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

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

***Abstract*—In this paper, we introduce Mayflower, a new distributed filesystem that is co-designed from the ground up to work together with a network control plane. In addition to the standard distributed filesystem components, Mayflower includes a flow monitor and manager running alongside 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. Mayflower can perform global optimizations that are unavailable to conventional network-aware 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 75% compared to HDFS with ECMP.**

***Index Terms*—Distributed filesystems; Network flow scheduling; Replica selection; Software-defined networks**

I. 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. Most network-aware distributed filesystems only use static network information in making their filesystem decisions and are not reciprocally involved in making network decisions that affect filesystem performance. Therefore, they are only marginally effective at avoiding network bottlenecks.

An example of a widely used network-aware distributed filesystem is HDFS [1], which makes use of static network topology information to perform 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 [2], it is highly likely that a random client will be equally distant to all of the replica hosts. In this scenario, HDFS is effectively performing random replica selection for the vast majority of requests.

The poor performance of remote reads has led to the adoption of distributed data processing frameworks, such as MapReduce, that move computation to the data. Although this approach is effective for many workloads, there is a large class of workloads, such as large-scale sorts, distributed joins, and matrix operations [3], [4], that require data movement. Even for workloads where moving computation to the data is possible, improving the performance of remote reads can still greatly simplify job scheduling as computational nodes can be decoupled from storage nodes.

Sinbad [5] 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 alongside the SDN controller that monitors the bandwidth utilization at the network edge, models the path bandwidth of the elephant flows in the network, and performs both replica and network path selection for client requests. In addition to Mayflower read requests, the Flowserver can perform path selection for other applications, such as selecting a machine for task scheduling, through a public interface.

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 the filesystem and network control plane. For example, when performing a read operation, instead of first selecting a replica based on network information and then choosing a network path connecting the client and the replica host, Mayflower can evaluate all possible paths between the client and all of the replica hosts.

Furthermore, Mayflower's tight integration with the network control plane enables it to directly minimize average request completion time. Unlike optimizing for the bandwidth metrics used in previous systems [5], 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 more 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.

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.

II. 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 and explain how they relate to distributed filesystem design.

A. Distributed Filesystems

Big-data applications rely heavily on distributed filesystems for storing and retrieving large datasets. Several large-scale distributed filesystems have been developed including Google File System (GFS) [2], Hadoop Distributed File System (HDFS) [1], Quantacast File System [6], Colossus [7], and SpringFS [8]. Many of these filesystems are network-aware and can take advantage of network topology information to select the closest replica to service a read request. Mayflower's basic filesystem design is similar to these existing distributed filesystems. However, instead of simply being network aware, Mayflower is co-designed from the ground up with a software-defined network. This allows it to take advantage of additional cross-layer optimization opportunities.

B. Network Traffic Characteristics

A study using Facebook and Microsoft datacenter traces [5] reports that distributed filesystems contribute from 54% to 85% of the total datacenter network traffic. Therefore, with such a significant network requirement, the performance of a distributed filesystem can be very sensitive to network conditions. Specifically, in oversubscribed networks, the oversubscribed portions of the network often become performance bottlenecks [9].

Most distributed filesystems address network bottlenecks by taking advantage of network locality, which reduces the need to service requests from distant parts of the network. Alternatively, a full-bisection bandwidth network can be used to avoid these bottlenecks [3]. However, oversubscribed hierarchical topologies are still prevalent, as reported in several network measurement studies [9], [5], [10], [11], [12], [4]. Moreover, the increasing popularity of SSDs and in-memory applications suggest that the network will remain the primary bottleneck for many distributed datacenter applications.

C. Leveraging Data Locality

Placing data consumers close to the data can reduce the network footprint of file read operations. For example, in MapReduce, the Map tasks are typically scheduled on the same machine as where the data is located. This may lead to poor performance if the machine is a straggler because it is overloaded or occupied with a different MapReduce job. This problem can be partially addressed by selectively increasing the replication factor of popular data [13], or by modifying the job scheduler to limit resource contention [14]. However, these approaches either require additional resources, or is only effective when there is a queue of jobs that are waiting to run.

Many computational tasks, such as data sorting, distributed joins, and matrix operations, are solved using specialized computational frameworks and require data movement over network as part of the computation [3]. Moving consumers to the data is only partially effective for these tasks. A previous study [5] 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.

D. Replica Placement

Most distributed filesystems place replicas across multiple fault domains in the network for fault tolerance. By having multiple replicas in different parts of the network, a filesystem client can reduce the use of oversubscribed links by issuing read requests to its closest replica.

Sinbad [5] recognizes that there is significant flexibility in selecting replica locations. It improves write performance by selecting replica locations that avoid network congestion while meeting fault tolerance requirements. Sinbad relies on end-point network information to infer in-network link utilization.

E. Network Traffic Engineering

In order to take advantage of path diversity in a datacenter network, protocols such as ECMP [15] randomly select a path from the available shortest paths for each flow. This approach works well for short unpredictable flows, but may lead to persistent congestion on some links for elephant flows. Hedera [16] and MicroTE [17] address this problem by making centralized path selection decisions using global network information. In these systems, elephant flows or predictable short flows are identified and moved to non-conflicting paths to reduce network congestion. Alternatively, decentralized protocols, such as MPTCP [18] and LocalFlow [19], use only local information to make path selection decisions. Mayflower introduces a centralized multipath scheduling algorithm where,

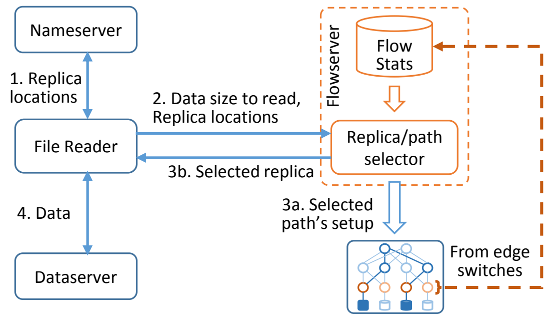


Fig. 1. Mayflower system components and their interaction in a file-read operation.

for a read request, the data source can be any one of the available replicas instead of a single pre-selected replica.

