View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Waterloo's Institutional Repository

Improving Distributed Filesystem Performance by Combining Replica and Network Path Selection

by

Xi Li

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2014

© Xi Li 2014

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Distributed filesystems are often the primary bandwidth consumers of large-scale datacenter networks. Unsurprisingly, the datacenter network is often the performance bottleneck for distributed filesystems. Yet even with this close relationship, current distributed filesystems and networks are designed independently and communicate over narrow interfaces that expose only their basic functionalities. Even network-aware distributed filesystems only make use of rudimentary network information, and are not reciprocally involved in making network decisions that affect filesystem performance.

In this thesis, we introduce Mayflower, a new distributed filesystem co-designed with the control plane of its underlying datacenter network. This design approach enables Mayflower to combine both filesystem and network information to make replica selection and dynamic flow scheduling decisions. By having more information and controlling both the filesystem and the network, Mayflower can perform optimizations that are unavailable to conventional distributed filesystems and network control planes. Our evaluation results using a real implementation show that Mayflower reduces average read completion time by more than 60% compared to HDFS with ECMP.

Acknowledgements

I would like to thank all the people who made this thesis possible.

Dedication

This thesis is dedicated to the one I love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Overview

Many data-intensive distributed applications rely heavily on a shared distributed filesystem to exchange data and state between nodes. As a result, distributed filesystems are often the primary bandwidth consumers for datacenter networks, thus the file placement and replica selection decisions can significantly affect the amount and location of network congestion. Similarly, with the rapid adoption of high-performance SSDs in the datacenter, it is becoming increasingly common for the datacenter network to be the performance bottleneck for large-scale distributed filesystems.

However, despite their close performance relationship, current distributed filesystems and network control planes are designed independently and communicate over narrow interfaces that expose only their basic functionalities. Network-aware distributed filesystems can therefore only use rudimentary network information in making their filesystem decisions, and are not reciprocally involved in making network decisions that affect filesystem performance. Consequently, they are only minimally effective at avoiding network bottlenecks.

An example of a network-aware distributed filesystem is HDFS [34], which is one of the most widely deployed distributed filesystems. It can make use of network topology information to perform static replica selection based on network distance. However, in a typical deployment where there are hundreds to thousands of storage servers across dozens of racks and a replication factor of just three [16], it is highly likely that a random client performing a read request will be equally distant from all of the replicas of its requested file. In this scenario, HDFS is just performing random replica selection. Moreover, even when the network distances of the replicas are not equal, the closest replica may be a poor choice due to (1) network congestion between the requester and the replica host, (2) high load of the replica host. As a result, network distance-based static replica selection is only partially effective at improving distributed filesystem performance.

Deploying a datacenter-wide dynamic network flow scheduler [2, 6] can reduce network congestion and improve distributed filesystem performance. However, flow schedulers are limited to finding the least congested path between the source and destination of each flow. They are unable to take advantage of the redundancies in the distributed filesystems, which makes them ineffective when all paths between the requester and the pre-selected replica are congested.

Figure 1.1 illustrates how HDFS and a dynamic flow scheduler solve the replica selection and path selection problem independently. A file is replicated to *three* different servers (marked red). The distributed filesystem picks one replica (marked green) based on a network distance scheme (e.g. the nearest replica host) or an end-host information scheme (e.g. the least loaded replica host). The location of the client and the chosen replica host is passed to the flow scheduler. The flow scheduler estimates the bandwidth of all paths (marked yellow) between the request client and the data server, and chooses the least congested path (marked green). However, a better replica selection choice may be a more distant replica host, if the path to the host has more remaining bandwidth and the host is less loaded.

Sinbad [8] is the first system to leverage replica placement flexibility in distributed filesystems to avoid congested links for its write operations. It monitors the end-host information such as bandwidth utilization of each server and uses this data together with network topology information to estimate the bottleneck link for each replica write request. Sinbad is a significant improvement over random or static replica placement strategies, but by working independently of the network control plane, it has a number of limitations. For example, in a cloud deployment, Sinbad is unable to monitor the bandwidth usage of other tenants. Additionally, by not accounting for the bandwidth of individual flows and the total number of flows on each link, Sinbad cannot accurately estimate path bandwidths, which can lead to poor replica placement decisions. Bandwidth estimation errors would be even more problematic if Sinbad was used for reads since, with a only a small number of replicas to choose from, selecting the second best replica instead of the best replica can significantly reduce read performance. This is not a flaw in Sinbad since it was not designed for read operations. Instead, it illustrates the need for an approach that considers network status when making replica select decisions for distributed filesystem.

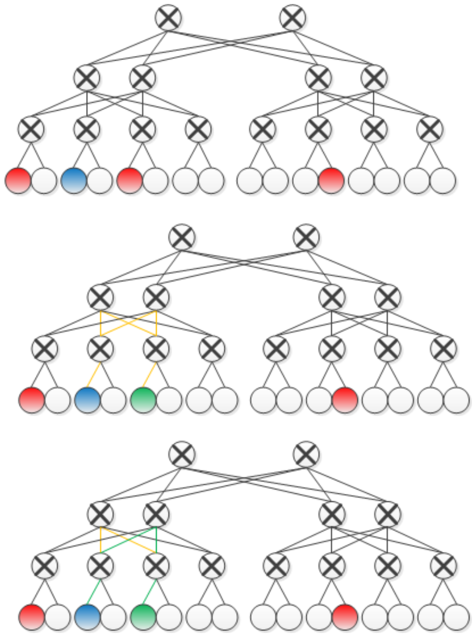

Figure 1.1: Replica Selection and Path Selection of DFS with a Flow Scheduler

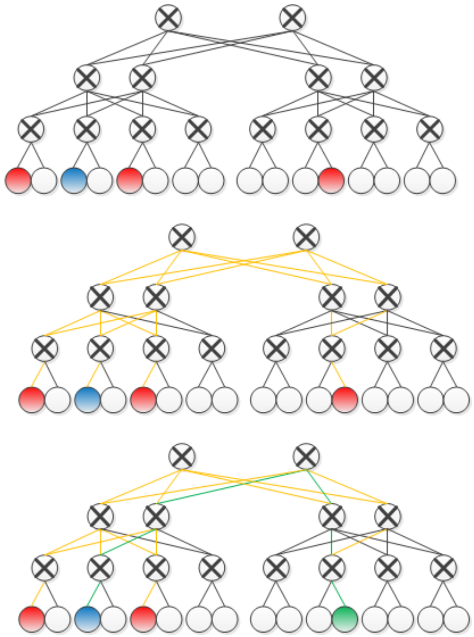

Figure 1.2: Replica Selection and Path Selection of Mayflower

1.2 Mayflower

This thesis introduces Mayflower, a high-performance distributed filesystem designed from the ground up to work together with a Software-Defined Networking (SDN) controller. The focus of Mayflower is in improving read performance given the majority of datacenter workloads are read-dominant. Mayflower consists of four main components: a NameServer that manages all filesystem metadata information, a FlowServer that monitors global network status and makes replica selection and path selection decisions, multiple DataServers that perform data storage, handle access and mutation to data, and a client library linked into user applications that provides a POSIX-like interface.

Mayflower models after the Google filesystem [16] (GFS) in its API and parts of its storage architecture. Both filesystems support random reads, sequential reads, and append operations. The main difference between Mayflower and GFS is its network control interface. Mayflower can be coupled with a centralized network flow scheduler, namely the FlowServer, via the network control interface. The FlowServer is an SDN controller application that monitors network status, such as per-port bandwidth utilization of each SDN switch and the bandwidth of current flows, and performs replica and network path selection based on global network information and replica location information from the NameServer. The FlowServer sets up the chosen path by sending out control messages to the SDN switches on the path. The network control plane interface enables filesystem and network decisions to be made based on information from both sides.

The FlowServer is an independent component in Mayflower. Users can replace or modify the FlowServer to use different objective functions and optimization algorithms. By default, the FlowServer minimizes average job completion time which accounts for both the expected completion time of each request and the expected increase in completion time of other in-flight requests. However, certain workloads may benefit from a scheduler that optimizes for maximum network utilization or ensures that time-sensitive requests finish within a specific deadline.

