PERFUSE: A Location Aware Distributed File System

Shravan Achar
Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC
Email: bachar@ncsu.edu

Guru Darshan Pollepalli Manohara
Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC
Email: gpollep@ncsu.edu

***Abstract*—This paper proposes a distributed file system with location-aware features. In decentralized delay-sensitive systems such as media streaming networks where massive amounts of data are distributed and shared among the nodes, the end-user experience is impacted by poor network latencies. Todays distributed file storage systems provide faster data access by allowing parallel reads and having local caches. We propose a that this can further be improved with our distributed algorithm making it a scalable solution for delay sensitive enterprise applications. Read and write requests are split and routed across all the nodes thereby increasing the overall read throughput. File sharing applications can make use of efficient data placement by the algorithm to improve the performance of peer-to-peer protocols such as BitTorrent which are commonly employed in media streaming networks. We have validated our claims by comparing the performance of reads and writes for storage servers spread across two continents.**

***Keywords*—distributed file system, gRPC, FUSE**

I. Introduction

In modern data centers, a need to manage large pools of storage devices geographically distributed across different data centers often arises. Further, performance of a distributed application is impacted by network latencies associated with storage nodes. We have designed and implemented a distributed file system using gRPC and FUSE. Perfuse is designed to manage storage nodes and ,specifically, to minimize the impact of network latency in a distributed file system. It shares many of the same goals of other distributed file systems such as reliability and availability. It constructs a distributed network coordinate system, using which relative distances between any two client and server can be determined. We, finally, construct an experimental setup with two storage devices with different network latencies. We analyze and document the performance improvements gained by reducing network latency to a storage device for any given client.

II. System Architecture

Perfuse is a concoction of various three components: (i) the storage unit, (ii) the metadata unit and (iii) the client. A logical and physical separation exists between the three main components, namely, the client, metadata server and the storage server. Each of these components are logically isolated from each other and one or more of any component can interact with one or more of the other two components. For example, multiple clients could be using the same metadata server but the metadata server may be using a pool of storage servers. The storage server is the storage unit which is ideally hosted on a public or private cloud. The meta-data server provides meta-information about the files and data to its clients, manages the storage servers that hold the actual data. The client application provides a POSIX based file system interface to the user.

A. Meta-data Server

meta-data server contains all the supporting logic for the functioning of the entire system All the meta-data associated with the files including location in the tree, their last access dates, etc. are stored in the meta-data server. Blocks of file data (called chunks) are identified with a collision-resistant hash. meta-data server stores information about chunks of all the files in the system in its database. These information includes its hash, offset, length and a seeder list. A seeder is an identifier for a storage node that has raw bytes associated with the chunk. The server also constructs a logical 2D map of the entire system using the algorithm in (**?**). Further, the server receives constant heart-beat messages from all the storage nodes. Any data movement or data addition in the system is notified to the meta-data Server. The meta-data server must go live before other components begin their process.

B. Storage Server

Storage servers provide storage media to store persistent data. The data in stored in the form of chunks. Typically, a file is made up of many chunks. This is in conformance to the idea of having de-duplication in all the components of the system. A single chunk may belong to more than one file, but only a single copy of it is stored. Further, in the event of a failure, the data is backed-up to cold storage with no further de-duplication efforts. The pool of storage servers form a distributed system. Unlike traditional distributed systems where a single master is chosen across the entire system, in Perfuse a logical master is chosen for every single client. This is similar to having a primary server for every single client with the difference being that the primary server changes as the client changes its location. The master server is often the closest storage server with respect to the client. Closest means that the server which has minimum network latency, with respect to the client, along with highest disk performance. This is to say that given two storage servers with equal network latencies, SSD storage is chosen as the master server.

