Distributed file systems: a current overview and future outlook

Boudewijn Braams
University of Amsterdam/Vrije Universiteit
UvANetID: 10401040/VUnetID: bbs820 (2527663)
bbx1992@gmail.com

***Abstract*—Many applications require access to data sets that do not fit on a single server. Additionally, such application might have varying degrees of performance, availability and fault-tolerance requirements. One way to address these issues is by making use of a distributed file system (DFS). In this paper we give an overview of several established DFS solutions (GFS, HDFS, GlusterFS, Ceph, Lustre and cloud-based object storage), while also taking a look at future challenges we expect to be relevant for the advancement of DFS design. For the established solutions, we highlight differences in their design and outline characterizing aspects. For the future challenges, we discuss novel research projects and how these attempt to tackle the challenges. Besides focussing on fault tolerance and scalability characteristics, we briefly consider the solutions in the context of exa-scale computing to see whether they would be appropriate.**

I. Introduction

The capacity of storage devices has been ever growing since the inception of the first hard disk drives (HDD's) more than 60 years ago. State-of-the-art technology currently limits HDD storage capacity for a single device at around ∼10TB, with technology to support ∼100TB HDD's expected by 2025 [9] [1]. However, many modern applications deal with data sets sized beyond what fits on a single machine or server. Moreover, these applications potentially require varying degrees of performance, availability and fault-tolerance. To facilitate this, one has to make use of a *distributed file system* (DFS). DFSs are not a new technology, traditional solutions like the Network File System (NFS) have been around since the 1980s [31]. Although these traditional technologies are still being used, it are the developments in high-performance computing (HPC) and the advent of big data and cloud-based applications that have lead to the development of many novel DFS designs and architectures. Moreover, since we are slowly but steadily heading towards the era of *exa-scale* computing. It will be interesting to see how DFS designs are developing in this regard.

With this paper, we will provide an overview of several established state-of-the-art DFS solutions while also highlighting novel research projects and regarding them in an exa-scale computing context. We will start by giving a high-level overview of the main concepts of DFS design in Section II, in which we will briefly discuss desired properties, potential architectures, the CAP theorem and exa-scale computing. In Section III we will evaluate various established DFS solutions from a design perspective, reflecting back on the design aspects introduced in the previous section. Finally, in Section IV we will discuss future challenges and highlight recent research projects.

II. DFS design

In a traditional, non-distributed file system, the file system manages access to storage devices on a *single* machine, providing well known file system semantics for reading/writing files and managing hierarchical namespaces. It acts as the layer between an application or operating system (OS) and the block-based interface provided by the storage devices *directly* attached to the machine. The storage devices are thus mounted on a single machine and can only be accessed directly by this machine. However, in the case of a DFS, the file system manages access to storage devices attached to *multiple* machines or servers. In order for the file system to actually manage data stored on multiple servers, these servers will have to communicate with each other over a network. Thus in the case of a DFS, another layer is placed on top of the traditional file system in the form of a network protocol to facilitate the communication between servers. This additional layer of abstraction should allow for a DFS to mount storage devices physically attached to different servers on multiple client machines.

In this section we will touch upon the desired properties of a DFS, potential architectures, the CAP theorem and briefly discuss exa-scale computing and what it implies for DFS design.

A. Desired properties

From a design perspective, there are three properties that are desirable for a DFS, namely: *transparency*, *fault tolerance* and *scalability* [23]. *Transparency* means that ideally the complexity of the distributed file system should be completely abstracted away and thus be hidden. Interaction with the DFS should mimic that of interacting with a local file system, the user or client should not be concerned with the intricacies of where and how the data is distributed and/or replicated. *Fault tolerance* means that in the event of a transient server failure (e.g. a failing HDD) or partial network failure (i.e. network partition) the system should continue to function, ideally without any compromising of data integrity. Lastly, *scalability* means that the system be able to withstand high load and allow for new resources (such as servers) to be integrated into the system with relative ease.

B. Architecture

In essence, a DFS will always comprise of a set of servers. There are however many ways in which these can be arranged

architecturally to realize distributed storage. In Sections III and IV we will see various different approaches. From architectures in which all servers manage data *and* metadata to architectures in which data and metadata are handled by a different server or servers entirely. Moreover, one can choose handle client request directly on data/metadata servers or introduce a additional layer of servers acting as a proxy. It is important to note that these architectural decisions can have serious implications for both fault tolerance and scalability. We will therefore reflect on this the aforementioned sections.

C. CAP theorem

Since data in a DFS will be distributed among several servers and will likely have to be replicated in places to ensure availability in case of a server failure, the notion of state becomes blurry. In 2000, Brewer proposed the CAP theorem, which essentially suggests that it impossible for a distributed system (and thus a DFS) to provide *Consistency* (C), *Availability* (A) and *Partition tolerance* (P) guarantees all at the same time [16]. The theorem was formalized and proved correct in 2002 by Gilbert and Lynch and states that at most two out of the three guarantees can be provided simultaneously [22].

The probability of server failure increases with the number of servers used, some form of replication is thus required to provide fault-tolerance. However, since data is replicated, failures in such a concurrent system can lead to state inconsistencies. *Consistency* in this context refers to the notion that replicated copies are identical. More formally, the *strict* consistency guarantee states that there must be a *single* total order on all concurrent data operations such that to an outside observer, they appear as a single operation, essentially behaving as though the operation was performed on a single machine.

While occasional server failures are inevitable, it is desirable for a distributed system to be continuously available. *Availability* thus refers to the guarantee of a distributed system to always provide a response to a request eventually (i.e it should not fail or give an error).