III. 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 provide the details of its system architecture.

A. Assumptions

Our design assumptions are heavily influenced by the reported usage models of Google filesystem (GFS) and HDFS. Mayflower assumes the following workload properties:

- The system only stores a modest number of files (on the order of tens of millions). File sizes typically range from hundreds of megabytes to terabytes. 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.
- File writes are primarily large sequential appends to files; random writes are very rare. Applications mutate data by either extending it through appends, or by creating new versions of it in the application layer and then overwriting the old version using a move operation.
- The workloads are heavily read-dominant. Read requests come from both local and remote clients.
- The network is the bottleneck resource due to a combination of high performance SSDs, efficient in-memory caching, and oversubscription in datacenter networks.

B. Architecture

Mayflower's basic system architecture consists of three main components: Dataserver, Nameserver, and Flowserver. The design principles of Dataserver and Nameserver are the same as their counterparts in GFS and HDFS. Therefore, we leave out their details and only discuss them briefly in this section.

Dataserver: The Dataserver stores actual file data. Each file is partitioned into chunks of configurable size. The chunk size is a system-wide parameter that is typically equal to or larger than 128 MB. Each file is represented as a directory in the Dataserver's local filesystem. The file-chunks are stored as individual files along with other metadata information in the file's directory. 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. In order to support atomic `append` requests, the Dataserver only services one `append` request at a time for each file.

Nameserver: Nameserver stores the metadata of the filesystem, including the 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, and cache the results for a Nameserver-specified amount of time to reduce lookup traffic. The mappings in the Nameserver are stored in a persistent database 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 files stored at the Dataservers.

For replica placement decisions, Nameserver takes into account system-wide fault-tolerance constraints, such as the replication factor and the number of fault domains. Currently, the Nameserver makes replica placement decisions using only static network topology information. This is because the focus of this work is on improving read performance for read-dominant workloads.

Nameserver is designed as a logically centralized service. We can improve its fault-tolerance by replicating its state across multiple nodes using a state machine replication system such as Paxos [20].

Flowserver: The Flowserver's primary task is to select both the replica and the network path for the Mayflower read operations. The Flowserver models the whole network and make its selections based on the estimated current network state. It builds its network model by tracking the network paths assigned to the read and write requests, and estimating the network utilization of each flow. The Flowserver can work together with existing network managers, such as DIFANE [21] and DevoFlow [22], to identify non-Mayflower-related elephant flows and estimate their bandwidth utilization. Using this model, the Flowserver can select a replica and path combination that minimizes the average completion time of all non-mice flows in the network, including existing network flows.

As the Flowserver does not explicitly control the bandwidth usage of each flow through bandwidth reservations, the network utilization estimates are often inaccurate. To avoid error propagation and reduce the effect of inaccuracy, the Flowserver periodically fetches the flow-stats from the edge switches. The flow-stats give the recent byte-counts for the flows. As the flow-stats are not collected very frequently, these byte-counts translate to the approximate bandwidth usage of the flows and their remaining size. These approximations reduces the inaccuracy in our estimations during replica and path selection.

In between measurements, the Flowserver tracks flow add and drop requests for the read jobs, and recomputes an estimate of the path bandwidth of each affected flow after each request. This ensures that completion time estimates are accurate, and also reduces the need to poll the switches at short intervals. The polling interval defaults to three seconds in our implementation. The Flowserver can be implemented as an integrated application in an existing SDN controller or as a standalone application that interacts with the SDN controller to perform path setup and to retrieve flow-stats information.

File read operation: Figure 1 illustrates Mayflower components and their interactions with a client in a file-read operation. 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. The Flowserver executes its replica-path selection algorithm (§ IV-B) for replica and path selection. In addition to selecting replicas, the Flowserver also installs the flow path for the request in the OpenFlow switches. Finally, the client contacts the Dataservers to retrieve the file.

Append-only semantics: Mayflower provides file-specific tunable consistency to its clients. In order to reduce reader/writer contention and 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.

C. 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, and the append completes once the primary receives an acknowledgment from all of the replica hosts. 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 be configured to provide linearizability with respect to `read` and `append` requests. The traditional approach to ensure strong consistency is to order all requests by the file's primary replica. However, Mayflower leverages its append-only semantics to only require sending the last chunk's `read` requests to the primary replica host. All other chunk requests can be sent to 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. Files that are deleted can temporarily appear readable due to client-side caching of the file-to-Dataserver mapping. However, we believe that this is a reasonable tradeoff for improving read performance given that delete requests are relatively rare.

Providing linearizability with interleaving delete requests can be achieved in the following way: When the Nameserver receives a delete request, it marks the file for deletion, but delays the delete for T time, where T is the maximum expiration period for the client-side file-to-Dataserver cache. During the delay period, if a client contacts the Nameserver for the file-to-Dataserver mapping, the Nameserver returns the result with a cache timeout equal to the remaining delay period. After T time, the Nameserver performs the delete operation. In this way, all the clients see the same interleaving of delete requests and cannot read the file from any replica after deletion. The performance impact is small for workloads where delete requests are rare.