Figure 1.2 illustrates how Mayflower can combine replica selection with path selection. In this example, a file is replicated to *three* different servers (marked red). The client retrieves the file metadata from the NameServer which includes the location of the three replica hosts. The client passes the metadata to the FlowServer. The FlowServer determines `all` possible paths to `all` replica hosts and estimates the available bandwidth on each path. It then returns the replica host with the most available bandwidth to the client, and sets up the path between the client and the selected replica host (marked green). Upon receiving the reply from FlowServer, the client can directly establish a connection to the chosen DataServer. The SDN switches guarantee that the data flow follows the path that was determined by the FlowServer. This example illustrates how Mayflower is able to perform a wider search in the solution space compared to the conventional approach illustrated in Figure 1.1. By exploring a larger solution space, Mayflower can often find a solution that has a significantly lower job completion time. Moreover, Mayflower can also use flow bandwidth estimates to determine whether reading concurrently from multiple replica hosts can improve performance, and what fraction of the file should be read from each replica to maximize the performance gain. Furthermore, it can select the paths to the replicas as a group instead of one by one. This allows Mayflower to choose paths that individually have low bandwidth, but together provide higher aggregate bandwidth than other path combinations.

By combining the network status information, such as the current bottleneck links and locations of the network congestions, with the file popularity information, such as the counter of reads to one file and the locations of the clients, Mayflower can perform *chunk migration* to migrate chunks closer to the clients. This increases the probability of the clients to do local reads, and decrease the amount of cross-rack traffic, which again helps to improve Mayflower's performance.

Overall, this thesis makes three contributions:

- The design of a new high-performance distributed filesystem that provides an SDN control plane interface.
- A prototype `C++` implementation of Mayflower.
- Experimental results that show Mayflower reduces average job completion time by more than 20% compared to HDFS.

1.3 The Organization of the Thesis

Chapter 2 introduces the state-of-art in distributed filesystems and flow schedulers. Chapter 3 presents the design of Mayflower filesystem. Chapter 4 evaluates the performance of Mayflower. Chapter 5 concludes this thesis.

Chapter 2

Background and Related Work

2.1 Datacenter Network Architecture

Current datacenter networks usually consist of hierarchical topologies, commonly two or three tiers of switches connecting thousands of servers. The top tier, middle tier and bottom tier of a topology are named the core tier, the aggregation tier and the edge tier respectively. The topology usually contains multiple redundant core switches for fault tolerance. A cluster with less than 8K servers requires only a two-tier topology (only the core and the edge tier). Clusters with more than 8K servers generally require three-tier architectures.

An edge switch usually connects to a rack of servers, and an aggregation switch normally connects to multiple edge switches. Therefore switches in network core require significant switching capacity to aggregate and transfer packets between edges. However the cost of switches increases non-linearly with the switching capacity. Therefore it is too costly to build a full bisection bandwidth network because it requires a number of high-end switches with enormous switching capacity which come at immense expense. Further more, research on network traffic characteristics also reveals that the network is often under utilized [8, 4]. In most datacenters, only a subset of core layer links are saturated, while other links remain mostly empty. Therefore, due to cost and network traffic pattern reasons, current large datacenters are often built with an oversubscribed network. The network core is commonly oversubscribed with only a portion of the aggregate bandwidth available at the edge of the network. The term oversubscription is defined as the ratio of the worst case achievable aggregate bandwidth among the end hosts to the total bisection bandwidth of a particular communication topology [1]. An oversubscription value of 1:5 means that only 20% of the host bandwidth is available for some communication patterns. The oversubscription of current large datacenter network is usually at 1:10 or higher [8].

2.2 Network Traffic Characteristics

A datacenter runs a wide variety of applications, ranging from web services, file storage services to large-scale data-intensive applications. This makes the network flow patterns hard and complicated to predict. Recent studies show that datacenter network traffic is often uneven and bursty, and the network traffic through switches modeled an ON/OFF network traffic pattern with positive skew and heavy tails [4, 23, 5]. Dealing with congestion in datacenter networks is difficult because, despite the bursty nature of network traffic and the fact that links are underutilized most of the time, as the links become congested, they have a tendency to remain continuously congested for extended periods of time. Core tier links tend to have higher utilization rates than other links. However, despite network oversubscription, only a subset of the core links (less than 25% of the total number of core links) are fully utilized at any point in time.

Moreover, the skew in application communication patterns may cause substantial imbalance in the usage of bottleneck links. This might hurt the performance of distributed applications in the case of all-to-all communication pattern such as the shuffle phase of MapReduce [9]. To reduce congestion in oversubscribed network, there has been significant past work on building full bisection bandwidth networks, such as Fat-Tree [1] and VL2 [17]. Although these solutions are more cost-effective compared with the conventional approaches, they still incur significant amount of cost and increase the wiring complexity to a large extent. Furthermore, network-oblivious applications are often unable to take advantage of multiple paths, therefore they can hardly utilize the additional bandwidth.

An alternative approach to alleviate the network bottleneck problem is to change the user applications. For example, the user applications can allow most of the clients to access data locally or at least within the same rack by improving the data placement algorithm. Or the applications can be improved by multi-path routing algorithms to alleviate multipath to avoid hot spots or congested links. This approach normally requires dynamic network information collection and global coordination.

2.3 Distributed Filesystems

As distributed and big-memory computations become more commonplace, an urgent need for systems to store very large datasets arises. Distributed filesystems allow for the easy dissemination and access of files across nodes, enabling seamless integration with data-intensive distributed applications. Distributed filesystems are designed to run on thousands of commodity servers. At this scale, machine failures or network connectivity outages happen frequently, thus shouldn't be considered as exceptions. Therefore the distributed filesytems need to deal with common component failures while providing high availability. Multiple distributed filesystems have been designed to fulfill these needs, such as the Google File System (GFS) [16] and the Hadoop Distributed Filesystem (HDFS) [34].

2.3.1 Google File System

Design Assumptions

While sharing some common goals with the previous distributed filesystems in terms of performance, fault tolerance and scalability, the GFS design made several unique assumptions to meet the workload requirement at Google:

1. Files stored in GFS are normally log files or web documents that can grow to multiple GBs. Files are replicated several times (typically *three*) for reliability and fault tolerance.

2. Appending is the most common file mutate operation. Random writes are almost non-existent, and can be supported by re-creating a new version of the file at the application level.

3. Sequential file reads are the most common operation.

4. Files are divided into fixed-size chunks, and stored on dedicate storage servers (chunkservers). Each chunk is identified by a unique 64 bit chunk handle.

Architecture

GFS follows a single-master multi-chunkserver design. The master node manages all filesystem metadata and controls filesystem wide activities. A single master node for storing metadata greatly simplifies the GFS. The master node which has global filesystem information can make sophisticated decisions such as chunk placement. The involvement of the master node in GFS data flow is very limited. This is because that the clients cache the file metadata information, and can often connect with the chunkservers directly for data access or mutation. The master node is replicated to provide fault tolerance. When the current master dies, the shadow master node takes over the old master node, which ensures that GFS is still available to clients.

Consistency Model

The data being appended to chunks may result in three states: `consistent`, `defined` and `inconsistent`. `Consistent` means that data on different replica hosts remain byte-wise identical after one or several append operations. The data is `defined` only if a client can clearly identify its own mutation in the chunk. Multiple concurrent append operations may lead the data to be `consistent` but not `defined`, since no client can identify its own mutation. The resulting data consists of mingled fraction of data from multiple clients. If any of the append operations failed, the state of the data is `inconsistent`. In this case, the clients might see different data depending on which replica they are reading from.

2.3.2 Hadoop Distributed File System

Hadoop Distributed File System (HDFS) is the most widely used distributed filesystem in industry. Similar to GFS, HDFS stores metadata on a dedicated server – the NameNode. Application data are stored on other servers called DataNodes. The NameNode and the DataNodes communicate over TCP.

In HDFS, data are also duplicated for (1) reliability, (2) to increase the chance of localized data access, and (3) to improve system thought by multiply data transfer bandwidth if no local data access available.

NameNode

Directories and files in HDFS are represented by *inodes* on the NameNode. Inode record file metadata such as permissions, creation, modification and access times, namespace and disk quotas. Metadata are kept in RAM on the NameNode. A background thread keep storing the metadata as *images* on the local disk. The NameNode restores from the local *image* after normal shutdown or exceptional failure.

