Engineering Big-Data Systems Assignment 3:

Name: Abhilash Kosaraju **NEUID :**001205393 **Date:** 10/13/2018

PART-3. Read the following paper (attached), and write a short summary/report.

1. **Google's File System**

Google File System, a scalable distributed file system for large distributed data-intensive applications has been designed to provide a fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.The design has been driven by observations of the application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points. Presented the file system interface extensions designed to support distributed applications and many aspects of the design, and report measurements from both micro-benchmarks and real world use have been discussed.

Assumptions: Component failures on a routine basis due to the use of inexpensive commodity components. Modest volume of files will are stored in the system. Workloads consists of two types of streams (large streaming and small streaming) and also large sequential writes.The system must efficiently implement well-defined semantics for multiple clients that concurrently append to the same file.High sustained bandwidth is more important than low latency.

Interface: GFS provides a familiar file system interface, though it does not implement a standard API such as POSIX. Files are organized hierarchically in directories and identified by pathnames. Moreover, GFS has snapshot and record append operations. Snapshot creates a copy of a file or a directory tree at low cost.It is useful for implementing multi-way merge results and producer consumer queues that many clients can simultaneously append to without additional locking.

Architecture: A GFS cluster consists of a single master and multiple chunk-servers and is accessed by multiple clients. It is easy to run both a chunk-server and a client on the same machine, as long as machine resources permit and the lower reliability caused by running possibly flaky application code. Files are divided into fixed-size chunks. Each chunk is identified by an immutable and globally unique 64 bit chunk handle assigned by the master at the time of chunk creation. By default, store three replicas, though users can designate different replication levels for different regions of the file namespace.The master periodically communicates with each chunk server in HeartBeat messages to give it instructions and collect its state.The master maintains all file system metadata. This includes the namespace, access control information, the mapping from files to chunks, and the current locations of chunks. GFS client code linked into each application implements the file system API and communicates with the master and chunk servers to read or write data on behalf of the application.

Single Master: A single master vastly simplifies our design and enables the master to make sophisticated chunk placement.and replication decisions using global knowledge. However, we must minimize its involvement in reads and writes so that it does not become a bottleneck.Clients never read and write file data through the master. Instead, a client asks the master which chunk servers it should contact. It caches this information for a limited time and interacts with the chunk servers directly for many subsequent operations. The client then sends a request to one of the replicas, most likely the closest one. The request specifies the chunk handle and a byte range within that chunk. Further reads of the same chunk require no more client-master interaction until the cached information expires or the file is reopened. In fact, the client typically asks for multiple chunks in the same request and the master can also include the information for chunks immediately following those requested.

Chunk Size: Chunk size is one of the key design parameters. We have chosen 64 MB, which is much larger than typical file system block sizes. Each chunk replica is stored as a plain Linux file on a chunk server and is extended only as needed. Lazy space allocation avoids wasting space due to internal fragmentation, perhaps the greatest objection against such a large chunk size. First, it reduces clients' need to interact with the master, Second, since on a large chunk, a client is more likely to perform many operations on a given chunk.Third, it reduces the size of the metadata stored on the master.On the other hand, a large chunk size, even with lazy space allocation, has its disadvantages. A small file consists of a small number of chunks, perhaps just one. The chunk servers storing those chunks may become hot spots if many clients are accessing the same file. In practice, hot spots have not been a major issue because our applications mostly read large multi-chunk files sequentially. this problem by storing such executables with a higher replication factor and by making the batch queue system stagger application start times.

Metadata: The master stores three major types of metadata: the file and chunk namespaces, the mapping from files to chunks, and the locations of each chunk's replicas.All metadata is kept in the master's memory. The first two types (namespaces and chunk-mapping) are also kept persistent by logging mutations to an operation log stored on the master's local disk and replicated on remote machines.***In-Memory Data Structures,***Since metadata is stored in memory, master operations are fast. Furthermore, it is easy and efficient for the master to periodically scan through its entire state in the background. ***Chunk Locations,****The master does not keep a persistent record of which chunk servers have a replica of a given chunk. It simply polls chunk servers for that information at startup. The master can keep itself up-to-date thereafter because it controls all chunk placement and monitors chunk server status with regular HeartBeat messages.****Operation Log,*** *the operation log contains a historical record of critical metadata changes. It is central to GFS. Not only is it the only persistent record of metadata, but it also serves as a logical time line that defines the order of concurrent operations.*