Communication among servers happens via a network. Similar to how server failures are inevitable, it is likely that there will be errors and failures in the network (e.g. loss of network messages or corruption). *Partition tolerance* refers to the ability of a system to cope with partitioning within a network, a situation that can occur when a certain set of servers is unable to deliver network messages to other servers in the network. In accordance with this guarantee, only complete network failure should allow the system respond incorrectly. In any other case, the system should behave as expected.

The CAP theorem essentially divides systems into the following three categories (see Figure 1):

- **CA**: These systems offer consistency and availability guarantees, but sacrifice partition tolerance. Such systems are not designed to gracefully handle network failures. It is evident that this design is not a good fit when designing a large scale DFS. However, for example for a relational database running on a single machine, sacrificing partition tolerance is perfectly acceptable.
- **CP**: These systems offer consistency and partition tolerance, but sacrifice availability. Such systems take into account the unreliable nature of computer networks, and prefer responding with an error to a request over responding with stale state. When a system requires atomic reads and writes (e.g. in a banking system), a design like this is preferred.
- **AP**: These systems offer availability and partition tolerance, but sacrifice consistency. Such a design is applicable when the consistency of the data is not an integral part of for example the business value (e.g. in the case of social networks).

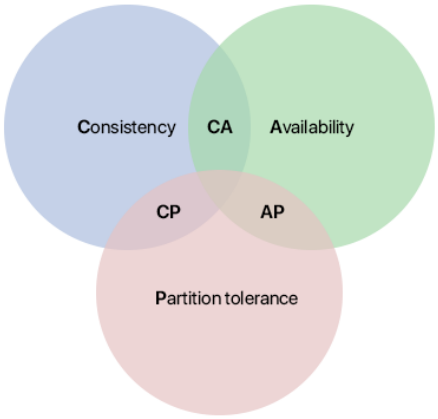

Fig. 1. Visual representation of the traditional CAP theorem

Given the seemingly hard limitations presented by the CAP theorem, the choice for a DFS will almost always come down to a CP or AP design, as a we would like our DFS to continue functioning in the case of network errors (i.e. we can not give up P). However, regarding the 'two out of three' formulation as a hard limitation on distributed system design is an oversimplification. In a 2012 article reflecting on his initial proposed idea, Brewer argues that the traditional notion of CAP has served its purpose (which was to *'open the minds of designers to a wider range of systems and tradeoffs'*) and that system designers should be freed of these perceived limitations [15]. In it he states that while networks can behave unreliably and partitions can occur, such events are relatively rare. Why should a system therefore have to choose between C or A when the system is *not* partitioned? Systems can be designed to deal with partitions explicitly and perform recovery afterwards. The choice between the three guarantees should not be thought of as a hard choice (e.g. choosing P means losing either C or A), but in terms of probabilities (e.g. the probability of P is deemed lower than other failures).

When a distributed system is able to detect partitions, it can explicitly account for them in a three stage process: partition detection, partition mode, recovery mode. In the first stage, the network partition must be detected and the system should enter partition mode. While in partition mode, the system should either impose limits on certain operations (reducing A) or record extra information regarding operations by storing the intent of an operation so it can be executed after partition recovery (hiding the reduction of A). In the last stage, the

system will have to reimpose C by essentially re-computing the state. Note that the last stage can lead to merge conflicts which can either required to be resolved manually or automatically (by imposing a deterministic choice on what option to choose).

Ultimately the key insight here is that when there is a network partition, there need not be a hard choice between either C or A: a choice between C and A can be made on sub system level granularity.

D. Exa-scale context

In the realm of HPC, supercomputers comprising of thousands of interconnected servers are being used to solve the world's most complex problems. We see application of HPC in both the scientific domain and industry: ranging from modeling global climate phenomenon to designing more efficient drugs. Currently, the state of the art in HPC is at the peta-scale (in the order of 10^{15} FLOPS[1]), first achieved in 2008 [6]. However, we now see an enormous increase in the size of commercial and scientific datasets. Consequently, it is likely that the current peta-scale technologies will not be able to handle this and that we will require new solutions to prepare ourselves for the next milestone: *exa*-scale computing (10^{18} FLOPS).

Exa-scale systems are expected to be realized by 2023 and will likely comprise of $\sim 100,000$ interconnected servers; simply scaling up peta-scale solutions will likely not suffice [36]. Evidently, this raises many challenges in how the required hardware but also how to design applications that can make use of that many computing nodes. However, what is relevant for this paper is how the large volumes of data involved will be stored. Exa-scale systems will need novel DFSs, sufficiently scalable to accommodate for the increase in memory capacity requirements. While not the solely focus of this paper, we will attempt to take this into account when discussing DFSs and recent research projects to see how well they might be suited for an exa-scale computing context.

III. Established DFS solutions

In this section we will start of with a brief description of the traditional approach to networked file systems in the form of NAS/SAN and why these techniques are by themselves not a good fit for modern DFS requirements. After that we will discuss several established DFS solutions (GFS, HDFS, GlusterFS, Ceph, Lustre) and will also regard cloud-based object storage as a higher level alternative. For each of the solutions we will describe the architectural design, characterizing aspects, relation to the CAP theorem and put them in an exa-scale context.

