Scalable Storage for Data-Intensive Computing

Abhishek Verma, Shivaram Venkataraman, Matthew Caesar, and Roy H. Campbell

Abstract Cloud computing applications require a scalable, elastic and fault tolerant storage system. We survey how storage systems have evolved from the traditional distributed filesystems, peer-to-peer storage systems and how these ideas have been synthesized in current cloud computing storage systems. Then, we describe how metadata management can be improved for a file system built to support large scale data-intensive applications. We implement Ring File System (RFS), that uses a single hop Distributed Hash Table, to manage file metadata and a traditional client-server model for managing the actual data. Our solution does not have a single point of failure, since the metadata is replicated. The number of files that can be stored and the throughput of metadata operations scales linearly with the number of servers. We compare our solution against two open source implementations of the Google File System (GFS): HDFS and KFS and show that it performs better in terms of fault tolerance, scalability and throughput. We envision that similar ideas from peer-to-peer systems can be applied to other large scale cloud computing storage systems.

1 Introduction

Persistent storage is a fundamental abstraction in computing. It consists of a named set of data items that come into existence through explicit creation, persist through temporary failures of the system, until they are explicitly deleted. Sharing of data in distributed systems has become pervasive as these systems have grown in scale in terms of number of machines and the amount of data stored.

{verma7, venkata4, caesar, rhc}@illinois.edu
Department of Computer Science,
University of Illinois at Urbana-Champaign,
201 N. Goodwin Avenue, Urbana, IL 61801.

The phenomenal growth of web services in the past decade has resulted in many Internet companies needing to perform large scale data analysis such as indexing the contents of the billions of websites or analyzing terabytes of traffic logs to mine usage patterns. A study into the economics of distributed computing [1] published in 2008, revealed that the cost of transferring data across the network is relatively high. Hence moving computation near the data is a more efficient computing model and several large scale, data-intensive application frameworks [2, 3] exemplify this model.

The growing size of the datacenter also means that hardware failures occur more frequently making such data analysis much harder. A recent presentation about a typical Google datacenter reported that up to 5% of disk drives fail each year and that every server restarts at least twice a year due to software or hardware issues [4]. With the size of digital data doubling every 18 months [5], it is also essential that applications are designed to scale and meet the growing demands.

To deal with these challenges, there has been a lot of work on building large scale distributed file systems. Distributed data storage has been identified as one of the challenges in cloud computing [6]. An efficient distributed file system needs to:

1. provide large bandwidth for data access from multiple concurrent jobs
2. operate reliably amidst hardware failures
3. be able to scale to many millions or billions of files and thousands of machines

The Google File System (GFS) [7] was proposed to meet the above requirements and has since been cloned in open source projects such as Hadoop Distributed File System (HDFS)[1] and Kosmos File System (KFS)[2] that are used by companies such as Yahoo, Facebook, Amazon, Baidu, etc.

The GFS architecture was picked for its simplicity and works well for hundreds of terabytes with few millions of files [8]. One of the direct implications of storing all the metadata in memory is that the size of metadata is limited by the memory available. A typical GFS master is capable of handling a few thousand operations per second [8] but when massively parallel applications like a MapReduce [2] job with many thousand mappers need to open a number of files, the GFS master becomes overloaded. Though the probability of a single server failing in a datacenter is low and the GFS master is continuously monitored, it still remains a single point of failure for the system. With storage requirements growing to petabytes, there is a need for distributing the metadata storage to more than one server.

Having multiple servers to handle failure would increase the overall reliability of the system and reduce the downtime visible to clients. As datacenters grow to accommodate many thousands of machines in one location, distributing the metadata operations among multiple servers would be necessary to increase the throughput. Handling metadata operations efficiently is an important aspect of the file system as they constitute up to half of file system workloads [9]. While I/O bandwidth available for a distributed file system can be increased by adding more data stor-

[1] http://hadoop.apache.org/

[2] http://kosmosfs.sourceforge.net/

age servers, scaling metadata management involves dealing with consistency issues across replicated servers.

Peer-to-peer storage systems [10], studied previously, provide decentralized control and tolerance to failures in untrusted-Internet scale environments. Grid Computing has been suggested as a potential environment for peer-to-peer ideas [11]. Similarly we believe that large scale cloud computing applications could benefit by adopting peer-to-peer system designs. In this work, we address the above mentioned limitations and present the design of Ring File System (RFS), a distributed file system for large scale data-intensive applications. In RFS, the metadata is distributed among multiple replicas connected using a Distributed Hash Table (DHT). This design provides better fault tolerance and scalability while ensuring a high throughput for metadata operations from multiple clients.

The major contributions of our work include:

1. A metadata storage architecture that provides fault tolerance, improved throughput and increased scalability for the file system.
2. Studying the impact of the proposed design through analysis and simulations.
3. Implementing and deploying RFS on a 16-node cluster and comparison with HDFS and KFS.

The rest of this chapter is organized as follows: We first provide a background on two popular traditional distributed file systems NFS and AFS in Section 2. Then, we discuss how peer-to-peer system ideas have been used to design distributed storage systems in Section 3. Section 4 discusses how current cloud storage systems are designed based on the amalgamation of ideas from traditional and P2P-based file systems. We describe the design of our system, RFS, in Section 5 and analyze its implications in Section 6. We then demonstrate the scalability and fault tolerance of our design through simulations followed by implementation results in Section 7. Section 8 summarizes the metadata management techniques in existing distributed file system and their limitations. We discuss possible future work and conclude with Section 9.