Consistency Model: GFS has a relaxed consistency model that supports our highly distributed applications well but remains relatively simple and efficient to implement.***Guarantees by GFS,*** File namespace mutations (e.g., file creation) are atomic. They are handled exclusively by the master: namespace locking guarantees atomicity and correctness (Section 4.1); the master's operation log defines a global total order of these operations. The state of a file region after a data mutation depends on the type of mutation, whether it succeeds or fails, and whether there are concurrent mutations. Table 1 summarizes the result. A file region is consistent if all clients will always see the same data, regardless of which replicas they read from. A region is defined after a file data mutation if it is consistent and clients will see what the mutation writes in its entirety. Data mutations may be writes or record appends. A write causes data to be written at an application-specified file offset. A record append causes data (the "record") to be appended atomically at least once even in the presence of concurrent mutations, but at an offset of GFS's choosing.After a sequence of successful mutations, the mutated file region is guaranteed to be defined and contain the data written by the last mutation. GFS achieves this by (a) applying mutations to a chunk in the same order on all its replicas (Section 3.1), and (b) using chunk version numbers to detect any replica that has become stale because it has missed mutations while its chunk server was down. In the other typical use, many writers concurrently append to a file for merged results or as a producer-consumer queue. Record append's append-at-least-once semantics preserves each writer's output. Readers deal with the occasional padding and duplicates as follows.

SYSTEM INTERACTIONS: Leases and Mutation Order,A mutation is an operation that changes the contents or metadata of a chunk such as a write or an append operation. Each mutation is performed at all the chunk's replicas. We use leases to maintain a consistent mutation order across replicas. The master grants a chunk lease to one of the replicas, which we call the primary. ***Data Flow,*** We decouple the flow of data from the flow of control to use the network efficiently. While control flows from the client to the primary and then to all secondaries, data is pushed linearly along a carefully picked chain of chunk servers in a pipelined fashion. To avoid network bottlenecks and high-latency links (e.g., inter-switch links are often both) as much as possible, each machine forwards the data to the "closest" machine in the network topology that has not received it.Finally, we minimize latency by pipelining the data transfer over TCP connections. ***Atomic Record Appends,*** GFS provides an atomic append operation called record append. In a traditional write, the client specifies the offset at which data is to be written. Concurrent writes to the same region are not serializable: the region may end up containing data fragments from multiple clients. ***Snapshot,***The snapshot operation makes a copy of a file or a directory tree (the "source") almost instantaneously, while minimizing any interruptions of ongoing mutations. Our users use it to quickly create branch copies of huge data sets (and often copies of those copies, recursively), or to checkpoint the current state before experimenting with changes that can later be committed or rolled back easily.

MASTER OPERATION: The master executes all namespace operations, ***Namespace Management and Locking,*** Many master operations can take a long time: for example, a snapshot operation has to revoke chunk server leases on all chunks covered by the snapshot. We do not want to delay other master operations while they are running. Therefore, we allow multiple operations to be active and use locks over regions of the namespace to ensure proper serialization. One nice property of this locking scheme is that it allows concurrent mutations in the same directory. ***Replica Placement,*** A GFS cluster is highly distributed at more levels than one. It typically has hundreds of chunk servers spread across many machine racks. These chunk servers in turn may be accessed from hundreds of clients from the same or different racks. Communication between two machines on different racks may cross one or more network switches.The chunk replica placement policy serves two purposes: maximize data reliability and availability, and maximize network bandwidth utilization. ***Creation, Re-replication, Rebalancing,***When the master creates a chunk, it chooses where to place the initially empty replicas. It considers several factors. (1) We want to place new replicas on chunk servers with below-average disk space utilization. Over time this will equalize disk utilization across chunk servers. (2) We want to limit the number of "recent" creations on each chunk server. Although creation itself is cheap, it reliably predicts imminent heavy write traffic because chunks are created when demanded by writes, and in our append-once-read-many workload they typically become practically read-only once they have been completely written. (3) As discussed above, we want to spread replicas of a chunk across racks. ***Garbage Collection,*** After a file is deleted, GFS does not immediately reclaim the available physical storage. It does so only lazily during regular garbage collection at both the file and chunk levels.***Discussion,*** Although distributed garbage collection is a hard problem that demands complicated solutions in the context of programming languages, it is quite simple in our case. We can easily identify all references to chunks: they are in the chunk mappings maintained exclusively by the master. ***Stale Replica Detection,*** *Chunk replicas may become stale if a chunk server fails and misses mutations to the chunk while it is down. For each chunk, the master maintains a chunk version number to distinguish between up-to-date and stale replicas.*

