3.1 Indexing

To find the location of a blob in a partition replica with low latency, the Datanode maintains a light-weight in-memory indexing per replica, as shown in Figure 6. The indexing is sorted by blob id, mapping the blob id to the start offset of the blob entry. The indexing is updated in an online fashion whenever blobs are put (e.g., blob 60) or deleted (e.g., blob 20).

Similar to SSTables [5], Ambry limits the size of the index by splitting it into segments, storing old segments on disk,

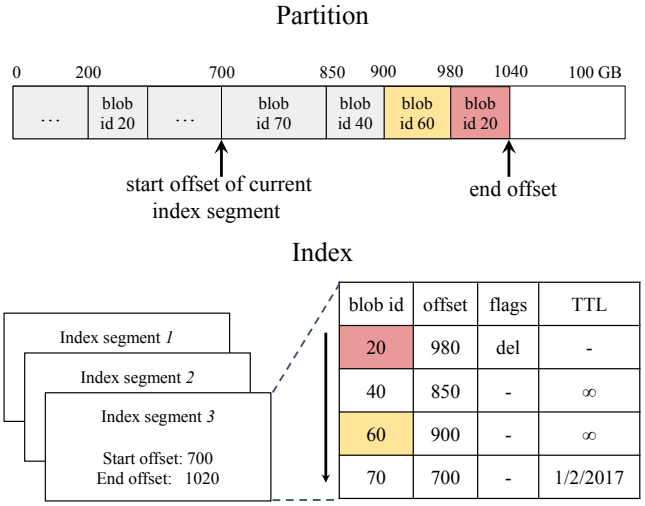


Figure 6: Indexing of blob offsets in a partition replica. When blobs are put (blob 60) or deleted (blob 20) the indexing stucture is updated.

and maintaining a Bloom filter for each on-disk segment (Section 4.3.2).

The indexing also stores a flag indicating if a blob has been deleted and optionally a time-to-live (TTL). During get operations, if the blob has expired or been deleted, an error will be returned before reading the actual data.

Note that the indexing does not contain any additional information affecting the correctness of the system, and just improves performance. If a Datanode fails, the whole indexing can be reconstructed from the partition.

4.3.2 Exploiting The OS Cache

Recently written data, which is the popular data as well, is automatically cached without any extra overhead (by the operating system). By exploiting this feature, many reads can be served from memory, which significantly improves performance. Thus, Ambry limits the memory usage of other data structures in the Datanode. Ambry bounds the indexing by splitting it into segments, with only the latest segment remaining in-memory (Figure 6). New entries are added to the in-memory segment of the indexing. When the in-memory segment exceeds a maximum size it is flushed to disk as a read-only segment. This design also helps toward failure recovery since only the in-memory segment needs to be reconstructed. Looking up blob offsets is done in reverse chronological order, starting with the latest segment (in-memory segment). Thus, a delete entry will be found before a put entry when reading a blob. This ensures deleted blobs are not retrievable.

Bloom Filters: To reduce lookup latency for on-disk segments, Ambry maintains an in-memory Bloom filter for each segment, containing the blob ids in that index segment. Using Bloom filters, Ambry quickly filters out which on-disk segment to load. Thus, with high probability, it incurs only one disk seek. However, due to our skewed workload a majority of reads just hit the in-memory segment, without any disk seeks.

5. REPLICATION

Replicas belonging to the same partition can become out of sync due to either failures, or asynchronous writes that write to only one datacenter. In order to fix inconsistent

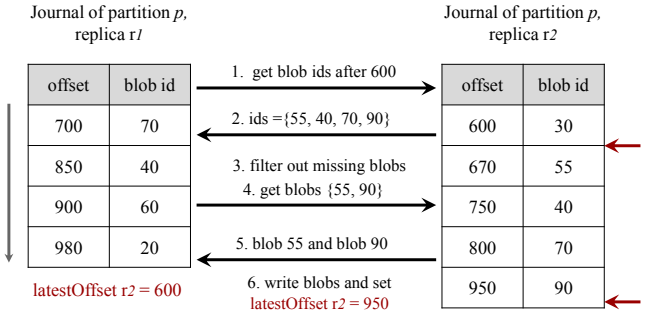


Figure 7: Journals for two replicas of the same partition and an example of the replication algorithm.

replicas, Ambry uses an asynchronous replication algorithm that periodically synchronizes replicas. This algorithm is completely decentralized. In this procedure each replica individually acts as a master and syncs-up with other replicas, in an all-to-all fashion. Synchronization is done using an asynchronous two-phase replication protocol as follows. This protocol is a pull-based approach where each replica independently requests for missing blobs from other replicas.

- **First Phase:** This phase finds missing blobs since the last synchronization point. This is done by requesting blob ids of all blobs written since the latest syncing offset and then filtering the ones missing locally.
- **Second Phase:** This phase replicates missing blobs. A request for only the missing blobs is sent. Then, the missing blobs are transferred and appended to the replica.

