

A Review of Storage and Fault Tolerance Approaches Used in Cloud Computing

Amit Batra[#], Rajender Kumar^{*}, Arvind Kumar[#]

[#]CSE Department,

H.C.T.M Kaithal India

^{*}H.C.T.M Kaithal India

Abstract— Cloud computing has gained significant traction in recent years. Companies such as Google, Amazon and Microsoft have been building massive data centers over the past few years. Spanning geographic and administrative domains, these data centers tend to be built out of commodity desktops with the total number of computers managed by these companies being in the order of millions. Additionally, the use of virtualization allows a physical node to be presented as a set of virtual nodes resulting in a seemingly inexhaustible set of computational resources. By leveraging economies of scale, these data centers can provision cpu, networking, and storage at substantially reduced prices which in turn underpins the move by many institutions to host their services in the cloud. In this survey we will be surveying the most dominant storage and fault tolerant strategies that are currently being used in cloud computing settings. There are several unifying themes that underlie the systems that we survey.

Keywords— Cloud, GFS, Azure, NFS, RAID, Dynamo.

I. INTRODUCTION

Cloud computing can be defined as a new style of computing in which dynamically scalable and often virtualized resources are provided as a services over the Internet. Cloud computing has become a significant technology trend, and many experts expect that cloud computing will reshape information technology (IT) processes and the IT marketplace. With the cloud computing technology, users use a variety of devices, including PCs, laptops, smartphones, and PDAs to access programs, storage, and application-development platforms over the Internet, via services offered by cloud computing providers. Advantages of the cloud computing technology include cost savings, high availability, and easy scalability.

II. LAYERS OF CLOUD COMPUTING

Cloud computing can be viewed as a collection of services, which can be presented as a layered cloud computing architecture, as shown in Fig. 1 below:

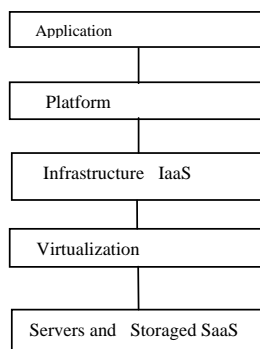


Fig. 1 Layered architecture of Cloud Computing

The services offered through cloud computing usually include IT services referred as to SaaS (Software-as-a-

Service), which is shown on top of the stack. SaaS allows users to run applications remotely from the cloud. Infrastructure-as-a-service (IaaS) refers to computing resources as a service. This includes virtualized computers with guaranteed processing power and reserved bandwidth for storage and Internet access.

Platform-as-a-Service (PaaS) is similar to IaaS, but also includes operating systems and required services for a particular application. In other words, PaaS is IaaS with a custom software stack for the given application. The data-Storage-as-a-Service (dSaaS) provides storage that the consumer is used including bandwidth requirements for the storage.

III. TYPES OF CLOUD COMPUTING

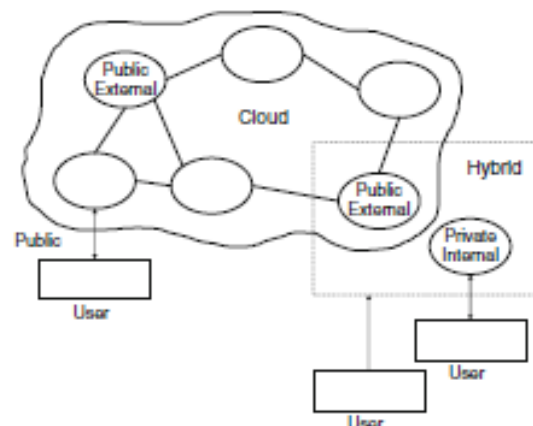


Fig. 2 Three types of cloud computing

In the public cloud (or external cloud) computing resources are dynamically provisioned over the Internet via Web applications or Web services from an off-site third-party provider. Public clouds are run by third parties, and applications from different customers are likely to be mixed together on the cloud's servers, storage systems, and networks. Private cloud (or internal cloud) refers to cloud computing on private networks. Private clouds are built for the exclusive use of one client, providing full control over data, security, and quality of service. Private clouds can be built and managed by a company's own IT organization or by a cloud provider.

A hybrid cloud environment combines multiple public and private cloud models. Hybrid clouds introduce the complexity of determining how to distribute applications across both a public and private cloud.

There are several unifying aspects that underlie the systems that we survey:

Aspect 1: Voluminous Data

The datasets managed by these systems tend to be extremely voluminous. It is not unusual for these datasets to

be several terabytes. The datasets also tend to be generated by programs, services and devices as opposed to being created by a user one character at a time.