FAULT TOLERANCE AND DIAGNOSIS, One of our greatest challenges in designing the system is dealing with frequent component failures. The quality and quantity of components together make these problems more the norm than the exception: we cannot completely trust the machines, nor can we completely trust the disks. Component failures can result in an unavailable system or, worse, corrupted data.***High Availability,*** Among hundreds of servers in a GFS cluster, some are bound to be unavailable at any given time. ***Fast Recovery,*** both the master and the chunk server are designed to restore their state and start in seconds no matter how they terminated. ***Chunk Replication,***Users can specify different replication levels for different parts of the file namespace. The default is three. The master clones existing replicas as needed to keep each chunk fully replicated as chunk servers go offline or detect corrupted replicas through checksum verification. ***Master Replication,*** The master state is replicated for reliability. Its operation log and checkpoints are replicated on multiple machines. A mutation to the state is considered committed only after its log record has been flushed to disk locally and on all master replicas.***Data Integrity,*** Each chunk server uses checksumming to detect corruption of stored data. Given that a GFS cluster often has thousands of disks on hundreds of machines, it regularly experiences disk failures that cause data corruption or loss on both the read and write paths.During idle periods, chunk servers can scan and verify the contents of inactive chunks. This allows us to detect corruption in chunks that are rarely read. Once the corruption is detected, the master can create a new uncorrupted replica and delete the corrupted replica. This prevents an inactive but corrupted chunk replica from fooling the master into thinking that it has enough valid replicas of a chunk.***Diagnostic Tools,*** Extensive and detailed diagnostic logging has helped immeasurably in problem isolation, debugging, and performance analysis, while incurring only a minimal cost. Without logs, it is hard to understand transient, non-repeatable interactions between machines. The RPC logs include the exact requests and responses sent on the wire, except for the file data being read or written. By matching requests with replies and collating RPC records on different machines, we can reconstruct the entire interaction history to diagnose a problem. The logs also serve as traces for load testing and performance analysis. **Real World**

Clusters, We now examine two clusters in use within Google that are representative of several others like them. Cluster A is used regularly for research and development by over a hundred engineers. A typical taskis initiated by a human user and runs up to several hours. It reads through a few MBs to a few TBs of data, transforms or analyzes the data, and writes the results backto the cluster. Cluster B is primarily used for production data processing.

CONCLUSIONS: The Google File System demonstrates the qualities essential for supporting large-scale data processing workloads on commodity hardware. While some design decisions are specific to our unique setting, many may apply to data processing tasks of a similar magnitude and cost consciousness.We started by reexamining traditional file system assumptions in light of our current and anticipated application workloads and technological environment. system provides fault tolerance by constant monitoring, replicating crucial data, and fast and automatic recovery. Chunk replication allows us to tolerate chunk server. The frequency of these failures motivated a novel online repair mechanism that regularly and transparently repairs the damage and compensates for lost replicas as soon as possible. GFS has successfully met our storage needs and is widely used within Google as the storage platform for research and development as well as production data processing. It is an important tool that enables us to continue to innovate and attack problems on the scale of the entire web.

2. MapReduce: Simplified Data Processing on Large Clusters

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real-world tasks are expressible in this model, as shown in the paper. The implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

As a reaction to the computing complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Programming Model:

The computation takes a set of input key/value pairs and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: Map and

Reduce. ***Map***, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function. The ***Reduce*** function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

Distributed Grep, Count of URL Access Frequency, Reverse Web-Link Graph, Term vector per host, Inverted Index, Distributed Sort are some of the examples of the map-reduce problems.

Implementations: Many different implementations of the MapReduce interface are possible. The right choice depends on the environment.

Execution Overview: The Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of M splits. The input splits can be processed in parallel by different machines. Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., hash(key) mod R). The number of partitions (R) and the partitioning function are specified by the user. When the user program calls the MapReduce function,

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.
2. One of the copies of the program is special – the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.
3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.
4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers.
5. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.
7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.

Fault Tolerance: Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.

Worker Failures: The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker has failed. Any map tasks completed by the worker are reset back to their initial idle state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to idle and becomes eligible for rescheduling.