In order to find the recently written blobs quickly, the replication algorithm maintains an additional data structure per replica, called a *journal*. The journal is an in-memory cache of recent blobs ordered by their offset. Figure 7 shows example journals of two replicas (r_1 and r_2) and the two phase replication procedure for r_1 syncing with r_2 from latestOffset 600. In the first phase, r_1 requests all recently added blobs in r_2 after latestOffset; using the journal r_2 returns a list B ={55, 40, 70, 90} of blob ids; and r_1 filters out the missing blobs (blob 55 and 90). In the second phase, r_1 receives the missing blobs, appends the blobs to the end of the replica, and updates the journal, indexing and latestOffset.

To provide improved efficiency and scalability of the system, the replication algorithm employs a number of further optimizations:

- Using separate thread pools for inter- and intra-datacenter replication with different periods.
- Batching requests between common partition replicas of two Datanodes, and batching blobs transferred across datacenters.
- Prioritizing lagging replicas to catch up at a faster rate (by using dedicated threads for lagging replicas).

6. EXPERIMENTAL RESULTS

We perform three kinds of experiments: small cluster (Section 6.1), production cluster (Section 6.2), and simulations (Section 6.3).

6.1 Throughput and Latency

In this section we measure the latency and throughput of the system using a micro-benchmark that stress-tests the system (Section 6.1.1), under read-only, write-only and read-write workloads.

6.1.1 Micro-Benchmark

We first measure the peak throughput. We designed a micro-benchmark that linearly adds load to the system (by adding more clients), until the saturation point where no more requests can be handled. Each client sends requests one at a time with the next request sent right after a response.

This benchmark has three modes: Write, Read, and Read-Write. In Write mode, random byte array blobs are put with varying number of clients. In Read mode, first blobs are written at saturation rate for a *write-period* of time. Then, randomly chosen blobs are read from the written blobs[4]. In most experiments we set the write-period long enough that most read requests (>80%) are served by disk, rather than RAM. Read-Write is a similar to Read, but serving 50% reads and 50% writes after the write-period.

Since latency and throughput are highly correlated with blob size, we use fixed-size blobs in each run, but vary the blob size.

6.1.2 Experimental Setup

We deployed Ambry with a single Datanode. The Datanode was running on a 24 core CPU with 64 GB of RAM, 14 1TB HDD disks, and a full-duplex 1 Gb/s Ethernet network. 4 GB of the RAM was set aside for the Datanode's internal use and the rest was left to be used as Linux Cache. We created 8 single-replica 100 GB partitions on each disk, with a total of 122 partitions. Using 14 disks with a 1 Gb/s network might look like an overkill. However, disk seeks are the dominant latency factor for small blobs. Since a large portion of blobs are small (< 50 KB), we need the parallelism created by using multiple disks (more detail in Section 6.1.4). Note that Ambry is designed as a cost-effective system using cheap HDD disks.

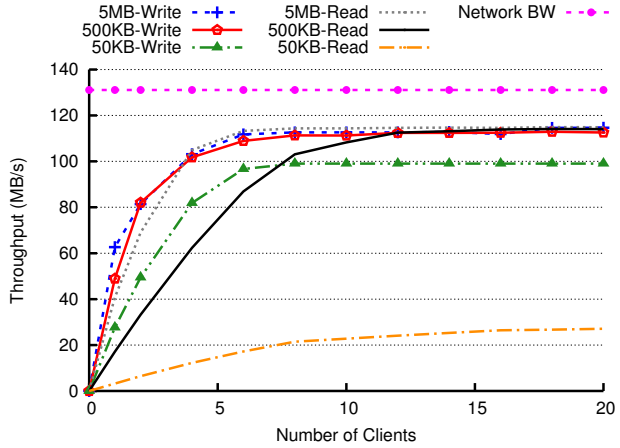
Clients send requests from a few machines located in the same datacenter as the Datanode. These clients, that are acting as Frontends, directly send requests to the Datanode. The micro-benchmark discussed above was used with varying blob sizes {25 KB, 50 KB, 100 KB, 250 KB, 500 KB, 1 MB, 5 MB}. We did not go above 5 MB since blobs are chunked beyond that point.

6.1.3 Effect of Number of Clients

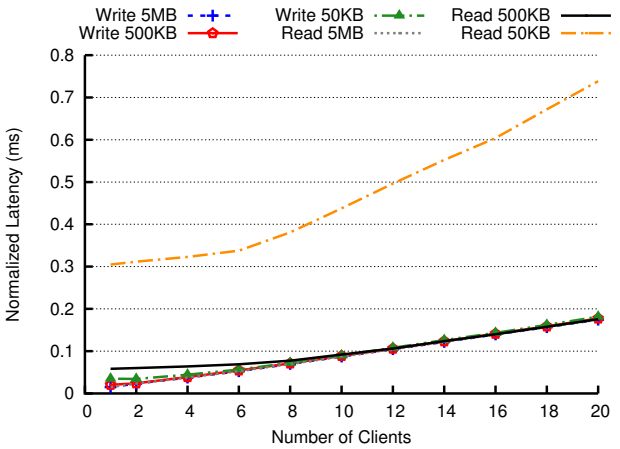
We ran the micro-benchmark with varying blob sizes, while linearly increasing the number of clients. For Read mode, the write-period was set such that 6 times the RAM size, was first written. Figure 8a shows the throughput in terms of MB/s served by the system. Adding clients proportionally increases the throughput until the saturation point. Saturation occurs at 75%-88% of the network bandwidth. The only exception is reading small blobs due to frequent disk seeks (discussed in Section 6.1.4). Saturation is reached quickly (usually $\leq$ 6 clients) since clients send requests as fast as possible in the benchmark.