A. Traditional solutions

Before the advent of big data and cloud computing there were three main prevalent storage architectures: direct-attached storage (DAS), storage area networks (SANs) and network-attached storage (NAS) [26]. With DAS, the storage device is directly connected to server and accessed via the block-based I/O bus of the machine. The limitation of DAS is obvious, namely that access to a storage device is limited to the hosting server. Both SAN and NAS attempt to overcome this limitation by allowing the sharing of access to storage devices over a network. A SAN is a network in which *block-level* access to storage devices is shared among different servers. This is done via a specialized infrastructure comprising of servers and switches running a specialized network layer (layer 2 in both the TCP/IP and Open Systems Interconnection (OSI) models) protocol. In contrast, in the case of NAS, access to the storage is presented at *file system* level over a general purpose network via an application layer protocol (TCP/IP layer 4, OSI layer 5).

In the case of both DAS and SAN, the file system running on a client node will see the storage device as being attached locally since all the administrative complexity is manifested in the lower level protocols and hardware. In the case of NAS, a specialized higher level OS level protocol like the Network File System (NFS) is required to interface with the server's storage device. This means that all access to the file system will have to go through a single file server, which is evidently a major bottleneck in terms of performance. It must be noted that a NAS/SAN hybrid where the backing storage of the NAS is actually a SAN is possible, this however only mitigates part of the inherent performance issue [30]. The advantages of SAN compared to NAS are thus performance and scalability, however these come at the cost of tight coupling between servers in the form of expensive dedicated supporting hardware and specialized network layer switches [35].

The inherent lack of scalability of NAS makes it a poor fit for achieving large scale distributed data storage. With the advent of cloud computing and open-source big data toolkits there is a trend of solving scalability issues using commodity hardware at the application level of abstraction. The need for specialized hardware and infrastructure and lack of application level configurability of SAN therefore largely rules it out as a solution for current day large scale data storage needs. Ideally what we want is solution that was designed from the ground up with modern scalability needs in mind running on commodity hardware.

B. GFS

To cope with the ever growing storage demands of their search engine, Google decided developed their own DFS in 2003 dubbed the Google File System (GFS). Even though the software is proprietary and not open-source, at the time of release they did write a paper describing the design of the DFS [21]. While the original design has since been replaced with a newer version called *Colossus*, no such paper has been released describing the updated design [19]. Since the original GFS was one of the pioneering DFSs for internet-scale data and the fact that its design has heavily inspired open-source alternatives such as HDFS, we will discuss it here.

The design of GFS is based around a few key assumptions. The first of which states that component failure should be regarded as the norm, rather than as an exception (which was common for distributed system designs at the time). Seeing as the DFS would be running on hundreds of commodity servers each with several HDDs, it is evident failures will definitely occur. Secondly, the system will be designed to deal with a moderate number of large files. Dealing with billions of KB-sized files inconvenient, which is why in GFS all files

[1] Floating point operations per second

are stored in fixed size 64MB chunks. Not only does this drastically reduce the size of the metadata required for each chunk, it also reduces the number of requests. Furthermore, they noticed that for their most common workloads, data mutations usually take the form of appending data, rather than replacing existing data at a random offset. This insight lead to a design that features an atomic append operation for multiple clients appending data, aiming for minimal synchronization overhead. Lastly they stated that high sustained bandwidth is preferred over low latency.

A GFS cluster runs on commodity hardware as a user-space application, providing a file like interface (although not fully POSIX compliant). Note that this makes the DFS not completely transparent, as it required a user space application instead of being mounted directly in the OS. Architecturally, such a cluster consists of a single *master* server and numerous *chunkservers*. Files are split into fixed-size chunks and each get assigned a unique identifier. By default, each chunk is replicated on at least 3 chunkservers. The master manages all the metadata such as the namespace, access control information, the mapping of files to chunks and the current locations of chunks in main memory to speed up the operations. At regular intervals, the master communicates with each chunkserver via a *HeartBeat* message, giving it instructions and receiving a reply with the server state and the list of chunks a chunkserver contains. Note that these messages are also used to check whether a chunkserver is still functioning. To minimize interaction with the master as it is a potential performance bottleneck, clients only interact with it for metadata operations, the actual transfer of data happens directly between the clients and chunkservers.

The choice for only a single master server was made since it lead to a relatively simple design, especially in contrast to having multiple servers managing a global metadata state. To provide fault tolerance in the case of a master failure, all metadata mutations are logged, stored on disk and replicated on remote servers. To keep the log small, the master periodically saves its state to disk and subsequently resets the log. Keeping track of all mutations allows for data recovery in the event of a master failure, in which case the log is simply played back on the last checkpoint. Note that a client operation is only responded to when the relevant log record is flushed to disk locally *and* on the remove servers.

With respect to consistency, GFS employs a *relaxed* consistency model. For metadata there is strict consistency with a global ordering of operations defined in the log. As there is only a single server managing metadata, the metadata operations are made atomic by means of a locking mechanism. For non-concurrent data mutations, file regions are said to be *consistent* (all clients always see the same data, regardless of which replica) and *defined* (consistent, with all mutations processed serially). For concurrent data mutations, a single ordering will be defined for all replicates, and the state will be consistent but potentially *undefined* (mutations may be interleaved).

To summarize, GFS can be regarded as having laid a solid foundation for modern DFS designs and research to build upon. GFS is a good fit for batch processing applications on large files, providing reliability by actively replicating chunks on multiple servers. With respect to the CAP theorem, the design favors availability and partition tolerance over consistency, given its relaxed consistency model. For latency sensitive applications however, the GFS is not a good fit. The original GFS design has proven to be scalable up to thousands of servers, hosting approximately 50M files comprising in the order of 10 petabytes of data [19]. However, the single metadata server design combined with the scalability limits stated above mean that it is unlikely to be suited for exa-scale computing.

