INFO8002: Megastore: Providing Scalable, Highly Available Storage for Interactive Services

BULUT Stephan,[1] **HELD Jan**,[2] **MUKOLONGA Jean-David**,[3] and **SIBOYABASORE Cedric**[4]

[1] *stephan.bulut@student.uliege.be (s172244)*
[2] *jan.held@student.uliege.be (s165516)*
[3] *jdmukolonga@student.uliege.be (s170679)*
[4] *c.siboyabasore@student.uliege.be (s175202)*

I. INTRODUCTION

In today's world of interactive online services, it is of utmost importance that data storage systems be highly available, highly scalable, have low latency, rapid development and provide users with a consistent view of the data and fault tolerance. However, many of these requirements are in conflict with each other : relational databases (RDBMS) like MySQL provide a strong API but offer poor scalability. NoSQL databases such as Google's Bigtable scale well but have a limited API and loose consistency.
In this paper [1], Google presents Megastore, a data storage system that satisfies the aforementioned requirements by building a bridge between the advantages of relational databases and those of NoSQL datastores. To do so, Megastore uses synchronous replication to achieve high availability and consistent view of the data. Megastore partitions the data into smaller groups of data and the data inside of a partition can be accessed using ACID transactions similar to RDBMS transactions. The communication between different partitions is provided by a NoSQL database, which gives the scalability of a NoSQL database while having the advantages of a RDBMS.
Megastore provides replication and fault tolerance using the Paxos consensus algorithm [2], which has been used by Google for years [3].

II. METHODOLOGY

Availability is given to Megastore by the use of a synchronous fault tolerant log replicator. Because it's synchronous, all other systems that are in communication with this will receive the same data at the same time. Since it is fault-tolerant, if a system fails or a packet is lost, it's able to recover from that.
Scalability is provided by the partitioning of data into small databases. Each of these small databases has its own separate log that keeps track of everything that happens to it and all this is stored in a NoSQL data store, which provides high scalability.
NoSQL gives a way of organizing data across disparate servers but it doesn't give a way of making replications across those servers. The authors first evaluated three different techniques for data replication (Synchronous Master/Slave, Asynchronous Master/Slave, Optimistic Replication) but discarded all of them because they could lead to data losses, inability to perform ACID transactions or involve a heavyweight master which can fail and make the entire system not fault-tolerant. Instead, the authors use the Paxos consensus algorithm[2]. This removes the need for a master, which means that all nodes are symmetric peers and the single point of failure problem is not encountered and this increases availability. For the replication scheme, the authors replicate a write-ahead log over a group of symmetric peers. Having symmetric peers means any node can initiate reads and writes. Log appends, which are the creation of new transactions, cause a block until the majority of replicas accept it.

To scale their replication scheme and maximize performance of the underlying datastore, the authors give applications fine-grained control over their data's partitioning and locality. They partition the data into entity groups which are collection of individual entities. An entity is thus the fundamental unit of a Megastore record[4]. Entity groups are combined together to create partitions that are replicated and then distributed over a geographically wide area to increase the locality; by spreading copies of the data around to geographically distinct areas, the average distance between the average user and the nearest copy of that data is decreased. It also helps to improve the availability of the system by reducing the likelihood of all of the copied data being in a failed state. Each of these partitions inside of a single data center are stored inside of a NoSQL data store.
Entities inside of an entity group are mutated using single-phase ACID transaction. Entity groups communicate with each other through sending and receiving entities. Across logically distant entity groups, the authors use asynchronous messaging.
Selecting appropriate entity group boundaries is an important step, as too fine-grained boundaries force excessive cross-group operations and placing too much unrelated data in a single entity group degrades throughput.

Once the data has been partitioned, Megastore stores it in Bigtable, a NoSQL datastore which is another Google product [5]. The authors use Bigtable because it provides scalable and fault-tolerant storage within a single data center and it allows applications to minimize

latency by controlling the placement of their data: if they can place their data physically near their users then they can reduce latency. They assign each entity group to the region from which it is accessed most and within that region, the data is replicated three or four times.
Just by using BigTable, MegaStore provides a scalable, fault-tolerant, low latency data storage system, which is half of the requirements the authors set out to solve at the beginning of the paper.

Megastore maps this architecture onto a feature set carefully chosen to encourage rapid development of scalable applications. The authors wanted a *cost-transparent* API: they wanted developers to have an intuitive understanding of the runtime cost of performing a particular transaction. This choice was justified by the fact that Megastore is designed to work with high-volume workloads which would rather have predictable performance, reads dominate writes in the applications targeted by the authors and Bigtable enables the storing and querying of hierarchical data in a very straightforward manner.