IV. Replica and Path Selection

When a Flowserver receives a replica selection request from a client, it executes our replica-path selection algorithm (§ IV-B) to select a replica host and the network path for the request. The replica-path selection process is based on estimated network state that is built using bandwidth estimations and remaining flow size approximations.

Target performance metrics: Our target performance metric is the average *job completion times*. 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 [16], [5] that are based on link utilization measurements, Mayflower maximizes the max-min bandwidth share because, in addition to link capacity, the max-min calculations also capture the number of existing flows and their bandwidth share in each link. 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 effect on existing flows when selecting the replica and the network path for a new request.

A. Problem Statement

Informally, our optimization goal is to select the replica and network path, among the paths from all replicas to the client, that minimizes the average completion time of Mayflower read jobs. In other words, our goal is to select the network path that minimizes the completion time of both the new flow as well as existing flows, including the previously existing elephant flows belonging to other applications. Our replica and path selection algorithm considers the following criteria: The paths of existing flows, the capacity of each link, the data size of each request, the estimated bandwidth shares of existing flows, and the remaining untransferred data size of existing flows.

A formal description of our problem, which resembles a minimum-cost flow problem, is as follows: Let G be the graph representing the paths from source to destination. Let $c_{i,j}$ be the cost of impact on existing flows on link (i,j). Let $b_{i,j}$ be the bottleneck bandwidth on the paths containing link (i,j). Let $d_{i,j}$ be the data flowing on link (i,j). Let $I_{i,j}$ be a binary indicator that a flow is assigned to link (i,j). Let s be the supersource, connected to all replicas with 0-cost paths. Let t be a sink node and x be the data size. Given G, as well as $c_{i,j}$ and $b_{i,j}$ $\forall$ $(i,j) \in G$, we have:

$$\min_{d_{i,j}, I_{i,j}} \sum_{(i,j)\in G} \frac{d_{i,j}}{b_{i,j}} + I_{i,j} c_{i,j}$$

subject to:

$$\sum_{j:(i,j)\in G} d_{i,j} = \sum_{j:(j,i)\in G} d_{j,i}, \forall i \neq s,t, i \in G$$

$$\sum_{i:(s,i)\in G} d_{s,i} = \sum_{i:(i,t)\in G} d_{i,t} = x$$

$$0 \leq d_{i,j} \leq x I_{i,j}$$

The objective function is formulated such that, in the optimal solution, the cost will be minimized if the binary variable $I_{i,j} = 1$, i.e., if and only if the link (i,j) is on the path used by the flow.

This formulation assumes that the bandwidth of flows along each link in a path is equal to the bottleneck bandwidth in that path. However, a link may be part of multiple paths, and the bandwidth of that link can depend on the path taken by the flow. To accommodate this, we construct a graph G' to be a tree with the sink node (client) as the root, and all the possible paths from a replica encoded as source nodes, i.e., one link per path from a replica to the client. The cost of using each link in G' is the aggregate cost of using all the links on the corresponding path in G, with the bandwidth set to the bottleneck bandwidth of that path. If paths and costs are precomputed, we can find the least-cost path for the flow by calculating the cost of the links and choosing the link in G', i.e., the path in G, with lowest cost. This serves as the basis of our replica-path selection algorithm. The next section discusses the replica-path selection algorithm and the method of calculating the cost of a path.

B. 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:

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

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.

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 portion of equation 2 estimates the cost of the new flow, while the second portion estimates the impact of the 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 that use the link and their bandwidth demands, 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 the link's capacity. The estimated bandwidth b_j of the new flow is its bandwidth in the bottleneck link in the path. The new bandwidth estimate of the existing flows is their bandwidth share when a new flow with bandwidth demand b_j is added to the links in the path. The total number of max-min fair share calculations needed to determine the bandwidth share of the new flow is bounded by the shortest path length between the client and the replica, which is usually small in a datacenter.

Background traffic: Mayflower can include information about elephant flows generated by other applications, which we refer to as background flows, to improve the accuracy of its bandwidth utilization estimates. Several existing systems have been proposed to detect elephant flows [16], [21], [23]. We assume an existing system performs elephant flow detection and informs the path taken by these flows to the Flowserver. The Flowserver adds these flows in its network view which allows correct bandwidth estimates for replica-path selection.

Unknown flow size: Occasionally, the size of a flow is not known (as is the case with background flows), or the provided information is inaccurate. Flow size information is not critical when estimating the cost of a new flow because, as shown in equation 2, the variation in the cost of a new flow derives from its estimated bandwidth. The flow size is used in the estimation of completion time increase for existing flows. If the flow size information is unknown during replica-path selection, then the average elephant flow size is used as an estimate.

Slack in updating bandwidth utilization: When a path is selected by the Flowserver using the replica-path selection algorithm, the bandwidth utilization for the new flow is set to its estimated bandwidth share. The bandwidth share of the existing flows in the selected path are updated with their new estimated values. To avoid a herd effect, these flows are then placed in an *update-freeze state* for a second. Flows take time to converge, and because their stats are collected periodically, the first update can coincide with a flow's unstable starting time. This incorrect bandwidth usage information can cause a large number of flows to be assigned to the same path, congesting the network. Therefore, the bandwidth usage of the flows in the update-freeze state is not updated. However, the remaining flow size information is allowed to be updated.