C. HDFS

The Hadoop Distributed File System (HDFS) was developed as part of the popular open-source big data framework Hadoop and officially released in 2011 [32]. The design of HDFS was heavily inspired by that of GFS, one might even go as far as stating that is an open-source implementation of GFS. However, in contrast to GFS, Hadoop (and thereby HDFS) is an Apache project and is thus available under the Apache open-source license. This has lead to widespread use of HDFS, in the context of Hadoop but also as a separate entity [8]. Since it is nearly identical in design as GFS, we will not go over it in its entirety, but will only highlight the main differences.

One of the more superficial differences between HDFS and GFS is the terminology. Instead of the *master* and *chunkservers*, we deal with the *NameNode* and *DataNodes* respectively, and instead of *chunks* we deal with *blocks*. In GFS, the locations of chunks are not persistently stored on the master (they are determined by the HeartBeat messages) while in HDFS the NameNode does maintain persistent location information. Although GFS is optimized for append-only operations, it does allow for random writes within files. In HDFS random writes are not possible, only append write operations are allowed. Lastly, as described in Section III-B, concurrent writes in GFS can lead to consistent but undefined state. HDFS on the other hand employs a single-writer, multiple-reader model meaning that files will always be consistent *and* defined.

The limitations of HDFS are identical to those of GFS. While scalable to thousands of servers and capable of storing tens of petabytes of data, the single server metadata management severely limits its further scalability. Note that new solution based on HDFS are being researched and developed to address these issues, several of these will be discussed in Section IV.

D. GlusterFS

Originally developed by Gluster and now owned by Red Hat after acquiring them in 2011, GlusterFS is an open-source networked file system designed for running on commodity hardware [7]. It is marketed as being a scale-out NAS system, relying on a more traditional client server design rather than employing a metadata server based architecture like GFS or HDFS. GlusterFS is fully POSIX compliant, runs on commodity Ethernet networks but also on Infiniband RDMA (an alternative networking standard often used in the context of HPC). While it is a user-space file system like GFS and HDFS, it can be mounted as native storage on Linux via Filesystem in Userspace (FUSE) or NFS.

In GlusterFS, data is stored on sets of servers called *volumes*. Each volume is a collection of *bricks*, where each

brick is a directory exported from a server. Data is subsequently distributed and potentially replicated on servers within a volume. The level of redundancy within a volume is customisable. *Distributed volumes*: files are simply distributed without replication, limiting the placement of a file to at most one brick. *Replicated volumes*: for better availability and reliability, files can be replicated across bricks within a volume. *Distributed replicated volumes*: files are distributed across replicated *sets* of bricks in the volume, potentially providing improved read performance. *Dispersed volumes*: based on erasure codes, stores encoded fragments of a file on multiple bricks in such a way that the original file can be recovered with only a subset of the encoded fragments. *Distributed dispersed volumes*: similar to replicated vs. distributed replicated volumes, providing higher reliability.

Besides the customisable replication within GlusterFS, what makes its design stand out is the fact that it doe snot maintain a separate metadata index of file locations. Instead, the file location is determined algorithmically using its *elastic hashing algorithm* (EHA). Given a path and file name, the hashing algorithm will determine where this file will be placed. Not having to explicitly store a metadata record is huge advantage in terms of scalability. However, a possible disadvantage of such a placement strategy is that while file *placement* might be uniform, uniformity of the actual *distribution* of data need not be, since not all files are equal in size.

With respect to CAP, GlusterFS favors consistency and availability by default. However, with replicated enabled, GlusterFS allows you to set up a *server quorum* which effectively allows you to trade in availability for partition tolerance. In the event of a network partition, i.e. when different sets of servers are concurrently handling different writes, we can end up with inconsistent state for the same files. When no quorum is set up, clients are always allowed to perform write operations, regardless of any network partition (high availability). As a consequence, any file that is left in an inconsistent state will be marked as such, and will require manual intervention to resolve the merge conflict. When a quorum *is* set up, only a single set of servers is allowed to perform write operations. This means that availability is sacrificed for the servers not in this set, write operations will simply. However, this does mean that the system is able to handle network partitions.

To summarize, there are several clear advantages to the design of GlusterFS. First of all, it offers POSIX file semantics, which means that it is mountable like any other traditional file system and adheres to strict consistency requirements. Secondly, its replication via erasure codes is a more space efficient way of replicating data than naively storing multiple copies. But the main advantage however is the fact the design does not feature a server explicitly storing file location metadata. With respect to scalability, not requiring a metadata server that can potentially be a performance bottleneck is a significant benefit. For certain workloads, a disadvantage of the design of GlusterFS is that it works on file granularity (as opposed to aggregated data blocks or chunks). Such a design can introduce more internal administrative overhead when for example replicating huge numbers of small files. However, we deem it likely that its approach of having a decentralized namespace will manifest itself in exa-scale DFS solutions of the future.

E. Ceph

Ceph is an open-source storage system providing object, block and file storage. It was initially presented in 2006 by Weil et al. as a research project [37]. In 2011, Weil founded Inktank Storage to lead the development of Ceph, which was subsequently acquired by Red Hat in 2014, now making them the lead development contributor. As with the previously discussed DFSs, Ceph is designed to be run on commodity hardware. It is POSIX compliant and natively mountable in Linux.

Like with GFS and HDFS, there is a decoupling of data and metadata. Data transfer occurs between client and data storage servers directly, while metadata operations are handled by metadata servers. However, unlike GFS and HDFS which employ a single metadata server, in Ceph, metadata is managed by an actual cluster of metadata servers (MDSs). It thereby attempts to tackle the problem of metadata management scalability, allowing for up to tens or hundreds of MDSs to form a metadata cluster [25]. Within this cluster, the metadata workload is attempted to be evenly distributed by the use of *dynamic subtree partitioning*, delegating the workload for parts of the file hierarchy to certain MDSs taking into account metadata popularity. Additionally, it allows MDSs to serve in standby mode, ready to take over in the event of an MDS failure, thereby increasing fault tolerance. Similarly to GlusterFS's EHA, Ceph uses its Controlled Replication Under Scalable Hashing (CRUSH) to algorithmically determine where objects are to be placed.