With all this in mind, Megastore created their own custom schema and custom data model to offer control over physical locality. The schema is also designed to make use of hierarchical data. Denormalization of the data allows Megastore to reduce the number of joins in order to get a more predictable query language. Join is not provided as a fundamental command in Megastore: instead, it must be implemented in application.

Megastore defines a data model that is somewhere between RDBMS and NoSQL. The data model is defined in a schema which is strongly typed: the data type must be explicitly specified at variable creation. There are fundamentally three types: Strings (different types of them), Numbers (different types of them) and Google's Protocol Buffers [6]. Each schema has a set of tables, each containing entities, which in turn contain properties. These properties are very similar to the columns in a traditional RDBMS. Just like in RDBMS, every entity has to be uniquely identifiable by a primary key in Megastore. Primary keys are formed using one or more properties.

Each entity group has its own set of local indexes which are used to find data within that group. On the other hand, global indexes span entire entity groups and are used to find entities when it is not know which entity group contain entities.

By adding a `STORING` clause to an index, applications can store additional properties from the primary table for faster access at read time.

All data has to be mapped to and stored in Bigtable. The concatenation of a Megastore table name and its properties names gives the Bigtable columns names. This allows different different entities to be mapped into the same row without collision because each entry is unique.

The Bigtable column name is a concatenation of the Megastore table name and the property name, allowing entities from different Megastore tables to be mapped into the same Bigtable row without collision.

Bigtable rows primarily store the transaction and replication metadata for the entity group, which allows the creation of atomic updates through a single Bigtable transaction. Each entry in an index is a single BigTable row.

Megastore entities can be thought of as miniature databases that have access to ACID semantics. Each entity group has write-ahead log to which a transaction writes its mutations. Then, the mutations are applied to the data.

BigTable allows the authors to store multiple values in the same row and column as long as they have different timestamps; each row-column pair can be thought of as 3D. The authors use this feature to implement multiversion concurrency control (MVCC) so that the values resulting from a mutation are written at the timestamp of their transaction.

A read action uses the most recent completed transaction by default. Because MegaStore uses Bigtable and because Bigtable has this notion of timestamping, it means that reads and writes do not need to block each other because read actions use the most recently completed transaction while write actions write to a later timestamp. Megastore supports three types of read: current, snapshot and inconsistent reads. Both current and snapshot reads are performed within a single entity group and read at the timestamp of the latest committed transaction. Before doing this action, current reads first ensure that all writes that have previously been committed are applied. Inconsistent reads ignore the state of the log and read the latest values directly.
Megastore supports one type of write transactions. They always begin with a current read to determine the next available log position. Then, it reads from Bigtable and gathers all mutations, assigns a timestamp to them and then append it to a log which using Paxos. The protocol uses optimistic control which means that multiple writes can be sent to the same log using Paxos, but only one of them will win. The non-victorious writes will be aborted and their writers will have to retry operations. To reduce contention, advisory locking or batching writes can be done. The writes are then applied to the individual entities and individual indexes in Bigtable. Then, there's a cleanup. All the temporary data is garbage collected.

Queues are used to allow entity groups to communicate with each other. They are used for sending and receiving atomic messages. If the sending and receiving queues are the same then the delivery is synchronous. Otherwise, it's asynchronous. Megastore supports two-phase commit but discourages its use because queues exist; they have lower latency and reduce the risk of contention. Megastore supports other features such as integrated backups (periodic full snapshot, incremental snapshot,..) to recover from programmer or operator errors and it also supports entity and transaction logs encryp-

tion.

For synchronous replication, Megastore uses a slightly modified version of Paxos that offers lower latency to provide consistent view of all data stored in its replicas. Reads and writes can be initiated from any replica and ACID semantics are provided regardless of the replica. Replication is performed per entity group by sending a transaction log to a quorum of replicas. The original Paxos algorithm [2] is not suitable for high-latency network links because it requires multiple rounds of communication. In order to make Paxos practical, real world systems reduce the number of roundtrips perform by write and read operations. To reduce the amount of latency involved, many of these systems use a master to which all reads and writes are directed. The master is always up-to-date because it participates in every write. It can serve reads of the current consensus state without any network communication and it can batch writes to improve throughput. Writes are reduced to a single round of communication [3]. However, this master-based approach comes with problems: it limits flexibility for reading and writing, transactions must be performed near the master replica if a small amount of network latency is desired and extra resources are needed to support the master. To solve these issues, Megastore uses the notion of fast reads and fast writes. The authors recognized that writes usually succeed on all replicas so it was realistic to allow local reads everywhere, which give low latency in all regions. To achieve fast reads, the authors allowed local reads everywhere which give low latency in all regions because they recognized that writes usually succeed on all replicas. The authors designed a service called the Coordinator attached to each replica. It tracks every entity group for which its replica has observed all Paxos writes. To achieve fast writes, instead of using dedicated masters, Megastore use *leaders*. Each log position has an instance of Paxos running and chooses a replica as a leader which arbitrates which value may use proposal number zero. Writes have to be submitted to the leader before they are submitted to other replicas, which minimizes latency. The next write's leader is selected by using the closest replica. Megastore supports full replicas which are the types of replicas that have been so far mentioned but it also supports witness replica which only vote in Paxos rounds and store some write-ahead data. It also supports read-only replicas which store full snapshots of the data and reflect a consistent view of some point in the recent past.