MapReduce is resilient to large-scale worker failures. For example, during one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The MapReduce master simply re-executed the work done by the unreachable worker machines, and continued to make forward progress, eventually completing the MapReduce operation.

Locality: Network bandwidth is a relatively scarce resource in our computing environment. We conserve network bandwidth by taking advantage of the fact that the input data (managed by GFS [8]) is stored on the local disks of the machines that make up our cluster. GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines. The MapReduce master takes the location information of the input files into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task's input data (e.g., on a worker machine that is on the same network switch as the machine containing the data). When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no network bandwidth.

Backup Tasks: One of the common causes that lengthens the total time taken for a MapReduce operation is a "straggler": a machine that takes an unusually long time to complete one of the last few maps or reduce tasks in the computation. Stragglers can arise for a whole host of reasons. We have a general mechanism to alleviate the problem of stragglers. When a MapReduce operation is close to completion, the master schedules backup executions of the remaining in-progress tasks.

We have a general mechanism to alleviate the problem of stragglers. When a MapReduce operation is close to completion, the master schedules backup executions of the remaining in-progress tasks. The task is marked as completed whenever either the primary or the backup execution completes. We have tuned this mechanism so that it typically increases the computational resources used by the operation by no more than a few percents.

Counters: The MapReduce library provides a counter facility to count occurrences of various events. For example, user code may want to count the total number of words processed or the number of German documents indexed, etc.

Grep: The grep program scans through 1010 100-byte records, searching for a relatively rare three-character pattern (the pattern occurs in 92,337 records). The input is split into approximately 64MB pieces (M = 15000), and the entire output is placed in one file (R = 1).

Sort: The sort program sorts 1010 100-byte records (approximately 1 terabyte of data). This program is modeled after the TeraSort benchmark [10]. The sorting program consists of less than 50 lines of user code. A three-line Map function extracts a 10-byte sorting key from a text line and emits the key and the original text line as the intermediate key/value pair. We used a built-in Identity function as the Reduce operator. These functions pass the intermediate key/value pair unchanged as the output key/value pair. The final sorted output is written to a set of 2-way replicated GFS files (i.e., 2 terabytes are written as the output of the program).

Large-Scale Indexing: One of our most significant uses of MapReduce to date has been a complete rewrite of the production indexing system that produces the data structures used for the Google web search service. The indexing system takes as input a large set of documents that have been retrieved by our crawling system, stored as a set of GFS files. The raw contents for these documents are more than 20 terabytes of data. The indexing process runs as a sequence of five to ten MapReduce operations. Using MapReduce (instead of the ad-hoc distributed passes in the prior version of the indexing system) has provided several benefits:

The indexing code is simpler, smaller, and easier to understand, because the code that deals with fault tolerance, distribution and parallelization are hidden within the MapReduce library. For example, the size of one phase of the computation dropped from approximately 3800 lines of C++ code to approximately 700 lines when expressed using MapReduce.

The performance of the MapReduce library is good enough that we can keep conceptually unrelated computations separate, instead of mixing them together to avoid extra passes over the data. This makes it easy to change the indexing process. For example, one change that took a few months to make in our old indexing system took only a few days to implement in the new system.

The indexing process has become much easier to operate because most of the problems caused by machine failures, slow machines, and networking hiccups are dealt with automatically by the MapReduce library without operator intervention. Furthermore, it is easy to improve the performance of the indexing process by adding new machines to the indexing cluster.

Conclusions: The MapReduce programming model has been successfully used at Google for many different purposes. We attribute this success to several reasons. First, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing. Second, a large variety of problems are easily expressible as MapReduce computations.

Many things have been learned several things from this. First, restricting the programming model makes it easy to parallelize and distribute computations and to make such computations fault-tolerant. Second, network bandwidth is a scarce resource. A number of optimizations in our system are therefore targeted at reducing the amount of data sent across the network: the locality optimization allows us to read data from local disks, and writing a single copy of the intermediate data to local disk saves network bandwidth. Third, redundant execution can be used to reduce the impact of slow machines and to handle machine failures and data loss.

3. Bigtable: A Distributed Storage System for Structured Data

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance.

