Mayflower: Improving Distributed Filesystem Performance Through SDN/Filesystem Co-Design

Sajjad Rizvi Xi Li Bernard Wong Xiao Cao Benjamin Cassell
School of Computer Science, University of Waterloo
{ sm3rizvi, x349li, bernard, x7cao, becassel}@uwaterloo.ca

Abstract

The increasing prevalence of oversubscribed networks and fast solid-state storage devices in the datacenter has made the network the new performance bottleneck for many distributed filesystems. As a result, distributed filesystems need to be network-aware in order to make more effective use of available network resources.

In this paper, we introduce Mayflower, a new distributed filesystem that is not only network-aware, it is co-designed from the ground up to work together with a network control plane. In addition to the standard distributed filesystem components, Mayflower has a flow monitor and manager running inside a software-defined networking controller. This tight coupling with the network controller enables Mayflower to make intelligent replica selection and flow scheduling decisions based on both filesystem and network information. It also enables Mayflower to perform global optimizations that are unavailable to conventional distributed filesystems and network control planes. Our evaluation results from both simulations and a prototype implementation show that Mayflower reduces average read completion time by more than 25% compared to current state-of-the-art distributed filesystems with an independent network flow scheduler, and more than 80% compared to HDFS with ECMP.

1. INTRODUCTION

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, and file placement and replica selection decisions can significantly affect the amount and location of network congestion. Similarly, with oversubscribed network architectures and 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 makes use of network topology information to perform static replica selection based on network distance. However, network distance does not capture dynamic resource contention or network congestion. Moreover, in a typical deployment with thousands of storage servers and a replication factor of just three [22], it is highly likely that a random client will be equally distant to all of the replica hosts. In this scenario, HDFS is just performing random replica selection.

Deploying a datacenter-wide dynamic network flow scheduler [6, 11] can reduce network congestion and improve distributed filesystem performance. However, because they are not involved in making the replica selection decision, flow schedulers are limited to finding the least congested path between the requester and the pre-selected replica. Therefore, they are unable to take advantage of redundancies in the distributed filesystem, which makes them ineffective when all paths between the requester and the pre-selected replica are congested.

Sinbad [12] is the first system to leverage replica placement flexibility in distributed filesystems to avoid congested links for their write operations. It monitors end-host information, such as the bandwidth utilization of each server, and uses this information together with the network topology to estimate the bottleneck link for each 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, by not accounting for the bandwidth of individual flows and the total number of flows in each link, Sinbad cannot accurately estimate path bandwidths, which can sometimes lead to poor replica placement decisions. Bandwidth estimation errors would be even more problematic if a Sinbad-like approach was used for read operations since, with only a small number of replicas to choose from, selecting the second best replica instead of the best replica can significantly reduce read performance.

In this paper, we introduce Mayflower, a novel distributed filesystem co-designed from ground up with a Software-Defined Networking (SDN) control plane. Mayflower consists of three main components: dataservers that perform reads from and appends to file chunks, a nameserver that manages the file to chunk mappings, and a Flowserver running within the SDN controller that monitors the bandwidth utilization at the network edge, models the path bandwidth of each Mayflower-related flow, and performs both replica selection and network flow assignment.

The focus of this work is in improving read performance since (1) the majority of datacenter workloads are read-dominant, (2) moving computation to the data is only feasible for a small set of data-intensive applications, and (3) current read replica selection strategies do not effectively utilize the network.

The main advantage of using an SDN/filesystem co-design approach is that it enables both filesystem and network decisions to be made collaboratively by Mayflower and the network control plane. For example, when performing a read operation, instead of first selecting a replica based on coarse grained network information and then choosing a network path connecting the client and the replica host, Mayflower can evaluate all the possible paths between the client and all of the replica hosts.

Furthermore, Mayflower's tight integration with the network control plane enables it to select a replica and path that greedily minimizes average request completion time. Unlike optimizing for the bandwidth metrics used in previous systems [12], minimizing average request completion time requires accounting for both the expected completion time of the pending request, and the expected increase in completion time of other in-flight requests. This is significantly difficult for a filesystem or flow scheduler to do independently, and we show in our evaluation that this is critically important for achieving good read performance. An additional benefit to this approach is that Mayflower can avoid unfortunate selection decisions that repeatedly increase the completion time of the same in-flight request, which can significantly reduce straggler-related problems.

Finally, Mayflower can also use flow bandwidth estimates to determine if reading concurrently from multiple replica hosts will improve performance, and what fraction of the file should be read from each replica to maximize the performance gain. This allows Mayflower to choose paths that individually have low bandwidth, but together provide higher aggregate bandwidth than other path combinations.

Overall, this paper makes three contributions:

- We present Mayflower, a new distributed filesystem that can make filesystem and network decisions collaboratively with a co-designed SDN control plane.
- We show that Mayflower can directly optimize for average request completion time and perform optimizations that are unavailable to conventional distributed filesystems and network control planes.
- We evaluate Mayflower's performance using both simulations and a real implementation running on Mininet [29]. Our results show that Mayflower reduces average read completion time by more than 50% compared to other distributed filesystem and network control plane designs.

2. BACKGROUND AND RELATED WORK

In this section, we outline past work on distributed filesystems, replica selection, and network path selection. We also describe the results of previous studies on datacenter network traffic patterns and explain how they relate to distributed filesystem design.

2.1 Distributed Filesystems

Big-data applications rely heavily on distributed filesystems for storing and retrieving large datasets. As a result, they have potentially high performance impact due to the underlying filesystem. Several high performance distributed filesystems have been developed including Google File System (GFS) [22], Hadoop Distributed File System (HDFS) [34], Quantacast File System [33], Colossus [21], and several others [36, 38, 20]. Mayflower's basic filesystem design aligns with these filesystems, yet it is significantly different towards its approach for performance optimizations. These filesystems can take advantage of network topology information to select a replica for read operations that minimizes the network distance to the client. However, Mayflower stresses the need for coordinated operations of both the filesystem and the network for their collective performance improvements. Mayflower's active coordination with the network manager allows it to select the replica and the network path that achieves higher throughput, and have least impact on the existing flows in the system.

2.2 Network Traffic Characteristics

A recent study using Facebook and Microsoft datacenters [12] reports that the network activity due to the filesystems ranges from 54% to 85% of the total network traffic. Distributed filesystems are therefore the primary source of network bandwidth utilization in many datacenters. A number of different network topologies [24, 23, 32, 25, 5], have been proposed to increase the bisection bandwidth in

the network. This has been shown to improve storage system performance when using flat storage model [31]. Nevertheless, oversubscribed multi-tier hierarchical topologies are still prevalent [12, 19] and are by far the most cited in network measurement studies [27, 10, 9]. Moreover, the increasing popularity of SSDs and in-memory applications suggest that the network will remain the primary bottleneck for many distributed datacenter applications.

A previous study [27] has found that network hotspots in a datacenter is typically not at the edge tier in oversubscribed networks. This suggests that there is an opportunity to improve read performance by reading multiple replicas in parallel if the paths to the replicas do not share a bottleneck link.