Figure 8b shows the latency normalized by blob size (i.e., average latency divided by blob size). Latency stays almost constant before reaching saturation point, and then increases linearly beyond the throughput saturation point. The linear increase in latency after saturation indicates the system does not add additional overhead beyond request serving.

[4]The random strategy gives a lower bound on the system's performance since real-world access patterns are skewed toward recent data.



(a) Throughput



(b) Latency normalized by blob size

Figure 8: Throughput and latency of read and write requests with varying number of clients on different blob sizes. These results were gathered on a single Datanode deployment.

6.1.4 *Effect of Blob Size*

In Figures 9a and 9b we analyzed the maximum throughput (with 20 clients) under different blob sizes and workloads. For large objects (>200 KB), the maximum throughput (in MB/s) stays constant and close to maximum network bandwidth across all blob sizes. Similarly, throughput in terms of requests/s scales proportionally.

However, for Read and Read-Write, the read throughput in terms of MB/s drops linearly for smaller sizes. This drop is because our micro-benchmark reads blobs randomly, incurring frequent disk seeks. The effect of disk seeks is amplified for smaller blobs. By further profiling the disk using Bonnie++ [1] (an IO benchmark for measuring disk performance), we confirmed that disk seeks are the dominant source of latency for small blobs. For example, when reading a 50 KB blob, more than 94% of latency is due to disk seek (6.49 ms for disk seek, and 0.4 ms for reading the data).

Read and Write workload mostly utilize only the outbound and inbound link on the Datanode, respectively. However, Read-Write workload has a better mix and evenly utilizes both links reaching higher throughput. Therefore, in our full-duplex network infrastructure the Read-Write mode is able to saturate both links reaching almost double the network bandwidth ($\simeq$ 1.7 Gb/s in total out of the 2 Gb/s available). For smaller size objects it reaches twice the Read throughput, since Read-Write is a 50-50 workload with reads being the limiting factor.

Figure 9c demonstrates the trend in latency under various blob sizes. These results are before the saturation point with 2 clients. Similar to throughput, latency grows linearly except for reading small blobs. The higher latency in Read is because most read requests incur a disk seek, while write requests are batched and written together. The Read-Write latency falls halfway between Read and Write latency because of its mixed workload.

	Average	Max	Min	StdDev
Disk Reads	17 ms	67 ms	1.6 ms	9 ms
Cached Reads	3 ms	5 ms	1.6 ms	0.4 ms
Improvement	**5.5x**	13x	0	23x

Table 3: Comparision of get request latency when most of requests (83%) are served by disk (Disk Reads) and when all requests are served by linux cache (Cached Reads) for 50 KB blobs.

6.1.5 *Variance in Latency*

The tail and variance in request latencies are important. Figure 10 shows the CDFs of Write, Read, and Read-Write mode experiments with 2 clients. The CDF of Read and Write mode is very close to a vertical line, with a short tail and a majority of values close to the median. The jump around 0.15 in Read mode (for 50 KB blob size) is because a small fraction of requests are served using the Linux cache which is orders of magnitudes faster than disk (Section 6.1.6). The Read-Write mode is a mixture of the Read and Write CDF with a change around 0.5, following the 50% read - 50% write workload pattern.

6.1.6 *Effect of Linux Cache*

We ran the micro-benchmark on 50 KB blobs and 2 clients in two configurations: 1) writing 6 times more than the RAM size before reading, so that most requests (83 %) are served by disk (Disk Read), and 2) writing data equal to the RAM size to keep all data in RAM (Cached Read). Table 3 compares these two modes.

The Cached Read experiment performed more than 2100 requests/s (104 MB/s reaching 79% network bandwidth) matching the maximum write throughput (Section 6.1.3), compared to 540 requests/s for Disk Reads. We also measured the average, maximum, minimum, and standard deviation of latency, shown in Table 3. In both cases, the minimum is equal to reading from the Linux Cache. However, the Cached Read case improves the average and max latency by 5.5x and 13x, respectively. This shows the significance of exploiting the Linux Cache (Section 4.3.2).