Aspect 2: Commodity Hardware

The storage infrastructure for these datasets tend to rely on commodity hard drives that have rotating disks. This mechanical nature of the disk drives limits their performance. While processor speeds have grown exponentially disk access times have not kept pace. The performance disparity between processor and disk access times is in the order of 14,000,000:1 and continues to grow.

Aspect 3: Distributed Data

A given dataset is seldom stored on a given node, and is typically distributed over a set of available nodes. This is done because a single commodity hard drive typically cannot hold the entire dataset. Scattering the dataset on a set of available nodes is also a precursor for subsequent concurrent processing being performed on the dataset.

Aspect 4: Expect Failures

Since the storage infrastructure relies on commodity components, failures should be expected. The systems thus need to have a failure model in place that can ensure continued progress and acceptable response times despite any failures that might have taken place. Often these datasets are replicated, and individual slices of these datasets have checksums associated with them to detect bit-flips and the concomitant data corruptions that often taken place in commodity hardware.

Aspect 5 Tradeoff Between Consistency and Availability

Having a consistent distributed system means that no matter what node you connect to, you will always find the same exact data. Here, we take availability to mean that as long as a request is sent to a node that has not failed it will return a result. This definition has no bound on time limit, it simply states that eventually a client will get a response. Last, there is partition tolerance. A partition occurs when one part of your distributed system can no longer communicate with another part, but can still communicate with clients. The simplest example of this is in a system with 2 nodes, **A** and **B**. If **A** and **B** can no longer communicate with each other, but both can and do keep serving clients, then the system is partition tolerant. With a partition-tolerant system, nothing short of a full system failure keeps the system from working correctly. As a quick example, let's look at a partition-tolerant system with two nodes **A** and **B**. Let's suppose there is some network error between **A** and **B**, and they can no longer communicate with each other, but both can still connect to clients. If a client were to write a change a file *v* hosted on both **A** and **B** while connected to **B**, the change would go through on **B**, but if the client later connects to **A** and reads *v* again, the client will not see their changes, so the system is no longer consistent. You could get around this by instead sacrificing availability – if you ignore writes during a network partition, you can maintain consistency.

Aspect 6 Tune for Access by Applications

Though these storage frameworks are built on top of existing file systems, the stored datasets are intended to be processed by applications and not humans. Since the dataset is scattered on a large number of machines, reconstructing the dataset requires processing the *metadata* (data describing the data) to identify the precise location of specific portions of the datasets. Manually accessing any of

the nodes to look for a portion of the dataset is futile since these portions have themselves been modified to include checksum information.

Aspect 7 Optimize for Dominant Usage

Another important consideration in these storage frameworks is optimizing the most general access patterns for these datasets. In some cases, this would mean optimizing for long, sequential reads that puts a premium on conserving bandwidth while in others it would involve optimizing small, continuous updates to the managed datasets.

Aspect 8 Enabling Technologies

Key technologies that enabled cloud computing are described in this section; they include virtualization, Web service and service-oriented architecture, service flows and workflows, and Web 2.0 and mash up.

Virtualization

The advantage of cloud computing is the ability to virtualize and share resources among different applications with the objective for better server utilization. In non-cloud computing three independent platforms exist for three different applications running on its own server. In the cloud, servers can be shared, or virtualized, for operating systems and applications resulting in fewer servers (in specific example two servers). Virtualization technologies include virtual machine techniques such as VMware and Xen, and virtual networks, such as VPN. Virtual machines provide virtualized

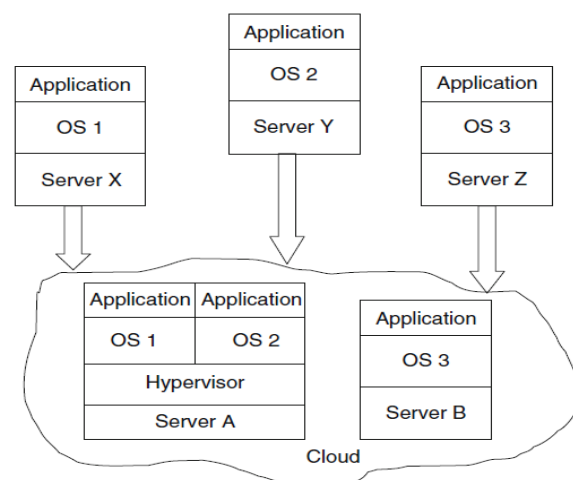
IT-infrastructures on-demand, while virtual networks support users with a customized network environment to access cloud resources.