2 Traditional Distributed Filesystems

In this section, we examine two traditional distributed filesystems NFS and AFS and focus on their design goals and consistency mechanisms. Traditional distributed filesystems are typically geared towards providing sharing capabilities among multiple (human) users under a common administrative domain.

2.1 NFS

The NFS [12] protocol has been an industry standard since its introduction by Sun Microsystems in the 1980s. It allows remote clients to mount file systems over the network and interact with those file systems as if they were mounted locally. Although the first implementation of NFS was in a Unix environment, NFS is now implemented within several different OS environments. File manipulation primitives supported by NFS are: read, write, create a file or directory, remove a file or directory. The NFS protocol is designed to be machine, operating system, network architecture, and transport protocol independent. This independence is achieved through the use of Remote Procedure Call (RPC) primitives built on top of an eXternal Data Representation (XDR).

NFS uses the Virtual file system (VFS) layer to handle local and remote files. VFS provides a standard file system interface and allows NFS to hide the difference between accessing local or remote file systems.

NFS is a stateless protocol, i.e., the server does not maintain the state of files – there are no open or close primitives in NFS. Hence, each client request contains all the necessary information for the server to complete the request and the server responds fully to every client's request without being aware of the conditions under which the client is making the request. Only the client knows the state of a file for which an operation is requested. If the server and/or the client maintains state, the failure of a client, server or the network is difficult to recover from.

NFS uses the mount protocol to access remote files, which establishes a local name for remote files. Thus, the users access remote files using local names, while the OS takes care of the mapping to remote names. Most NFS implementations provide session semantics for performance reasons – no changes are visible to other processes until the file is closed. Using local caches greatly improves performance at the cost of consistency and reliability. Different implementations use different caching policies. Sun's implementation allows cache data to be stable for up to 30 seconds. Applications can use locks in order to ensure consistency.

Unlike earlier versions, the NFS version 4 protocol supports traditional file access while integrating support for file locking and the mount protocol. In addition, support for strong security (and its negotiation), compound operations, client caching, and internationalization have been added. Client checks cache validity when the file is opened. Modified data is written back to the server when the file is closed.

Parallel NFS (pNFS) is a part of the NFS v4.1 standard that allows clients to access storage devices directly and in parallel. The pNFS architecture eliminates the scalability and performance issues associated with NFS servers in earlier deployments. This is achieved by the separation of data and metadata, and moving the metadata server out of the data path.

2.2 AFS

The Andrew File System (AFS) was developed as a part of the Andrew project at Carnegie Mellon University. AFS is designed to allow users with workstations to share data easily. The design of AFS consists two components: a set of centralized file servers and a communication network, called Vice, and a client process named Venus that runs on every workstation. The distributed file system is mounted as a single tree in every workstation and Venus communicates with Vice to open files and manage the local cache.

The two main design goals of AFS are scalability and security. Scalability is achieved by caching relevant information in the clients to support a large number of clients per server. In the first version of AFS, clients cached the pathname prefix information and directed requests to the appropriate server. Additionally the file cache used in AFS-1 was pessimistic and verified if the cache was up to date every time a file was opened.

AFS-2 was designed to improve the performance and overcome some of the administrative difficulties found in AFS-1. The cache coherence protocol in AFS-2 assumes that the cache was valid unless notified by a callback. AFS-2 also introduces the notion of having data volumes to eliminate the static mapping from files to servers. Volumes consist of a partial subtree and many volumes are contained in a single disk partition. Furthermore, using volumes helps the design of other features like read-only snapshots, backups and per-user disk quotas. AFS-2 was used for around four years at CMU and experiments showed that its performance was better than NFS [13].

The third version of AFS was motivated by the need to support multiple administrative domains. Such a design could support a federation of cells but present users with a single unified namespace. AFS was also commercialized during this time and the development of AFS-3 was continued at Transarc Corporation in 1989. The currently available implementation of AFS is a community supported distribution named OpenAFS.

AFS has also played an important role in shaping the design of other distributed file system. Coda a highly-available distributed file system, also developed at CMU, is a descendant of AFS-2. The design of NFSv4 published in 2003 was also heavily influenced by AFS.

3 P2P-based Storage Systems

The need for sharing files over the Internet led to the birth of peer-to-peer systems like Napster and Gnutella. In this section, we describe the design of two storage systems based on Distributed Hash Tables (DHTs).

3.1 OceanStore

OceanStore [10] is a global persistent data store that aims to provide a consistent, highly available storage utility. There are two differentiating design goals:

1. the ability to be constructed from untrusted infrastructure; and
2. aggressive promiscuous caching

OceanStore assumes that the infrastructure is fundamentally untrusted – any server may crash without warning, leak information to third parties or be compromised. OceanStore caches data promiscuously anywhere, anytime, in order to provide faster access and robustness to network partitions. Although aggressive caching complicates data coherence and location, it provides greater flexibility to optimize locality and trades off consistency for availability. It also helps to reduce network congestion by localizing access traffic. Promiscuous caching requires redundancy and cryptographic techniques to ensure the integrity and authenticity of the data.