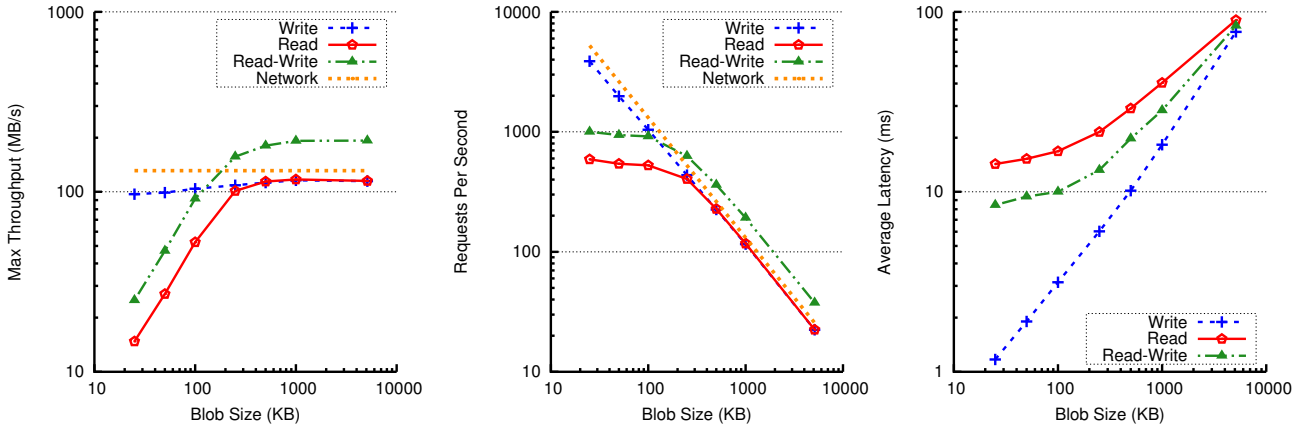
6.2 Geo-distributed Optimizations

We analyzed our replication algorithm among 3 different datacenters at LinkedIn {DC1, DC2, DC3}, located all across the US. All experiments in this section are from production workloads.

6.2.1 *Replication Lag*

We define *replication lag* between a pair of replicas (r_1, r_2) as the difference of r_2's highest used offset and r_1's latest synced offset with r_2. Note that not all data in the replication lag needs to be transfered to r_1, since it could receive the missing blobs from other replicas as well.

We measured the replication lag among all replicas of a given Datanode and the rest of the cluster, and found that



(a) Throughput (MB/s) (b) Throughput (requests/s) (c) Latency

Figure 9: Effect of blob size on maximum throughput, both in terms of MB/s and requests/s, and latency. Results were gathered on a write-only, read-only, and mixed (50%-50%) workload. Reads for small blob sizes (<200 KB) are slowed down by frequent disk seeks, while other requests saturate the network link.

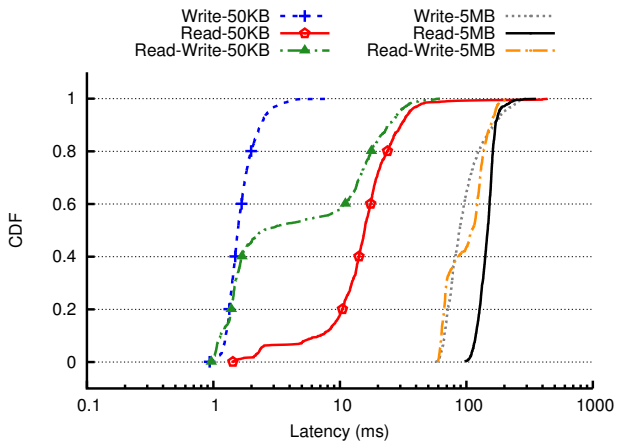


Figure 10: CDF of read and write latency with 2 clients for various blob sizes. Read-Write falls in the middle with change around 0.5 due to the 50-50 mixed workload.

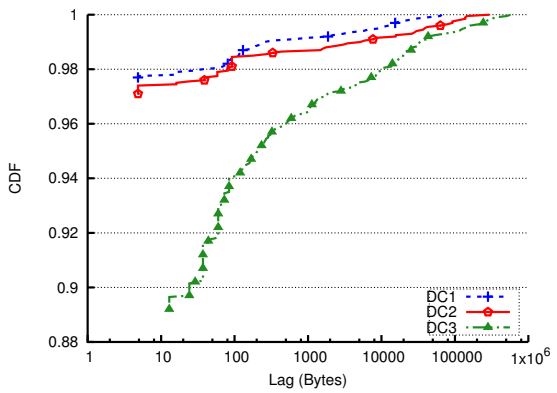


Figure 11: CDF of Replication Lag among replica pairs of a given Datanode and the rest of the cluster. Most values were zero, and are omitted from the graph.

more than 85% of the values were 0. Figure 11 shows the CDF of non-zero values grouped by datacenter. The 95th percentile is less than 1 KB for 100 GB partitions (in all datacenters), with slightly worse lag in datacenter 3 since it is placed farther apart from others.

6.2.2 Replication Bandwidth

Ambry relies on background replication to write data to other datacenters. We measured the aggregate network bandwidth used for inter-datacenter replication during a 24 hour period, shown in Figure 12. The aggregate bandwidth is small (< 10 MB/s), similar across all datacenters, and correlated to request rates with a diurnal pattern. This value is small because we batch replication between common replicas and due to the read-heavy nature of the workload.

Figure 13 demonstrates the CDF of average replication bandwidth per Datanode, for both intra- and inter-datacenter replication. Intra-datacenter bandwidth is minimal (< 200 B/s at 95th percentile), especially compared to inter-datacenter with 150-200 KB/s at 95th percentile (1000x larger). The higher value for inter-datacenter is because of asynchronous writes. However, the inter-datacenter bandwidth is still negligible (≈ 0.2% of a 1 Gb/s link). The small difference among the 3 datacenters is due to the different request rates they receive.