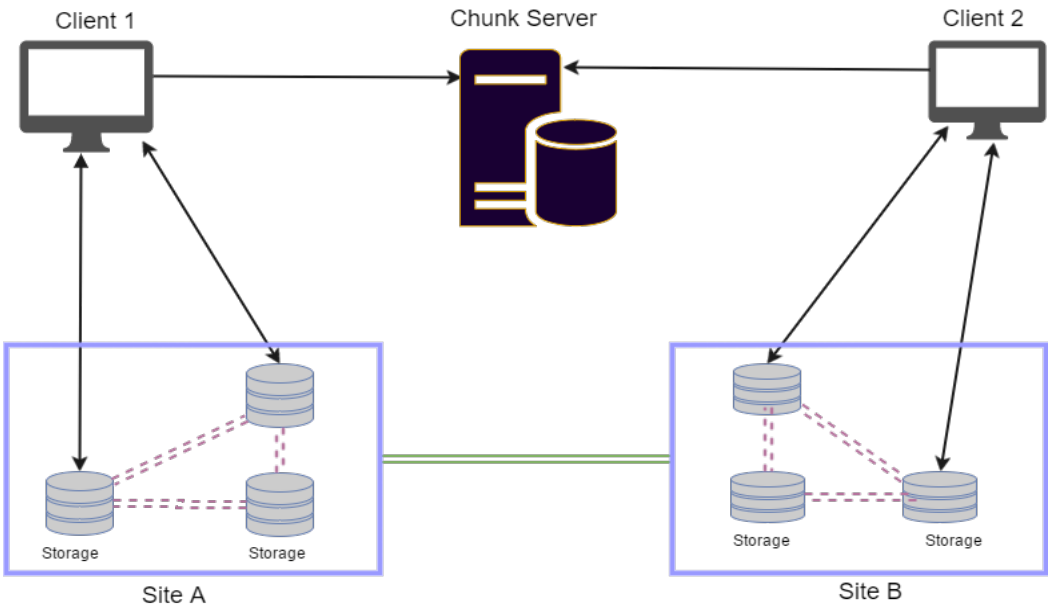

Fig. 1: System Architecture

C. Client

client provides a uniform file system interface to the users. It supports all operations expected of a POSIX-compliant local file system such as read(), write(), mkdir(), rmdir(), unlink() etc. The client is implemented in the user-space using FUSE libraries. The client also contains local cache to hold data chunks at the time of reads and writes. All the changes to a file are performed locally on the client and the resultant changes are sent to the storage server and meta-data server upon file close (or flush). client and storage server are primarily responsible for maintaining data de-duplication in the system. client fragments the files and computing hashes on the fragments (chunks) and then reports it to other components.

III. System Design

A. Data De-duplicaton

All the storage servers store data in the form of chunks. Mapping between the chunks and Files is stored on the Meta-Data Server and provided to the client upon request. Perfuse uses fixed size chunks of 4KB. To compute the signature associated with a chunk, SHA1 hash is used. This creates a unique signature for the content associated with the Chunk. All the data transfers in the system are performed at the Chunk level. The client is responsible for fragmentation of any new file and computes hash for the individual chunks before transferring the data to a nearby storage server. When an application opens a file, client downloads all the hashes associated with the files constituent chunks. The client downloads the Chunk Data on demand from the storage server based on the Read and Write requests from the application. Upon file close, the client expunges the hash entries associated with the file from his database.

B. Vivaldi Localization

As stated earlier in the paper, an important goal of the project is to improve the network delay associated with traditional distributed file systems. This is achieved by measuring the client's proximity to the storage server. We examined multiple localization solutions which could provide better delay metrics and reduce overhead associated with the computation. Most algorithms require complete information about end-to-end delays between all the nodes in the system. This means that every node in the system has to measure delay with respect to the other n-1 nodes, creating an algorithmic complexity of $O(n^2)$. A Network Coordinate System as described in (1), on the other hand provides a localization solution with an algorithmic complexity of O(n), and it accomplishes it with very little computational overhead and delay information from just three other nodes. It uses simplified triangulation techniques to provide an approximate geographical distribution of nodes. Any point on a 2D Cartesian plane can be located if its distance from three different points are known. But in real world, when measuring network delays, we must bring transmission delays and queuing delay into the picture along with propagation delay. These additional components adds uncertainty in the computation as triangle inequality is violated on multiple occasions. To reduce the error associated with the computation, Vivaldi visualizes the network as a spring system, where springs are connected between nodes whose delay information is available. The measured delays are the natural length of the spring and any decrease or increase in length would cause the spring to change its potential energy. Vivaldi algorithm attempts to compute the network node distribution with minimum potential energy.