DataNode

DataNodes register to the NameNode before joining an HDFS cluster. Every DataNode gets a DataNode ID assigned by the NameNode during registration. The DataNodes and the NameNode communicates via *HeartBeat* messages. Operation commands issued by the NameNode are not sent separately but contained in the *HeartBeat* messages. Like GFS, files in HDFS are also split into *blocks* (like the *chunks* in GFS) on the DataNodes.

HDFS Client

Similar to most conventional filesystems, HDFS client supports `read`, `write`, `create` and `delete` operations to files. Additionally, HDFS client supports `create` and `delete` of directories. When reading a file, HDFS client first contacts the NameNode for the locations of data blocks comprising the file and then reads block contents from the DataNode *closest* to the client. If multiple DataNodes have same distance away from the client, an arbitrary one is selected. When writing data, the client requests the NameNode to nominate a suite of N DataNodes to host the block replicas where N is the replication factor. The client writes data to HDFS in a pipeline manner: the client send data to the first DataNode, then first DataNode relays data to the second DataNode. The process is repeated until the data gets propagated to the last DataNode. Once a block has been filled up to the configured *block size*, the client repeats the above process, contacting the NameNode for another suite of N DataNodes. The order for the pipeline might be different for the new set.

Consistency Model

Compared with GFS, HDFS takes a stronger consistency model – a single-writer, multiple-reader model. Before writing to a file in HDFS, an HDFS client has to attain a lease for the file from the NameNode. The writer client may extend the lease by periodically sending *HeartBeat* messages to the NameNode. Data region will always remain in the `defined` state since there exists a only one writer. However, HDFS does not guarantee the reader clients fetch exactly same data during another ongoing append, because appending data is still propagating in the pipeline manner. Once the append operation finishes or the writer client releases the file lease, HDFS broadcasts *flush* requests to the set of DataNodes, guaranteeing the data is populated on all. A writer client can also explicitly call *hflush()* to forcefully flush data to all replica hosts.

2.3.3 Other Related Work

Past work have proposed ways to improve the performance of HDFS, such as merging small files into large one and combining HDFS with a prefetch mechanism [11], and selecting replica hosts based on RTTs between the clients [28]. Other distributed storage systems have been proposed since GFS and HDFS [24, 10, 35, 36]. Dynamo [10] is a storage system in Amazon for storing and maintaining user shopping cart information. It supports offline read and write operations by providing a set of conflict resolution techniques. Cassandra [24] is decentralized structured storage system which focuses on scaling while providing seamless accessibility to Facebook users. Ceph [35] and PPFS [36] builds on object storage devices (OSD) to provide high-performance, scalable distributed filesystems.

2.4 End-Point Location Selection

Most distributed filesystems store multiple replicas on different servers to increase fault tolerance. Traditional replica selection algorithms [34] select the closest replica in order to reduce aggregation and core tier network traffic. Recent work [8] recognizes that there is significant flexibility in selecting a replica location and investigates algorithms for performing congestion-aware replica placement. The ability to choose from intelligently-located replicas is important, as closer replicas do not always translate into shorter flow completion times.

Although Google File System and Hadoop Distributed File System provide a reliable, scalable and high performance storage service, they are completely network-oblivious. This means that reads and writes in distributed applications may occur over saturated links or under other adverse conditions that can negatively affect the completion time of the computation. In contrast, Mayflower collects network stats at the FlowServer in order to select a replica and the least congested network path that makes full use of the available resources in the system, especially the limited available bandwidth in network core. By making better use of network resources, Mayflower is able to reduce flow completion time for requests and minimize the effect of new requests on existing flows.

2.5 Dynamic Flow Scheduling

In order to take advantage of path diversity in a datacenter network, protocols such as ECMP [19] use the hash of flow-related packet header information to determine which of the shortest paths to use for each flow. This approach works well for short flows, but may lead to persistent congestion on some links for elephant flows. Recent flow scheduling systems such as Hedera [2] and MicroTE [6] solve this problem by making centralized path selection decisions using global network information. Mayflower uses a custom multi-path scheduling algorithm, centrally controlled by the FlowServer, in order to direct flows to replicas that will minimize flow completion time. Mayflower's dynamic flow scheduling is additionally constrained by the desire to have a minimal impact on existing flows in the network.

Chapter 3

Design Overview

3.1 Assumptions

Mayflower's design assumptions are heavily influenced by the reported usage models of Google filesystem (GFS) and Hadoop File System (HDFS). Given GFS and HDFS's large combined user base, their usage models are representative of current data-intensive distributed applications. Therefore, Mayflower filesystem assumes the following workload properties:

- The system only stores a modest number of files (on the order of millions). File sizes typically range from hundreds of megabytes to tens of gigabytes. The metadata for the entire filesystem can be stored in memory on a single high-end server.

- Most reads are large and sequential, and clients often fetch entire files. This is representative of applications that partition work at the file granularity. In these applications, clients fetch and process one file at a time, and the file access pattern is often determine by the file content (e.g., graph processing where edges are embedded in the data). Large sequential reads are also common for applications that need to prefetch or scan large immutable data objects, such as sorted string tables (SSTs), or retrieve large media files in order to perform video processing or transcoding.

- File writes are primarily large sequential appends to files; random writes are incredibly rare. Applications primarily mutate data by either extending it through appends, or by creating new versions of it in the application layer and appending the new version to the file while retaining the previous versions.

- Files are often used as producer-consumer queues, or as a broadcast medium for multiple readers where each reader is applying a different transform to the same data. Since transforms are often applied sequentially in stages and can take multiple files generated by other stages as inputs, strong consistency may be necessary to ensure correctness for some applications.

- The workloads are heavily read-dominant. Read requests come from both local and remote clients.

- Replicas are placed with some constraints with respect to fault domains. For example, replicas should not be on the same rack and at least one of the replicas should be on a different pod, where we define a pod as the collection of servers that share the same aggregation switch in a 3-tier tree topology.

- The network is the bottleneck resource due to a combination of high performance SSDs, efficient in-memory caching, and oversubscription in datacenter networks.

3.2 Design Goals

Mayflower's primary design goal is to provide high-performance reads to large files by circumventing network hotspots. Additional design goals include offering application-tunable consistency in order to meet different application-specific correctness and performance requirements, and providing similar scalability, reliability, fault tolerance and availability properties to that of current widely-deployed distributed filesystems, namely, GFS and HDFS. These design goals are based on the workload assumptions from Section 3.1, and motivate Mayflower's system architecture.

3.3 Interface

Mayflower provides a similar file system interface as HDFS [34]. Files are organized hierarchically in directories and identified by full pathnames. Mayflower supports operations to *create*, *delete*, *read* and *append* files.

Create takes one string parameter to specify the name of the create file.

```
int Create(const string &file_name);
```

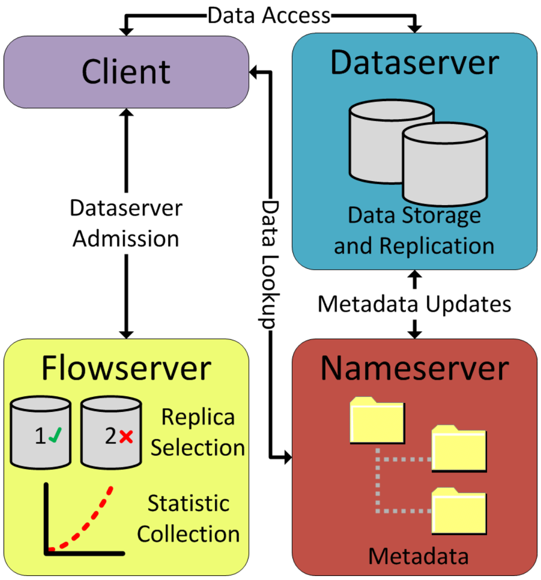

Figure 3.1: Mayflower Architecture

Read takes four parameters: a pointer to the read return value, a string to specify the read filename, two unsigned integer values for the offset and length of the read operation.

```
int Read(char *result, const string &filename, const uint offset, const uint length);
```

Append takes four parameters: a pointer to the pending append data, a string to specify the append filename, an unsigned integer value for the length of append data.

```
int Write(const char *data, const string &file_name, const uint length);
```

Delete takes one parameter to specify the name of the delete file.

```
int Delete(const string &file_name);
```

All Functions return 0 on success, -1 otherwise.