Figure 14 shows a zoomed in graph of only inter-datacenter bandwidth. Inter-datacenter replication has a short tail with the 95th to 5th percentile ratio of about 3x. This short tail is because of the load-balanced design in Ambry. Intra-datacenter replication has a longer tail, as well as many zero values (omitted from the graph). Thus, replication either uses almost zero bandwidth (intra-datacenter) or almost balanced bandwidth (inter-datacenter).

6.2.3 Replication Latency

We define *replication latency* as the time spent in one iteration of the replication protocol, i.e., $T_{missing_blobs_received}$ minus $T_{replication_initiated}$. Figure 15 demonstrates the CDF of average replication latency for intra- and inter-datacenter, in our production workload.

Inter-datacenter replication latency is low with a median of less than 150 ms, and a very short tail. Although this la-

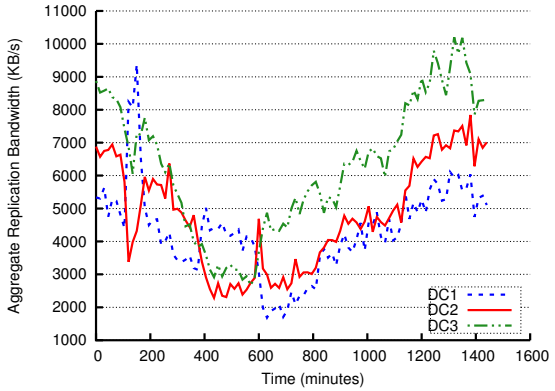


Figure 12: Aggregate network bandwidth used for inter-datacenter replication during a 24 hour period in production.

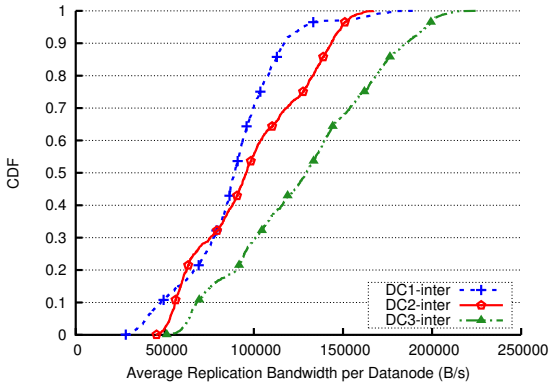


Figure 14: CDF of average inter-datacenter network bandwidth used per Datanode over a 24 hour period in production.

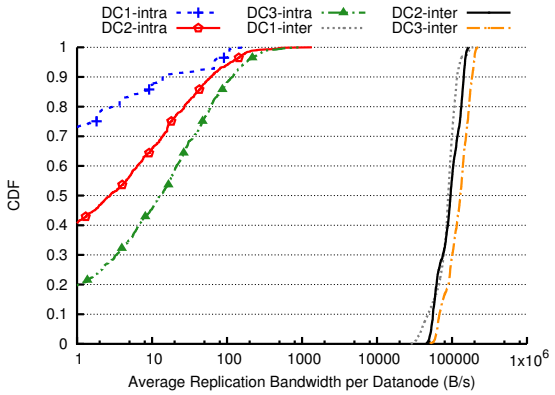
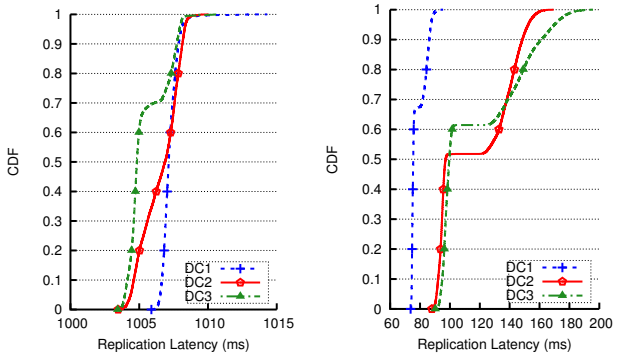


Figure 13: CDF of average network bandwidth used per Datanode for intra- and inter-datacenter replication over a 24 hour period in production.



(a) Intra Datacenter (b) Inter Datacenter

Figure 15: CDF of average replication latency (i.e., time spent to recieve missing blobs) for intra- and inter-datacenter in production environment.

tency might appear to be high, the number of proxy requests remain near-zero ($< 0.001\%$). This is because users usually read data from the same local datacenter to which they have recently written. Therefore, replication has a minimal effect on user experience.

Surprisingly, intra-datacenter replication latency is relatively high (6x more than inter-datacenter) and with little variance. This pattern is because of a pre-existing and prefixed artificial added delay of 1 second, intended to prevent incorrect blob collision detections. If a blob is replicated in a datacenter faster than the Datanode receives the initial put request (which is possible with less strict policies), the Datanode might consider the put as a blob collision and fail the request. The artificial delay is used to prevent this issue in intra-datacenter replication. This relatively small delay does not have a significant impact on durability or availability of data since intra-replication is only used to fix failed/slow Datanodes, which rarely occurs.

6.3 Load Balancing

Since cluster growth occurs infrequently (every few months at most), we implemented a simulator to show the behavior of Ambry over a long period of time (several years), and at large scale (hundreds of Datanodes). We tried a workload that is based on production workloads. All results in this section are gathered using the simulator.