We extend the definition of delay to include the Disk I/O latency of the node. So, the delay supplied to Vivaldi is a combination of network delay and disk I/O latency. The additional coupling with disk I/O latency ensures that the storage servers with a SSD is given a higher preference than

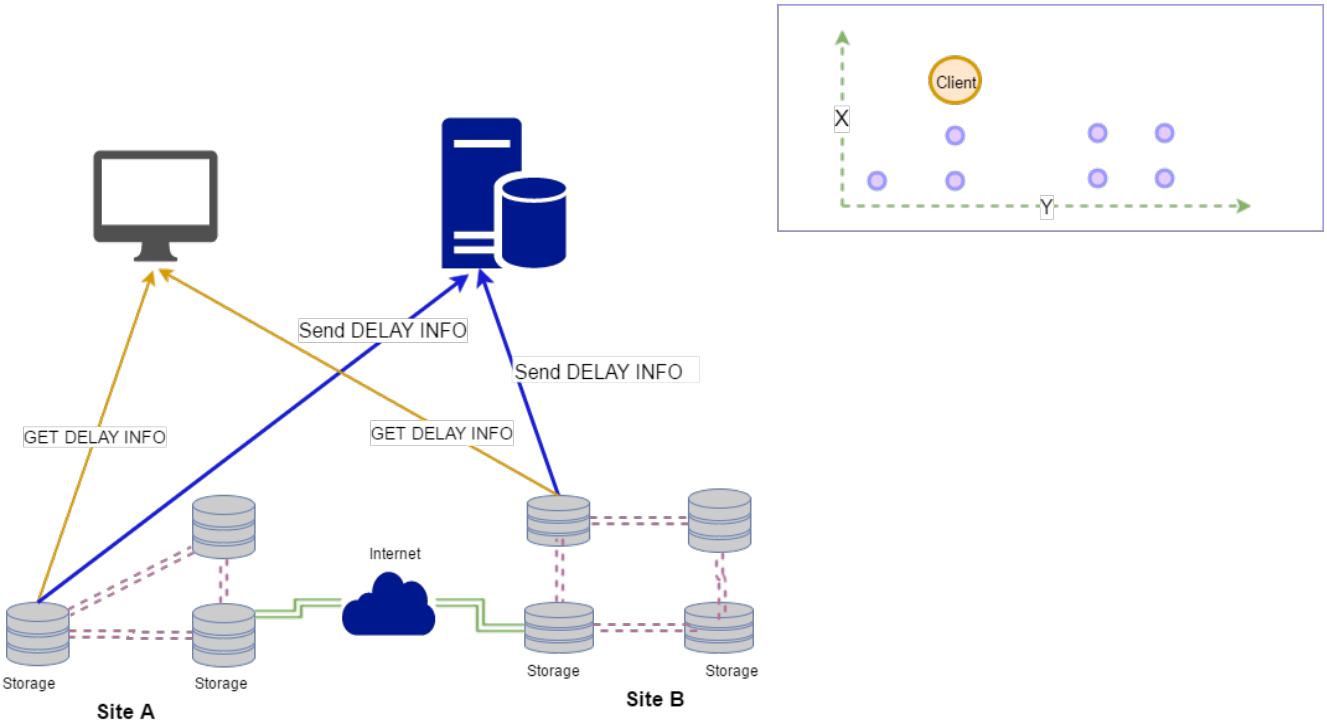

Fig. 2: client Addition using Vivaldi

HDD during read/write operations. The equation below shows the delay equation. (α is a constant less than 1)

$$T_{delay} = T_{network} + \alpha * T_{disk-latency}$$

Meta-data server stores the topological map of the network using the Vivaldi algorithm. It is notified when a new storage server is added to the system or an existing storage server leaves the system. Any changes to the delay associated with the links are immediately updated in the network topology map. When a new client joins the system, chunks Server attempts to locate it with respect to the storage servers. The following events occur when a new client joins: (i) 3 randomly chosen storage servers measure delays from the client to themselves. (ii) The delay info is consolidated and transmitted to the meta-data server. (iii) meta-data server uses Vivaldi algorithm to update the clients location in the map. The following events occur when a new Server joins the system: (i)3 randomly chosen storage nodes measure delay from the new server to themselves. If less than 3 nodes are present in the system, then all the nodes measure delays from the new server. (ii)Delay info (including disk delays) is sent to the metadata server (iii)The topological map is updated. These operations are shown in Figure 2.