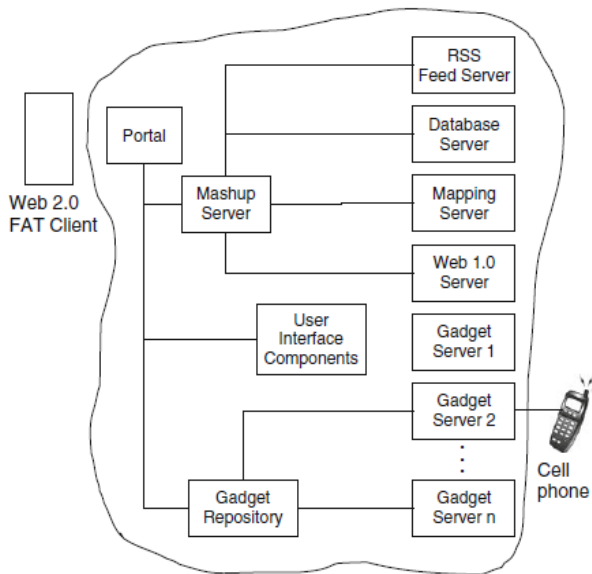
Web Service and Service Oriented Architecture

Web Services and Service Oriented Architecture (SOA) are not new concepts; however they represent the base technologies for cloud computing. Cloud services are typically designed as Web services, which follow industry standards including WSDL, SOAP, and UDDI. A Service Oriented Architecture organizes and manages Web services inside clouds. A SOA also includes a set of cloud services, which are available on various distributed platforms.

Service Flow and Workflows

The concept of service flow and workflow refers to an integrated view of service based activities provided in clouds. Workflows have become one of the important areas of research in the field of database and information systems.





For completeness we have included a description of the systems, which explored ideas that have now found its way into several of the systems.

1. Microsoft Azure

Azure is Microsoft’s cloud computing solution. It consists of three parts: storage, scalable computing, and the base fabric to hold everything together across a heterogeneous network. Figure 3 shows a high level overview of Azure’s structure:

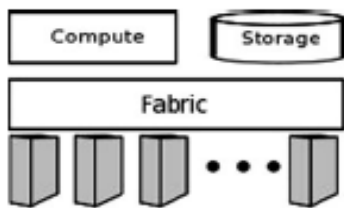


Fig 3. Azure overview

1.1 Failure

Azure’s fabric layer is made up of machines in a Microsoft Data Center. The data center is divided into *fault domains*. Microsoft defines a fault domain as a set of machines which can be brought down by the failure of a single piece of hardware. All machines dedicated to Azure are controlled by 5–7 fabric controllers. Each machine inside the fabric has a fabric controller process running which reports the status of all applications running on that machine (this includes user apps in different VMs as well as the storage service). While we are not exactly clear on how storage is handled inside the fabric, we do know that the fabric controllers see the storage service as just another application. If an application dies for any reason, the controllers are responsible for starting up another instance of the application.

1.2 Consistency and Guarantees

Azure’s storage guarantees read-what-you-wrote consistency – worker threads and clients will be able to immediately see changes it just wrote. Unfortunately, there is no clear picture of what this means for other threads/clients. It also guarantees a replication level of 3 for

all stored data. There have also been no specific guarantees as to latency or specific mention of SLAs.

1.3 Data Placement

The Azure fabric layer is responsible for the placement of data. While it is not directly aware of replicas, it is able to ensure that instances of the storage service are running in different fault domains. From the whitepapers Microsoft has made available, it looks like a fabric controller only operates in one data center. There is a chance that users are able to choose which data center to use.

2 Bigtable

As the name suggests, Bigtable stores large amounts of data in a table. While it is not a full relational model, it is essentially a multidimensional database.

2.1 Replication

As mentioned above, the Bigtable master server makes sure that only one server is actually modifying a tablet at a time. While this looks like Bigtable is ignoring replication entirely, every tablet’s SSTables are actually being stored in GFS. Bigtable neatly bypasses the problem of replication and lets GFS handle it. Bigtable will inherit the replication level of the folders where the SSTables are stored.

2.2 Failures

All failure detection for Bigtable eventually comes down to Chubby. When a tablet server first starts up, it contacts Chubby and makes a server-specific file, and obtains an exclusive lock on it. This lock is kept active as long as the tablet has a connection to Chubby, and will immediately stop serving tablets if it loses that lock. If a tablet server ever contacts Chubby and finds the file gone, it will kill itself. The master server is responsible for periodically polling the tablet servers and checking to see if they are still up. If the master cannot contact a tablet server, it first checks to see if the tablet server can still communicate with Chubby. It does so by attempting to obtain an exclusive lock on the tablet server file. If the master obtains the lock, Chubby is alive and the tablet can’t communicate with Chubby. The master then deletes the server file, ensuring that the server will not attempt to serve again. If the master’s Chubby session expires, the master immediately kills itself without effecting tablet serving. A cluster management system running alongside Bigtable is responsible for starting up a new master server if this happens.