Placing data consumers close to the data can be an effective approach to reducing the network footprint of file read operations. A number of techniques have been proposed [39, 7] that try to increase opportunities to place the data and the client on the same machine (data locality) for reducing network traffic. However, data locality is not always achievable [11], and, in some cases, it has been shown to be irrelevant [8]. A previous study [12] has found that, even by taking advantage of data locality, 14% of Facebook traffic and 31% of Microsoft traffic are from remote distributed filesystem read operations.

2.3 End-Point Location Selection

Most distributed filesystems place replicas in multiple fault domains over the network for fault tolerance. Traditional replica selection algorithms [34] select the closest replica in order to reduce aggregation and core tier network traffic. Recent work [12] 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 faster flow completion times. Mayflower's Flowserver uses a more complex, network-aware metric to select replicas to read operations.

2.4 Network Traffic Engineering

In order to take advantage of path diversity in a datacenter network, protocols such as ECMP [26] select a path from the available shortest paths for each flow based only on flow-related packet header information. 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 [6] and MicroTE [11] solve this problem by making centralized path selection decisions using global network information. Mayflower uses a custom multipath scheduling algorithm, centrally controlled by the Flowserver, in order to direct flows to replicas that will minimize flow completion time.

There is a lengthy list of flow scheduling and network engineering techniques [13, 15, 16, 17, 14, 35] for network performance optimization. Unlike Mayflower, these techniques cannot take advantage of the availability of multiple replica choices.

3. DESIGN OVERVIEW

In this section, we first describe our assumptions regarding the typical usage model for Mayflower and detail the properties of Mayflower's target workload. We then outline Mayflower's main design goals, and provide the details of its system architecture.

3.1 Assumptions

Our design assumptions are heavily influenced by the reported usage models of Google filesystem (GFS) and HDFS. We believe that, given their large combined user base, their usage models are representative of current data-intensive distributed applications. Mayflower 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 files one at a time, and the file access pattern is often determined by the file contents (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 less frequent. Applications primarily mutate data by either extending it through appends, or by creating new versions of it in the application layer while retaining the previous versions.
- 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 network.
- The network is the bottleneck resource due to a combination of high performance SSDs, efficient in-memory caching, and oversubscription in datacenter networks.

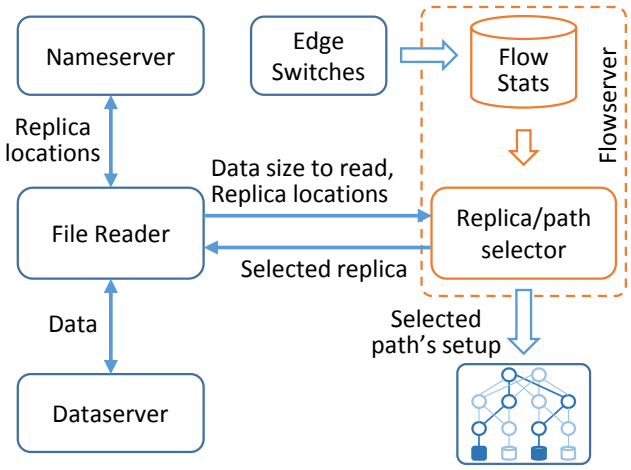


Figure 1: Mayflower system components and their interaction in a read-file operation.

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.

3.3 Architecture

Mayflower's basic system architecture consists of three main components: dataserver, nameserver, and Flowserver. The dataserver is responsible for storing and reading the actual data, the nameserver manages the filesystem's namespace, and the Flowserver, which runs within the SDN controller, monitors the network, and performs replica and path selection. Figure 1 illustrates the different components and their interactions with the client. In this example, the file reader contacts the nameserver to determine the replica locations of its requested file and then contacts the Flowserver to determine which replicas to read from. In addition to selecting the replicas, the Flowserver will also install the flow path for this request in the OpenFlow switches. Finally, the client will contact the dataservers to retrieve the file.

Each file in Mayflower is partitioned into large chunks, where the chunk size is a system parameter that is typically equal or larger than 128 MB. The nameserver is responsible for storing file to chunks and file to dataservers mappings. 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.

Mayflower provides file-specific tunable consistency to its clients. In order to reduce reader/writer contention and strong consistency-related overhead, Mayflower 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 a `move` operation to overwrite the original file.

Append-only semantics also simplify client-side caching of file to chunks 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. A client can discover new entries to the map, created through appends by other clients, when it attemps to read from the file since the dataserver includes the file's size with each read result it returns. File to dataservers 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. This is because the focus of this work is on improving read performance for read-dominant workloads. We expect that it would be relatively straightforward to implement a Sinbad-like replica placement strategy by having the nameserver make the placement decision collaboratively with the Flowserver.

3.3.1 Nameserver

The nameserver stores and manages file-to-chunks and file-to-dataservers mappings. Clients contact the nameserver when they need to `create` or `delete` a file, or lookup the size of a file. The mappings are stored using LevelDB [2], a persistent key-value database. By default, LevelDB is configured with `fsync` off in order to speed up file creation and deletion, and the nameserver should be configured with enough memory to ensure that LevelDB serves requests entirely from memory. The purpose of using a persistent database to store the mappings is to speed up nameserver restarts after a graceful shutdown. After an unexpected restart, instead of reading from the possibly stale database, the nameserver rebuilds the mappings by scanning the file metadata stored at the dataservers.

The nameserver is currently implemented as a centralized service. We can improve the fault-tolerance of the nameserver by using a state machine replication algorithm, such as Paxos [28], to replicate the nameserver to multiple nodes.

3.3.2 Dataserver

The dataserver is responsible for storing and retrieving file chunks. Each file is represented as a directory in the dataserver's local filesystem, and the name of the directory

is the file's UUID [4]. Each directory has a file that includes name and other metadata information. Chunks are stored as numbered files inside the directory (e.g., the first and second chunks have filenames of *1* and *2* respectively). In order to support atomic `append` requests, the dataserver only services one `append` request at a time for each file. Because of Mayflower's append-only semantics, the dataserver can concurrently service `read` requests with an `append` request as long as the `read` requests are not requesting the last chunk.

Each file has a primary dataserver, which is responsible for ordering all of the `append` requests for the file. The primary dataserver relays `append` requests to the other replica hosts while servicing the request locally.

3.3.3 Flowserver

The Flowserver is responsible for monitoring the per port bandwidth utilization of each edge switch, modeling the path bandwidth of each Mayflower-related flow based on the measured link bandwidth at the edge and the number of flows sharing each link, and performing replica selection and network flow assignment. Bandwidth monitoring involves periodically fetching from the edge switches the byte counters for both Mayflower-related flows and each switch port. This allows the Flowserver to compute the bandwidth utilization of the flows, and determine the unused bandwidth of each edge link.

The measured bandwidth information is used as an instantaneous snapshot of the network state. In between measurements, the Flowserver tracks flow add and drop requests, and recomputes an estimate of the path bandwidth of each flow after each request. This ensures that completion time estimates are accurate, and also reduces the need to poll the switches at very short intervals.

3.4 Consistency

By default, Mayflower provides sequential consistency for each file where 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.