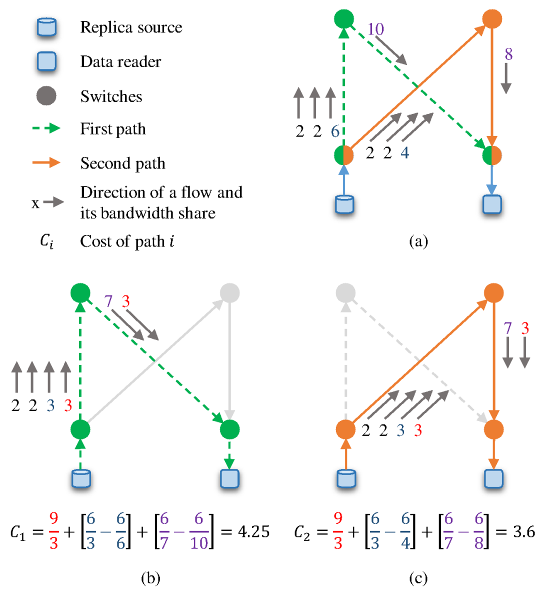
Simplifying bandwidth estimations: The reduction in bandwidth share of flows in a path may result in the increase in bandwidth share of flows in other paths. This can indirectly affect nearly all flows in the system. To accurately measure the impact of adding a new flow on a path, we need to not only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

For simplicity, we ignore the secondary effects of changes in bandwidth, 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 edge switches' flow-stats. We also use the flow stat information to update the bandwidth utilization of the remaining flows in the network. This significantly reduces the complexity of the problem while still providing good bandwidth utilization approximations.

However, polling the switches to collect flow-stats incurs additional overhead on the switches and the Flowserver. Mayflower reduces the overhead of stats collection by polling only the edge switches. Stats collection for non-edge switches is unnecessary because their link utilizations can be inferred by combining flow path information with flow stats from the edge switches. The stats collection period can be relatively large without affecting Mayflower's performance. This is because flow stats are primarily used to reduce small approximation errors in our network model. Mayflower uses a one second stats collection interval by default, and our experimental results are unaffected when we increase the interval period from one to five seconds. Finally, polling across edge switches is staggered to further reduce the impact of stats collection on the network.

An illustrative example: Consider an example where a client reads 9Mb data from a replica source, illustrated in Figure 2. The Figure shows two edge switches connected to two aggregate switches through 10Mbps links. 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 switch to the aggregate switch, and one flow on the third link, which connects the aggregate switch to the edge switch.

We first evaluate the first path, shown in Figure 2b. The second link is the bottleneck link for a new flow because it gives a 3Mbps share to the new flow, compared to a 5Mbps share on the third link. Therefore, the new flow will have a 3Mbps share on the first path and the read job will take $9/3 = 3$ seconds to complete. 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 sizes 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 would be 3.6, and therefore the second path will be selected for the read operation.



Replica source
Data reader
Switches
First path
Second path
x Direction of a flow and its bandwidth share
C_i Cost of path i
10
8
2 2 6
2 2 4
(a)
7 3
2 2 3 3
$C_1 = \frac{9}{3} + \left[\frac{6}{3} - \frac{6}{6}\right] + \left[\frac{6}{7} - \frac{6}{10}\right] = 4.25$
(b)
7 3
2 2 3 3
$C_2 = \frac{9}{3} + \left[\frac{6}{3} - \frac{6}{4}\right] + \left[\frac{6}{7} - \frac{6}{8}\right] = 3.6$
(c)

Fig. 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 is C_2.

In this example, the bandwidth share of the new flow is the same for both paths. The difference in cost is due to the increase in completion time of the existing flows. The second path has an increase of 0.6 seconds in the completion time of the existing flows, compared to 1.4 seconds for the first path.

C. Reading from Multiple Replicas

Mayflower reads from multiple replicas in parallel if doing so results in a reduction of the completion time. A read job is split into two subflows if the combined cost of the two subflows is smaller than the cost of reading from only one replica. The cost function remains the same as in equation 2. The bandwidth estimate for the two subflows is adjusted as if two new flows are being added in the system.

After selecting the network paths for the subflows and estimating their bandwidth shares, the data read size for each subflow is divided such that the subflows finish at the same time. The subflows are assigned different replicas to avoid encountering the same network bottlenecks.

Multiple replicas selection process: First, a replica-path p_1 is selected for the first subflow f_1 using the replica-path selection algorithm (§ IV-B). Assume that the estimated bandwidth share of f_1 is b_1 and its cost is c_1. A temporary flow is then added in the selected path to update the bandwidth estimates of the existing flows in that path. Then another replica-path p_2 is selected for the second subflow f_2, which has estimated bandwidth share b_2 and cost c_2. As p_1 and p_2 may have common links in their paths, f_2 may reduce the bandwidth share of f_1. Therefore, the bandwidth share and cost of f_1 are adjusted to b'_1 and c'_1 according to the reduction in b_1. If the combined cost, $c_t = c'_1 + c_2$, of the two subflows is smaller than c_1, which is the cost of using only one flow for the read job, the replica-paths p_1 and p_2 are selected for the subflows. Otherwise, the temporary changes done by adding a temporary flow in p_1 are rolled back, and only p_1 is selected by the Flowserver. If two subflows are selected, the flow size S_i for each subflow is adjusted proportionally to its estimated bandwidth share: $S_i = d * b_i / b$, where d is the requested data size and $b = b'_1 + b_2$.