The master server will kill itself and the cluster manager will repeatedly try to kick start a new master until Chubby starts responding again.

2.3 Accessing Data

Every client is initially sent a library of tablet locations, so they should initially be able to directly contact the correct tablet server. Over time, tablet servers die, some may be added, or tablets may be deleted or split. Bigtable has a 3-tier hierarchy for tablet location. First, there is a file stored in Chubby that contains the location of the root tablet. Every Bigtable instance has its own root tablet. A root tablet specifies the location of all tablets in a METADATA table. This METADATA table holds the locations of all user tables as well as some tablet-specific information useful for debugging purposes. The root tablet is simply the first tablet of the METADATA table. The root tablet is treated specially – it is never split so that the tablet location

hierarchy doesn't grow. With this scheme, 2^{34} tablet locations can be addressed.

The client library caches the tablet locations from the METADATA table, and will recursively trace through the hierarchy if it doesn't have a tablet, or the tablet location is stale. With an empty cache, it will take 3 round trips but may take up to 6 with a stale cache. None of these operations need to read from GFS, so the time is negligible. The tablet servers have access to sorted SSTables, so they can usually locate required data (if not already in memory) with a single disk access.

2.4 Data Integrity

Bigtable is not directly involved with maintaining data integrity. All Bigtable data is stored in GFS, and that is what is responsible for actually detecting and fixing any errors that occur in data. When a tablet server goes down there is a chance that a table modification was not committed, or a tablet split was not properly propagated back to Chubby. Keeping all tablet operation logs in GFS as well solves the first problem: a new tablet server can read through the logs, and ensure all tablets are up to date. Tablet splits are even less of a problem, as a tablet server will report any tablets it has that are not referenced by Chubby.

2.5 Consistency and Guarantees

Bigtable guarantees eventual consistency – all replicas are eventually in sync. Tablet servers store any tablet modifications in memory, and will write permutations to a log, but will not necessarily wait for GFS to confirm that a write has succeeded before confirming it with users. This helps to improve latency, and give users a more interactive experience, such as when using Google Earth. Bigtable inherits all of the GFS guarantees pertaining to data replication, error recovery, and data placement.

2.6 Metadata

The METADATA table contains the metadata for all tablets held within an instance of Dynamo. This metadata includes lists of the SSTables which make up a tablet, and a set of pointers to commit logs for the tablet. When a tablet server starts serving a file, it first reads the tablet metadata to learn which SSTable files need to be loaded. After loading the SSTables into memory, it works through the commit logs, and

brings the version in memory up to the point it was at when the tablet was last accessed.

2.7 Data Placement

All of Bigtable's data placement is handled by GFS – it has no direct concern for data placement. As far as Bigtable is concerned, there are only single copies of files – it uses GFS handles to access any files needed. While Bigtable is not

directly aware of multiple versions of files, it can still take advantage of replicas through GFS.

2.8 Security

Bigtable is designed to run in a trusted environment, and does not really have much in the way of security measures. Theoretically, a user may be able to have encrypted row and column names, as well as the data in the fields. This would be possible since these are all arbitrary strings. While encrypting row names means you could potentially use some of the grouping abilities, there is no reason a user would not be able to gain some security with this method.

3. Google File System

The Google File System (GFS) is designed by Google to function as a backend for all of Google's systems. The basic assumption underlying its design is that components are expected to fail. A robust system is needed to detect and work around these failures without disrupting the serving of files. GFS is optimized for the most common operations – long, sequential and short, random reads, as well as large, appending and small, arbitrary writes. Additionally, a major goal in designing GFS was to efficiently allow concurrent appends to the same file. As a design goal, high sustained bandwidth was deemed more important than low latency in order to accommodate large datasets. A GFS instance contains a *master server* and many *chunk servers*. The master server is responsible for maintaining all file system metadata and managing *chunks* (stored file pieces). There are usually also several master replicas, as well as shadow masters which can handle client reads to help reduce load on a master server. The chunk servers hold data in 64 MB-sized chunks.

3.1 Checkpointing

In GFS, the master server will keep logs tracking all chunk mutation. Once a log file starts to become too big, the master server will create a checkpoint. These checkpoints can be used to recover a master server, and are used by the master replicas to bring a new master process up.