Alternatively, Mayflower can configured to provide strong consistency with respect to `read` and `write` requests. The traditional approach to ensuring strong consistency is to require all `read` requests are also ordered by the file's primary replica. However, Mayflower leverages its append-only semantics to only require sending last chunk `read` requests to the primary replica host. All other chunk requests can be sent to the any of the replica hosts since every chunk except the last one 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 `read` and `append` requests are interleaved with `delete` requests; deleted files in Mayflower can briefly appear to be readable due to client-side caching. However, we believe that this is a reasonable consistency concession for improving read performance.

4. REPLICA AND PATH SELECTION

When a Flowserver receives a replica selection request from a client, it executes our replica–path selection algorithm (§ 4.2) to select a replica host and network path for the request. In order to perform the replica and path selection, the Flowserver keeps track of the bandwidth share estimates of the existing flows belonging to the filesystem's read jobs, and network path assignments for each flow.

To estimate bandwidth utilization of flows and the remaining amount of data that still needs to be transferred for existing flows, the Flowserver periodically fetches flow stats from the edge switches. The flow stats are collected for only those flows that originate from dataservers attached to the edge switch being queried.

Target performance metric: Our target performance metric is average and 95th percentile *job completion time*. If we assume the network is the bottleneck for a read operation, the job completion time for the read request can be minimized by maximizing the max-min bandwidth share of the job's flows. Unlike other flow scheduling systems [6, 12] which are based on link utilization measurements, Mayflower maximizes max-min bandwidth share because, in addition to link capacity, it also captures the number of existing flows in each link and the bandwidth share of each flow. In contrast, absolute link utilization only provides information on available link capacity.

Even though the path with the most bandwidth share is a good choice, it is not always the best choice in highly dynamic settings. This is because new flows affect the path selection for already scheduled flows. We must therefore account for the impact on existing flows when making a scheduling decision for a new request.

4.1 Problem Statement

Our optimization goal is to select the network path among the paths from all replicas to the client that minimizes the completion time of both the new flow as well as existing flows. The replica and path selection algorithm has at its disposal the path of existing flows, capacity of each link, data size of each request, current bandwidth share of existing flows, and remaining size of existing flows.

4.2 Replica–Path Selection Process

Mayflower's replica–path selection algorithm evaluates all the paths from each replica to the client and selects the path which has minimum cost:

Pseudocode 1 Replica and Path Selection

```
1: procedure SELECTREPLICAANDPATH(Request R)
2:     costs ← []
3:     P ← paths from R.replicas
4:     for each path p ∈ P do
5:         est_bw ← MAXMINSHARE(p.links)
6:         costs[p]← FLOWCOST(p, est_bw, R.size)
7:     end for
8:     path, replica ← MINCOST(P, costs)
9:     for each flow ∈ path do ▷ Only those flows whose BW is changed
10:        SETBW(flow, flow.new_bw)
11:    end for
12:    return path, replica
13: end procedure
```

Pseudocode 2 Cost of the new flow for the given path

```
1: procedure FLOWCOST(Path p, est_bw, flow_size)
2:     cost ← flow_size / est_bw
3:     flows ← { flows assigned to p.links}
4:     for each f ∈ flows do
5:         new_bw ← NEWBANDWIDTH(f, p, est_bw)
6:         cur_bw ← CURRENTBANDWIDTH(f)
7:         r ← remaining size of f                    ▷ From flow stats
8:         cost = cost + [r / new_bw] − [r / cur_bw]
9:     end for
10:    return cost
11: end procedure
12: procedure UPDATEBW(Flow f, bw)
13:                                          ▷ Called by flow stats collector
14:    if f.freezed = False or T > f.T then
15:        f.bw ← bw
16:        f.freezed ← False
17:    end if
18: end procedure
19: procedure SETBW(Flow f, bw)
20:    f.bw ← bw
21:    f.T ← T + (f.remaining / bw)
22:    f.freezed ← True
23: end procedure
```

$$\text{Path}_{\min}(P) = \underset{\forall p \in P}{\operatorname{argmin}} \text{Cost}(p) \tag{1}$$

where P is the set of all distinct paths between the data reader and the replica sources. We restrict to selecting from only the shortest paths between two endpoints. This limits the network path lengths to be 2, 4 or 6 in a traditional three-tier tree network.

The cost of each path $p \in P$ is the completion time of the new read job j, and the increase in completion time of the existing jobs in each link along that path:

$$\text{Cost}(p) = \frac{d_j}{b_j} + \sum_{\forall f \in F_p} \left[\frac{r_f}{b'_f} - \frac{r_f}{b_f} \right] \tag{2}$$

where d_j is the requested data size and b_j is the estimated bandwidth share of a new flow on path p. The first half of the equation is estimating the cost of the new flow while the second half is estimating the impact of the new flow on existing flows F_p in path p. The cost of an existing flow $f \in F_p$ is the estimated increase in completion time to download its remaining data r_f when the current bandwidth b_f is decreased to b'_f due to the addition of the new flow in the path. The current bandwidth share and remaining sizes of the existing flows are measured through flow stats collected from the edge switches.

The bandwidth share of the new flow, and the change in bandwidth share of the existing flows are estimated through max-min fair share calculations. For each link, given a set of flows with their bandwidth demands that use the link and the link's capacity, we equally divide the bandwidth across each flow up to the flow's demand while remaining within the link's capacity. The demand for the existing flows is set to their current bandwidth share whereas the demand of the new flow is set to infinity. The estimated bandwidth b_j of the new flow is its bandwidth in the bottleneck link in the path. However, the new bandwidth estimate of the existing flows is their bandwidth share when a new flow with bandwidth demand b_j is added in the links in the path.

Slack in updating bandwidth utilization: When a path is selected by the Flowserver using our replica–path selection algorithm, the bandwidth utilization for the new flow is set to its estimated bandwidth share. Moreover, the bandwidth share of the existing flows whose estimated share is changed in the selected path are updated with their estimated values and these flows are then placed in an *update-freeze state*. The flows remain in this state for the time proportional to their expected completion time (lines 12 to 18 in Pseudocode 2).

The flows are kept in the update-freeze state because a flow's recently updated bandwidth state can be overwritten too soon in the next flow stats collection cycle. This will invalidate the previous estimates and lead to incorrect calculations for the forthcoming flows. When a flow-stats update is received by the Flowserver, the values are not updated for a flow if it is in the update-freeze state. The values are updated when the time period of the freeze state expires (lines 19 to 23 in Pseudocode 2).

Simplifying bandwidth estimations: The reduction in bandwidth share of flows in a path may result in the increase in bandwidth share of flows in other paths. This can in turn have additional secondary and tertiary impact on nearly all the flows in the system. To accurately measure the impact of adding a new flow on a path, we need to not only update the state of the flows on the selected path but also identify and update the changes in the bandwidth utilization of flows in other paths. This greatly increases the cost of bandwidth estimation.