3.4 Architecture

A Mayflower cluster consists of (1) a centralized filesystem master node (NameServer), (2) a centralized network flow scheduler (FlowServer), (3) multiple data storage nodes

(DataServers), (4) a client library linked into user applications. Each of these components is a separate user-level process. Figure 3.1 illustrates the different components and their interactions with the client.

The NameServer maintains all of the filesystem's metadata information. It sends heartbeat messages to the DataServers periodically to collect stats or give instructions. A DataServer registers with the NameServer at start up to join a Mayflower cluster. The FlowServer is an SDN controller application. It monitors the current network status, models the network, and performs replica and network path selection based on global network information and replica location information from the NameServer. When a client issues a request, the FlowServer estimates the path bandwidth to each replica. Based on the current network status, it may pick one or more replicas and the corresponding path(s) to them. Then the FlowServer installs rules on the OpenFlow switches to assign specific routes for the data flows. However, the FlowServer is an optional component for Mayflower. A Mayflower filesystem cluster without the FlowServer behaves like HDFS in the way that requests always go to the nearest replica and the path between a client and a replica host is assigned by ECMP. The Mayflower client library provides a set of file system API to user applications. The client library communicates with the NameServer to retrieve file metadata information and the FlowServer to make replica selection and path selection queries. A client connects to a DataServer directly without the involvement of the NameServer or the FlowServer for data exchange.

Files in Mayflower is divided into fixed-size chunks. The *chunk size* is a filesystem level parameter, and is set to 64 MB by default. DataServers store data chunks as normal files in the local Linux filesystem. Files are associated with an immutable and globally unique 64 bit `UUID` generated by the NameServer at the time of *create*. A chunk can be identified with the `UUID` of the file it belongs to with the index of the chunk. For reliability and fault tolerance, each chunk (file) is replicated on multiple servers across different racks. The *replication factor* is a system parameter. All files have the same number of replicas. Mayflower's default replication factor is *three*. Users may tune the parameters to meet different kinds of application specific requirements. The locations of the replicas are determined by the replica placement strategy. Mayflower's default replica placement strategy takes a rack-aware approach to improve data reliability, availability, and network bandwidth utilization. This guarantees that data remains accessible to clients, even if one rack becomes unavailable. More details regarding replica placement algorithm are discussed in section 3.4.1. By monitoring the current network congestion via the FlowServer and the popularity of chunks via the NameServer, Mayflower can also perform chunk migration to migrate chunks closer to clients, to increase the probability of clients to do local reads and decrease the amount of cross-rack traffic.

In order to reduce reader/writer contention and strong consistency-related overhead, Mayflower filesystem does not support random writes. Instead, files can only be modified using atomic *append* operations. Random writes can be emulated in the application layer by creating and modifying a new copy of the file and using *delete* and *create* operations to overwrite the original file. Append-only semantics also simplify client-side caching of file to chunk mappings by ensuring that existing map entries cannot change unless the file is deleted. Clients can therefore safely cache these mappings to reduce the load on the NameServer. File to DataServer mappings can also be safely cached with cache expiry times that depend on the mean time between replica migration and node failure.

Replica placement decisions are made by the NameServer when a file is initially created. The NameServer takes into account system-wide fault-tolerance constraints, such as the replication factor and the number of fault domains, when determining replica locations. Currently, the NameServer makes replica placement decisions independently using only static information. The NameServer provides an interface to plug-in a more advanced replica placement algorithm, such as Sinbad [8] which takes network status quo into consideration.

3.4.1 NameServer

A centralized master node, the NameServer, manages all filesystem wide metadata, including the filesystem namespace, the mapping from file to chunk, the mapping from chunk to DataServer address and et al. The NameServer also serves the metadata queries by clients. To simplify chunk and replica management, replication is performed at the file level instead of the chunk level. Each file is replicated to a fixed number of DataServers, and each of these DataServers has an entire copy of the file consisting of one or more chunks.

The single master design largely simplifies Mayflower and enables the NameServer to make sophisticated chunk placement, replication decisions and migration decisions using global knowledge. To address single node performance bottleneck issue, data flows never go through the NameServer. A client may query the NameServer for file metadata, then caches the information for a limited duration before this piece of metadata times out. Clients interact with the DataServers directly for data exchange. This design limits NameServer's involvement in *read* and *write*, minimizes the possibility that the NameServer becomes a system performance bottleneck.

All metadata is stored in LevelDB [15], a persistent key-value database. The NameServer machine should have enough RAM to ensure that the LevelDB metadata and storage data is entirely in cache. A persistent database expedites the restart of the NameServer after a graceful shutdown. In the case of an unexpected restart, instead of reading from the possibly stale database, the NameServer rebuilds the database by scanning the file metadata stored at the DataServers. The current implementation of Mayflower filesystem contains only one NameServer. A usual way to provide a more reliable service like GFS and HDFS, the NameServer can be backed up with several secondary NameServers. The consistency between the NameServers can be guaranteed by state machine algorithms (e.g. Paxos [25]).

Namespace Management

Unlike traditional filesystems that has a per-directory data structure (e.g. Linux inode) which contains all directory related metadata such as list of file, file size and directory ACL, Mayflower filesystem stores its namespace as a lookup table, mapping file full pathname to its metadata. The metadata has three levels: *file*, *chunk* and *block*.

Every file corresponds to one *file* level metadata. A *file* level metadata structure contains the following fields: *filename*, *UUID*, a list of *chunk* level metadata and *size* which counts the length of the list. *Filename* stores the full pathname of the file. Full pathname begins with /, representing the root directory. Mayflower filesystem distinguishes no difference between a directory or a normal file. For example, while */foo* can be a normal log file, *foo* represents a directory in */foo/doo* . Mayflower only guarantees that full pathname is not duplicated in file namespace, thus */foo* and */foo/doo* can both exist in Mayflower filesystem. *UUID* is generated when a client *creates* the file in Mayflower. As a centralized service, it is easy for the NameServer to guarantee that every *UUID* is globally unique. The NameServer records existing *UUIDs* and reuses one UUID if a file is deleted from Mayflower filesystem. *Size* is the number of the chunks this file consists of. This *size* field works as a rough estimation of the file size, e.g. given a file *doo* has $five$ chunks and the chunk size is configured to 64 MB. The *doo*'s size can be estimated between 256 MB and 320 MB. This design reduces the involvement of the NameServer in the *append* process. The client doesn't need to notify the NameServer about how much data it has appended to a file. The DataServer doesn't need to notify about how much data it has received either. Instead, the DataServer only notifies NameServer when the current last chunk reaches the configured chunk size, and a new empty chunk is added for file. The details of append process is discussed in section 3.4.2.

A *chunk* level metadata has the following fields: *split* flag, *chunk location* and a list of *block* level metadata. *Split* flag is a boolean value indicating whether or not this chunk is erasure coded (EC). If EC is turned off in Mayflower filesystem, or a chunk is not yet ECed, the *split* flag stays `False` and the *chunk location* field is valid, storing the location

(IP address and port) of the DataServer where this chunk lives in. In the other case, the chunk is ECed, then the *split* flag is flipped to `True` and a list of *block* level metadata is added. The *chunk location* field is invalid in this case.

A `block` level metadata has only one field: *location*. *Location* stores the IP address and port number of the DataServer where the block lives in.

Replica Placement

The locations of the replicas are determined in the *create* process. When the NameServer receives a client's request to *create* a new file, it chooses N DataServers to store the replicas of the file (given replication factor is N). One of the N DataServers works as the primary replica host, and the rest of them work as backup replica hosts. The primary replica host coordinates the *append* request between all replica hosts 3.4.2. Mayflower provides an interface for users to plug-in other replication placement algorithms such as Sinbad [8]. Mayflower's built-in replica placement takes a similar strategy like HDFS [34]. Below is a comparison of HDFS's and Mayflower filesystem's replica placement scheme when the replication factor is *three*.