3.2 Replication

By default, all GFS maintains a replication level of 3. This is, however, a configurable trait: “. . . users can designate different replication levels for different regions of the file namespace”. For example, a temp directory generally has a replication level of 1, and is used as a scratch space. The master server is responsible for ensuring that the replication level is met. This not only involves copying over chunks if a chunk server goes down, but also removing replicas once a server comes back up. As a general rule, the master server will try to place replicas on different racks. With Google's network setup, the master is able to deduce the network topology from IP addresses.

3.3 Failures

When it comes to failures, GFS always expects the worst. The master server regularly exchanges heartbeats with the chunk servers. If the master server does not receive a heartbeat from a chunk server in time, it will assume the server has died, and will immediately start to spread the chunks located on that server to other servers to restore replication levels. Should a chunk server recover, it will start to send heartbeats again and notify the master that it is back up. At this point the master server will need to delete chunks in order to drop back down to replication level and not waste space. Because of this approach, it would be possible to wreak havoc with a GFS instance by repeatedly turning on and off a chunk server. Master server failure is detected by an external management system. Once this happens, one of the master server replicas is promoted, and the master server process is started up on it. A full restart usually takes about 2 minutes – most of this time is spent polling the chunk servers to find out what chunks they contain

3.4 Data Access

Clients initially contact the master server to gain access to a file, after which the client interacts directly with the necessary chunk server(s). For a multi terabyte file, a client

can keep track of all chunk servers in its cache. The chunk server directly interacting with clients is granted a chunk *lease* by the master server, and is now known as the *primary*. The primary is then responsible for ordering any operations on the data serially. It is then responsible for propagating these changes to the other chunk servers that hold the chunk. If a client is only looking to read data, it is possible for the client to go through the shadow master as opposed to the master server. It is possible for concurrent writes to get interleaved in unexpected ways, or for failed write attempts to show themselves as repeated data in chunks. GFS assumes that any application using it is able to handle these possible problems though redundant data may hurt the efficiency of reads.

3.5 Data Integrity

Each chunk in GFS keeps track of its own checksum information this information is unique for each chunk – it is not guaranteed to be the same even across replicas. Chunk servers are responsible for checking the checksums of the chunks they are holding. With this, it is possible for the system to detect corrupted files. If a corrupted chunk is detected, the chunk is deleted, and copied from another replica.

3.6 Consistency and Guarantees

GFS is built to handle multiple concurrent appends on a single file. It is up to a primary chunk server to order incoming permutation requests from multiple clients into a sequential order, and then pass these changes on to all other replicas. Because of this, it is possible that a client will not see exactly what they wrote on a sequential read – there is a possibility that permutations from other clients have been interleaved with their own. Google describes this state as consistent but undefined – all clients will see the same data, regardless of which replica is primary, but mutations may be interspersed. When there is a write failure, a chunk may become inconsistent.

This is a case where there may be redundant lines of data in some but not all replicas. As GFS was built to maintain bandwidth, as opposed to meet a targeted latency goal there are no guarantees that pertain to latency. GFS does guarantee maintenance of the specified replication level which is achieved using system heartbeats. GFS also cannot guarantee full consistency in the face of write failures. A slightly looser definition of consistency – at least a single copy of all data is fully stored in each replica – is what GFS supplies. Any application built on top of GFS that can handle these possible inconsistencies should be able to guarantee a stronger consistency.

3.7 Metadata

In GFS, the master server contains metadata about all chunks contained in the system. This is how the master server keeps track of where the chunks are located. Each chunk has its own set of metadata as well. A chunk has a version number, as well as its own checksum information.

3.8 Data Placement

The master server attempts to place replicas on separate racks, a feat made possible by Google's network scheme. The master server also attempts to balance network load, so it will try to evenly disperse all chunks.

3.9 Security Scheme

GFS expects to be run in a trusted environment, and has no major security approaches. If a user could bring down a chunk server, modify the chunk versions held on it, and

reconnect it to the system, GFS would slowly grind to a halt as it believes that that server has the most up-to-date chunks and begins deleting and rewriting all these chunks. This would create a lot of network traffic, and theoretically bring down not only any service that relies on GFS, but also anything else that requires network bandwidth to work.

4. Dynamo

Dynamo is the back end for most of the services provided by Amazon. Like S3 it is a distributed storage system. Dynamo stores data in key-value pairs, and sacrifices consistency for availability. Dynamo has been designed to store relatively small files and to retrieve them very quickly. A web page may have several services which each have their own Dynamo instance running in the background – this is what leads to the necessity of making sure latency is low when retrieving data.

Dynamo uses consistent hashing to make a scalable system. Every file in the system identified by a key is hashed, and this hash value is used to determine which node in the system it is assigned to. This hash space is treated as a ring, which is divided into Q equally sized partitions. Each node (server) in the system is assigned an equal number of partitions. In this figure, there are a total of 8 partitions. Nodes **A**, **B**, and **C** are responsible for keeping copies of all files where the hashed key falls into the striped partition that they manage.