Therefore, for simplicity, we ignore the secondary and tertiary effects and only estimate and update the bandwidth share of flows in the paths between replicas and the client. Estimation errors do not accumulate because we periodically update our bandwidth estimates from the switches' flow stats. We also use the flow stat information to update

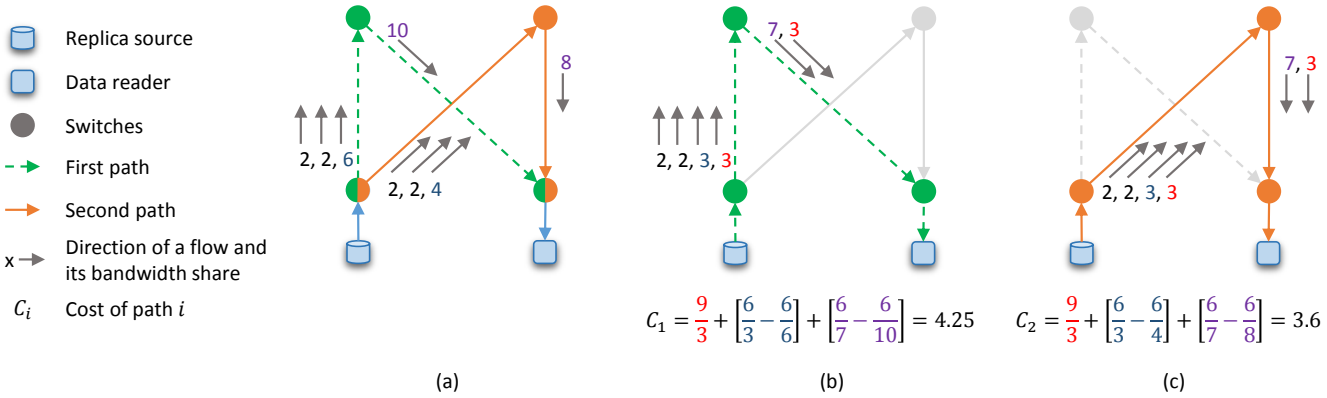


Figure 2: An example of cost calculation for replica–path selection: a) Existing flows' bandwidth share along two paths (10Mbps links). b) Bandwidth share of the flows if a new flow is added on the first path where C_1 is the cost of adding the new flow on the path. New flow's size is 9Mb whereas the remaining size of the existing flows is 6Mb. c) Bandwidth share of the flows if the new flow is added on the second path and its corresponding cost C_2.

the bandwidth utilization of the remaining flows in the network. In this way, we significantly reduce the complexity of the problem while still having good bandwidth utilization approximations.

An illustrative example: Consider an example of a client reading 9Mb data from a replica source, illustrated in Figure 2. The figure shows the arrangement of two edge switches connected with two aggregate switches. All the links have 10Mbps bandwidth capacity. There are two equal length paths between the reader and the data source. Both paths have three flows on the second link, which connects the edge to the aggregate, and one flow on the third link, which connects the aggregate to the edge.

To evaluate the first path (as shown in Figure 2b), first the max-min fair share of the new flow is calculated on that path. The second link is the bottleneck link because it gives 3Mbps share to the new flow as compared to 5Mbps share on the third link. Therefore, the new flow will have 3Mbps share on the first path and the read job will take $9/3 = 3$ seconds to complete. Adding this new flow may result in a reduction of the bandwidth share of some of the existing flows and will increase their completion times. According to the max-min fair share calculations, the existing flow with 6Mbps share in the second link will be reduced to 3Mbps and the 10Mbps-flow on the third link will be reduced to 7Mbps. As a result, the completion time of the flows will be increased by $[(6/3) - (6/6)] = 1$ and $[(6/7) - (6/10)] = 0.25$ seconds respectively, assuming the flows' remaining size to be 6Mb. Therefore, the estimated cost of the first path, which is a measure of the increase in total completion time, is $3 + 1 + 0.25 = 4.25$. Similarly, the cost of the second path turns out to be 3.6 and therefore the second path will be selected for the read operation.

In this example, the bandwidth share of the new flow is the same for both paths. The difference in cost is due to the impact of the new flow on the existing flows. The second path has an increase of 0.6 seconds in the job completion time of the existing flows, compared to 1.4 seconds for the first path. In a more realistic network setting, there are additional factors affecting the bandwidth share and completion time of flows, such as oversubscription and heterogeneous link capacities. For instance, if we assume that the second link in the first path has 20Mbps capacity, then the cost of the first path will become 2.4 seconds and thus the first path will be selected.

4.3 Reading from Multiple Replicas

Mayflower reads from multiple replicas in parallel if doing so results in a reduction of the completion time. In our replica–path selection algorithm, a read job is only split into multiple sub-jobs if the combined estimated bandwidth share of the subflows is greater than the bandwidth share of the flow to a single replica. After selecting the network paths for the subflows and estimating their bandwidth shares, the data read size for each flow is divided such that the subflows finish at the same time. The subflows are assigned different replicas to avoid the same network bottlenecks.

Multiple replicas selection process: While estimating the bandwidth share and path for two subflows, for the first subflow f_1, select a replica–path p_1 using the replica–path selection algorithm (§ 4.2). Assume that the estimated bandwidth share of f_1 is b_1. Add a temporary flow in path p_1 and temporarily update the bandwidth shares of existing flows in the path. For the second subflow f_2, select another replica–path p_2 using the same replica–path selection algorithm. Assume that the estimated bandwidth share to be b_2. The second subflow may reduce the bandwidth share of f_1 and thus b_1 is adjusted to b_1' accordingly. If the combined bandwidth share $b = b_1' + b_2$ exceeds b_1, the two replica-paths p_1 and p_2 are selected for the subflows. Otherwise, the temporary changes

are rolled back and only p_1 with estimated bandwidth b_1 is selected. If the subflows are selected, the flow size S_i for each subflow i is adjusted proportional to their bandwidth share: $S_i = d * b_i/b$, where d is the size of the requested data. The Flowserver returns the replica–path and the data size associated with each of them to the client.

Our results show that the completion time of read jobs is further reduced up to 10% on average. Moreover, the average difference of finish time between the two subflows of a read job is less than a second when reading a 256 MB block.

5. IMPLEMENTATION

We implemented Mayflower in C++, with the exception of the Flowserver which is built on top of the Java-based Floodlight [18] controller. Our Mayflower prototype consists of 7500 lines of C++ code and 3700 lines of Java code. We used Apache Thrift [1] RPC framework for control messages between the servers and the clients. Files are transferred using the Linux `Sendfile` function. `Sendfile` is a zero-copy mechanism which reduces I/O latency and complexity by transferring data directly from kernel buffers to the NIC, avoiding copying to userspace.

The Flowserver is implemented as a Floodlight controller application. The replica-path function is exposed as an RPC service. The Flowserver implementation is not tied to Mayflower, and can be integrated with any distributed application through its RPC framework. The RPC call to the Flowserver accepts a list of source/destination IP addresses, port numbers, and the size of the data to be transferred. The RPC call returns a list of replicas and the corresponding data size to be downloaded from those replicas.