The Flowserver returns the replica-paths and the associated data sizes to the client. The client concurrently downloads from the given replicas in chunks of K MB, where K is a tunable parameter that we set to be 32 MB for 128 MB block sizes. If one of the flows finish earlier, it starts downloading the remaining chunks of the other flow, including the last chunk.

Our results show that reading from multiple replicas further reduces the completion time of read jobs by 10% on average. Moreover, the average difference in completion times between the two subflows of a read job is less than one second when reading a 128 MB block.

V. Implementation

We implemented Mayflower in C++, with the exception of the Flowserver which runs as an application in the Java-based Floodlight controller. Our prototype consists of 7,500 lines of C++ code and 3,700 lines of Java code. The replica-path selection function in the Flowserver is exposed as an RPC service. The Flowserver is implemented as a stand alone service and can be integrated with any distributed application through its RPC framework.

The Nameserver stores the filesystem information in a LevelDB [24] key-value store. Files are divided in 128 MB chunks and replicated on three servers. The default replica placement strategy follows an HDFS's placement approach: Two replicas are placed in the same rack as the client, and the third replica is placed in another randomly selected rack.

VI. 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 with HDFS.

A. Experimental Setup

We conducted our experiments by emulating a 3-tier datacenter network topology using Mininet [25]. 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 emulate a network with 1 Gbps edge links. 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 a total of 12 cores running at 2.1 GHz. The machines are connected through a Mellanox SX6012 switch via 10 Gbps links.

Our testbed consists of 64 virtual hosts distributed across four pods. In our topology, a pod consists of four racks connected to two common aggregation switches. Each pod is emulated using three physical machines. Two machines emulate the hosts and the top-of-rack switches while the third machine emulates the aggregation 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 IP and MAC address translation. Our emulated network has 1 Gbps edge links, and oversubscription is achieved by varying the capacity of higher tier links.

Traffic Matrix: We evaluate Mayflower's performance using several synthetic workloads. Our workloads have the following properties: (1) job arrival follows the Poisson distribution, (2) file read popularity follows the Zipf distribution [13] with the skewness parameter $\rho = 1.1$, and (3) the clients are placed on virtual hosts based on the staggered probability described by Hedera [16]; 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 randomly selected server, the second replica is placed in the same pod as the primary, and the third replica is placed in a different pod.

B. Selection Schemes

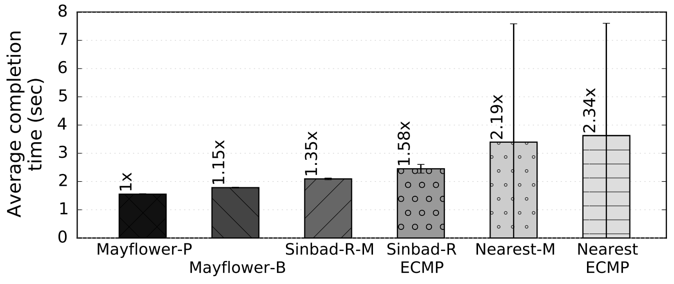
We compared Mayflower's replica-path selection with four other schemes that are a combination of static and dynamic replica selection with various network load balancing methods:

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

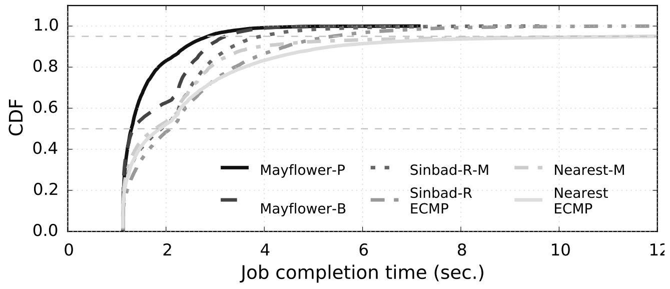
Sinbad-R with ECMP: In this scheme, a replica is selected based on the current network state by using our read-variant implementation of Sinbad [5], which we call Sinbad-R. Sinbad was originally designed for dynamic replica selection for file write operations. It collects end-hosts' network load information to estimate the network utilization for higher tier links.

In order to support file-read operations, Sinbad-R estimates the link utilization for traffic travelling up the network hierarchy (i.e., from edge to core) instead of down the hierarchy, which is the case for the original Sinbad scheme. This modification is necessary because file-reads and file-writes have opposing data-flow directions.

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 replica source for these schemes. We refer to these methods as Nearest-MF and Sinbad-R-MF in our results.



(a) Average completion times.



(b) CDF of the jobs completion times.

Fig. 3. Average job completion times of different replica and path selection methods.

Mayflower replica-path selection: Mayflower has two replica selection methods: Mayflower-B and Mayflower-P. In Mayflower-B, the Flowserver selects one replica, where as in Mayflower-P, it selects up to two replicas for each request. The client downloads from the selected replicas concurrently.

C. Replica-Path Selection Performance

We first evaluate the performance of Mayflower's replica-path selection using micro-benchmarks. In these experiments, we run a simple client/server application to emulate read jobs. For each read job, a client contacts the Flowserver for replica selection and path setup. The client then downloads 128 MB from the server selected by the Flowserver. Following the design described in Section IV-C, the client further breaks down the 128 MB block into 32 MB chunks, and downloads chunks from the selected replicas concurrently. When it finishes downloading a chunk from a replica, it will issue a request for the next chunk from the same replica until all four chunks have been downloaded. To avoid disk bottlenecks in these experiments, the data is stored in memory.