Introduction: Over the last two and a half years we have designed, implemented, and deployed a distributed storage system for managing structured data at Google called Bigtable. Bigtable is designed to reliably scale to petabytes of data and thousands of machines. Bigtable has achieved several goals: wide applicability, scalability, high performance, and high availability. In many ways, Bigtable resembles a database: it shares many implementation strategies with databases. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and al- lows clients to reason about the locality properties of the data represented in the underlying storage.Finally, Bigtable schema parameters let clients dynamically control whether to serve data out of memory or from disk.

A Bigtable is a sparse, distributed, persistent multi- dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes. The row keys in a table are arbitrary strings (currently up to 64KB in size, although 10-100 bytes is a typical size for most of our users). Column keys are grouped into sets called column families, which form the basic unit of access control. A column key is named using the following syntax: family:qualifier. Each cell in a Bigtable can contain multiple versions of the same data; these versions are indexed by timestamp. Bigtable timestamps are 64-bit integers.

API: The Bigtable API provides functions for creating and deleting tables and column families. It also provides functions for changing cluster, table, and column family metadata, such as access control rights. Bigtable can be used with MapReduce, a framework for running large-scale parallel computations developed at Google.

Building Blocks: Bigtable is built on several other pieces of Google infrastructure. Bigtable uses the distributed Google File System (GFS) [17] to store log and data files. A Bigtable cluster typically operates in a shared pool of machines that run a wide variety of other distributed applications, and Bigtable processes often share the same machines with processes from other applications. Bigtable de- pends on a cluster management system for scheduling jobs, managing resources on shared machines, dealing with machine failures, and monitoring machine status.

The Google SSTable file format is used internally to store Bigtable data. An SSTable provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings.

Bigtable relies on a highly-available and persistent distributed lock service called Chubby [8]. A Chubby service consists of five active replicas, one of which is elected to be the master and actively serve requests. The service is live when a majority of the replicas are running and can communicate with each other. Chubby uses the Paxos algorithm [9, 23] to keep its replicas consistent in the face of failure.Bigtable uses Chubby for a variety of tasks: to ensure that there is at most one active master at any time; to store the bootstrap location of Bigtable data.

Implementation: The Bigtable implementation has three major components: a library that is linked into every client, one master server, and many tablet servers.

The master is responsible for assigning tablets to tablet servers, detecting the addition and expiration of tablet servers, balancing tablet-server load, and garbage collection of files in GFS.

A Bigtable cluster stores a number of tables. Each ta- ble consists of a set of tablets, and each tablet contains all data associated with a row range. Initially, each table consists of just one tablet. As a table grows, it is automatically split into multiple tablets, each approximately 100-200 MB in size by default.

Tablet Location: A three-level hierarchy analogous to that of a B+- tree to store tablet location information.The first level is a file stored in Chubby that contains the location of the root tablet. The root tablet contains the location of all tablets in a special METADATA table. Each METADATA tablet contains the location of a set of user tablets. The root tablet is just the first tablet in the METADATA table, but is treated specially—it is never split—to ensure that the tablet location hierarchy has no more than three levels. The METADATA table stores the location of a tablet under a row key that is an encoding of the tablet's table identifier and its end row. We also store secondary information in the METADATA table, including a log of all events pertaining to each tablet (such as when a server begins serving it). This information is helpful for debugging and performance analysis.

Tablet Assignment: Each tablet is assigned to one tablet server at a time. The master keeps track of the set of live tablet servers, and the current assignment of tablets to tablet servers, including which tablets are unassigned. When a tablet is unassigned, and a tablet server with sufficient room for the tablet is available, the master assigns the tablet by sending a tablet load request to the tablet server.

When a master is started by the cluster management system, it needs to discover the current tablet assign- ments before it can change them. The master executes the following steps at startup. (1) The master grabs a unique master lock in Chubby, which prevents con- current master instantiations. (2) The master scans the servers directory in Chubby to find the live servers. (3) The master communicates with every live tablet server to discover what tablets are already assigned to each server. (4) The master scans the METADATA table to learn the set of tablets. Whenever this scan encounters a tablet that is not already assigned, the master adds the tablet to the set of unassigned tablets, which makes the tablet eligible for tablet assignment.