The Nameserver stores the filesystem information in a LevelDB [2] key-value store. LevelDB provides data persistence and fast lookup through in-memory data caching. File metadata consists of filenames and block information, occupying at least 67 bytes per file. The block size and replication factor are system-wide configurable parameters with default values of 256 MB blocks and 3 replicas, respectively. The default replica placement strategy follows an HDFS rack-aware placement approach: Two replicas are placed in the same rack and any further replicas are placed in other randomly selected racks.

The Mayflower client library provides an interface similar to HDFS. It supports create, read, write and delete functions. The client caches file metadata to reduce load on the Nameserver. During read operations, clients query the Flowserver to select a replica to read from.

6. EVALUATION

We evaluate the effectiveness of Mayflower's replica-path selection using micro-benchmarks that compare it with several other replica-path selection schemes. We also compare the performance of our prototype implementation of Mayflower with HDFS, which is widely used in commercial datacenters.

6.1 Experimental Setup

We conducted our experiments by emulating a 3-tier datacenter network topology with 8:1 oversubscription using Mininet [29]. Mininet leverages Linux virtualization techniques to emulate hosts, and uses software switches, such as Open vSwitch [3], to emulate the OpenFlow switches. As emulating a complete datacenter network in a single machine imposes network size and bandwidth limitations, we partitioned our virtual network into several slices, and distributed these slices across a cluster of 13 machines. This allowed us to increase edge link capacity up to 1 Gbps.

Each machine in our cluster consists of a 64 GB RAM, a 200 GB Intel S3700 SSD, and two Intel Xeon E5-2620 processors having total 12 cores of 2.1 GHz each. The machines are connected through a Mellanox SX6012 switch via 10 Gbps links.

Our testbed consists of 64 virtual hosts distributed across four aggregation groups called pods; a pod is a grouping of four racks connected with two common aggregate switches, as shown in Figure 3(a). Each pod is distributed across three physical machines. Two machines emulate the hosts and the rack switches, while the third machine emulates the aggregate switches belonging to that pod. The pods are connected through two core switches that are emulated in a separate dedicated machine.

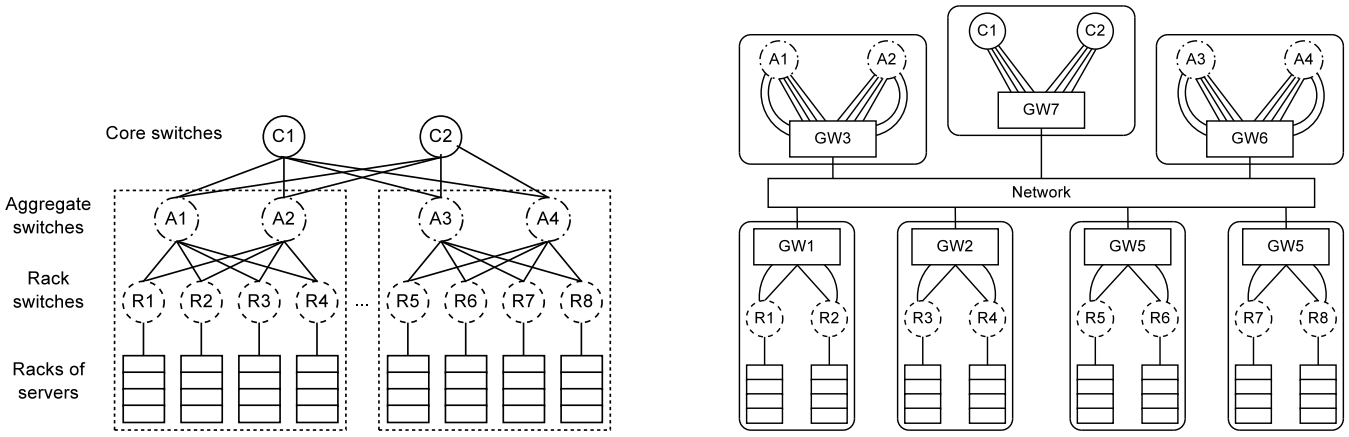
We stitched these network slices together through a combination of IP and MAC address translations. We also considered using GRE tunneling and Maxinet [37]. However, we developed our scheme that is tailored to our environment in order to improve performance. We use 1 Gbps edge links in the virtual topology and vary the higher tier links capacity for different oversubscription ratios.

6.1.1 Traffic Matrix

To simulate and address a variety of traffic patterns for a distributed filesystem deployed in a datacenter, we synthesized the workload using probabilistic methods, where: (1) job arrival follows the Poisson distribution, (2) file read popularity follows the Zipf distribution [7] with the skewness parameter $\rho = 1.1$, and (3) the clients are placed based on the staggered probability described by Hedera [6]: a client is placed in the same rack as the primary replica with probability R, in another rack but in the same pod with probability P, and in a different pod with probability $O = 1 - R - P$. The replica placement follows conventional constraints of fault tolerance domains. The primary replica is placed in a uniform-randomly selected server. The second replica is placed in the same pod as the primary replica, and the third replica is placed in a different pod.

6.2 Replica/Path Selection Comparison

We compared Mayflower's replica-path selection with four other methods that are a combination of static and dynamic replica selection with various network load balancing methods:



(a) Logical structure of the testbed

(b) Physical structure of the testbed

Figure 3: Testbed setup

Nearest with ECMP: In this method, the closest replica to the client is selected, and the flows are spread across redundant links using ECMP. This represents the methods where only the static information is used for replica selection and network load balancing.

Sinbad-R with ECMP: In this scheme, a replica is selected based on the current network state by using our read-variant implementation of Sinbad [12], which we call `Sinbad-R`. Sinbad was originally designed for dynamic replica selection during file write operations. It collects end-hosts' network load information to estimate the network utilization for higher tier links.

To tailor Sinbad for file-read operations in `Sinbad-R`, we made two necessary modifications in the original method. First, `Sinbad-R` estimates the network utilization for the links facing towards the core layer, which is opposite to the direction of the links used for estimation in the original Sinbad scheme. This modification is necessary because the data-flow direction is opposite for file-reads in comparison with the file-write operations. Second, if there exists a pod where both the client and any replica are co-located, the replica search space is restricted to only that pod. There are no such restrictions in the original Sinbad replica selection, because all the hosts in the network are considered as potential replica write locations.

Dynamic path selection: To evaluate the effectiveness of `Nearest` and `Sinbad-R` replica selection combined with dynamic network load balancing, we coupled them with Mayflower's network flow scheduler. However, unlike Mayflower's combined replica and path selection, the optimization space is limited to the pre-selected source and destination pairs for these schemes. We refer to these methods as `Nearest Mayflower` and `Sinbad-R Mayflower` in our results.