C. Reliability

Perfuse can be configured to have a replication factor of more than 1. The default replication factor is 1. When the replication factor is more than 1, the primary server is responsible to push the data to all other backup servers. The primary server, however, returns immediately to the client call and the data transfer happens in the background. This forces the system to have a non-zero minimum RPO. At the same time, by returning the client call after the primary is written to, offers higher performance. When a storage node goes down, the metadata server skips heart beats from the node. This causes the metadata server to re-compute the seeders list for every chunk it maintains its database. The storage node is removed from the seeder list for every chunk. Though the system provides reliability with respect to storage nodes, the metadata server still remains a single point of failure.

IV. Implementation

A. File Operations

Perfuse handles all data transfers at chunk granularity. Any operation performed on a file directly affects its constituent chunks. Currently Perfuse supports open, close, read, write, mkdir, rmdir, unlink, flush and truncate. Upon opening a file, client requests a list of all the chunks and their metadata (like length, offset, storage servers to contact) associated with a file from the Metadata Sever. At this point, only the hashes associated with the chunks are available with the client, the data associated with the file are not available to the client. Corresponding read() calls triggers Chunk downloads from the storage server (to the clients local cache). This is accomplished by using the storage server details in the Chunk metadata (downloaded at the time of file open). If the download fails, the client sends a request to the meta-data server and obtains an updated location for the Chunk. The number of chunks downloaded are computed based on the length and offset parameters obtained as part of the read() call. Similarly, any writes to a file are performed by fetching the Chunk data from the storage servers and then writing the data to the Chunk locally. When the application closes (or flushes) the file, all the updated chunks hashes are recomputed and sent