Each tablet is assigned to one tablet server at a time. The master keeps track of the set of live tablet servers, and the current assignment of tablets to tablet servers, including which tablets are unassigned. When a tablet is unassigned, and a tablet server with sufficient room for the tablet is available, the master assigns the tablet by sending a tablet load request to the tablet server. Bigtable uses Chubby to keep track of tablet servers. When a tablet server starts, it creates, and acquires an exclusive lock on, a uniquely-named file in a specific Chubby directory.The master is responsible for detecting when a tablet server is no longer serving its tablets, and for reassign- ing those tablets as soon as possible. To detect when a tablet server is no longer serving its tablets, the master periodically asks each tablet server for the status of its lock. When a master is started by the cluster management system, it needs to discover the current tablet assign- ments before it can change them. The set of existing tablets only changes when a table is created or deleted, two existing tablets are merged to form one larger tablet, or an existing tablet is split into two smaller tablets.

Tablet Serving: The persistent state of a tablet is stored in GFS, as illustrated in Figure 5. Updates are committed to a commit log that stores redo records. Of these updates, the recently committed ones are stored in memory in a sorted buffer called a memtable; the older updates are stored in a sequence of SSTables.When a read operation arrives at a tablet server, it is similarly checked for well formed ness and proper authorization.When a write operation arrives at a tablet server, the server checks that it is well-formed, and that the sender is authorized to perform the mutation. Authorization is performed by reading the list of permitted writers from a Chubby file. Incoming read and write operations can continue while tablets are split and merged.

Compactions: As write operations execute, the size of the memtable in- creases. When the memtable size reaches a threshold, the memtable is frozen, a new memtable is created, and the frozen memtable is converted to an SSTable and written to GFS. This minor compaction process has two goals: it shrinks the memory usage of the tablet server, and it reduces the amount of data that has to be read from the commit log during recovery if this server dies. Incoming read and write operations can continue while compactions occur.

Conclusion: Bigtable, a distributed system for storing structured data at Google. Bigtable clusters have been in production use since April 2005, and we spent roughly seven person-years on design and implementation before that date. As of August 2006, more than sixty projects are using Bigtable. Our users like the performance and high availability provided by the Bigtable implementation, and that they can scale the capacity of their clusters by simply adding more machines to the system as their resource demands change over time.

4. The Chubby lock service for loosely-coupled distributed systems

The Chubby lock service, which is intended to provide coarse-grained locking as well as reliable (though low-volume) storage for a loosely-coupled distributed system. Chubby provides an interface much like a distributed file system with advisory locks, but the design emphasis is on availability and reliability, as opposed to high performance.

The lock service Chubby is intended for use within a loosely-coupled distributed system consisting of moderately large numbers of small machines connected by a high-speed network. For example, a Chubby instance (also known as a Chubby cell) might serve ten thousand 4-processor machines connected by 1Gbit/s Ethernet. The purpose of the lock service is to allow its clients to synchronize their activities and to agree on basic information about their environment. The primary goals included reliability, availability to a moderately large set of clients, and easy-to-understand semantics; throughput and storage capacity were considered secondary. Chubby is expected to help developers deal with coarse-grained synchronization within their systems, and in particular to deal with the problem of electing a leader from among a set of otherwise equivalent servers.

Readers familiar with distributed computing will recognize the election of a primary among peers as an instance of the distributed consensus problem, and realize we require a solution using asynchronous communication; this term describes the behavior of the vast majority of real networks, such as Ethernet or the Internet, which allow packets to be lost, delayed, and reordered. The asynchronous consensus is solved by the Paxos protocol. Paxos maintains safety without timing assumptions, but clocks must be introduced to ensure liveness

Rationale: One might argue that we should have built a library embodying Paxos, rather than a library that accesses a centralized lock service, even a highly reliable one. A client Paxos library would depend on no other servers (besides the name service), and would provide a standard framework for programmers, assuming their services can be implemented as state machines. Indeed, we provide such a client library that is independent of Chubby. Nevertheless, a lock service has some advantages over a client library.

The arguments suggest two key design decisions: We chose a lock service, as opposed to a library or service for consensus, and we chose to serve small-files to permit elected primaries to advertise themselves and their parameters, rather than build and maintain the second service. Some decisions follow from our expected use and from our environment: A service advertising its primary via a Chubby file may have thousands of clients. Therefore, we must allow thousands of clients to observe this file, preferably without needing many servers. Clients and replicas of a replicated service may wish to know when the service's primary changes. This suggests that an event notification mechanism would be useful to avoid polling. Even if clients need not poll files periodically, many wills; this is a consequence of supporting many developers. Thus, caching of files is desirable. Our developers are confused by non-intuitive caching semantics, so we prefer consistent caching. To avoid both financial loss and jail time, we provide security mechanisms, including access control. A choice that may surprise some readers is that we do not expect lock used to be fine-grained, in which they might be held only for a short duration (seconds or less); instead, we expect coarse-grained use. Chubby is intended to provide only coarse-grained locking. Fortunately, it is straightforward for clients to implement their own fine-grained locks tailored to their application.