6.3 Performance

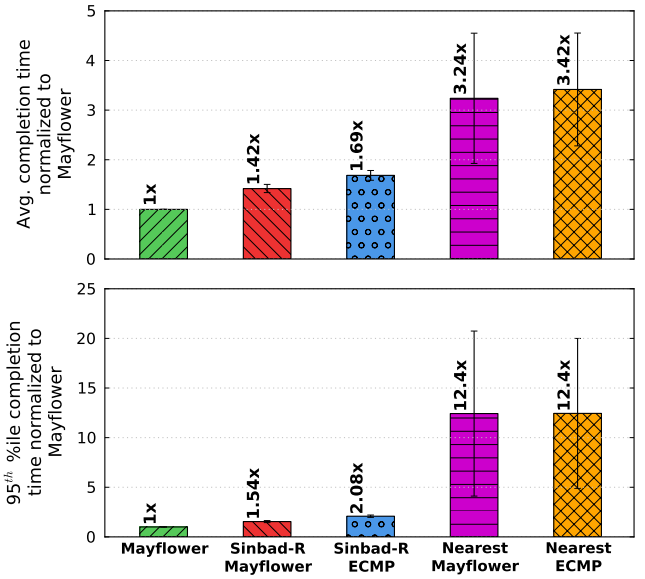


Figure 4: Average (top) and 95th percentile (bottom) job completion times normalized to Mayflower (first bar). 50% of the clients are located on the same rack as the primary replica of the requested file.

We first evaluate the performance of Mayflower's replica-path selection. In these experiments, we run a simple client/server application. Each client is given a set of files to read.

Figure 4 illustrates the performance of Mayflower's replica–path selection in comparison with the other methods (§ 6.2). The bars in the figure show the average and 95th percentile job completion times normalized to Mayflower results. The error bars represent 95% confidence interval calculated using Fieller's Method [30].The workload in these experiments represents the common scenario where most of

the clients are co-located with the data source in the same rack[1], following the locality distribution of (0.5, 0.3, 0.2), which implies that half of the clients are co-located with the primary replica.

The results highlight the benefits of following the network-filesystem co-design approach in Mayflower; the other approaches require on average 1.4x to 3.4x the job completion time compared to Mayflower. At 95th percentile, the the completion times increases to 12.4x, which highlights the impact of stragglers on the system performance.

The `Nearest` replica selection based approaches perform poorly because the replica selection is static and oblivious to the network state. As half of the clients are located in the same rack as the replica, the clients have only one replica and path option. Therefore, the edge link of the primary replica becomes congested. Moreover, the dynamic network load balancing cannot help in this case as the congestion location is at edge of the data source. As we see from the results with other locality distributions (§6.4), such link congestion can be reduced by placing more replicas at equal distance.

For similar reasons, `Sinbad-R` also suffers with lower performance. However, `Sinbad-R` has more replicas and paths to choose from; the pod having the primary replica also has a secondary replica, which implies that 80% of the clients have the choice of two replicas, and rest of them have the choice of all three replicas. `Sinbad-R` selects the replica based on the edge link's utilization as compared to distance based metric used in `Nearest` replica selection. Therefore, it performs significantly better than that, but still perform poorly than Mayflower. The performance difference between `Sinbad-R Mayflower` and `Sinbad-R ECMP` shows the benefits of dynamic network load balancing on top of dynamic replica selection.

Mayflower, on the other hand, has all three replica options for the clients. More replica options also increases the number of possible network paths between the client and the replica. Mayflower evaluates all possible replica-path options, and selects the one which is estimated to reduce overall job completion time. If the network paths from the closer replicas are congested, and a farther replica is a better choice, Mayflower selects the remote replica. As a result, it effectively avoids network hotspots close to the edge tier.

At 95th percentile, the wider performance gap between Mayflower and the other approaches shows Mayflower's strength in minimizing stragglers in the system. Stragglers adversely affect the performance of the applications that have barrier stages.

The presented results evaluate the performance of Mayflower for a specific workload which is common in datacenters. However, several variables affect the network dynamics and the performance of the replica and path selection methods, including: (1) the location of the clients relative to the replica, it impacts the location of the hotspots in the network, (2) the rate at which the clients start a new read job, it defines the load over the network and the filesystem, and (3) the network oversubscription ratio, which impacts the probability of congestion in the links. The subsequent sections discuss and evaluate the performance of Mayflower with these variations in the network and the workload.

[1]Results with other distributions are discussed later (§ 6.4).

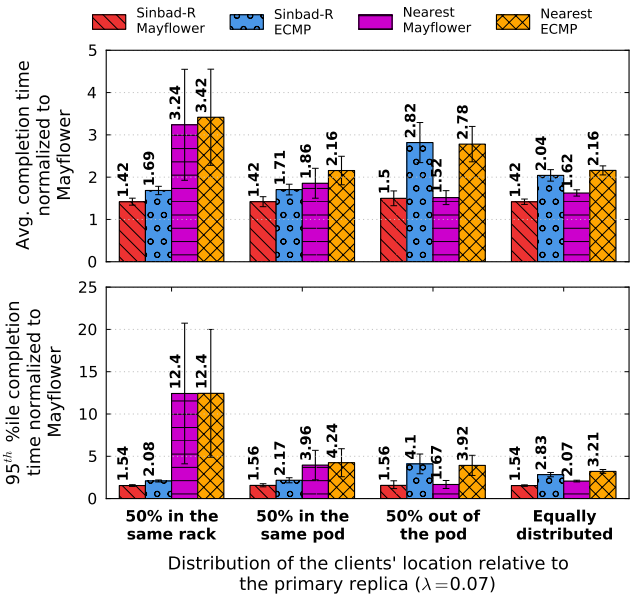


Figure 5: Average (top) and 95th percentile (bottom) job completion times normalized to Mayflower. The x-axis shows the distribution of clients relative to primary replica of the file (*R*,*P*,*O*), being in the same rack *R*, in the same pod *P*, and in another pod *O*. The groups have the distributions (0.5, 0.3, 0.2), (0.3, 0.5, 0.2), (0.2, 0.3, 0.5) and (0.33, 0.33, 0.33) in sequence from left to right.

6.4 Impact of the Clients' Locality

A client can be located in several locations relative to the replicas, that defines the network path length, and . A client can be co-located with the data source in the same machine. We ignore this scenario due to lack of network activity. A client can be located in the same rack as one of the replicas, in which case the closest replica is one hop away. If it is located in the same rack as the primary replica, the nearest replica is one hop away. If the client is located in the same pod as the primary replica,

Figure 5 shows the effectiveness of Mayflower's replica-path selection with different distributions of client locality relative to the primary replica. The bars are grouped according to the probability distributions (*R*, *P*, *O*) of the clients being in the same rack *R*, in the same pod *P* and in another pod *O* relative to the location of the primary replica. The groups have the probability distributions (0.5, 0.3, 0.2), (0.3, 0.5, 0.2), (0.2, 0.3, 0.5) and (0.33, 0.33, 0.33) in sequence from left to right. The bars in the first group are replicated from the previous figure for completeness and better visual

comparison.

The second group of bars in Figure 5 shows the performance when 50% of the clients have to fetch data from a remote rack in the same pod. That places more burden on the aggregation tier of the network compared with that of the edge tier. The results are not significantly different than the first group where the edge tier has higher load. Mayflower performs consistently well despite the higher oversubscription of the aggregate tier as compared to the edge tier. Its implications are significant for job scheduling in distributed data processing applications. Mayflower increases the flexibility of scheduling the jobs when the jobs cannot be scheduled on the same machine where the input data is located. As a result, Mayflower can increase the number of possible locations where jobs can be scheduled.