In HDFS, the replica placement strategy depends on whether it has the topology information of the cluster. If HDFS has the topology information and the client runs within the HDFS cluster, the first replica will be stored on the same physical machine that the client is running on. The second and third replicas are stored on two machines in a different rack. If HDFS is unaware of the topology or the client resides outside the HDFS cluster, HDFS arbitrarily chooses *three* servers for replicas. Mayflower's replica placement strategy is similar to HDFS. In Mayflower filesystem, the second replica is placed in a different pod as with the first (primary) replica where a pod is a collection of racks of servers that share an aggregation switch in a three-tier tree topology. The third replica resides within the same pod of the second replica, but in a different rack.

Namespace locking

Google File System[16] provides a very complex namespace locking scheme: Before a client modifies `/d1/d2/d3/.../dn/leaf`, it has to acquire the read lock for `/d1, /d1/d2, /d1/d2/d3, ... to /d1/d2/.../dn` and the write lock for `/d1/d2/d3/.../dn/leaf`. This leaf can either be a directory or a file depending on the operation. However, in Mayflower, file *read* or *append* operations do not require any locking on the NameServer side. The DataServers manages concurrent *append* requests and guarantees that every *read* request it receives can be served. On the NameServer side, locking is only necessary when clients mutate the namespace in *create* and *delete*. The design that clients cache the file metadata information for *read* and *append* minimizes the number of client requests that require communication with the NameServer. However, even when a client has to query the NameServer for file metadata information or mutate the namespace, the LevelDB can respond in micro-seconds, which introduces only marginal overhead given most requests in Mayflower are elephant flows that lasts for seconds. Table 3.1 lists the read & write benchmarks for LevelDB.

	Random	Sequential
Read	5.215 micros/op	0.476 micros/op
Write	2.460 micros/op	1.765 micros/op

Table 3.1: LevelDB Benchmark [15]

3.4.2 DataServer

The DataServer handles requests to access or mutate data chunks. It serves one *append* request at a time for each file. *Append* requests are serviced in FIFO order. Each chunk is replicated across N DataServers with one DataServer serving as the primary replica. The primary DataServer coordinates between all replica hosts, making sure those *append* requests are performed in a same order on all hosts. For each *append* request, the primary replica DataServer relays the appending data to the other replica hosts while concatenating the data to the end of the local copy. When a DataServer detects that a chunk is full, it stops appending data to that chunk, instead creates a new chunk file and append remaining data in the new one. The primary replica host informs the NameServer about the adding of a new chunk to a file. When serving *read* requests, a DataServer pulls the file metadata from the client, opens the requested chunk locally, and sends the data over socket connection to the client.

Lock Table & Atomic Appends

The DataServer implements a lock table to (1) parallelize requests for different files and (2) guarantee the order of requests to the same file. The lock table stores the mapping from a file `UUID` to a condition variable. An *append* request has to acquire the file lock before mutation. If more requests to the same file arrived before the first request finishes, they wait on the corresponding conditional variable of that file. All waiting requests get notified when the first request completes, and an arbitrary one may grab the lock. The order of the appends is not deterministic, but is guaranteed to be same on all replica hosts. Requests to different files acquire different locks, and they are not affected. A unique lock is designed for the lock table itself to guarantee that lock table access and mutation is thread-safe. The *read* requests are served in a lock free scheme, given data chunks in Mayflower filesystem is immutable once written.

3.4.3 FlowServer

The FlowServer monitors the per-port bandwidth utilization of each SDN switch, models the cluster network, and performs replica selection and network flow assignment. Bandwidth monitoring involves periodically fetching from the switches the byte counters for Mayflower-related flows and the bytes counters of each switch port. This allows the FlowServer to compute the bandwidth utilization of the flows, and determine the unused bandwidth of each link.

Using the measured bandwidth information as an instantaneous snapshot, the FlowServer knows the bandwidth of each Mayflower-related flow at that time point. In between measurements, the FlowServer tracks flow add and drop requests, and recomputes an estimation of the path bandwidth of each flow after each request. This ensures that completion time estimations are accurate, and also reduces frequency to pull stats from the switches.

The FlowServer performs replica selection and path assignment for every read request. The default algorithm of FlowServer is to greedily select the least congested path to one of the replicas while minimizing the impact of this request over the existing flows, then return this replica as the chosen replica to the client.

3.4.4 Client

Mayflower provides a client library that is complied into the user application. The client library provides an interface for four basic operations: *create*, *append*, *read* and *delete*. Details of each operation are described in the following sections.

Create

To create a file in Mayflower filesystem, the client issues a *create* request to the NameServer. The filename is passed along with the *create* request. The NameServer (1) generates an

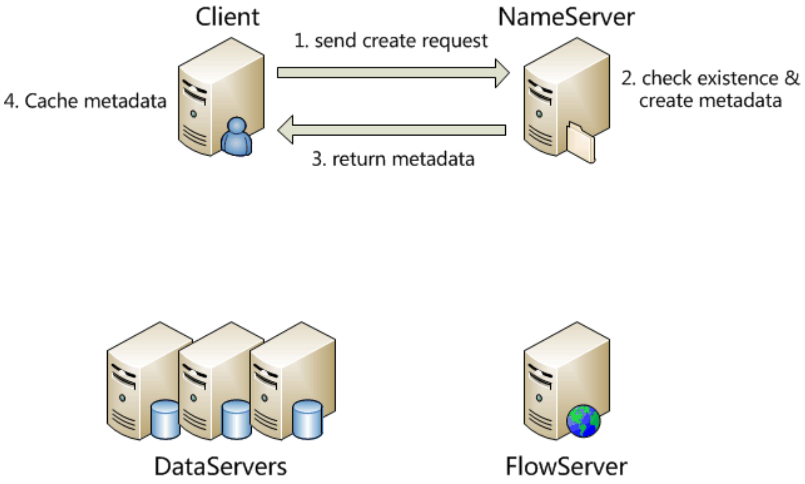

Figure 3.2: Create Operation Timeline

1. Client sends *create* request along with the filename to the NameServer
2. The NameServer chooses replica location and generates a *UUID* for the file
3. The client receives metadata from the NameServer, then save the metadata in cache

UUID for the file, (2) selects N DataServers to store the replicas, (3) returns the metadata, the *UUID* and *replica locations*, to the client. The client caches the metadata information to speed up future requests. Figure 3.2 illustrates the process of *create* request. The *create* process does not create any data chunks on the DataServers. The chunks are created when the DataServers receive the first *append* request to the file.

Append

A file may have N replicas living on N different DataServers. The DataServers' location information is stored as a list in the file metadata (discussed in section 3.4.1). The order of the list is determined by the replica placement algorithm. The first DataServer in the list is considered as the `primary DataServer` for this particular file.

Assuming the client already has the file metadata in its cache, the client establishes a socket connection to the primary DataServer, sending a *append* request along with the *UUID* of the file. After the *append* request arrives at the primary DataServer, it tries to acquire the lock for the specified *UUID*, and waits for an available worker to execute. Once

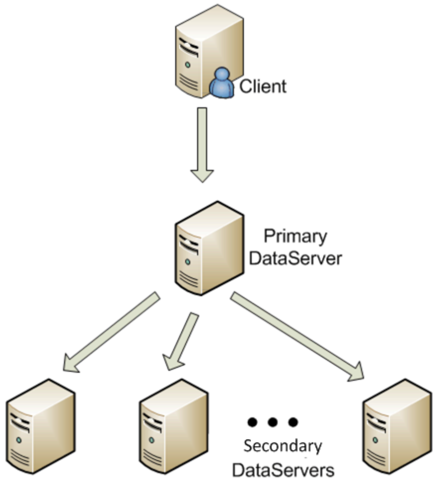

Figure 3.3: Data relay in append operation

a thread worker is assigned for the request, the primary DataServer (1) appends data to the current last chunk of the file on local disk, (2) relays data to the other replica DataServers via socket. The data relay process is shown in Figure 3.3.