In Ceph, data is stored in a distinct cluster referred to as the Reliable Autonomic Distributed Object Storage (RADOS), exposing an single autonomous object store to clients and the metadata cluster. Within this cluster, objects are stored in conceptual object storage devices (OSDs) which are stored on the local file system of a server. Since 2017, besides allowing OSDs to be run on top of conventional Linux file systems, Ceph now offers its own file system called BlueStore which is optimized for the OSD design (thereby removing the translation overhead due between OSD semantics and traditional file system semantics) [3]. Within the data cluster, data replication is available and customizable. Similar to GlusterFS's *dispersed volume*, Ceph too uses erasure codes in replication.

Ceph offers a lot of customizability with respect to consistency and availability. Not only does it support MDS/OSD clusters ranging from a handful to hundreds of servers, allowing for high availability, it also offers a choice between strict or relaxed consistency models within these clusters. With regard to partition tolerance, Ceph supports a cluster of monitors which maintain a map of the storage cluster, keeping track of the state of the storage servers allowing for action to be taken when necessary. In agreement with the revised take on the CAP theorem and given the amount of customizability Ceph offers, it is not appropriate to label it as either a CA, CP or AP system, thus we will refrain from doing so.

The design of Ceph, allowing for clusters of not only data, but also metadata and monitor servers provides it with excellent scalability characteristics. Currently, it is currently already being used by Yahoo to store petabytes of data and is chosen as the technology to prepare their infrastructure for storing exabytes of data [12]. This in combination with the

level of customizability makes Ceph a good candidate for an exa-scale computing DFS.

F. Lustre

Originally started as a research project in 1999, it was officially released in 2003. Lustre is an open-source DFS tailored to HPC environments, scaling up to thousands of servers and storing petabytes of data [10]. It is fully POSIX compliant, natively mountable on Linux and can be run over standard Ethernet or HPC oriented interconnects such as Infiniband.

In Lustre, storage and transfer of data and metadata are performed by different servers. Similar to the design of Ceph, metadata is stored on a cluster of metadata targets (MDTs). The metadata is distributed over these servers via the Lustre Distributed Namespace (DNE), an approach along the same lines as Ceph's dynamic subtree partitioning. However, unlike Ceph, clients do not access MDTs directly. Instead the client endpoint for metadata operations are the metadata servers (MDSs). The same goes for data storage; files are stored in objects on object storage targets (OSTs) while the client request for the objects are handled by object storage servers (OSSs), which delegate the requests to the ODTs. Thus we see a clear separation of the actual storage of data/metadata and the handling of their respective client requests. This should help in distributing the workload, at the cost of requiring more servers.

Lustre was originally designed for use in HPC, whereby it was assumed that the storage devices provide internal redundancy and fault tolerance. Therefore Lustre does not provide any file system level replication. However, a high level design is provided by the Lustre team for file level replication [11]. In terms of consistency, Lustre is able to provide strong consistency via a locking mechanism. Lustre aims to achieve high availability by providing failover mechanism, essentially allowing redundant servers to take over in the event of server failures.

In conclusion, there are several clear advantages to Lustre's design. The first of which is that it allows for clusters of data *and* metadata, like Ceph. Secondly, the handling of client requests and actual storage of data and metadata occurs on different machines. In terms of scalability this is a clear advantage since it allows for explicit control over how many server to dedicate to the handling of client requests and actual storage. Similarly, availability can be customized by introducing redundant backup servers to a cluster. The number of files that are stored in a single object is customizable as well, which means that Lustre is not necessarily tied to single type of workload with respect to file size. However, the lack of replication at the software level makes it a poor fit for failure sensitive commodity hardware, especially when the cluster size grows. That being said, its metadata and data cluster architecture, given hardware providing built-in redundancy and fault tolerance, make it a good candidate for an exa-scale computing DFS.

G. Cloud-based object storage

Although arguably not a DFS at the same level of abstraction as those previously discussed, we deemed it necessary to discuss the object storage services provided by practically all major cloud providers. These services essentially provide a complete virtualization of storage, not burdening application developers with the intricacies of cluster management or the cost of running and maintaining a private cluster. Additionally, these services are comparably trivial to set up and provide easy to use interfaces. Here we will briefly discuss the advantages and disadvantages of these services and discuss the consistency models offered by the three major cloud providers: Amazon, Microsoft and Google.

There are many advantages to cloud-based object storage. For system designers and application developers, an object interface that abstracts away nearly all of the administrative complexities that come with distributed data storage is a major selling point. The relative ease of use and pricing of cloud services make it so that large scale data storage is available to small companies or even individuals, and not just in the reach of large enterprises or universities. Since the major cloud providers have data centers all around the globe, data can be replicated over these data centers and can thus lead to better performance when accessed from varying locations.

With respect to the interfaces provided for these services, in addition to the availability of language bindings for practically all popular programming languages in the form of official or third-party libraries, they all provide very similar RESTful API's for interfacing with them over HTTP/HTTPS. Since the interfaces are web-based it means that cross platform data sharing is trivial. It must be noted however that while the interfaces are similar, they are not identical. Thus there is the risk of vendor lock-in. Fortunately, efforts are being made by the SNIA to standardize the interface for clients in the form of the Cloud Data Management Interface (CDMI) [13].