Consider a network with 40 servers per rack and 500 racks in a pod. The data is replicated three times in the usual fault domains. The preference of rack locality for job placement requires the jobs to be scheduled on the same rack where the input data is located. Thus, there are only 120 out of 20000 (0.6% of the total) servers where a job can be scheduled. If the data can be fetched from a remote rack with equal performance, it will increase the number of possible job scheduling locations from 120 to 20000 servers. Therefore, Mayflower can significantly improve the performance of the applications by eliminating the restriction to schedule the jobs only on those machines where the data is located.

The third group of bars in Figure 5 represents the case where 50% of the clients have to fetch the data by traversing the core tier. With the network having 8:1 core-to-rack oversubscription in these experiments, the core tier has higher load as it is the most oversubscribed. In this scenario, the replica and path selections become more important because of the higher utilization of the core tier and the replicas being equally distant for half of the clients. In such cases, `Nearest` replica selection becomes merely a random replica selection. The higher performance of `Sinbad-R Mayflower` and `Nearest Mayflower` relative to their counterparts with ECMP path selection shows the strength of Mayflower's path selection method. Mayflower picks the path that maximizes bandwidth for the new flow and minimizes its impact on the existing flows.

Finally, the last set of bars in the Figure 5 show the results for the case when the clients can be placed anywhere relative to the primary replica with equal probability. Mayflower consistently outperforms other techniques in this scenario.

6.5 Impact of High Job Rates

Network and filesystem performance varies with the load or the number of jobs in the system. For a system to sustain under higher load, it has to maintian the job completion time, or in other words, the average number of jobs in the system. Otherwise, due to various congestion points and overloaded links, the number of jobs in the system will keep increasing, and the system will reach a stage where no job will be able to finish within an acceptable time limit. We observe the same effect in our experiments where `Nearest Mayflower` and `Nearest ECMP` start failing at higher job arrival rate.

Figure 6 shows the variation of the average and 95th percentile job completion times with the increasing job arrival rate. The job arrival is modeled as a poisson process and the job arrival (λ) rate is defined per server. Thus the job arrival rate of 0.07 means that, system wide, about 5 new read jobs are started every second. Note that the y-axis in the figure shows the completion time instead of the improvement factor, which was shown in the previous figures. Moreover, the error bars show the 95% confidence interval calculated using student-t distribution.

Figure 6(a) shows the results for the common scenario in which mojority of the clients are situated in the same rack as the primary replica of the requested file. As expected, all the methods perform equally well at lower job rate because of the light burden on the system.At higher job rates, links start to become congested and the performance degrades quickly for all the methods except Mayflower, which has a small increase in completion time.

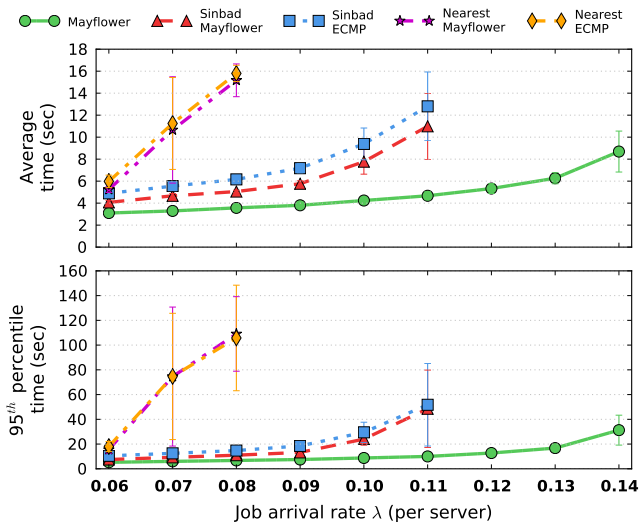
Moreover, the gap between Mayflower's performance and that of the other systems increases with the job rate. The relatively small completion time of Mayflower at higher job rates suggests that it effectively avoids congestion points in the network which reduces the load over the system. Moreover, the smaller growth in completion time with the job rate suggests that Mayflower can increase the throughput of the system by serving relatively a larger number of jobs without degrading performance.

Figure 6(b) shows a similar trend for the case where the core tier has the highest load. The core tier is the most oversubscribed and 50% of the clients read data by traversing the core tier. In this workload, we see a higher rate of increase in the completion time for all of the replica and path selection methods. However, Mayflower shows relatively smaller increases in completion time. The overlapping performance of `Sinbad ECMP` and `Nearest ECMP`, and the consistent performance difference of Mayflower compared to `Sinbad Mayflower` and `Nearest Mayflower` further corroborates the the effectiveness of Mayflower's path selection mechanism.

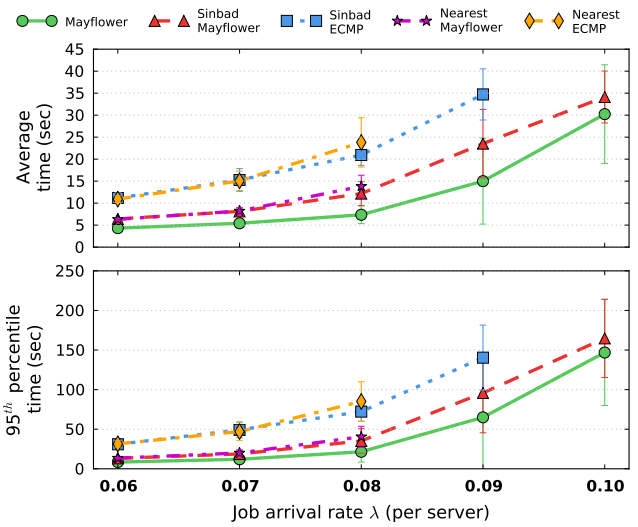
6.6 Impact of Oversubscription

Figure 7 shows the performance of Mayflower with different network oversubscription ratios. The results shown in the previous sections are with 8:1 core-to-rack oversubscription ratio. Higher oversubscription increases the chances for network congestion. We show the results for Mayflower and `Sinbad-R Mayflower` as those are the best among all the methods. Both Mayflower and `Sinbad-R Mayflower` have higher impact due to higher oversubscription. Job completion times almost double when we double the oversubscription ratio.

6.7 Comparison With HDFS



(a) When the clients locality distribution is $(0.5, 0.3, 0.2)$ — 50% of the clients are located in the same rack as the primary replica.



(b) When the clients locality distribution is $(0.2, 0.3, 0.5)$ — 50% of the clients fetch data which traverses the core tier.

Figure 6: Impact of the arrival rate of the new jobs per client. Mayflower can handle a relatively large number of jobs in the system with sub-linear scalability.