OceanStore employs a Byzantine-fault tolerant commit protocol to provide strong consistency across replicas. The OceanStore API also allows applications to weaken their consistency restrictions in exchange for higher performance and availability.

A version-based archival storage system provides durability. OceanStore stores each version of a data object in a permanent, read-only form, which is encoded with an erasure code and spread over hundreds or thousands of servers. A small subset of the encoded fragments are sufficient to reconstruct the archived object; only a global-scale disaster could disable enough machines to destroy the archived object.

The OceanStore introspection layer adapts the system to improve performance and fault tolerance. Internal event monitors collect and analyze information such as usage patterns, network activity, and resource availability. OceanStore can then adapt to regional outages and denial of service attacks, pro-actively migrate data towards areas of use and maintain sufficiently high levels of data redundancy.

OceanStore objects are identified by a globally unique identifier (GUID), which is the secure hash (for e.g., SHA1 [14]) of the owner's key and a human readable name. This scheme allows servers to verify and object's owner efficiently and facilitates access checks and resource accounting.

OceanStore uses Tapestry [15] to store and locate objects. Tapestry is a scalable overlay network, built on TCP/IP, that frees the OceanStore implementation from worrying about the location of resources. Each message sent through Tapestry is addressed with a GUID rather than an IP address; Tapestry routes the message to a physical host containing a resource with that GUID. Further, Tapestry is locality aware: if there are several resources with the same GUID, it locates (with high probability) one that is among the closest to the message source.

3.2 PAST

PAST is a large scale, decentralized, persistent, peer-to-peer storage system that aims to provide high availability and scalability. PAST is composed of nodes connected to the Internet and an overlay routing network among the nodes is constructed using Pastry [16]. PAST supports three operations:

1. **Insert**: Stores a given file a *k* different locations in the PAST network. *k* represents the number of replicas created and can be chosen by the user. The file identifier is generated using a SHA-1 hash of the file name, the owner's public key and a random salt to ensure that they are unique. *k* nodes which have an identifier closest to the fileId are selected to store the node.
2. **Lookup**: Retrieves a copy of the file from the nearest available replica.
3. **Reclaim**: Reclaims the storage of *k* copies of the file but does not guarantee that the file is no longer available.

Since node identifiers and file identifiers are uniformly distributed in their domains, the number of files stored by each node is roughly balanced. However, due to variation in the size of the inserted files and the capacity of each PAST node there could be storage imbalances in the system.

There are two schemes proposed to handle such imbalances. In the first scheme, called replica diversion, if one of the k closest nodes to the given fileId does not have enough space to store the file, a node from the leafset is chosen and a pointer is maintained in the original node. To handle failures, this pointer is also replicated. If no suitable node is found for replica diversion the entire request is reverted and the client is forced to choose a different fileId by using a different random salt. This scheme is called file diversion and is a costly operation. Simulations show that file diversion is required only for up to 4% of the requests when the leaf set size is 32.

The replication of a file in PAST, to *k* different locations, is to ensure high availability. However, some popular files could require more than *k* replicas to minimize latency and improve performance. PAST uses any space unused in the nodes to cache files which are frequently accessed. When a file is routed through a node during an insert or lookup operation, if the size of the file is less than a fraction of the available free space, it is cached. The cached copies of a file are maintained in addition to the *k* replicas and are evicted when space is required for a new file.

4 Cloud Storage Systems

Motivated by the need for processing terabytes of data generated by systems, cloud storage systems need to provide high performance at large scale. In this section, we examine two cloud computing storage systems.

4.1 Google File System

Around 2000, Google designed and implemented the Google File System (GFS) [7] to provide support for large, distributed, data-intensive applications. One of the requirements was to run on cheap commodity hardware and deliver good aggregate throughput to a large number of clients. GFS is designed for storing a modest (millions) number of huge files. It is optimized for large streaming reads and writes. Though small random reads and writes are supported, it is a non-goal to perform them efficiently.

The GFS architecture comprises of a single GFS *master* server which stores the file metadata of the file system and multiple slaves known as *chunkservers* which store the data. The GFS master stores the metadata, which consists of information such as file names, size, directory structure and block locations, in memory. The chunkservers periodically send heartbeat messages to the master to report their state and get instructions.

Files are divided into chunks (usually 64 MB in size) and the GFS master manages the placement and data-layout among the various chunkservers. A large chunk size reduces the overhead of the client interacting with the master to find out its location. For reliability, each chunk is replicated on multiple (by default three) chunkservers.

Having a single master simplifies the design and enables the master to make sophisticated chunk placement and replication decisions using global knowledge. It's involvement in reads and writes is minimized so that it does not become a bottleneck. Clients never read and write file data through the master. Instead, a client asks the master which chunkservers it should contact. Clients cache this information (for a limited time) and directly communicate with the chunkservers for subsequent operations.

GFS supports a relaxed consistency model that is simple and efficient to implement at scale. File namespace mutations at the master are guaranteed to be atomic. Record append causes data to be appended atomically at least once even in the presence of concurrent mutations. Since clients cache chunk locations, they may read stale data. However, this window is limited by the cache entry's timeout and the next open of the file. Data corruption (like bit rot) is detected through checksumming by the chunkservers.