4.1 Checkpointing

Dynamo nodes share information via a gossip based protocol. There are no regular heartbeats sent between the nodes. All communication is pushed by client requests. If there is no request for data, the nodes do not communicate and do not care if another node is down. Periodic tests to see if a node is available occur only if a node is found to be unreachable during a client request.

4.2 Replication

With Dynamo, a quorum-like system is used to determine if a read or write was successful. If enough nodes reply that a write/read was successful, the whole operation is considered successful – even if not all N replicas are written to or read from. Dynamo allows the service writer to specify not only N , but R and W as well. R is the number of successful reads necessary for the whole operation to be successful, and W is the number of writes. Dynamo will report a successful write if $W-1$ nodes report success, so to make a system that is always up, and will never reject a write, W can be set to 1. Generally, W and R are both less than N , so that the system can make progress in the presence of failures. A suggested configuration for Dynamo is to have $R + W > N$. A general configuration of (N,R,W) is $(3,2,2)$.

4.3 Failures

Dynamo operates under the assumption that hardware failures are expected, and trades data consistency guarantees for availability. It uses a gossip-based system to detect failures of nodes. Once a node stops responding, other nodes will eventually propagate knowledge of the failure. As a design feature, nodes are not considered removed unless an administrator issues an explicit removal command – this means the system will gracefully handle transient downtimes. If a coordinator cannot reach a node for a write, it will simply pass the data on to the next available node in the hash ring. This will contain an extra

bit of metadata that marks it as belonging elsewhere. Once a node comes back online, this information can be passed back to it. If a node is not available, the data presumed to be on that node is not immediately replicated on another node this only happens when an administrator explicitly removes the node via a command. Dynamo is built under the expectation that there will be many transient failures, so there is no scramble to ensure replication levels are met when a node stops responding to requests. Because of this, some reads may fail if R is set equal to N. Once a node has been explicitly removed, all key ranges previously held by that node are reassigned to other nodes while ensuring that a given node is not overloaded as a result of this redistribution.

4.4 Accessing Data

Dynamo's gossip-based protocol for node discovery ensures that all nodes know in one step the exact node to send a read or write request to. There are two main methods of accessing data: (1) using a dedicated node to handle client requests or (2) having several dedicated nodes, or coordinators, that process client requests and forward them to the appropriate nodes. The former approach can lead to unbalanced network nodes while the latter approach results in a more balanced network and a lower latency can be assured.

4.5 Data Integrity

There is no specific mention of detecting corruptions in data, or how any corresponding error recovery may occur. Since data is stored as a binary object, it may be left up to the application developers to detect data corruption, and handle any sort of recovery. Reported results in live settings do not indicate permanent data loss. Amazon requires regular archival of every system – there is a chance that this archival data is used for recovery if errors in data are found

4.6 Consistency and Guarantees

Dynamo guarantees eventual consistency – there is a chance that not all replications contain the same data. Due to transient network failures and concurrent writes, some changes may not be fully propagated. To solve this problem, each object also contains a context. This context contains a version vector, giving the ability to track back through changes and figure out which version of an object should carry the most precedence. There are several different schemes for handling this. Dynamo

itself supports several simple schemes, including a last-write-wins method. There is also an interface that allows developers to implement more complex and data specific merging techniques. Merging of different object versions is handled on reads. If a coordinator retrieves multiple versions of an object on a read, it can attempt to merge differences before sending it to the client. Anything that cannot be resolved by the coordinator is passed onto the client. Any subsequent write from that client is assumed to have resolved any remaining conflicts. The coordinator makes sure to write back the resolved object to all nodes that responded to the object query. The only other base guarantee provided by Dynamo is performance geared towards the 99.99th percentile of users – millisecond latencies are assured. Aside from this, service developers are allowed to tweak the system to fit the guarantees necessary for their application through the N, R and W settings.

4.7 Metadata

In Dynamo, the object metadata is referred to as context. Every time data is written, a context is included. The context contains system metadata and other information specific to the object such as versioning information. There may also be an extra binary field which allows developers to add any additional information needed to help their application run. The metadata is not searchable, and only seems to interact with Dynamo when resolving version conflicts as mentioned above.