System structure: Chubby has two main components that communicate via RPC: a server, and a library that client applications link against. A Chubby cell consists of a small set of servers (typically five) known as replicas, placed so as to reduce the likelihood of correlated failure.

The replicas use a distributed consensus protocol to elect a master; the master must obtain votes from a majority of the replicas, plus promises that those replicas will not elect a different master for an interval of a few seconds known as the master lease. The master lease is periodically renewed by the replicas provided the master continues to win a majority of the vote. The replicas maintain copies of a simple database, but only the master initiates reads and writes of this database. All other replicas simply copy updates from the master, sent using the consensus protocol. Clients find the master by sending master location requests to the replicas listed in the DNS. Non-master replicas respond to such requests by returning the identity of the master. Once a client has located the master, the client directs all requests to it either until it ceases to respond, or until it indicates that it is no longer the master. If a replica fails and does not recover for a few hours, a simple replacement system selects a fresh machine from a free pool and starts the lock server binary on it.

Files, directories, and handles: Chubby exports a file system interface similar to, but simpler than that of UNIX.

Locks and sequencers: Each Chubby file and directory can act as a reader-writer lock: either one client handle may hold the lock in exclusive (writer) mode, or any number of client handles may hold the lock in shared (reader) mode. Like the mutexes known to most programmers, locks are advisory. That is the conflict only with other attempts to acquire the same lock: holding a lock called F neither is necessary to access the file F nor prevents other clients from doing- ing so. We rejected mandatory locks, which make locked objects inaccessible to clients not holding their locks:

Because Chubby's naming structure resembles a file system, we were able to make it available to applications both with its own specialized API and via interfaces used by our other file systems, such as the Google File System. This significantly reduced the effort needed to write basic browsing and namespace manipulation tools and reduced the need to educate casual Chubby users.

The namespace contains only files and directories, collectively called nodes. Every such node has only one name within its cell; there are no symbolic or hard links.

Events: Chubby clients may subscribe to a range of events when they create a handle. These events are delivered to the client asynchronously via an up-call from the Chubby library. Events include, file contents modified, child node added, removed and modified, chubby master failed over, a handle has become invalid, lock acquired and conflicting lock over another client. Events are delivered after the corresponding action has taken place. Thus, if a client is informed that file contents have changed, it is guaranteed to see the new data (or data that is yet more recent) if it subsequently reads the file.

API: Clients see a Chubby handle as a pointer to an opaque structure that supports various operations. Handles are created only by Open(), and destroyed with Close().

Open() opens a named file or directory to produce a handle, analogous to a UNIX file descriptor. Only this call takes a node name; all others operate on handles.Close() closes an open handle. Further use of the handle is not permitted. This call never fails.

Caching: To reduce read traffic, Chubby clients cache file data and node meta-data (including file absence) in a consistent, write-through cache held in memory. The cache is maintained by a lease mechanism described below and kept consistent by invalidations sent by the master, which keeps a list of what each client may be caching. The protocol ensures that clients see either a consistent view of the Chubby state or an error. When file data or meta-data is to be changed, the modification is blocked while the master sends invalidations for the data to every client that may have cached it; this mechanism sits on top of KeepAlive RPCs, discussed more fully in the next section. On receipt of an invalidation, a client flushes the invalidated state and acknowledges.

Only one round of invalidations is needed because the master treats the node as uncachable while cache invalidations remain unacknowledged. This approach allows reads always to be processed without delay; this is useful because reads greatly outnumber writes. An alternative would be to block calls that access the node during invalidation; this would make it less likely that over-eager clients will bombard the master with uncached accesses during invalidation, at the cost of occasional delays. If this were a problem, one could imagine adopting a hybrid scheme that switched tactics if overload were detected.

Sessions and KeepAlives:

A Chubby session is a relationship between a Chubby cell and a Chubby client; it exists for some interval of time and is maintained by periodic handshakes called KeepAlives. Unless a Chubby client informs the master otherwise, the client's handles, locks, and cached data all remain valid provided its session remains valid. A client requests a new session on first contacting the master of a Chubby cell. It ends the session explicitly either when it terminates, or if the session has been idle.