4.2 Dynamo

Dynamo [17] is a distributed key-value store used at Amazon and is primarily built to replace relational databases with a key-value store having eventual consistency.

Dynamo uses a synthesis of well known techniques to achieve the goals of scalability and availability. It is designed taking churn into account: storage nodes can be added or removed without requiring any manual partitioning or redistribution. Data is partitioned and replicated using consistent hashing, and consistency is provided using versioning. Vector clocks with reconciliation during reads are used to provide high availability for writes. Consistency among replicas during failures is maintained by a quorum-like replica synchronization protocol. Dynamo uses a gossip based distributed failure detection and membership protocol.

The basic consistent hashing algorithm assigns a random position for each node on the ring. This can lead to non-uniform data and load distribution and is oblivious to heterogeneity. Hence, Dynamo uses “virtual nodes”: a virtual node looks like a single node in the system, but each node can be responsible for more than one virtual node. When a new node is added to the system, it is assigned multiple positions in the ring.

Dynamo provides eventual consistency, which allows for updates to be propagated to all replicas asynchronously. A write request can return to the client, before the update has been applied at all the replicas, which can result in scenarios where a subsequent read operation may return stale data. In the absence of failures, there is a bound on the update propagation times. However under certain failures (like network partitions), updates may not propagate to all replicas for a long time. Dynamo shows that an eventually consistent storage system can be a building block for highly available applications.

5 RFS Design

In this section, we present the design of Ring File System (RFS), a distributed file system for large scale data-intensive applications. In RFS, the metadata is distributed among multiple replicas connected using a Distributed Hash Table (DHT). This design provides better fault tolerance and scalability while ensuring a high throughput for metadata operations from multiple clients.

Our architecture consists of three types of nodes: *metaservers*, *chunkservers* and *clients* as shown in the Figure 1. The metaservers store the metadata of the file system whereas the chunkservers store the actual contents of the file. Every metaserver has information about the locations of all the other metaservers in the file system. Thus, the metaservers are organized in a single hop Distributed Hash Table (DHT). Each metaserver has an identifier which is obtained by hashing its *MAC* address.

Chunkservers are grouped into multiple cells and each cell communicates with a single metaserver. This grouping can be performed in two ways. The chunkserver can compute a hash of its *MAC* address and connect to the metaserver that is its successor in the DHT. This makes the system more self adaptive since the file system is symmetric with respect to each metaserver. The alternative is to configure each chunkserver to connect to a particular metaserver alone. This gives more control over the mapping of chunkservers to metaservers and can be useful in configuring geographically distributed cells each having its own metaserver.

The clients distribute the metadata for the files and directories over the DHT by computing a hash of the parent path present in the file operation. Using the parent path implies that the metadata for all the files in a given directory is present at the same metaserver. This makes listing the contents of a directory efficient and is commonly used by MapReduce [2] and other cloud computing applications.

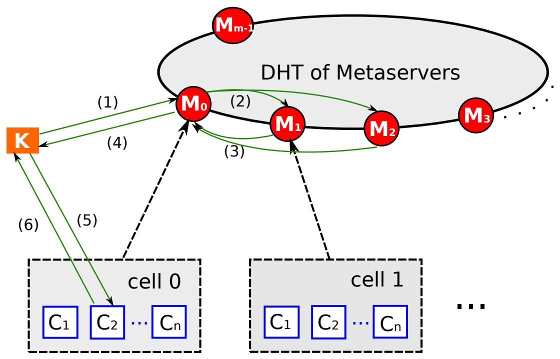

Fig. 1 Architecture of RingFS

5.1 Normal Operation

We demonstrate the steps involved in the creation of a file, when there are no failures in the system. The sequence of operations shown in Figure 1 are:

1. Client wishes to create a file named */dir1/dir2/filename*. It computes a hash of the parent path, */dir1/dir2*, to determine that it has to contact metaserver M_0 for this file operation.
2. Client issues a create request to this metaserver which adds a record to its *metatable* and allocates space for the file in Cell 0.
3. Before returning the response back to the client, M_0 sends a replication request to r of its successors, $M_1, M_2, \cdots, M_r$ in the DHT to perform the same operation on their *replica_metatable*.
4. All of the successor metaservers send replies to M_0. Synchronous replication is necessary to ensure consistency in the event of failures of metaservers.
5. M_0 sends back the response to the client.
6. Client then contacts the chunkserver and sends the actual file contents.
7. Chunkserver stores the file contents.

Thus, in all r metadata Remote Procedure Calls (RPCs) are needed for a write operation. If multiple clients try to create a file or write to the same file, consistency is ensured by the fact that these mutable operations are serialized at the primary metaserver for that file.

The read operation is similarly performed by using the hash of the parent path to determine the metaserver to contact. This metaserver directly replies with the metadata information of the file and the location of the chunks. The client then communicates directly with the chunkservers to read the contents of the file. Thus, read operations need a single metadata RPC.