4.8 Data Placement

According to DeCandia et al. , there are guarantees in place to ensure that replicas are spread across different data centers. It is likely that Amazon has a particular scheme that allows Dynamo to efficiently determine the locations of nodes. An object key is first hashed to find its location on the network ring. Moving around the ring clockwise from that point, the first encountered node is where the first copy of the data is placed. The next N-1 nodes (still moving clockwise) will contain replicas of the data. There are no current methods of data segregation in Dynamo – there is simply a get() and put() interface for developers, and no support for a hierarchical structure. Each service using Dynamo has its individual instance of it running. For example, your shopping cart will not be able to access the best seller's list. On the other hand, Dynamo has no guarantees that the different instances are not running on the same machine.

4.9 Security

Dynamo has been designed to run in a trusted environment, so there is no structure in place to handle security concerns. By design, each service that uses Dynamo has its own separate instance running. Because of this, users do have some sense of security, as there is some natural separation of data, and one application cannot access the data of another.

5. xFS

Unlike the other systems mentioned here, xFS never made it to a production environment. xFS is the original "Serverless File System", and several systems in production today build upon ideas originally brought up in. xFS was designed to run on commodity hardware, and expected to handle large loads and multiple users. Based on tracking usage patterns in an NFS system for several days, one assumption xFS makes is that users other than the creator of the file rarely modify files.

5.1 Failure Model

In xFS, when a machine fails it is not expected to come back online. Upon failure of a machine, data is automatically shuffled around to compensate for the loss. While failures are assumed to be permanent, the system was designed to be able to come back up from a full loss of functionality.

5.2 Replication

xFS does not support replication of files. Instead, it supports a RAID approach for storing data. In xFS, servers are organized into *stripe groups*. Each stripe group is a subset of the set of data servers. When a client writes to a file, it is gathered into a *write block* that is held in the client's cache. There are two clients, each building their own write block. Once the write block is full, the data is sent to the server group to be written to file. For a server

group with N servers, the file is split into N-1 pieces, and striped in a RAID pattern across all the servers. The Nth stripe is a parity block that contains the XOR of all the other pieces, and is shown as a striped block. This parity block will go to the parity server for the group. This way, if a server is lost, or a piece becomes corrupted it can be restored. One downside to this approach is that if multiple servers from a group go down, the data may be permanently lost, and xFS will stop working. In general, the replication level of a file can never be greater than the number of servers in the server group.

5.3 Data Access

In xFS a client will connect to a system manager, which will look up the appropriate server group, and have the client connect to the server group leader. In general, this takes about 3 hops (not including the actual transmission of data). Generally, the system will attempt to move data to be as close to the user as possible (in many cases, the design expects the client to be running on a machine that is also acting as a data server), incurring the short term penalty in network traffic of moving a file for the long term bonus of not needing further interaction with a system manager.

5.4 Integrity

Because of the RAID backend of xFS, data corruption can be detected and repaired using the parity block computed when data is written. xFS also uses this information to recover missing stripe blocks when a server in a stripe group fails.

5.5 Consistency and Guarantees

xFS guarantees *read-what-you-wrote* consistency, but it also allows users to read stale data – meaning that the best overall consistency guarantee that it can achieve is eventual. It is also not clear that the system can effectively handle concurrent writes. xFS never made it to a production environment, so there was never a strong need to establish any guarantees governing access times. Additionally, xFS was designed to

handle flux in the number of available servers.

5.6 Metadata

The main advantage of xFS is its fully dynamic structure. The idea is to be able to move data around to handle load fluctuations and to increase locality. The system uses metadata information to help locate all files and put them back together in order.

5.7 Data placement

Managers in xFS try to ensure that data is being held as close to the client accessing it as possible—in some cases even shifting the location of data as a client starts writing to it. While this seems unwieldy, xFS uses a log-based storage method, so there is not too much of a network hit as data is shifted with a new write closer to the current client.

5.8 Security

xFS was designed to be run in a trusted environment, and it is expected that clients are running on machines that are also acting as storage servers. It is, however, possible for xFS to be mounted and accessed from an unsafe environment. Unfortunately this is more inefficient and results in much more network traffic. It is also possible for a rogue manager to start indiscriminately overwriting data that can cause the entire system to fail.

6. Amazon S3

The Simple Storage Service (S3) from Amazon is used by home users, small businesses, academic institutions, and large enterprises. With S3 (Simple Storage Service), data can be spread across multiple servers around the US and Europe (S3-Europe). S3 offers low latency, infinite data durability, and 99.99% availability.