6.3.1 Simulator Design

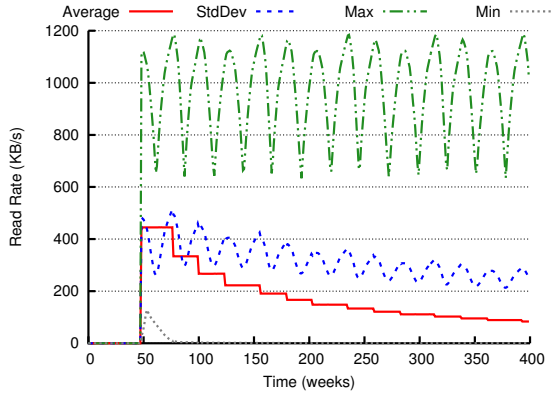
The simulator's design resembles Ambry's architecture and requests are served using the same path. However, there are no real physical disks. For handling requests, only the metadata (blob id and size) is stored/retrieved, and the effect of requests are reflected (e.g., disk space increase).

Workload: We use a synthetic workload closely resembling the real-world traffic at LinkedIn. This workload preserves the rate of each type of request (read, write, and delete), the blob size distribution, and the access pattern of blobs (based on age).

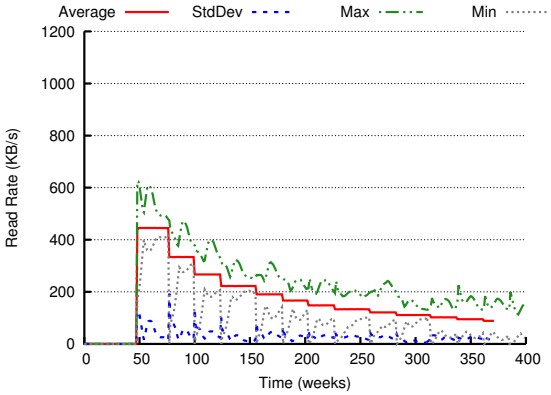
Cluster Expansion: The simulator starts with an initial set of Datanodes, disks, and partitions in one datacenter. Over time, when partitions reach the capacity threshold, a new batch of partitions are added using the replica placement strategy from Section 2.2. If partitions cannot be added (e.g., if there is not enough unallocated space on disks), a batch of new Datanodes are added.

6.3.2 Experiment Setup

The simulation is run in a single datacenter, with 10 Frontend nodes. The experiment starts with 15 Datanodes, each with 10 4TB disks, and 1200 100GB partitions with 3 replicas. At each partition and Datanode addition point, a batch of 600 partitions and 15 Datanodes are added, respectively. The simulation is run over 400 weeks (almost 8 years) and

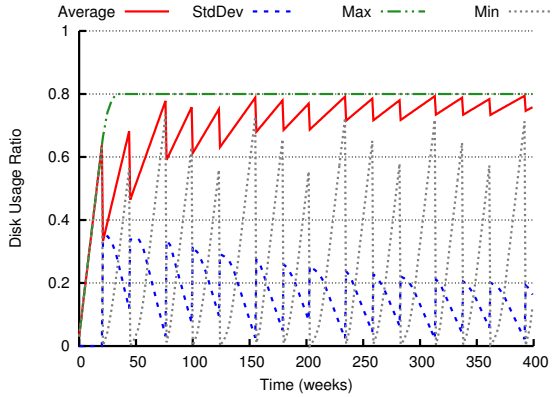


(a) Without rebalancing

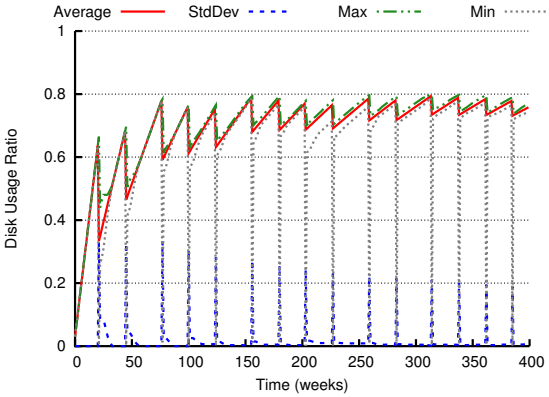


(b) With rebalancing

Figure 16: Average, standard deviation, maximum and minimum of average read rate (KB/s) among disks over a 400 week interval. The system is bootstrapping in the first few weeks, and the results are omitted.



(a) Without rebalancing



(b) With rebalancing

Figure 17: Average, standard deviation, maximum and minimum of disk usage ratio (i.e., used space divided by total space) over a 400 week interval.

up to 240 Datanodes. The simulation is run with and without rebalancing with the exact same configuration, while measuring request rates, disk usage, and data movement.

6.3.3 Request Rate

We measured the read rate (KB/s) for each disk at any point of time. Figure 16 demonstrates the average, standard deviation, maximum and minimum among these values, for the system with and without rebalancing. The results for write rates were similar and removed due to lack of space.