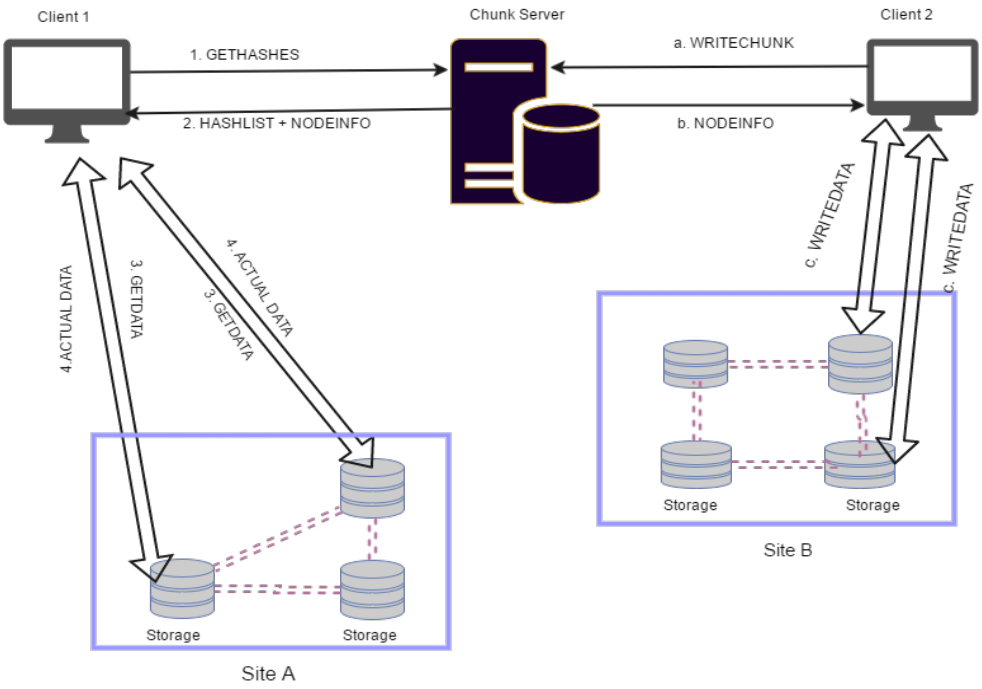

Fig. 3: Read and Write Operation

to storage server along with their data. The storage server ignores a Chunk if the incoming Chunks hash is already available with the storage server, else adds the Chunk into its database. storage server triggers a Route Update to the meta-data server and provides the updated hashes associated with the file and adds new Chunk into the System (if any) as shown in Figure 3. The file system follows close-to-open consistency. Any changes done to a file are visible to other clients when close() operation is performed. Directory creation and deletion operations are handled by sending file info updates to the meta-data server. Removal of a file causes the meta-data server to expunge all the Chunk hashes for the file, but the data associated with the constituent chunks is untouched on the storage server. This was done to allow faster file writes if the client recreates the file with similar contents in the file system. File truncation functions similar to write(). When the truncated file is larger than the actual file size, the additional zeros are appended to the last chunk of the file.

B. Inter-component Communication

Googles gRPC [2] framework is used to pass messages between the components (meta-data server, client, and storage server). Protocol Buffer [3] was chosen for serialization and deserialization of messages because it incurs minimal computational overhead and the resultant serialized data is smaller when compared to JSON. gRPC allows multiple gRPC clients to connect to gRPC servers at once. Multithreading and synchronization are handled internally by the gRPC suite. Data Streams are one of the essential components of a gRPC suite. They can provide significant performance boost when there are numerous data transactions. Streams band together individual requests and provide a consolidated (and iterable) object to the receiver. By using data streams for data path transactions between client and storage Server, we were able to achieve almost twice the performance than individual gRPC calls performed in succession. gRPC and Protocol Buffers support multiple languages like C++, Java, Python, Go, etc. meta-data server is implemented using Java and other components are implemented using Python.

V. Evaluation

The experimental scenario is constructed as per Figure 4. The primary storage server is set up on AWS. The secondary storage server is set up on VCL. The client, too, is located within the VCL environment. In traditional distributed file systems, the primary server is used to service the client even though the client changes its location. Farther the client goes from the primary server, larger network delay is incurred for file system operations. In perfuse, the client is serviced by the closest server.

We analyze read performance (both with and without cache), write performance experienced by the client when interacting with AWS server. Further, since de-duplication is performed at the storage server, a comparison of actual data received by the server versus the data stored in its media is also made. We perform the same experiment when the Vivaldi algorithm is in place, which chooses the closest server with respect to the client.

For this experiment, we construct 5 files with the following command

```
ruby -e 'a=STDIN.readlines;X.times do;b=[];Y.
    times do; b << a[rand(a.size)].chomp end;
    puts b.join(" "); end' < /usr/share/dict/
    words > file.txt
```

The command randomly chooses words from the dictionary of words in linux and constructs a text file. X denotes the