6.1 Data Access and Management

S3 stores data in 2 levels: a top level of *buckets* and *data objects*. Buckets are similar to folders, and can hold an unlimited number of data objects. Each Amazon Web Services (AWS) account can have up to 100 buckets. Charging for S3 is computed at the bucket level. All costs levels are tiered, but the basic costs as of January 2010 are as follows: storage costs \$0.15/GB/month in the US, \$0.165 in N California, and \$0.15/GB/month in Europe; \$0.10/GB for uploads and \$0.17/GB for downloads; and \$0.01/1,000 PUT, COPY, POST, or LIST operations, \$0.001/10,000 GET and all other operations. Each data object has a name, a blob of data (up to 5 GB), and metadata. S3 imposes a small set of predefined metadata entries, and allows for up to 4 KB of user generated *{name, value}* pairs to be added to this metadata. While users are allowed to create, modify, and delete objects in a bucket, S3 does not support renaming data objects or moving them between buckets—these operations require the user to first download the entire object and then write the whole object back to S3. The search functions are also severely limited in the current implementation. Users are only allowed to search for objects by the name of the bucket—the metadata and data blob itself cannot be searched. Amazon S3 supports three protocols for accessing data: SOAP, REST, and BitTorrent. While REST is most popularly used for large data transfers, BitTorrent has the potential to be very useful for the transfer of large objects.

6.2 Security

While clients use a Public Key Infrastructure (PKI) based scheme to authenticate when performing operations with S3, the user's public and private keys are generated by Amazon and the private key is available through the user's AWS site. This means that the effective security is down to the user's AWS password, which can be reset through email. Since S3 accounts are linked directly to a credit card, this can potentially cause the user a lot of problems. Access control is specified using access control lists (ACL) at both the bucket and data object level. Each ACL can specify access permissions for up to 100 identities, and only a limited number of access control attributes are supported: read for buckets or data objects, write for buckets, and, finally, reading and writing the ACL itself. The user can configure a bucket to store access log records. These logs contain request type, object accessed, and the time the request was processed.

6.3 Integrity

The inner workings of Amazon S3 have not been published. It is hard to determine their approach to error detection and recovery. Based on the reported usage, there was no permanent data loss.

IV CONCLUSIONS

In this survey we have surveyed several approaches to data storage in cloud computing settings. Data centers have, and will continue, to be built out of commodity components.

The use of commodity components combined with issues related to the settings in which these components operate such as heat dissipations and scheduled downtimes imply that failures are a common occurrence and should be treated as such. In these environments, it is no longer a matter of if a system or component will fail, but simply when. Datasets are dispersed on a set of machines to cope with their voluminous nature and to enable concurrent processing on them. To cope with failures, every slice of the dataset is replicated a preset number of times; replication allows applications to sustain failures to machines that hold certain slices of the dataset and also to initiate error corrections due to data corruptions. The European Network and Information Security Agency (ENISA) recently released a document outlining the security risks in cloud computing settings. Among the concerns raised in this document include data protection, insecure or incomplete data deletion, and the possibility of malicious insiders. Other security related concerns that have been raised include data segregation, control over a data's location, and investigative support. Most of the systems that we have described here do not adequately address several of these aforementioned security concerns and also exacerbate the problem by designing systems that are presumed to operate in a trusted environment: this allows us to construct situations, in some of these systems, where a malicious entity can wreak havoc. Issues related to security and trust need to be thoroughly addressed before these settings can be used for mission critical and sensitive information.

ACKNOWLEDGMENT

We would like to express our gratitude to all those who gave us the possibility to complete this paper. Furthermore we would like to thank International Journal of Computer Science and Information Technology for giving this this esteemed opportunity for publishing this paper. We would also like to thank Director, H.C.T.M, Kaithal, as well as Head of Department(CSE), H.C.T.M., Kaithal for their collaboration and helping us to make resources availability.

REFERENCES

- [1] Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., et al. (2009). Above the clouds: A Berkeley view of cloud computing (University of California at Berkeley, Tech. Rep. No. UCB/EECS-2009-28).
- [2] Brewer, E. A. (2009). Towards robust distributed systems. *Principles of Distributed Computing (PODC) Keynote, Portland, OR*.
- [3] Brodtkin, J. (2008). Gartner: Seven cloud-computing security risks. Retrieved Infoworld, July 02 2008
- [4] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., et al. (2006). Bigtable: A distributed storage system for structured data. *Proceedings of Operating Systems Design and Implementation (OSDI'06), Seattle, WA*, 205–218.
- [5] Chappell, D. (2009a). *Introducing windows azure* (Tech. Rep., Microsoft Corporation).
- [6] Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). The google file system. *19th Symposium on Operating Systems Principles*, New York, NY, 29–43.
- [7] Gilbert, S., & Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, Available, Partition-tolerant web services. *ACM SIGACT News*, 33(2), 51–59.
- [8] Lamport, L. (2001). Paxos made simple. *ACM SIGACT News*, 32(4), 18–25.