5.2 *Failure and Recovery*

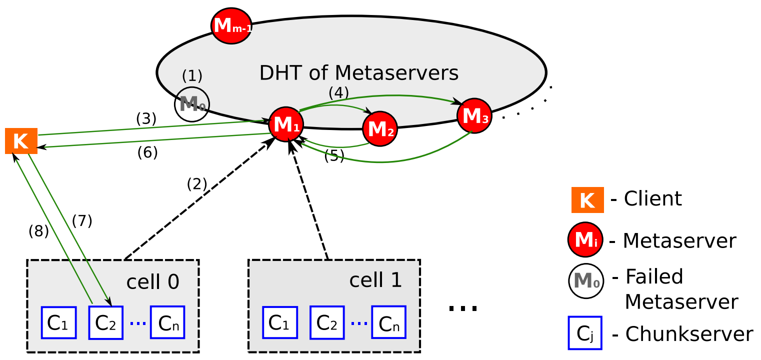

Fig. 2 Tolerating metaserver failures

Let us now consider a case now where metaserver M_0 has failed. The chunkservers in cell 0, detect the failure through heartbeat messages and connect to the next server M_1 in the DHT. When a client wishes to create a file its connection is now handled by M_1 in place of M_0. We replicate the metadata to r successive servers for this request. M_1 also allocates space for the file in cell 0 and manages the layout and replication for the chunkservers in cell 0.

Once M_0 recovers, it sends a request to its neighboring metaservers $M_1, M_2, \cdots, M_r$ to obtain the latest version of the metadata. On receipt of this request, M_1 sends the metadata which belongs to M_0 and also closes the connection with the chunkservers in cell 0. The chunkservers now reconnect to M_0 which takes over the layout management for this cell and verifies the file chunks based on the latest metadata version obtained. Also, M_r lazily deletes the $(r+1)^{th}$ copy of the metadata. Thus, our design guarantees strict consistency through the use of synchronous replication. In order to withstand the failure of a machine or a rack in a datacenter, a suitable number of replicas can be chosen using techniques from Carbonite [18].

If instead, M_k, one of the r successors of M_0 fails in step 3, then M_0 retries the replication request a fixed number of times. In the mean time, the underlying DHT stabilization protocol updates the routing table and M_{k+1} handles the requests directed to the namespace previously serviced by M_k. If M_0 is unable to replicate the metadata to r successors, then it sends an error message back to the client.

6 Analysis

In this section, we present a mathematical analysis comparing the design of GFS and RFS with respect to the scalability, throughput followed by failure analysis.

m : Number of metaservers
R, W : Baseline Read and Write throughputs
r : Number of times the metadata is replicated

Metric	**GFS**	**RFS**
Metaserver failures that can be tolerated	0	$r-1$
RPCs required for a read	1	1
RPCs required for a write	1	r
Metadata throughput for reads	R	$R \cdot m$
Metadata throughput for writes	W	$W \cdot m/r$

Table 1 Analytical comparison of GFS and RFS.

6.1 Design Analysis

Let the total number of machines in the system be n. In GFS, there is exactly 1 metaserver and the remaining $n-1$ machines are chunkservers that store the actual data. Since there is only one metaserver, the metadata is not replicated and the file system cannot survive the crash of the metaserver.

In RFS, we have m metaservers that distribute the metadata r times. RFS can thus survive the crash of $r-1$ metaservers. Although a single Remote Procedure Call (RPC) is enough for the lookup using a hash of the path, r RPCs are needed for the creation of the file, since the metadata has to be replicated to r other servers. Since m metaservers can handle the read operations for different files, the read metadata throughput is m times that of GFS. Similarly, the write metadata throughput is m/r times that of GFS, since it is distributed over m metaservers, but replicated r times. This analysis is summarized in Table 1.

6.2 Failure Analysis

Failures are assumed to be independent. This assumption is reasonable because we have only tens of metaservers and they are distributed across racks and potentially different clusters. We ignore the failure of chunkservers in this analysis since it has the same effect on both the designs and simplifies our analysis. Let $f = 1/MTBF$

be the probability that the meta server fails in a given time, and let R_g be the time required to recover it. The file system is unavailable for $R_g \cdot f$ of the time. If GFS is deployed with a hot standby master replica, GFS is unavailable for $R_g \cdot f^2$ of the time, when both of them fail. For example, if the master server fails once a month and it takes 6 hours for it to recover, then the file system availability with a single master is 99.18% and increases to 99.99% with a hot standby.

Let m be the number of metaservers in our system, r be the number of times the metadata is replicated, f be the probability that a given server fails in a given time t and R_r be the time required to recover it. Since the recovery time of a metaserver is proportional to the amount of metadata stored on it and we assume that the metadata is replicated r times, R_r will be roughly equal to $r \cdot R_g/n$. The probability that any r consecutive metaservers in the ring go down is $mf^r(1-f)^{m-r}$. If we have $m = 10$ metaservers, $r = 3$ copies of the metadata and $MTBF$ is 30 days, then this probability is 0.47%. However, a portion of our file system is unavailable if and only if all the replicated metaservers go down within the recovery time of each other. This happens with a probability of $F_r = m \cdot f \cdot \left(\frac{f \cdot R_r}{t}\right)^{r-1} \cdot (1-f)^{m-r}$, assuming that the failures are equally distributed over time. The file system is unavailable for $F_r \cdot R_r$ of the time. Continuing with the example and substituting appropriate values, we find that the recovery time would be 1.8 hours and the availability is 99.9994%.