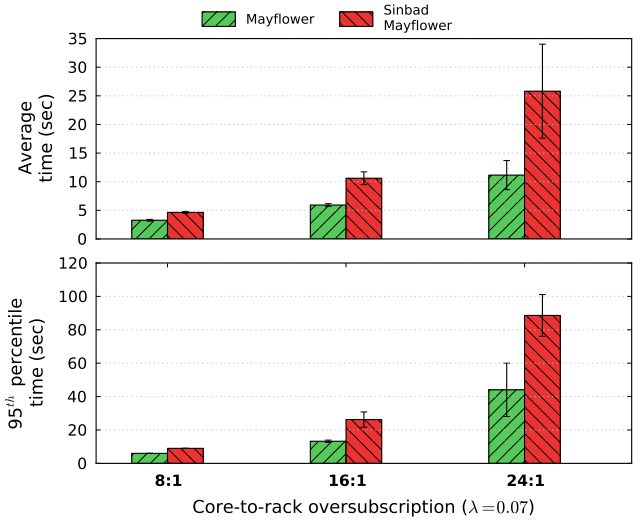


Figure 7: Impact of network oversubscription. 50% of the clients are located in the same rack as the primary replica.

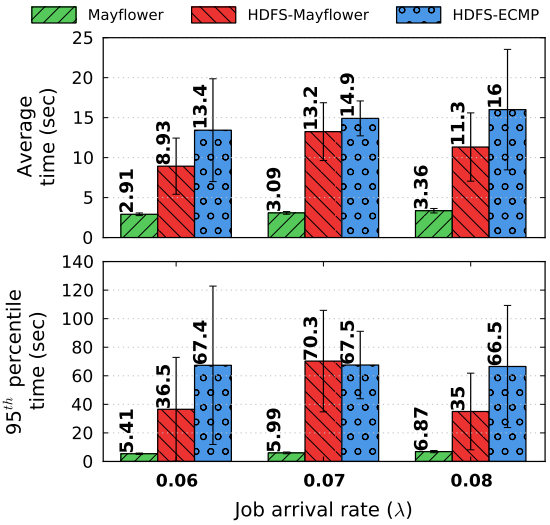


Figure 8: Mayflower real implementation comparison with HDFS.

Figure 8 shows the performance of our Mayflower implementation compared with HDFS. We used the same network setup and traffic matrix that we used for replica–path selection evaluation. However, we run the real filesystem in these experiments instead of running a client/server application. We configured HDFS to use rack awareness for replica selection – HDFS selects the replica in the same rack where the client is located, if any such replica exists. For network flow scheduling, we performed HDFS experiments with both ECMP and Mayflower flow scheduling. For file placement, we use the same primary replica location for both Mayflower and HDFS.

Mayflower's experimental results are consistent with our simulation results. Mayflower shows a small increase in the completion time as the job arrival rate grows. On the other hand, the completion times for HDFS grow quickly with the job rate. However, the default rack-aware replica selection limits the performance of HDFS. The better performance of

HDFS-Mayflower when compared with HDFS-Mayflower indicates that the network load balancing helps in reducing the congestion in the network. Nevertheless, coordinated replica selection with network management is key to achieving the best network and filesystem performance.

7. CONCLUSIONS

We presented Mayflower, a new distributed filesystem that follows a network/filesystem co-design approach to improving read performance. Mayflower's novel replica and network path selection algorithm can directly optimize for average job completion time using network measurement statistics collected by the SDN. We evaluated Mayflower using both simulations and a deployment on an emulated network using a fully functional prototype. Our results showed that existing systems require 1.5x the completion time compared to Mayflower using common datacenter workloads.

8. REFERENCES

[1] Apache Thrift RPC framework. https://thrift.apache.org.

[2] Leveldb key-value store. www.github.com/google/leveldb.

[3] Open vSwitch: Production Quality, Multilayer Open Virtual Switch. www.openvswitch.org.

[4] Universally unique identifier (UUID). www.bit.ly/1vjAJ2X.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM CCR*, volume 38, pages 63–74. ACM, 2008.

[6] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 19–19, 2010.

[7] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proceedings of the sixth conference on Computer Systems*, pages 287–300. ACM, 2011.

[8] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, pages 12–12. USENIX Association, 2011.

[9] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, pages 267–280. ACM, 2010.

[10] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. *SIGCOMM CCR*, 40(1):92–99, 2010.

[11] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *CoNEXT*, page 8. ACM, 2011.

[12] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *SIGCOMM 2013*, pages 231–242. ACM, 2013.

[13] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. In *SIGCOMM CCR*, volume 41, pages 98–109. ACM, 2011.

[14] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with varys. In *SIGCOMM*, pages 443–454. ACM, 2014.

[15] W. Cui and C. Qian. Difs: Distributed flow scheduling for data center networks. *arXiv preprint arXiv:1307.7416*, 2013.

[16] A. R. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *INFOCOM*, pages 1629–1637. IEEE, 2011.

[17] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized task-aware scheduling for data center networks. In *SIGCOMM*, pages 431–442. ACM, 2014.

[18] D. Erickson. Floodlight: open source SDN controller. www.projectfloodlight.org.

[19] N. Farrington and A. Andreyev. Facebook's data center network architecture. In *IEEE Optical Interconnects Conference*, pages 5–7, 2013.

[20] D. Fetterly, M. Haridasan, M. Isard, and S. Sundararaman. Tidyfs: A simple and small distributed file system. In *USENIX ATC*, 2011.

[21] A. Fikes. Storage architecture and challenges. http://bit.ly/1Q0jhh6.

[22] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[23] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *SIGCOMM*, pages 51–62. ACM, 2009.

[24] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: a high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, pages 63–74. ACM, 2009.

[25] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: a scalable and fault-tolerant network structure for data centers. In *SIGCOMM*, pages 75–86. ACM, 2008.

[26] C. E. Hopps. RFC 2992: Analysis of an equal-cost multi-path algorithm, 2000. www.tools.ietf.org/html/rfc2992.

[27] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *IMC*, pages 202–208. ACM, 2009.

[28] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[29] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *SIGCOMM HotNets workshop*, pages 19:1–19:6. ACM, 2010.

[30] H. Motulsky. *Intuitive biostatistics*, volume 1. Oxford University Press, New York, 1995.

[31] E. B. Nightingale, J. Elson, J. Fan, O. S. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *OSDI*, pages 1–15, 2012.

[32] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM*, pages 39–50. ACM, 2009.

[33] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The quantcast file system. *Proceedings of the VLDB Endowment*, 6(11):1092–1101, 2013.

[34] K. Shvachko, H. Kuang, S. Radia, and R. 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] F. P. Tso and D. P. Pezaros. Improving data center network utilization using near-optimal traffic engineering. *Parallel and Distributed Systems, IEEE Transactions on*, 24(6):1139–1148, 2013.

[36] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed

file system. In *OSDI*, pages 307–320. USENIX Association, 2006.

[37] P. Wette, M. Draxler, and A. Schwabe. MaxiNet: Distributed Emulation of Software-Defined Networks. In *Networking Conference, 2014 IFIP*, pages 1–9. IEEE, 2014.

[38] L. Xu, J. Cipar, E. Krevat, A. Tumanov, N. Gupta, M. A. Kozuch, and G. R. Ganger. Springfs: bridging agility and performance in elastic distributed storage. In *FAST*, pages 243–255, 2014.

[39] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, pages 265–278. ACM, 2010.