To guarantee incoming data is appended at the end of the target file, namely at the end of the current last chunk of the file, a DataServer maintains a redirection table. The redirection table maps *UUID* to the index of the current last chunk of the file. After a DataServer receives an *append* request and extracts the *UUID*, it checks the redirection table with the key *UUID*, gets the index of the current last chunk, then appends data to that chunk files. The DataServer updates redirection table by increasing the index count by `one` when it detects that the size of the current last chunk has reached *chunk size*. *Chunk size* is a filesystem level configurable value, e.g. default chunk size is 256 MB in Microsoft Azure [7]. In Mayflower, the default value is 64 MB. After the DataServer updates the redirection table, it creates an empty chunk file to store the remaining data of the *append* request. Future incoming data is appended to the new chunk, until it is full again, then the same process to update redirection table and create new chunk file is repeated. If this DataServer is the primary DataServer for a file, it notifies the NameServer that a new chunk is created. Upon receiving the notification, the NameServer makes corresponding changes in the file's metadata.

If multiple clients are appending to the same file simultaneously, the DataServer lock table guarantees that the same order of *append* requests are processed on all replica hosts.

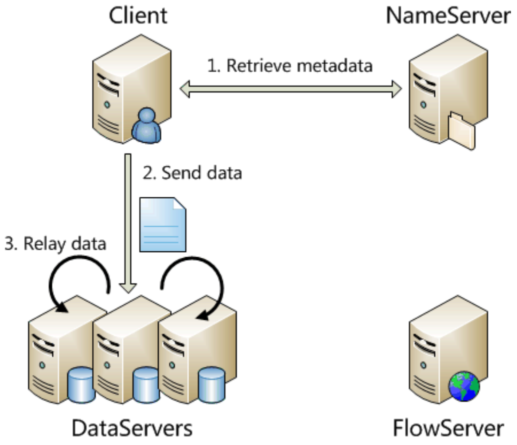

Figure 3.4: Append Operation Timeline

1. Client retrieves file metadata from the NameServer
2. Client sends data to the primary DataServer
3. The primary DataServer relays data to other backup DataServers

Only the request which acquired the lock on the primary DataServer is able to populate append requests to the backup replica DataServers, while others are blocked. Therefore replicas are guaranteed to be identical with an arbitrary *append* request arrival order. The *append* process is shown in Figure 3.4.

Read

Depending on the offset and length, a file read request might be split into several chunk read requests. For each chunk read request, the client queries the FlowServer to determine the replica it should read from. The FlowServer determines all the possible paths to all replica hosts and estimates the available bandwidth on each path. It then returns the replica host with the most available bandwidth to the client, and sets up the path between the client and the selected replica host. Upon receiving the reply from FlowServer, the client can directly establish a connection to the chosen DataServer. In some cases, the FlowServer may determine that the client can benefit from reading multiple replicas simultaneously, and the existing flows increase only little latency for this. The FlowServer selects multiple replicas for this request, and sets up path for each selected replica. After reading the

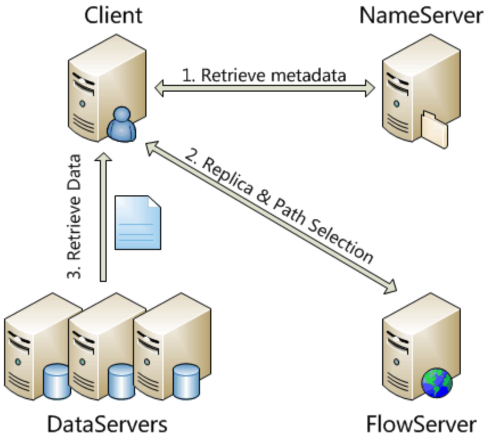

Figure 3.5: Read Operation Timeline

1. Client retrieves the metadata from the NameServer
2. Client sends requests and the metadata to FlowServer to install path in network
3. Client fetches data from DataServer(s)

data from the selected DataServer, the client informs the FlowServer when the request completes, so that the FlowServer can update its path bandwidth estimations. Figure 3.5 demonstrates the read process timeline.

Figure 3.6 demonstrates how a file read request is divided into two chunk read requests, and one of the chunk read request is further split into two concurrent chunk read requests. Assuming a file is replicated *three* times on DataServer *A, B and C*, and a client tries to read the file from offset 10 MB with length 64 MB. The *chunk size* is 64 MB in this example. The file read request covers the first and the second chunk of this file (index *zero* and *one*). Therefore the client creates two chunk read requests: one to read chunk *zero* from offset 10 MB to 64 MB, the other to read chunk *one* from offset 0 MB to 10 MB. The client queries the FlowServer for these two chunk read requests. The FlowServer selects DataServer *A* & *B* for chunk *zero* read request and DataServer *C* for chunk *one* read request. The client sets up socket connections with these three DataServers, reads chunk *zero* from 10 MB to 37 MB from DataServer *A* (27 MB), chunk *zero* from 37 MB to 64 MB from DataServer *B* (27 MB), and chunk *one* from 0 MB to 10 MB from DataServer *C* (10 MB), The client combines the three pieces of data, then returns to the user application (64 MB in total).

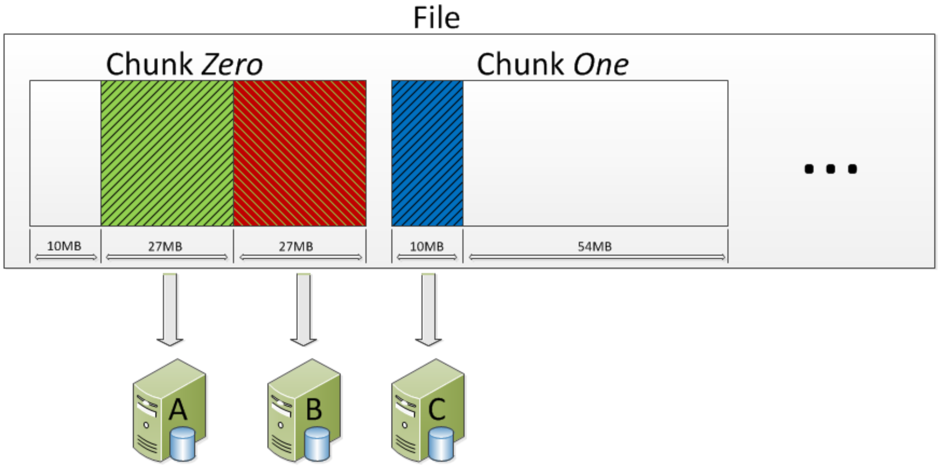

Figure 3.6: Concurrent Read from three DataServers

Delete

Since the clients cache file metadata information, a file chunk cannot be removed while there is still a valid cache entry to it. In Mayflower, the client sends a file *delete* request to the NameServer. The NameServer deletes the file's metadata immediately, ensuring that no other clients can retrieve the metadata of the deleted file past this point. Given the `TTL` of the file metadata cache on the client side, The NameServer then waits the `TTL` until the file metadata information has expired on all of the clients, before sending out an actual *delete* request to the DataServers to remove the file chunks. The `TTL` is user configurable parameter in Mayflower. Figure 3.7 illustrates the delete process timeline.

3.5 Consistency Model

Mayflower provides a user tunable consistency model between sequential consistency and strong consistency. In sequential consistency model, all clients see the same interleaving of operations. This requires that all *append* requests are sent and ordered by a file's primary replica host. Upon receiving an *append* request, the primary replica host relays the request to the other replica hosts while performing the append locally. Clients can however send read requests to any replica host and coordination between hosts is not required to service the read request.

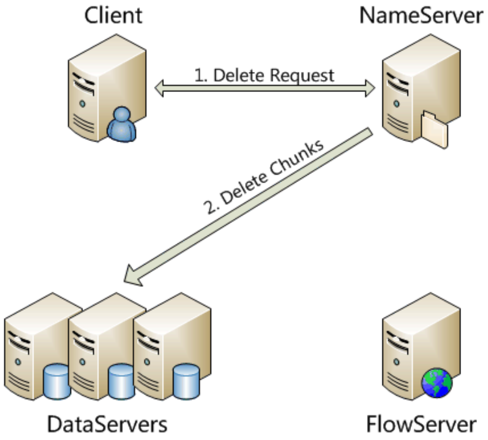

Figure 3.7: Delete Operation Timeline

1. Client sends *delete* request to the NameServer
2. The NameServer waits until all the metadata cache of the deleted file expired on client side
3. The NameServer sends the actual *delete* request to DataServers to remove data chunks

Alternatively, Mayflower can be configured to provide strong consistency (linearizability) with respect to *read* and *write* requests. The classic approach to ensuring strong consistency is to require all *read* requests to also be sent and ordered by the file's primary replica. However, Mayflower leverages append-only semantics to only require sending *read* requests to the last chunk to the primary replica host. All other chunk requests can be sent to the any of the replica hosts since these chunks are essentially immutable. Therefore, for large multi-gigabyte files, the vast majority of chunks can be serviced by any replica host while still maintaining strong consistency. The only limitation to this approach is that it cannot provide strong consistency when interleaved with *delete* requests; deleted files can briefly appear to be readable. However, we believe this is a reasonable consistency concession for improving read performance.