7 Experiments

In this section, we present experimental results obtained from our prototype implementation of RFS. Our implementation is based on KFS and has modified data structures for metadata management and the ability for metaservers to recover from failures by communicating with its replicas. To study the behavior on large networks of nodes, we also implemented a simulation environment.

All experiments were performed on sixteen 8-core HP DL160 (Intel Xeon 2.66GHz CPUs) with 16GB of main memory, running CentOS 5.4. The MapReduce implementation used was Hadoop 0.20.1 and was executed using Sun's Java SDK 1.6.0. We compare our results against Hadoop Distributed File System (HDFS) that accompanied the Hadoop 0.20.1 release and Kosmos File system (KFS) 0.4. For the HDFS and KFS experiments, a single server is configured as the metaserver and the other 15 nodes as chunkservers. RFS is configured with 3 metaservers and 5 chunkservers connecting to each of them. We replicate the metadata three times in our experiments.

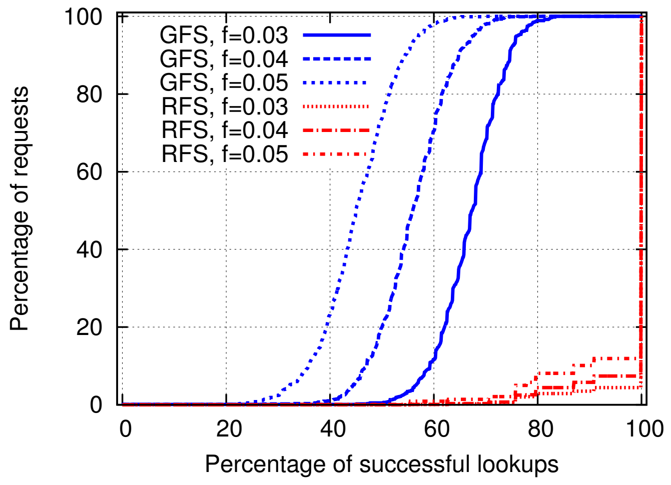

Fig. 3 CDF of number of successful lookups for different failure probabilities

7.1 Simulation

Fault tolerance of a design is difficult to measure without a large scale deployment. Hence, we chose to model the failures that occur in datacenters using a discrete iterative simulation. Each metaserver is assumed to have a constant and independent failure probability.

The results show that RFS has better fault tolerance than the single master (GFS) design. In the case of GFS, if the metaserver fails, the whole file system is unavailable and the number of successful lookups is 0 till it recovers after some time. In RFS, we configure 10 metaservers and each fails independently. The metadata is replicated on the two successor metaservers. Only a part of the file system is unavailable only when three successive metaservers fail. Figure 3 shows a plot of the CDF of the number of successful lookups for GFS and RFS for different probabilities of failure. As the failure probability increases, the number of successful lookups decreases. Less than 10% of the lookups fail in RFS in all the cases.

7.2 Fault Tolerance

The second experiment demonstrates the fault tolerance of our implementation. A client sends 150 metadata operations per second and the number of successful operations is plotted over time for GFS, KFS and RFS in Figure 4. HDFS achieves a steady state throughput, but when the metaserver is killed, the complete file system become unavailable. Around $t = 110s$, the metaserver is restarted and it recovers from its checkpointed state and replays the logs of operations that couldn't be checkpointed. The spike during the recovery happens because the metaserver buffers the requests till it is recovering and batches them together. A similar trend is observed in the case of KFS, in which we kill the metaserver at $t = 70s$ and restart it at $t = 140s$.

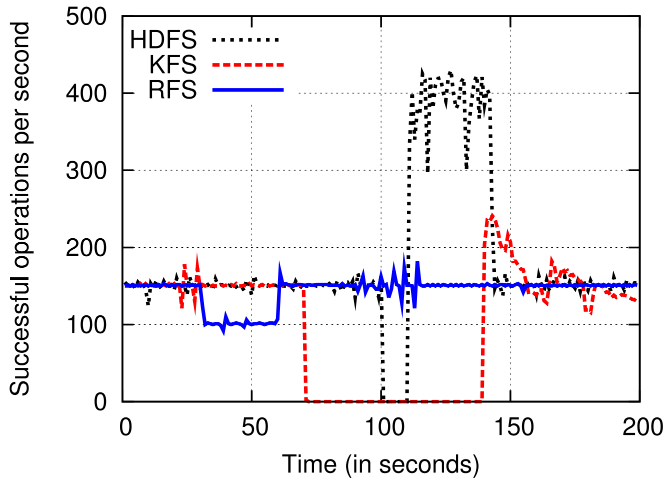

Fig. 4 Fault tolerance of HDFS, KFS and RFS

For testing the fault tolerance of RFS, we kill one of the three metaservers at $t = 20s$ and it does not lead to any decline in the throughput of successful operations. At $t = 30s$, we kill another metaserver, leaving just one metaserver leading to a drop in the throughput. At $t = 60s$, we restart the failed metaserver and the throughput stabilizes to its steady state.