Figure 3 shows the performance of Mayflower's replica-path selection in comparison with the other methods (§ VI-B). The bars show the mean completion time of the read requests and the error bars represent 95% confidence intervals. Mayflower's completion time also includes the query latency to the Flowserver for replica-path selection, which is approximately 4 ms in our experiments. The parameters for this experimental workload consists of $\lambda = 0.1$, $\rho = 1.1$, and a client locality distribution of ($R = 0.5$, $P = 0.3$, $O = 0.2$). The results show that Nearest-ECMP, which is commonly used in current

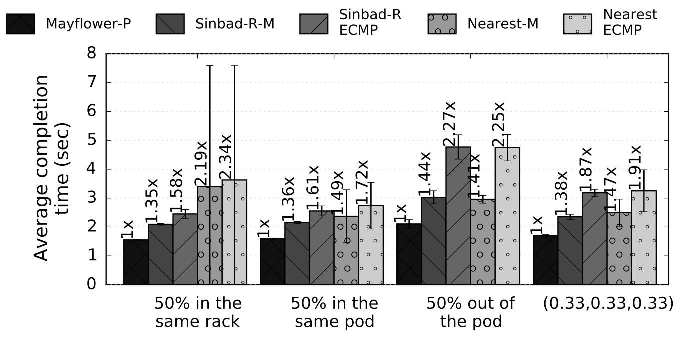


Fig. 4. Average job completion times with different client locality distributions.

deployments, has more than 2.3x the average completion time compared to Mayflower-P. It also shows that replacing ECMP with a dynamic path selection scheme (Nearest-M) only provides marginal improvements over ECMP.

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 in the same rack as the primary replica, the clients have only one replica and path option. Therefore, the edge link of the primary replica can become congested, resulting in higher completion time.

Sinbad-R-M and Sinbad-R ECMP illustrate that dynamic replica selection can reduce average completion time compared to static replica selection. However, because they select the replicas and paths independently, the Sinbad-based schemes still have an average completion time of more than 1.3x of Mayflower-P. Finally, by intelligently downloading from up to two replicas in parallel, Mayflower-P has 14% lower average completion time than Mayflower-B.

Figure 3(b) shows the CDF of job completion time for the same experiment. It illustrates that Mayflower-P has a significantly shorter tail completion time than the other schemes. Its 95th percentile job completion time is 2.81 seconds compared to 3.21 seconds for Mayflower-B and 3.82 seconds for Sinbad-R-M. These results suggest that, in the case where the paths to all replicas are partially congested, Mayflower-P can still achieve a low job completion time by reading from two replicas over two paths with different bottleneck links.

D. Impact of Client Locality

Figure 4 shows the effectiveness of Mayflower's replica-path selection with different client locality distributions. 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 bar groups in the Figure 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) from left to right. As Mayflower-P consistently performs between 10 to 15% better than Mayflower-B, we omit the results of Mayflower-B from the remaining graphs in this paper to improve their readability.

The results show that as we reduce client locality from (0.5, 0.3, 0.2) to (0.3, 0.5, 0.2) causing a larger portion of the

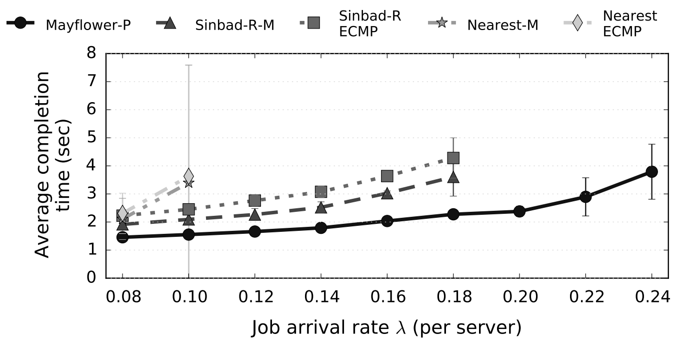


Fig. 5. Increase in average completion times due to higher job rate.

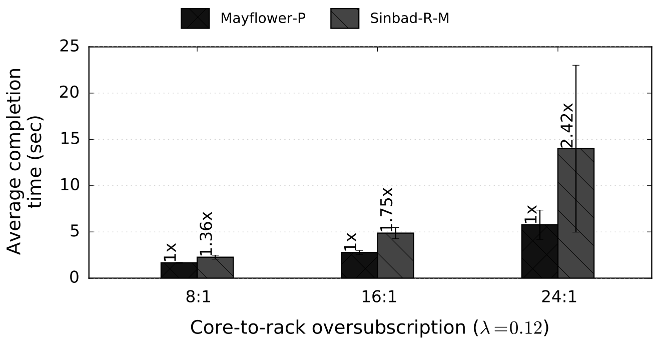


Fig. 6. Impact of network oversubscription on the average completion time.

traffic to leave the rack, the nearest replica selection schemes surprisingly show a reduction in their average completion time. This is due to a greater dispersion of traffic between replicas, because a client is equally distant from its two closest replicas for a larger percentage of its requests. For the other schemes, this reduction in client locality causes a small increase in average completion time, with the Sinbad and Nearest-based schemes having an average completion time of more than 1.36x compared to Mayflower-P.

Further reducing the client locality to (0.2, 0.3, 0.5) results in significant increase in average completion time for both Sinbad and Nearest-based approaches. Moreover, the average completion time of ECMP-based approaches increases by nearly 2.27x with this reduction in locality. This is because, with poor client locality, a larger fraction of the requests must traverse the heavily utilized core links. Therefore, efficient load balancing of the core links through dynamic path selection becomes far more important.