Chapter 4

Evaluation

4.1 Micro-benchmarks

In this section, we evaluate Mayflower's *read* and *append* performance using two micro-benchmarks.

4.1.1 Experiment Setup

We deployed Mayflower on a small cluster consisting of a single switch and nine machines. *Four* DataServers and *four* clients run separately on eight of the machines. The remaining machine runs the NameServer. All nine machines are connected to an `HP` ProCurve switch with 1 Gbps link. Because these micro-benchmarks are only used to exercise the filesystem, a FlowServer is not necessary for this experiment.

4.1.2 Read

We measured the performance of N clients reading simultaneously from Mayflower to determine the performance of Mayflower's *read* operation. Before the experiment, 100 files were preloaded into the filesystem. Each client reads a random 64 MB portion of an arbitrary file. This was repeated 10 times such that each client reads a total of 640 MB of data from multiple random files. We perform an initial warmup run before each experiment to ensure that all requests will be served from memory. Figure 4.1.2 shows the aggregate read throughput of the clients. The aggregate read throughput grows linearly with the

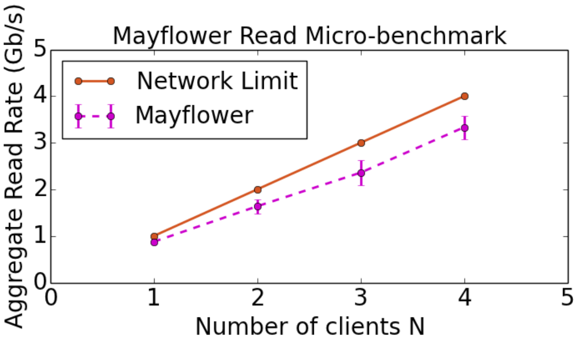

Figure 4.1: Mayflower Read Micro-benchmark

number of client and approaches the total line rate of the network cards. However, with additional clients, the per client read throughput falls relative to the total line rate due to client contention.

4.1.3 Append

To test Mayflower's *append* operation performance, we measure Mayflower's performance when N clients are appending simultaneously. Each client appends 64 MB of data to 10 different files sequentially in this experiment. Since each file is replicated to *three* servers, each client is writing a total of 1920 MB data. Figure 4.1.3 shows that the append throughput also grows linearily with the number of clients. Unlike read performance, which is bottlenecked by line rate of the network card, write performance is bottlenecked by the disk write throughput. With only one client, the append throughput is only approximately 45 MB/s. Additional clients do not provide a proportional increase in aggregate throughput because each append operation is replicated to three DataServers. Therefore, with only four DataServers, the replica set of two concurrent clients will always overlap with each other. This will therefore reduce the increase in aggregate write performance from each additional client. The aggregate write throughput of four clients is 137 MB/s or 34 MB/s per client, compared to 45 MB/s for a single client.

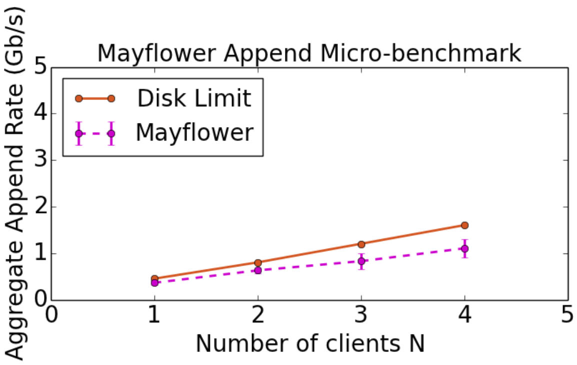

Figure 4.2: Mayflower Append Micro-benchmark

4.1.4 Micro-benchmark Conclusion

Due to the limitation of the number of machines available in lab, we only did the benchmark experiment using *four* clients at most. However, because every *read* request typically involves only one DataServer, we would assume the *read* requests can expand to very large scale as long as the number of DataServers grows linearly with the number of clients. In the *append* micro-benchmark, when there is *four* clients *appending* simultaneously, there exist 12 actual append flows in Mayflower, because the replication factor is three. That is every DataServer needs to handle *three append* requests. The Mayflower *append* operations remain close to the disk limits, even the DataServers are heavily loaded. Therefore, we believe it is reasonable to conclude that Mayflower I/O throughput can expand to large scale.

4.2 Simulated Network Result

To analyze the effectiveness of combining replica selection with path selection, we perform a comparison study between Mayflower and HDFS. Both filesystems were deployed on a simulated three-tier network using a synthetic workload.

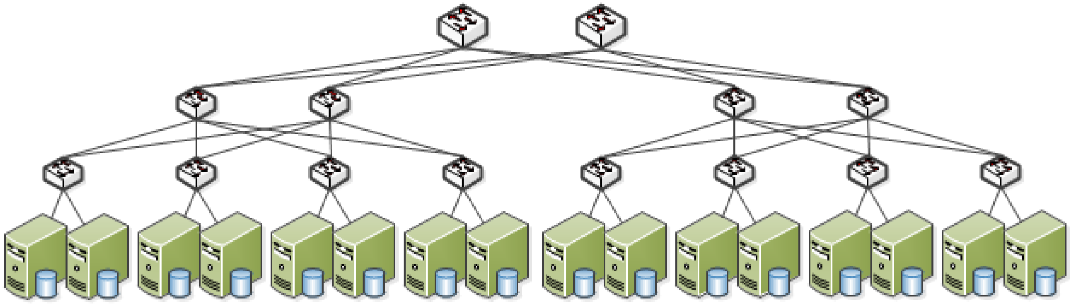

Figure 4.3: Mininet Simulated Network Topology

4.2.1 Testbed

The experiment was performed on a Mininet [26] simulated network. Mininet is an open source project that simulates large networks by creating Linux container that each has an isolated network namespace. Processes running in containers can communicate with each other via a virtual Ethernet pair (veth pair). A veth pair can be attached to a virtual switch such as OVS. Our simulated three-tier multi-root tree topology has *two* core switches and two pod, where a pod consists of *two* aggregation switches that are connected to $four$ edge switches. The edge switches serve as top-of-rack switches to a rack of servers. The oversubscription of this simulated network is 1:8. Figure 4.2.1 illustrates this topology with 16 hosts (2 hosts per rack).

4.2.2 Workload

The experimental workload is generated based on the following rules:

File Placement

5000 files are preloaded in both Mayflower and HDFS with the replication factor of *three*. Both filesystems use their default replica placement strategy.

Job Arrival Rate

The job arrival rate of a client follows a Poisson distribution. For example, if the experiment duration is T, and the job arrival rate of a client is λ, the total number of jobs for this client during the whole experiment should be $T * \lambda$. And if there exists N clients working simultaneously in the experiment with the same job arrival rate λ, and the job arrival rate is independent from each other, the overall job arrival rate of the whole system should be $N * \lambda$. The total number of jobs through out the whole experiment is $N * T * \lambda$.

Client Placement

Past work has shown that file popularity in a distributed filesystem follows a Zipf distribution [12]. Therefore, the target file for each job request is selected based on a Zipf popularity distribution. The client's location is determined by the staggered probability introduced in Hedera [2]. The stagger probability specifies that, relative to the primary replica, a reader client has a probability of $EdgeP$ to be in the same rack, a probability of $PodP$ to be in a different rack of the same pod , and a probability of $1 - EdgeP - PodP$ to be in a different pod.