7.3 Throughput

The third experiment demonstrates the metadata throughput performance. A multi-threaded client is configured to spawn a new thread and perform read and write metadata operations at the appropriate frequency to achieve the target qps. We then measure how many operations complete successfully each second and use this to compute the server's capacity. Figure 5 shows the load graph comparison for HDFS, KFS and RFS. The throughput of RFS is roughly twice that of HDFS and KFS and though the experiment was conducted with 3 metaservers, the speed is slightly lesser due to the replication overhead. Also, the performance of HDFS and KFS are quite similar and RFS with no metadata replication has the same performance as KFS.

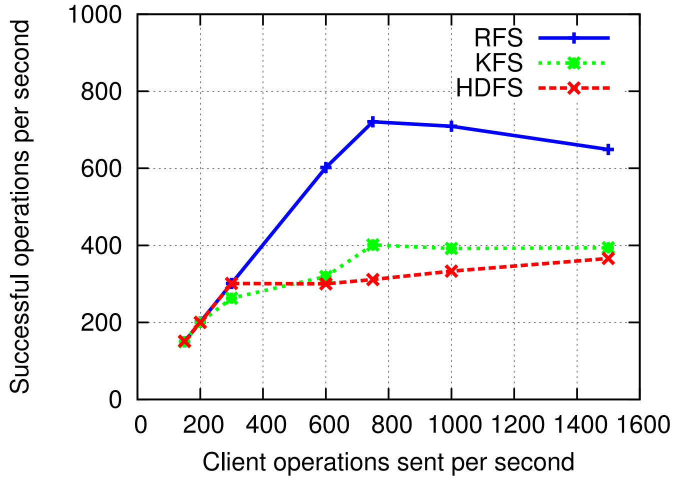

Fig. 5 Comparison of throughput under different load conditions

7.4 MapReduce Performance

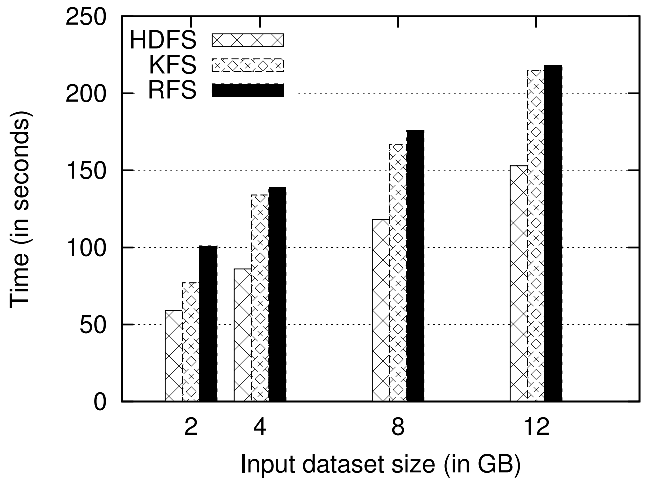

Fig. 6 MapReduce application - Wordcount

We ran a simple MapReduce application that counts the number of words on a Wikipedia dataset and varied the input dataset size from 2GB to 16GB. We measured the time taken for the job to compute on all three file system and a plot of the same is shown in Figure 6. We observed that for a smaller dataset the overhead of replicating the metadata increased the time taken to run the job, but on larger datasets the running times were almost the same for KFS and RFS.

8 Comparison with Related Work

Metadata management has been implemented in file systems such as NFS and AFS [13] by statically partitioning the directory hierarchy to different servers. This, however, requires an administrator to assign directory subtrees to each server but enables clients to know easily which servers have the metadata for a give file name. Techniques of hashing a file name or the parent directory name to locate a server have been previously discussed in file systems such as Vesta [19] and Lustre [20]. Ceph [21], a petabyte scale file system, uses a dynamic metadata distribution scheme where subtrees are migrated when the load on a server increases. Hashing schemes have been found to be inefficient while trying to satisfy POSIX directory access semantics as this would involve contacting more than one server. However studies have shown that most cloud computing applications do not require strict POSIX semantics [7] and with efficient caching of metadata on the clients, the performance overhead can be overcome.

8.1 Peer-to-Peer File systems

File systems such as PAST [22] and CFS [23] have been built on top of DHTs like Pastry [16] and Chord [24], but they concentrate on storage management in a peer-to-peer system with immutable files. Ivy [25] is a read/write peer-to-peer file system which uses logging and DHash. A more exhaustive survey of peer-to-peer storage techniques for distributed file systems can be found here [26].

Our work differs from these existing file systems in two aspects: (1) Consistency of metadata is crucial in a distributed file system deployed in a datacenter and our design provides stricter consistency guarantees than these systems through synchronous replication. (2) Existing peer-to-peer file systems place blocks randomly, while some can exploit locality. Our system can implement more sophisticated placement policies (e.g: placing blocks on the closest and the least loaded server), since the group of servers which store the metadata have global information about the file system.

8.2 Distributed Key-value Stores

Recently, there have been some efforts in deploying peer-to-peer like systems in distributed key-value stores used in datacenters. Cassandra [27] is a distributed key-value store that has been widely used and provides clients with a simple data model and eventual consistency guarantees. Cassandra provides a fully distributed design like Dynamo with a column-family based data model like Bigtable [28]. Key-value stores are often useful for low latency access to small objects that can tolerate eventual consistency guarantees. RingFS on the other hand, tries to address the problems associated with storing the metadata for large files in a hierarchical file system and provides stronger consistency guarantees.