Finally, the last bar group shows that, in the case where a client distribution is (0.33, 0.33, 0.33), the result is in between the results from (0.3, 0.5, 0.2) and (0.2, 0.3, 0.5).

E. Impact of Job Rate

In this experiment, we vary the job rate to determine the impact of system load on the completion time of the different replica and path selection methods. Job arrival is modelled as a Poisson process and the job arrival rate (λ) specifies the rate for each server. Thus the job arrival rate of 0.08 means that on average 6 new read jobs are started every second in a 64 node system.

Figure 5 shows the results for the common scenario in which a majority of the clients are situated in the same rack as the primary replica of the requested file. We find that, all the methods perform equally well at lower job rate because of the light burden on the system. At higher job rates, links become congested and the performance degrades quickly for all the methods. However, Mayflower has a slower growth in the completion time.

The relatively small increase in completion time of Mayflower at higher job rates suggests that Mayflower is more effective at avoiding congestion points in the network than the other methods.

F. Impact of Oversubscription

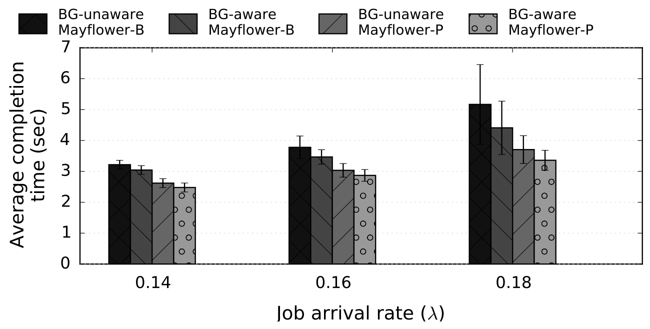
The experiments from the previous sections were conducted with an 8:1 oversubscription ratio. Figure 6 shows the performance of Mayflower with other network oversubscription ratios. There is no oversubscription within a rack in all of our tested configurations. Due to limitations in our test infrastructure, we are unable to experiment with lower oversubscription while maintaining a 1 Gbps edge link capacity. In this and subsequent Figures, we only show the results for Mayflower and Sinbad-R-M as those are the best among all the different methods.

Higher oversubscription increases the chances for network congestion. Both Mayflower and Sinbad-R-M have a higher average completion time in more oversubscribed networks. However, Mayflower performs better under higher oversubscription than Sinbad-R-M, difference of 2.4x, as it has more replica and path options.

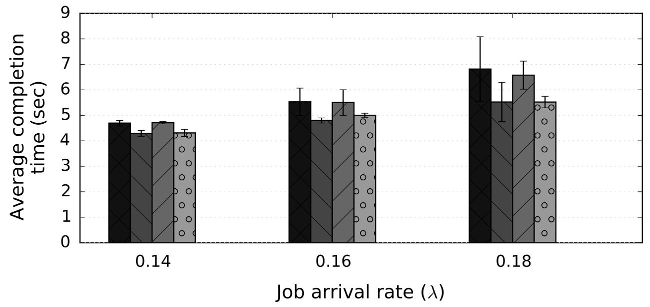
G. Effect on Background Traffic

In this experiment, we evaluate the performance of Mayflower-B and Mayflower-P in the presence of other flows in the network. We also evaluate the effect of Mayflower flows on background flows. To generate the background traffic, we run another client/server application in which the clients download data from a server selected randomly using the locality distribution of (0.2, 0.3, 0.5). The size of the background flows are based on a truncated normal distribution with the parameters $N(96,64)$ and with a lower and upper limit of [32, 256]. We use the job arrival rate $\lambda = 0.08$ for the background flows and vary the arrival rate of Mayflower file read requests, shown by the different bar groups in Figure 7.

Figure 7(a) shows the completion time of Mayflower read requests. The first bar, background unaware (BG-unaware) Mayflower-B, shows the completion time when Mayflower-B runs while being oblivious to the background flows. The second bar, background aware (BG-aware) Mayflower-B, shows the completion time when Mayflower-B is aware of the background flows. Without background flow awareness, Mayflower-B's bandwidth estimates are inaccurate, which can result in selecting paths that are congested with background flows. When Mayflower-B is aware of the background flows, it can accurately model the utilization of each link, which allows it



(a) Completion time of Mayflower file read requests.



(b) Completion time of the flows generated in the background.

Fig. 7. The performance of both Mayflower requests and background traffic.

to avoid congested links and reduce average read completion time compared to BG-unaware Mayflower-B. Compared to Mayflower-B, Mayflower-P is more effective at avoiding congested links by dispersing the load across two partially congested paths with different bottleneck links.

Figure 7(b) shows the completion time of the background flows. The Figure highlights the importance of better replica and path selection as it affects the performance of both the background flows and the file read operations. Poor path selection hurts all the elephant flows in the network. This is due to the interdependence between the network and the applications, suggesting the need for increased coordination between the endpoint applications and the network control plane. By making Mayflower aware of the background flows, our results show that we can reduce the completion of both Mayflower request and background flows compared to using background-unaware Mayflower.

H. *Effect of Data Size*

Figure 8 shows the performance of Mayflower-P with different block size and data rate combinations. Mayflower-P performs consistently better for both small and large block sizes. For smaller block sizes, the average completion time is shorter than the stats collection intervals. Therefore, Mayflower relies primarily on bandwidth utilization estimates to determine flow durations, which is in turn used to determine which replica-path it selects. As a result, for smaller block sizes, there is a smaller performance gap between Mayflower-P and the other methods because of its higher reliance on bandwidth estimates.