In terms of CAP, usually cloud-based object storage solutions used to fall in the AP category, providing high availability (through replication) and partition tolerance while offering only *eventual consistency* [24]. Informally put, eventual consistency only guarantees that given the absence of future updates, *eventually* all read operations on a particular data item will yield the latest version. It should be noted that preservation of the *ordering* of operations is not at all guaranteed. One of the most obvious use cases for eventual consistency is the hosting of large sets of unstructured data that are not subjected to frequent change such as static images or videos. Where it evidently falls short is when stale reads can compromise the business logic of an application. An extreme example of this would be a banking application.

In an effort to allow for stronger consistency guarantees for replicated cloud storage, Mahmood et al. propose a system enabling *causal consistency* for cloud storage services [24]. With causal consistency, preservation of the causal relationships between operations is guaranteed. More specifically it guarantees that write operations that are potentially causally related must always be observed in the same order, while concurrent writes may be seen in a different order [34]. For example, in the context of a web shop, a notification about a discount on a certain product (itself a write operation causally related to the previous write operation on the price of the product) should be followed by the customer seeing the discounted price when visiting the web shop. In this case it is evident that preservation of the causal relationship is desired, otherwise the customer could be presented with the

non-discounted price while having received the notification. In their work they note that while there has been a lot of research into achieving causal consistency in storage systems, their adoption has been limited and is impractical since they require *full* replication of the entire data set among all data centers. They propose a promising new system dubbed *Karma* that achieves causal consistency using only *partial* replication (making it both more feasible and economical). The system is able to offer causal consistency and guarantees availability under a single AZ[2] failure and simple network partitions. It retains availability by operating in a degraded mode upon network partition, aligning with the revised take on the CAP theorem discussed in Section II-C.

Today, the three major cloud providers offer different consistency guarantees for their object storage services. Amazon's S3 offers only eventual consistency for update and delete operation but does offer so called *read-after-write* consistency for newly created objects [5]. This means that when an object is newly created, it should be immediately available (no need for parts of an application to introduce an artificial delay to assure the creation has propagated). Google's Cloud Storage offers full consistency for creating, deleting objects and performing meta-data updates, while only offering eventual consistency for object access revocation [4]. Lastly, Microsoft's Azure Storage claims to offer full consistency in addition to availability and partition tolerance guarantees *'in practice'* by layering their system design around a specific fault model [17].

Since major cloud providers have immense backing infrastructures, they are able to offer virtually unlimited storage capacity. This means that at least in terms of memory capacity, cloud-based object storage is scalable to petabytes and even exabytes. They offer it at a level of abstraction that is easy to interface with from a system designer or application developer point of view. They major downside however will be the relatively high latency of the web based interfaces, making it unsuitable for latency sensitive (HPC) workloads. With regard to consistency, the major cloud providers initially only offered eventual consistency for their object storage services, ruling out their use in applications which rely on stronger consistency restraints. However, we are seeing efforts being made in both academia and industry to improve these services by providing stronger consistency models in the presence of high availability and partition tolerance. Although at a higher level of abstraction than the previously discussed DFSs, cloud-based object storage can be an appropriate solution when extremely low latency is not required.

IV. Future challenges and research

The open-source solutions discussed in the previous section are production ready and can be used today. However, there is also an active research community with respect to improving DFS design, both in academia and in open-source communities. For this section, we have chosen three challenges that we expect to be relevant for the advancement of DFS design. We will briefly describe them and novel research projects attempting to address these challenges.

[2]Availability zone: a term used by the major cloud providers referring to a logical data center with a physically distinct, independent infrastructure in terms of power, networking and cooling [2]

A. Scalability of metadata management

To ensure future use of current DFS designs means that they not only have to be scalable in terms of actual storage capacity, but also in metadata management. In a comparison of typical file system workloads, it was found that nearly half of all operations are metadata operations [29]. It is for this reason that there is a continuous effort being made to make metadata management more scalable. Here we will briefly go over the metadata scalability characteristics of the discussed DFSs and take a look at two recent research projects.

From the discussed DFSs we have observed the inherent lack of metadata scalability of GFS and HDFS, since they both feature a design with only a single metadata server. Lustre allows for multiple metadata servers, but relies on explicitly storing the locations of files. GlusterFS somewhat improves on this aspect by not explicitly storing metadata regarding file locations but instead it opting for algorithmic placement. It must be noted however that even with this in place, all the other metadata operations still happen on the data storage servers. The design of Ceph is probably the most scalable with respect to metadata management, since it allows for a cluster of metadata servers and also features algorithmic file placement and dynamic metadata workload distribution.

A notable recent development in relational databases is *NewSQL*, a class of databases seeking to combine the scalability characteristics of NoSQL databases with the transactional characteristics of traditional relational databases. In a 2017 paper Niazi et al. present HopFS, a DFS built on top of HDFS, replacing the single metadata server with a cluster of NewSQL databases storing the metadata [27]. They attempt to address the issue of metadata management scalability by storing all HDFS metadata in a Network Database (NDB), a NewSQL engine for MySQL Cluster. They tested their solution on a Spotify workload (a Hadoop cluster of 1600+ servers storing 60 petabytes of data) for which they observed a throughput increase of 16-37x compared to regular HDFS. What makes this solution noteworthy is that it is a drop-in replacement for HDFS, allowing it to be used in existing Hadoop environments, allowing them to scale beyond the limits imposed by the single metadata server approach.