Each session has an associated lease—an interval of time extending into the future during which the master guarantees not to terminate the session unilaterally. The end of this interval is called the session lease timeout. The master is free to advance this timeout further into the future, but may not move it backward in time. The master advances the lease timeout in three circumstances: on the creation of the session, when a master fail-over occurs (see below), and when it responds to a KeepAlive RPC from the client.

Fail-overs:

When a master fails or otherwise loses mastership, it discards its in-memory state about sessions, handles, and locks. The authoritative timer for session leases runs at the master, so until a new master is elected the session lease timer is stopped; this is legal because it is equivalent to extending the client's lease.

If a master election occurs quickly, clients can contact the new master before their local (approximate) lease timers expire. If the election takes a long time, clients flush their caches and wait for the grace period while trying to find the new master. Thus the grace period allows sessions to be maintained across fail-overs that exceed the normal lease timeout.

Database implementation :

Backup: Every few hours, the master of each Chubby cell writes a snapshot of its database to a GFS file server [7] in a different building. The use of a separate building ensures both that the backup will survive building damage, and that the backups introduce no cyclic dependencies in the system; a GFS cell in the same building potentially might rely on the Chubby cell for electing its master. Backups provide both disaster recovery and a means for initializing the database of a newly replaced replica without placing the load on replicas that are in service.

Mirroring: Chubby allows a collection of files to be mirrored from one cell to another. Mirroring is fast because the files are small and the event mechanism informs the mirroring code immediately if a file is added, deleted, or modified. Provided there are no network problems, changes are reflected in dozens of mirrors worldwide in well under a second. If a mirror is unreachable, it remains unchanged until connectivity is restored. Updated files are then identified by comparing their checksums.

Among the files mirrored from the global cell are Chubby's own access control lists, various files in which Chubby cells and other systems advertise their presence to our monitoring services, pointers to allow clients to locate large data sets such as Bigtable cells, and many configuration files for other systems.

Mechanisms for scaling: Chubby's clients are individual processes, so Chubby must handle more clients than one might expect; we have seen 90,000 clients communicating directly with a Chubby master—far more than the number of machines involved. Several approaches can be considered for request processing at master,

- We can create an arbitrary number of Chubby cells; clients almost always use a nearby cell (found with DNS) to avoid reliance on remote machines. Our typical deployment uses one Chubby cell for a data center of several thousand machines.

- The master may increase lease times from the default 12s up to around 60s when it is under heavy load, so it need process fewer KeepAlive RPCs.
- Chubby clients cache file data, meta-data, the absence of files, and open handles to reduce the number of calls they make on the server.
- protocol-conversion servers that translate the Chubby protocol into less-complex protocols such as DNS and others.
- Two further approaches have been worked out to further scale the chubby further, they are ***Proxies*** and ***Partitioning.***

Proxies: Chubby's protocol can be proxied (using the same protocol on both sides) by trusted processes that pass requests from other clients to a Chubby cell. A proxy can reduce server load by handling both KeepAlive and read requests; it cannot reduce write traffic, which passes through the proxy's cache. If a proxy handles Nproxy clients, KeepAlive traffic is reduced by a factor of Nproxy, which might be 10 thousand or more.
Partitioning: Chubby's interface was chosen so that the namespace of a cell could be partitioned between servers. Although we have not yet needed it, the code can partition the namespace by directory. If enabled, a Chubby cell would be composed of N partitions, each of which has a set of replicas and a master. Unless the number of partitions N is large, one would expect that each client would contact the majority of the partitions. Thus, partitioning reduces read and write traffic on any given partition by a factor of N but does not necessarily reduce KeepAlive traffic.

Summary:

Chubby is a distributed lock service intended for coarse-grained synchronization of activities within Google's distributed systems; it has found wider use as a name service and repository for configuration information. Its design is based on well-known ideas that have meshed well: distributed consensus among a few replicas for fault tolerance, consistent client-side caching to reduce server load while retaining simple semantics, timely notification of updates, and a familiar file system interface.

Chubby has become Google's primary internal name service; it is a common rendezvous mechanism for systems such as MapReduce the storage systems GFS and Bigtable use Chubby to elect a primary from redundant replicas; and it is a standard repository for files that require high availability, such as access control lists.