The average, which is also the ideal, is a dropping step function. The drops are points where new Datanodes were added to the cluster. In case of no rebalancing, a majority of the disks are old read-only disks with almost zero traffic, while a few disks receive most of the request. Thus, the minimum is close to zero. Also, the maximum and standard deviation are significant (3x-7x and 1x-2x larger than the average, respectively). When rebalancing is added, the minimum and maximum move close to the average, and the standard deviation drops close to zero. We conclude that Ambry's load balancing is effective.

6.3.4 Disk Usage

We analyzed the disk usage ratio, i.e., used space divided by total space among disks, with and without rebalancing. As seen in Figure 17, without rebalancing, the maximum stays at the capacity threshold since some disks become and remain full, while the minimum drops to zero whenever new Datanodes are added. With rebalancing, the maximum and minimum move closer to the average with temporary drops in the minimum until rebalancing is completed. Additionally, the standard deviation drops significantly, becoming almost zero with temporary spikes on Datanode addition points.

6.3.5 Evaluation Over Time

We evaluated the improvement over time by measuring the integral of range (max-min) and standard deviation for request rate and disk usage over the 400 week interval. As shown in Table 4, rebalancing has a prominent effect improving the range and standard deviation by 6x-9x and 8x-10x, respectively.

6.3.6 Data Movement

Whenever rebalancing is triggered, we measure the minimum data movement needed to reach an ideal state and the data movement caused by rebalancing. We calculate the minimum data movement by adding the difference between ideal and current disk usage among all disks above ideal disk usage. This value is a lower bound on the feasible minimum data movement since data is moved in granularity of partitions. As shown in Figure 18, the data movement of rebalancing is very close and always below the minimum. This is because the rebalancing algorithms trades off perfect balance (ideal state) for less data movement. Specifically, the rebalancing algorithm usually does not remove (or add)

Integral over 400 weeks	Write Avg	Read Avg	Disk Usage
Range w/o RB	63,000	340,000	200
Range w/ RB	8,500	52,000	22
Improvement	7.5x	6x	9x
StdDev w/o RB	21,00	112,000	67
StdDev w/ RB	2500	11,000	6.7
Improvement	8x	10x	10x

Table 4: Improvement of range (max-min) and standard deviation of request rates and disk usage over a 400 week interval. Results are from the system with rebalancing (w/ RB) and without rebalancing (w/o RB).

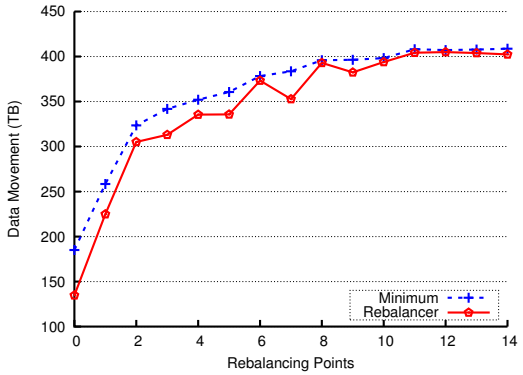


Figure 18: Data movement of the rebalancing algorithm at each rebalancing point (i.e., whenever new Datanodes are added) over a 400 week interval.

a partition from a disk if it would go below (or above) the ideal state, even if this were to cause slight imbalance.

7. RELATED WORK

File Systems: The design of Ambry is inspired by log-structure file systems (LFS) [21, 23]. These file systems are optimized for write throughput by sequentially writing in log-like data structures and relying on the OS cache for reads. Although these single machine file systems suffer from fragmentation issues and cleaning overhead, the core ideas are very relevant, especially since blobs are immutable. The main differences are the skewed data access pattern in our workload and additional optimization such as segmented indexing and Bloom filters used in Ambry.

There has been work on handling metadata and small files more efficiently. Some of these techniques include reducing disk seeks [9], using a combination of log-structured file systems (for metadata and small data) and fast file systems (for large data) [30], and storing the initial segment of data in the index block [17]. Our system resolves this issue by using in-memory segmented indexing plus Bloom filters and batching techniques.

Distributed File Systems: Due to the extremely large amount of data and data sharing needs, many distributed file systems such as NFS [22] and AFS [16], and even more reliable ones handling failures, such as GFS, HDFS, and Ceph [10, 24, 28] have emerged. However, all these systems suffer from the high metadata overhead and additional capabilities (e.g., nested directories, permissions, etc.) unnecessary for a simple blob store. In many of these systems (e.g., HDFS, GFS, and NFS) the metadata overhead is magnified by having a separate single metadata server. This server adds an extra hub in each request, becomes a single point of failure, and limits scalability beyond a point. Recent research has addressed this problem by either distributing the metadata [28] or caching it [20]. Although these systems alleviate accessing metadata, each small object still has a large metadata (usually stored on disk), decreasing the effective throughput of the system.

Distributed Data Stores: Many key-value stores, such as [2, 5, 8, 14], have been designed to handle a large number of requests per second. However, these systems cannot handle massively large objects (tens of MBs to GBs) efficiently, and add unnecessary overhead to provide consistency. Also, some systems [2, 8, 14] hash data to machines, creating large data movement whenever nodes are added/deleted.

PNUTS [6] and Spanner [7] are scalable geographically distributed systems, where PNUTS maintains load balance as well. However, both systems provide more features and stronger guarantees than needed in a simple immutable blob store.