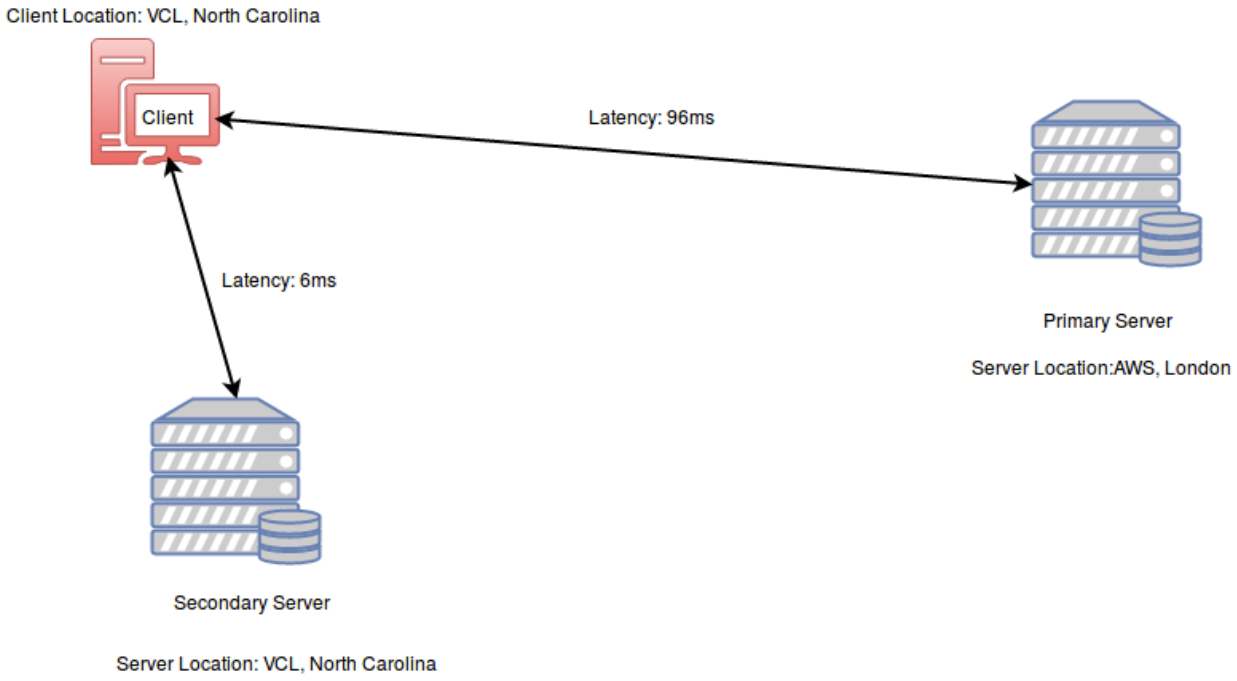

Fig. 4: Experimental Setup

number of lines and Y denotes the number of words per line. Five files, called base files, are 1MB, 2MB, 4MB, 8MB and 16MB in size respectively(numbered from file1.txt to file5.txt). The base files are concatenated in specific order to create 5 test files. This is done to create duplication of data in files. The same is tabulated in Table I

VI. Results

All the files are written to and read from server located in AWS. The latency between the client and server was recorded to be 96ms. The figure Figure 5 shows the write performance experienced by the client with storage server set up on AWS. From the figure it can be seen that the write times increase

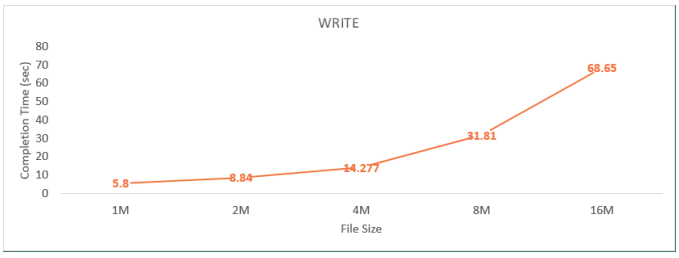

Fig. 5: Write Performance AWS

almost linearly with file size.

Figure 6 shows the read performance experienced by the client with the AWS server. Comparison between cached and non-cached reads is plotted for each file. The presence of client cache greatly improves read performance, having recorded a 3x improvement for each file.

Fig. 6: Read Performance AWS

Figure shows the improvement in capacity at the storage server with de-duplication efforts. Close to 50% data is deduplicated, thereby providing 2x capacity compared to a system with no de-duplication.

Each of the above files are written to and read from server located in VCL. Though the AWS server is the primary, the vivaldi algorithm employed in perfuse chooses the closest server with respect to the client.

Figure 7 shows the write performance of files A to E to the closest server. Ideally, if the client moves to a different location the data must automatically be transferred to the closest server without any intervention of the client. Such a feature is currently unimplemented in the system. The write performance is improved for each file due to lower network latency.

Figure 8 shows the read performance(first time reads) with

File Name	File Size	Base files
A.txt	1MB	file1.txt
B.txt	2MB	file1.txt file2.txt
C.txt	4MB	file1.txt file2.txt file3.txt
D.txt	8MB	file1.txt file2.txt file3.txt file4.txt
E.txt	16MB	file1.txt file2.txt file3.txt file4.txt file5.txt