Using a similar approach, Takatsu et al. present PPFS (Post-Petascale File System), a DFS optimized for high file creation workloads. In their paper they argue that modern DFSs are not optimized for high file creation workloads, and that for exa-scale computing this can turn out to be a serious performance bottleneck [33]. They have evaluated their system against IndexFS (2014), a middleware for file systems such as HDFS and Lustre aiming to improve metadata performance [28]. With respect to file creation performance, they observed a 2.6x increase in performance. They achieved this by employing a distributed metadata cluster design using key-value metadata storage and non-blocking distributed transactions to simultaneously update multiple entries. Although only tested on relatively small clusters comprising of tens of servers, it good to see that an effort is being made to improve upon aspects such as file creation performance, which might turn out to be a bottleneck in an exa-scale context.

B. Small file performance

There is a discrepancy in performance between DFSs with respect workloads involving many small files as opposed to workloads with fewer, but larger files. In a comparative analysis of various DFSs, it was established that the design of GFS, HDFS and Lustre make them not well suited towards workloads involving many small files, while Ceph and GlusterFS seemed to perform reasonably well for both types of workloads [18]. That being said, there is room left for improvement with regard to small file workload performance. Here we will briefly highlight two recent research projects attempting to address this issue.

In their paper, Zhang et al. present HybridFS, a DFS framework that runs on top of multiple DFSs and dynamically chooses which DFS to use for file placement [38]. As not to impose a tight coupling between an application and a specific DFS based on file size performance characteristics of the DFS, Zhang et al. present HybridFS. It is a DFS framework that runs on top of multiple DFSs and features an additional metadata management server that is in charge of file placement on the underlying DFSs, features a partial metadata store for small files and can perform dynamic file migration to balance storage usage on the DFSs. They have evaluated their solution on an 8 node cluster running Ceph, HDFS and GlusterFS. They observed a best-case performance increase of 30% in read/write operations when compared to single DFS. HybridFS is definitely a step in the right direction with regard to providing a workload agnostic DFS framework. With respect to exa-scale computing, the relatively small cluster size on which it was tested and the fact that the additional metadata management server introduces an additional single point of failure limit its scalability and thereby suitability for exa-scale computing.

A different approach is taken by Fu et al., who present iFlatLFS (Flat Lightweight File System), an OS level file system designed as a replacement for file systems such as Ext4, ReiserFS or XFS when deployed on storage servers in a DFS [20]. The file system was explicitly designed to address the issue of poor small file performance. In iFlatFS, data can be directly addressed from disk instead of through a hierarchical file tree as is the case for the conventional file systems. They achieve this by using a much simpler metadata scheme that occupies much less space, which allows it to be entirely cached in memory. This in turn means that all metadata requests can be server from memory, instead of having to go through disk. They have tested their solution using the Taobao File System (TFS, developed by Alibaba) in which they observed an almost twofold increase in 1KB-64KB file performance when compared to Ext4.

C. Decentralization

In the solutions discussed so far we have seen various approaches to positively influence the scalability characteristics of a DFS. A recurring concept is that of decentralization, distributing responsibility of certain aspects of the system to multiple non-authoritative servers instead of relying on a single or multiple dedicated centralised servers. Removing a single point of failure by distributing the workload should help to increase fault tolerance and scalability. We see such an approach in GlusterFS and Ceph, which both feature a decentralized approach towards file placement. Here we will briefly discuss a recent project that seeks to go even further, a completely decentralized peer-to-peer DFS.

Originally designed by Benet and currently an open-source project with active development from a community of developers, the InterPlanetary File System (IPFS) is a DFS protocol designed to allow all connected peers to access the same set of files [14]. The author describes it as being similar to the Web in a single BitTorrent swarm exchanging objects within a Git repository. It makes use of distributed hash tables, incentivized block exchange and a self certifying namespace to create content addressable peer-to-peer network of files. Any type of data can be stored as an object in IPFS, with integrity validation and automatic deduplication as built-in features.

Removing all single points of failure by taking a completely distributed peer-to-peer approach is very interesting, since it in theory provides infinite scalability. However, having to rely on servers beyond your control likely rules it out for latency sensitive or mission critical applications. That being said, leveraging a globally distributed network of interconnected machines as a DFS is very relevant to at least capacity requirements. One can envision that given a large peer count, storing exabytes of data becomes almost trivial. Generally, we expect that the concept of decentralization will play a significant role in the development of future DFSs to cope with ever increasing scalability demands.

V. Conclusion

The goal of this paper was to construct an overview of the state-of-the-art of DFSs. We have evaluated the design of several well established DFS solutions while also taking a look novel research attempting to solve the challenges of future DFSs.

With respect to the established solutions, we have seen drastically varying designs. From single metadata server architectures such as GFS/HDFS, to clusters of metadata servers in the case of Ceph and Lustre. GlusterFS and Ceph both employ algorithmic file placement, which mean that there is no need to explicitly store file location metadata. Purely based on high-level design comparison, Ceph seems to cater the most to modern scalability requirements and will therefore be a likely candidate to be used in an exa-scale computing context. Lastly we discussed cloud-based object storage, which with its immense backing infrastructure is be able to store exabyte sized data sets. However, for latency sensitive applications like HPC it is unlikely to be a good fit.

We have taken a look at recent research and presented them in the context of three challenges that we deemed relevant for future DFS designs. Firstly, we discussed the issue of scaling metadata management. We found that with respect to metadata clusters, we are seeing efforts being made to incorporate techniques from the database world such as distributed transactions and NewSQL databases to speed up metadata performance. Secondly, we discussed the issue of performance in small file workloads. We regarded two research projects attempting to tackle this issue. The first of which by running multiple DFSs, each optimized for certain file sizes, and having an additional layer on top of it dynamically

choosing an appropriate underlying DFS based on the type of workload. The second introduced an entirely new HDD level file system to be run on the storage servers in a DFS, increasing performance by reducing the size of stored metadata, allowing it to be stored entirely in main memory. Lastly we discussed the concept of decentralization with respect to DFSs. We regarded IPFS, a completely decentralized peer-to-peer DFS allowing for all peers to access the same set of files.