Blob Stores: A similar concept to partitions in Ambry has been used in other systems. Haystack uses logical volumes [3], Twitter's blob store uses virtual buckets [27], and Petal file system introduces virtual disks [15]. Ambry is amenable to some optimizations in these systems such as the additional internal caching in Haystack. However, neither Haystack nor Twitter's blob store tackle the problem of load-imbalance. Additionally, Haystack uses synchronous writes across all replicas impacting efficiency in a geo-distributed setting.

Facebook has also designed f4 [18], a blob store using erasure coding to reduce replication factor of old data (that has become cold). Despite the novel ideas in this system, which potentially can be included in Ambry, our main focus is on both new and old data. Oracle's Database [19] and Windows Azure Storage (WAS) [4] also store mutable blobs, and WAS is even optimized for a geo-distributed environment. However, they both provide additional functionalities such as support for many data types other than blobs, strong consistency guarantees, and modification to blobs, that are not needed in our use case.

8. CONCLUSION

This paper described Ambry, a distributed storage system designed specifically for storing large immutable media objects, called blobs. We designed Ambry to serve requests in a geographically distributed environment of multiple datacenters while maintaining low latency and high throughput. Using a decentralized design, rebalancing mechanism, chunking, and logical blob grouping, we provide load balancing and horizontal scalability to meet the rapid growth at LinkedIn.

As part of future work we plan to adaptively change the replication factor of data based on the popularity, and use erasure coding mechanisms for cold data. We also plan to investigate using compression mechanisms and its costs and benefits. Additionally, we are working on improving the security of Ambry, especially for cross-datacenter traffic.

9. ACKNOWLEDGMENTS

We wish to thank the following people for their invaluable contributions towards the development and deployment of Ambry: our site reliability engineers, Tofig Suleymanov, Arjun Shenoy and Dmitry Nikiforov; our alumni Jay Wylie; our interns Tong Wei and Sarath Sreedharan; and our newest member Ming Xia for his valuable review comments.

10. REFERENCES

[1] Bonnie++. http://www.coker.com.au/bonnie++/, 2001 (accessed Mar, 2016).

[2] A. Auradkar, C. Botev, S. Das, D. De Maagd, A. Feinberg, P. Ganti, L. Gao, B. Ghosh, K. Gopalakrishna, et al. Data infrastructure at LinkedIn. In *Proceeding of the IEEE International Conference on Data Engineering (ICDE)*, 2012.

[3] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *Proceeding of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[4] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, et al. Windows Azure storage: A highly available cloud storage service with strong consistency. In *Proceeding of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2), 2008.

[6] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *In Proceeding of the Very Large Data Bases Endowment (VLDB)*, 1(2), 2008.

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, et al. Spanner: Google's globally-distributed database. In *Proceeding of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceeding of the ACM SIGOPS Operating Systems Review (OSR)*, 2007.

[9] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *Proceeding of the USENIX Annual Technical Conference (ATC)*, 1997.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceeding of the ACM SIGOPS Operating Systems Review (OSR)*, 2003.

[11] Hortonworks. Ozone: An object store in HDFS. http://hortonworks.com/blog/ozone-object-store-hdfs/, 2014 (accessed Mar, 2016).

[12] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceeding of the USENIX Annual Technical Conference (ATC)*, 2010.

[13] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceeding of the USENIX Networking Meets Databases Workshop (NetDB)*, 2011.

[14] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. In *Proceeding of the ACM SIGOPS Operating Systems Review (OSR)*, number 2, 2010.

[15] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceeding of the ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[16] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM (CACM)*, 29(3), 1986.

[17] S. J. Mullender and A. S. Tanenbaum. Immediate files. *Software: Practice and Experience*, 14(4), 1984.

[18] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, et al. F4: Facebook's warm blob storage system. In *Proceeding of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[19] Oracle. Database securefiles and large objects developer's guide. https://docs.oracle.com/database/121/ADLOB/toc.htm, 2011(accessed Mar, 2016).

[20] K. Ren, Q. Zheng, S. Patil, and G. Gibson. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In *Proceeding of the IEEE High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[21] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1), 1992.

[22] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Proceeding of the USENIX Summer Technical Conference*, 1985.

[23] M. Seltzer, K. Bostic, M. K. Mckusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proceeding of the USENIX Winter Technical Conference*, 1993.

[24] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceeding of the IEEE Mass Storage Systems and Technologies (MSST)*, 2010.

[25] D. Stancevic. Zero copy I: User-mode perspective. *Linux Journal*, 2003(105), 2003.

[26] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceeding of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2001.

[27] Twitter. Blobstore: Twitter's in-house photo storage system. https://blog.twitter.com/2012/blobstore-twitter-s-in-house-photo-storage-system, 2011 (accessed Mar, 2016).

[28] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceeding of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[29] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceeding of the IEEE High Performance Computing, Networking, Storage and Analysis (SC)*, 2006.

[30] Z. Zhang and K. Ghose. hFS: A hybrid file system prototype for improving small file and metadata performance. In *Proceeding of the ACM European Conference on Computer Systems (EuroSys)*, 2007.