TABLE I: Test files

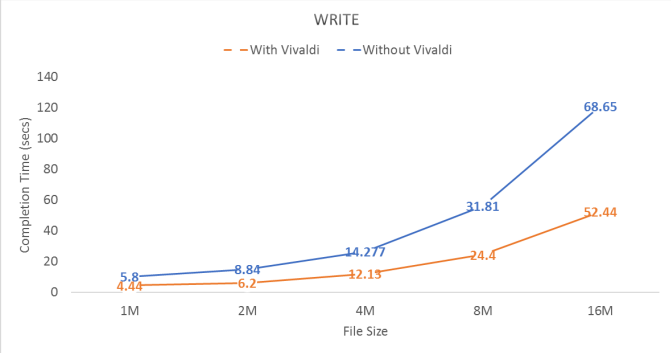

Fig. 7: Write Performance Comparison

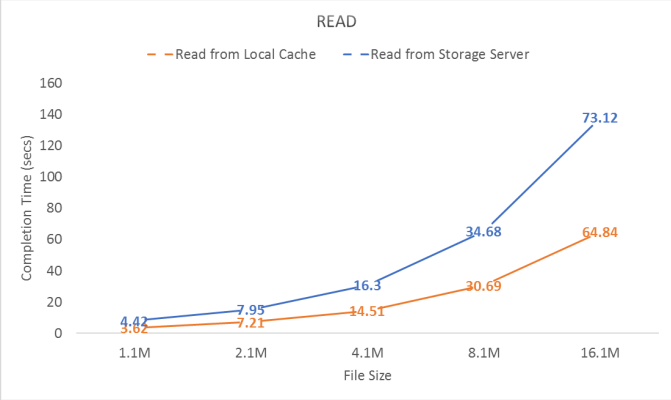

Fig. 8: Read Performance VCL

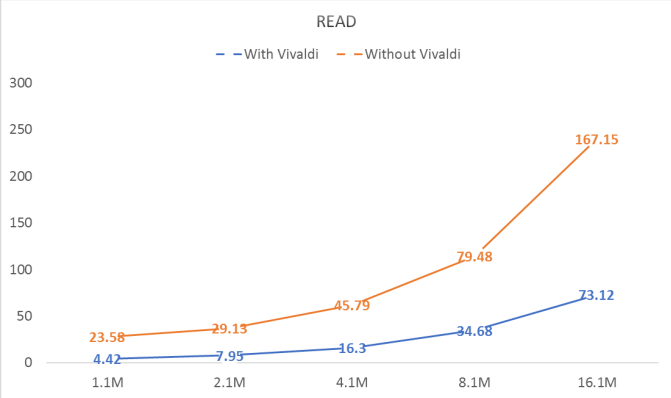

Fig. 9: Non-Cached Read Performance Comparison

respect to the VCL server. Figure 9 shows the comparison between read performances of VCL and AWS servers are made. Drastic improvements in reads are observed when vivaldi algorithm is employed to choose a lower-latency server. However, this read performance is only marginally worse than cached-reads. This leads us to conclude that when the network latency is low, the impact of having a local cache is less pronounced. Local cache provides significant benefits when network latency is very high.

VII. Conclusion

Perfuse provides a framework to effectively manage a pool of storage nodes. A file system is one such service provided using the framework. The read performance is greatly enhanced from having a local cache with the client. However, this benefit is less pronounced when the network latency is low. The system also offers a configurable replication factor to meet application needs. Triggering of a switchover event is done automatically by monitoring constant heart beat messages from storage nodes. The system employs a custom algorithm to minimize the impact of network latency by determining the closest storage server to serve client requests.Future improvements to the system may include more optimization to metadata storage to enable better scalability and remplement core modules in C++ to reduce the overheaded of Java and Python. Although REST APIs provide low file system performance, they can be used to effectively automate workflow management. The industry, in recent years, have been REST APIs to automate storage tasks such provisioning, data protection, replication. gRPC provides better performance than REST APIs and may well play a key-role in promoting software-defined storage.

References

[1] Frank Dabek, Russ Cox, Frans Kaashoek, Robert Morris, *Vivaldi: A Decentralized Network Coordinate System*, Cambridge, MA

[2] *Google gRPC*, http://www.grpc.io

[3] *Google Protocol Buffer*, https://developers.google.com/protocol-buffers

[4] Stribling, Jeremy and Sovran, Yair and Zhang, Irene and Pretzer, Xavid and Li, Jinyang and Kaashoek, M. Frans and Morris, Robert, *Flexible, Wide-area Storage for Distributed Systems with WheelFS*, 43–58, 2009. http://dl.acm.org/citation.cfm?id=1558977.1558981