Note that the presented information here is far from exhaustive and that there are a lot of current DFS solutions and research projects that we have not covered. Especially in the light of preparing systems for exa-scale computing, there is a lot of ongoing research into developing new techniques and designing refined architectures for future DFSs. That being said, we believe that what we have presented in this paper provides a reasonable representation of the state-of-the-art and future development challenges of DFSs.

REFERENCES

[1] Astc technology roadmap. http://idema.org/?page_id=5868. Accessed: 03-12-2017.

[2] Overview of availability zones in azure (preview). https://docs.microsoft.com/en-us/azure/availability-zones/az-overview. Accessed: 18-12-2017.

[3] Ceph v12.2.0 luminous released - ceph. http://ceph.com/releases/v12-2-0-luminous-released/. Accessed: 06-01-2018.

[4] Consistency — cloud storage documentation. https://cloud.google.com/storage/docs/consistency, . Accessed: 18-12-2017.

[5] Introduction to amazon s3 - amazon simple storage system. http://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html#ConsistencyModel, . Accessed: 18-12-2017.

[6] June 2008 — top500 supercomputer sites. https://www.top500.org/lists/2008/06/. Accessed: 04-01-2018.

[7] Gluster docs. http://docs.gluster.org/en/latest/. Accessed: 05-01-2018.

[8] Poweredby - hadoop wiki. https://wiki.apache.org/hadoop/PoweredBy. Accessed: 05-01-2018.

[9] Hgst ultrastar hs14. http://www.hgst.com/products/hard-drives/ultrastar-hs14. Accessed: 03-12-2017.

[10] Lustre. http://lustre.org/, . Accessed: 06-01-2018.

[11] Lustre - file level replication high level design. http://wiki.lustre.org/File_Level_Replication_High_Level_Design, . Accessed: 06-01-2018.

[12] Yahoo cloud object storage. https://yahooeng.tumblr.com/post/116391291701/yahoo-cloud-object-store-object-storage-at. Accessed: 06-01-2018.

[13] Storage Networking Industry Association. Cloud data management interface (cdmi) version 1.1.1. 2015.

[14] Juan Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[15] Eric Brewer. Cap twelve years later: How the" rules" have changed. *Computer*, 45(2):23–29, 2012.

[16] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

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

[18] Benjamin Depardon, Gaël Le Mahec, and Cyril Séguin. Analysis of six distributed file systems. 2013.

[19] Andrew Fikes. Storage architecture and challenges. *Talk at the Google Faculty Summit*, 2010.

[20] Songling Fu, Ligang He, Chenlin Huang, Xiangke Liao, and Kenli Li. Performance optimization for managing massive numbers of small files in distributed file systems. *IEEE Transactions on Parallel and Distributed Systems*, 26(12):3433–3448, 2015.

[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[22] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[23] Eliezer Levy and Abraham Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys (CSUR)*, 22(4):321–374, 1990.

[24] Tariq Mahmood, Shankaranarayanan Puzhavakath Narayanan, Sanjay Rao, TN Vijaykumar, and Mithuna Thottethodi. Achieving causal consistency under partial replication for geo-distributed cloud storage. 2016.

[25] Carlos Maltzahn, Esteban Molina-Estolano, Amandeep Khurana, Alex J Nelson, Scott A Brandt, and Sage Weil. Ceph as a scalable alternative to the hadoop distributed file system. *login: The USENIX Magazine*, 35:38–49, 2010.

[26] Mike Mesnier, Gregory R Ganger, and Erik Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–90, 2003.

[27] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. Hopsfs: Scaling hierarchical file system metadata using newsql databases. In *FAST*, pages 89–104, 2017.

[28] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 237–248, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-5500-8. doi: 10.1109/SC.2014.25. URL https://doi.org/10.1109/SC.2014.25.

[29] Drew S Roselli, Jacob R Lorch, Thomas E Anderson, et al. A comparison of file system workloads. In *USENIX annual technical conference, general track*, pages 41–54, 2000.

[30] David Sacks. Demystifying storage networking das, san, nas, nas gateways, fibre channel, and iscsi. *IBM storage networking*, 2001.

[31] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation or the sun network filesystem, 1985.

[32] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system.

In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. IEEE, 2010.

[33] Fuyumasa Takatsu, Kohei Hiraga, and Osamu Tatebe. Ppfs: A scale-out distributed file system for post-petascale systems. *Journal of Information Processing*, 25:438–447, 2017.

[34] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. 2016.

[35] John R Vacca. *Optical Networking Best Practices Handbook*. John Wiley & Sons, 2006.

[36] Thiruvengadam Vijayaraghavany, Yasuko Eckert, Gabriel H Loh, Michael J Schulte, Mike Ignatowski, Bradford M Beckmann, William C Brantley, Joseph L Greathouse, Wei Huang, Arun Karunanithi, et al. Design and analysis of an apu for exascale computing. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 85–96. IEEE, 2017.

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

[38] L. Zhang, Y. Wu, R. Xue, T. C. Hsu, H. Yang, and Y. C. Chung. Hybridfs - a high performance and balanced file system framework with multiple distributed file systems. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 796–805, July 2017. doi: 10.1109/COMPSAC.2017.140.