9 Conclusion

Today's cloud computing storage systems need to be scalable, elastic and fault tolerant. We surveyed how storage systems have evolved from traditional distributed filesystems (NFS and AFS) and peer-to-peer storage systems (OceanStore and PAST), and how these ideas have been synthesized in current cloud computing storage systems (GFS and Dynamo).

We presented and evaluated RFS, a scalable, fault-tolerant and high throughput file system that is well suited for large scale data-intensive applications. RFS can tolerate the failure of multiple metaservers and it can handle a large number of files. The number of files that can be stored in RFS and the throughput of metadata operations scales linearly with the number of servers. RFS performs better than HDFS and KFS in terms of fault tolerance, scalability and throughput.

Peer-to-peer systems are decentralized and self-organizing. Thus, they are attractive for datacenters built with commodity components with high failure rates, especially as the size of datacenters increases. We have shown how using a single-hop Distributed Hash Table to manage its metadata from peer-to-peer systems can be combined together with the traditional client server model for managing the actual data. We envision that more ideas from peer-to-peer systems research can be applied for building systems that scale to large datacenters with hundreds of thousands of machines distributed at multiple sites.

References

1. J. Gray, "Distributed computing economics," *Queue*, vol. 6, no. 3, pp. 63–68, 2008.
2. J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
3. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *EuroSys '07: Proc. of the 2nd ACM SIGOPS*, New York, NY, USA, 2007, pp. 59–72.
4. J. Dean, "Large-Scale Distributed Systems at Google: Current Systems and Future Directions," 2009.
5. J. Gantz and D. Reinsel, "As the economy contracts, the Digital Universe expands," *IDC Multimedia White Paper*, 2009.

6. M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, “Above the Clouds: A Berkeley View of Cloud Computing,” EECS Department, University of California, Berkeley, Tech. Rep., 2009.
7. S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google file system,” *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.
8. M. K. McKusick and S. Quinlan, “GFS: Evolution on Fast-forward,” *Queue*, vol. 7, no. 7, pp. 10–20, 2009.
9. D. Roselli, J. Lorch, and T. Anderson, “A comparison of file system workloads,” in *Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX Association, 2000.
10. J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, “Oceanstore: An architecture for global-scale persistent storage,” in *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.
11. J. Ledlie, J. Shneidman, M. Seltzer, and J. Huth, “Scooped, again,” *Lecture notes in computer science*, pp. 129–138, 2003.
12. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, “Design and implementation of the sun network filesystem,” in *Proceedings of the Summer 1985 USENIX Conference*, 1985, pp. 119–130.
13. J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, “Scale and performance in a distributed file system,” *ACM Transactions on Computer Systems (TOCS)*, vol. 6, no. 1, pp. 51–81, 1988.
14. D. Eastlake and P. Jones, “US secure hash algorithm 1 (SHA1),” RFC 3174, September, Tech. Rep., 2001.
15. B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz, “Tapestry: a resilient global-scale overlay for service deployment,” in *IEEE J. Selected Areas in Communications*, January 2003.
16. A. Rowstron and P. Druschel, “Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems,” in *ACM Middleware*, November 2001.
17. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: Amazon’s highly available key-value store,” *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, p. 220, 2007.
18. B. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. Kaashoek, J. Kubiatowicz, and R. Morris, “Efficient replica maintenance for distributed storage systems,” in *Proc. of NSDI*, vol. 6, 2006.
19. P. Corbett and D. Feitelson, “The Vesta parallel file system,” *ACM Transactions on Computer Systems (TOCS)*, vol. 14, no. 3, pp. 225–264, 1996.
20. P. Schwan, “Lustre: Building a file system for 1000-node clusters,” in *Proceedings of the 2003 Linux Symposium*, 2003.
21. S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn, “Ceph: A scalable, high-performance distributed file system,” in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.
22. P. Druschel and A. Rowstron, “PAST: A large-scale, persistent peer-to-peer storage utility,” in *Proc. HotOS VIII*, 2001, pp. 75–80.
23. F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica, “Wide-area cooperative storage with CFS,” *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 202–215, 2001.
24. I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, “Chord: a scalable peer-to-peer lookup service for Internet applications,” in *ACM SIGCOMM*, August 2001.
25. T. M. G. Athicha Muthitacharoen, Robert Morris and B. Chen, “Ivy: A Read/Write Peer-to-Peer File System,” in *OSDI*, December 2002.
26. R. Hasan, Z. Anwar, W. Yurcik, L. Brumbaugh, and R. Campbell, “A survey of peer-to-peer storage techniques for distributed file systems,” in *ITCC*, vol. 5, pp. 205–213.
27. A. Lakshman and P. Malik, “Cassandra: structured storage system on a P2P network,” in *Proc. of the 28th ACM symposium on Principles of distributed computing*, 2009.

28. F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI06)*, 2006.

Acknowledgements This work was funded in part by NSF IIS grant 0841765 and in part by NSF CCF grant 0964471. The views expressed are those of the authors only.