4.2.3 Result

We perform our experiment on an Amazon EC2 instance with 32 vCPUs, 60 GB RAM and two 320 GB SSDs. In this experiment, we compare the average job completion time (JCT) of three systems using the same workload: (1) Mayflower filesystem with its default FlowServer, (2) Mayflower filesystem with nearest replica selection and ECMP path selection, and (3) HDFS. HDFS runs nearest replica selection and ECMP path selection by default. The experiment results are from 3 runs with a duration of 900 seconds for each run. The average JCT is calculated from the results when the system is in steady state. The stagger probability is set to (0.5, 0.3) in this experiment.

Figure 4.4 and Figure 4.5 show the impact of the job arrival rate λ on the mean JCT and the 95th percentile JCT respectively. The Mayflower bar and the Nearest ECMP bar show that the default FlowServer, which aims to minimize the JCT of the current request and minimize the impact of this request over the current flows, provides significant improvement over the nearest replica selection scheme coupled with ECMP path selection. The improvement of the Nearest ECMP bar over the HDFS bar demonstrates the better I/O throughput of Mayflower over HDFS, because they both perform nearest replica selection and ECMP path selection. The result also show that the trend that the improvement of Mayflower over HDFS increases as the job arrival rate increases. When the job arrive rate is 0.04, Mayflower with FlowServer reduces the mean JCT by 40% compared with HDFS; this value grows to 54% when the job arrival rate is 0.06. Moreover, the 95th percentile

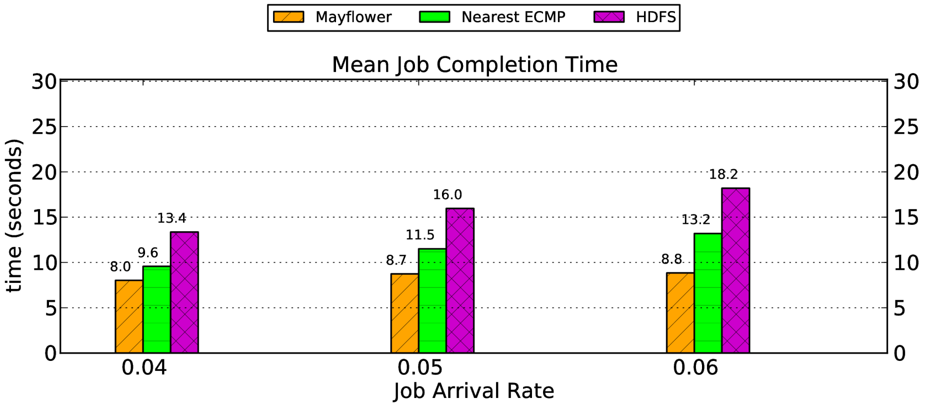

Figure 4.4: Mayflower vs HDFS Mean Job Completion Time

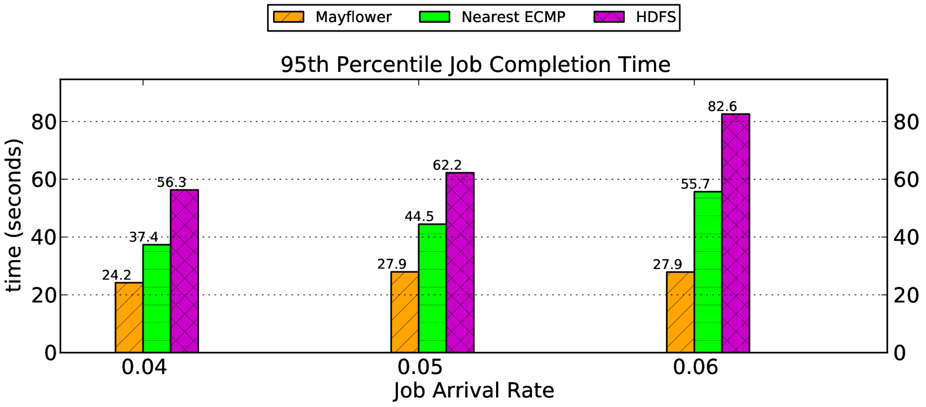

Figure 4.5: Mayflower vs HDFS 95th Job Completion Time

JCT results demonstrate that Mayflower is better at handling the straggler problem in distributed computations. In the worst case where the job arrival rate is 0.06, Mayflower reduces the 95th percentile JCT by 66% compared with HDFS. Our results show that (1) Mayflower which implemented in C++ provides a higher I/O throughput compared to HDFS, and (2) the FlowServer, which combines replica selection and path selection can significantly reduce the JCT compared with nearest replica selection and ECMP path selection.

Chapter 5

Conclusion

The prevalence of highly oversubscribed datacenter networks has led to the network being the primary bottleneck for high performance distributed filesystems. Past work has tried to tackle this problem by adding network-awareness to the filesystem. However, the filesystem does not have dynamic network information, and may therefore send read requests to replicas with highly congested network links, which in turn limits the performance of the filesystem. Mayflower addresses this problem by co-designing the filesystem with a software-defined network controller. This enables Mayflower to actively monitor the network status, and to perform replica and path selection based on the current network information.

In this thesis, we introduce the filesystem component of Mayflower. We describe its design assumptions based on requirements from large-scale distributed computation frameworks. Our design takes advantage of append-only filesystem semantics to both improve I/O performance and offer low-cost strong consistency. We also describe in detail its interface to a centralized replica and path selection service in order to support efficient network operations. We evaluate Mayflower using a fully functional prototype and found that it is highly efficient and performs significantly better than HDFS. We believe that, as datacenter networks continue to grow in scale, there will be even a greater need for filesystems that work together with software-defined network controllers in order to effectively utilize the network.

References

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[2] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 19–19, 2010.

[3] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, volume 10, page 24, 2010.

[4] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[5] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review*, 40(1):92–99, 2010.

[6] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh Conference on emerging Networking Experiments and Technologies*, page 8. ACM, 2011.

[7] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[8] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 231–242. ACM, 2013.

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[11] Bo Dong, Jie Qiu, Qinghua Zheng, Xiao Zhong, Jingwei Li, and Ying Li. A novel approach to improving the efficiency of storing and accessing small files on hadoop: a case study by powerpoint files. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 65–72. IEEE, 2010.

[12] John R Douceur and William J Bolosky. A large-scale study of file-system contents. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):59–70, 1999.

[13] Andrew Fikes. Storage architecture and challenges. Google Faculty Summit, 2010.

[14] Daniel Ford, François Labelle, Florentina I Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *OSDI*, pages 61–74, 2010.

[15] Sanjay Ghemawat and Jeffrey Dean. Leveldb @ONLINE, 2011.

[16] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[17] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 51–62. ACM, 2009.

[18] Brandon Heller, Srinivasan Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. Elastictree: Saving energy in data center networks. In *NSDI*, volume 10, pages 249–264, 2010.

[19] Christian E Hopps. Analysis of an equal-cost multi-path algorithm. *RFC 2992 Internet Engineering Task Force*, 2000.

[20] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, et al. Erasure coding in windows azure storage. In *USENIX Annual Technical Conference*, pages 15–26, 2012.

[21] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[22] Kuang-Ching Wang Jason Parraga. Project floodlight open sdn controller @ONLINE, 2014.

[23] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 202–208. ACM, 2009.

[24] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

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

[26] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 19:1–19:6. ACM, 2010.

[27] Paul J Leach, Michael Mealling, and Rich Salz. A universally unique identifier (uuid) urn namespace. 2005.

[28] Yuan Lin, Yang Chen, Guodong Wang, and Beixing Deng. Rigel: A scalable and lightweight replica selection service for replicated distributed file system. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 581–582. IEEE, 2010.

[29] Zaharia Matei, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, 2010.

[30] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[31] James S Plank, Scott Simmerman, and Catherine D Schuman. Jerasure: A library in c/c++ facilitating erasure coding for storage applications-version 1.2. *University of Tennessee, Tech. Rep. CS-08-627*, 23, 2008.

[32] Andrew Prunicki. Apache thrift.

[33] shinichi. Google-glog.

[34] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies*, pages 1–10. IEEE, May 2010.

[35] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[36] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *FAST*, volume 8, pages 1–17, 2008.

[37] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.