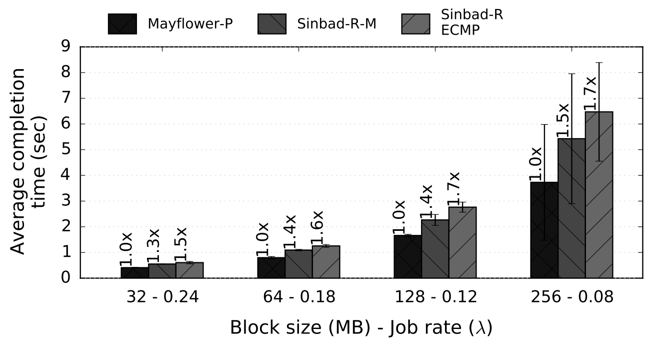


Fig. 8. Performance impact of the block size.

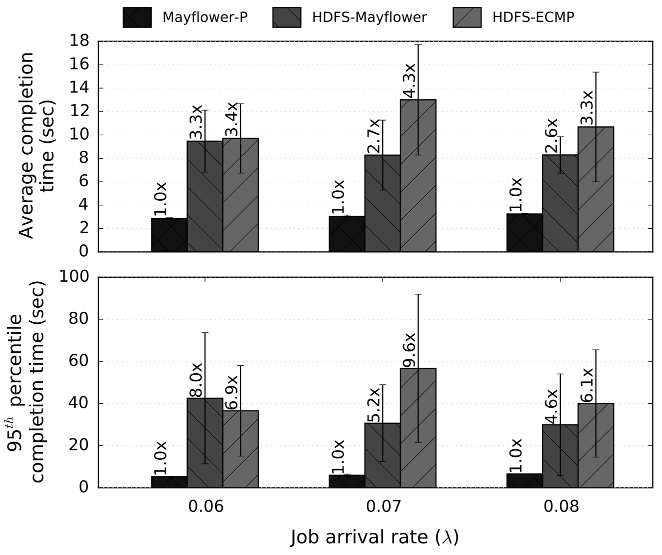


Fig. 9. Mayflower prototype versus HDFS.

I. *Comparison With HDFS*

Figure 9 shows the performance of our Mayflower prototype compared with HDFS. We used the same network setup and traffic matrix that were used for our replica-path selection evaluation. In this experiment, we use a block size of 256 MB, a client locality distribution of (0.5, 0.3, 0.2) and job arrival rates from $\lambda = 0.06$ to $\lambda = 0.08$. Instead of running a client/server application that only simulates file reads and writes, this experiment uses our Mayflower prototype implementation. 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 increases. In contrast, the completion times for HDFS grow rapidly with an increase in the job rate.

VII. 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 1.3x the completion time compared to Mayflower using common datacenter workloads.

VIII. Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), the David R. Cheriton Graduate Scholarship, the Go-Bell Graduate Scholarship, the University of Waterloo President's Scholarship, and the NSERC Alexander Graham Bell Graduate Scholarship. This work benefitted from the use of the CrySP RIPPLE Facility at the University of Waterloo.

References

[1] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of MSST*. IEEE, May 2010.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003.

[3] E. B. Nightingale, J. Elson, J. Fan, O. S. Hofmann, J. Howell, and Y. Suzue, "Flat datacenter storage." in *Proceedings of OSDI*, 2012.

[4] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proceedings of SIGCOMM*. ACM, 2015.

[5] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," in *Proceedings of SIGCOMM*. ACM, 2013.

[6] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The Quantcast file system," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, 2013.

[7] A. Fikes. Storage architecture and challenges. http://goo.gl/F4VQfT.

[8] 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 *Proceedings of FAST*, 2014.

[9] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proceedings of IMC*. ACM, 2009.

[10] N. Farrington and A. Andreyev, "Facebook's data center network architecture," in *Proceedings of the IEEE Optical Interconnects Conference*, 2013.

[11] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," *ACM SIGCOMM CCR*, vol. 40, 2010.

[12] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of IMC*. ACM, 2010.

[13] 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 EuroSys*. ACM, 2011.

[14] 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 *Proceedings of EuroSys*. ACM, 2010.

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

[16] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proceedings of NSDI*, vol. 10. USENIX, 2010.

[17] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine Grained Traffic Engineering for Data Centers," in *Proceedings of CoNEXT*. ACM, 2011.

[18] H. Han, S. Shakkottai, C. V. Hollot, R. Srikant, and D. Towsley, "Multipath tcp: a joint congestion control and routing scheme to exploit path diversity in the internet," *IEEE/ACM Transactions on Networking (TON)*, vol. 14, no. 6, 2006.

[19] S. Sen, D. Shue, S. Ihm, and M. J. Freedman, "Scalable, optimal flow routing in datacenters via local link balancing," in *Proceedings of CoNEXT*. ACM, 2013.

[20] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, 1998.

[21] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," in *Proceedings of SIGCOMM*, 2010.

[22] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: scaling flow management for high-performance networks," in *Proceedings of SIGCOMM*. ACM, 2011.

[23] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *Proceedings of INFOCOM*. IEEE, 2011.

[24] "LevelDB key-value store," www.github.com/google/leveldb.

[25] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the SIGCOMM HotNets workshop*. ACM, 2010.