Megastore is deployed through a client library and auxiliary servers. Each application server has a designated local replica. Replication servers create an intermediary between the application that is performing Paxos operations and the local Bigtable, which helps to reduce the number of round-trip communications needed. They can also detect and fix failed writes by submitting No-op values via Paxos to the log record with the failed write.

Read operations require five steps : querying the local replica's coordinator, determining the highest possibly-committed log position and selecting a replica that has applied through that log position, catching it up to the maximum known log position, updating the coordinator and then querying the data by reading the selected replica using the timestamp of the selected log position. Write operations always start with a read so they include those five steps but also 5 other steps including asking the leader to accept a proposed value, run the Paxos Prepare phase at all replicas with a higher proposal number, asking all the replicas to accept the proposal number, invalidating the coordinator at all full replicas that did not accept the value and applying these mutations to as many replicas as possible.

Coordinator processes run in each datacenter and are vulnerable to network failures. To detect these failures and recover from them, the authors use Google's Chubby [7] locking service. Coordinators obtain specific Chubby locks in remote datacenters at startup and must hold a majority of them to process requests. Otherwise, they revert to a conservative state where they consider all entity groups in they purview to be out-of-date.
A write timeout of around 10 seconds is possible when all writers must wait for the coordinator's Chubby locks to expire. This timeout is considered acceptable by the developers of Megastore because it is short and allows fast local reads. Other types of network failures such as asymmetric network partitions have been faced by the authors only a couple of times.

Coordinators are also vulnerable to race conditions. Invalidate messages are always safe but this is not always the case for validate messages. A validate message from an early write and a later invalidate are always safe. Higher numbered invalidates always trump lower numbered validates. Crashes are detected using an epoch system. Although coordinators are components that may fail, they increase local reads speed from any datacenter which is valuable because in Megastore, reads are more prevalent than writes.
In this paper, the authors introduce a slightly modified version of Paxos which has a unique set of trade-offs. Application servers in multiple datacenters may initiate writes to the same entity group but only one of them will succeed while the others need to retry their transactions. Limiting the per-entity-group commit rate to a few writes per second per entity group yields insignificant conflict rates. Bulk processing also reduces the conflict rate.

Megastore's performance can degrade when a particular full replica becomes unreliable or loses connectivity. The authors thought of a number of ways to respond to these failures, including rerouting traffic to application servers near other replicas, disable the replica's coordinators, ensuring that the problem has a minimal impact on write latency. Disabling the replica entirely can also be done but it is a rarely used action.

Development of Megastore was aided by a strong emphasis on testability. They wrote many cheap test as-

sertions and the code had a through unit coverage in their testing. The most effective bug-finding tool was a network simulator which created a set of pseudo-random tests. It tested all possible orderings and delays and it also did it deterministically. If bugs were found in a sequence of events, the sequence was added to the unit tests. With this pseudo-random simulation, the authors found problems that they would never have expected or tested on their own.

III. CONTRIBUTIONS AND RESULTS

With Megastore, the authors developed a data model and storage system that allows rapid development of interactive applications where high availability and scalability are built-in from the start.

The authors also provided an implementation of the Paxos replication and consensus algorithm optimized for low-latency operation across geographically distributed datacenters to provide high availability for the system. Megastore uses Paxos for the replication of primary data across datacenters on every write and is the largest system to do so, when many other systems only use it for locking, master election, or replication of metadata and configurations.

After the deployment of Megastore in the real world, the authors' expectations were met, as the majority of reads were indeed local and that writes cost about one roundtrip. They found that most applications that use Megastore have found the latency tolerable and actually use the Megastore schema language.

Megastore has been deployed within Google for several years and used by more than 100 production applications as a storage service. Most customers see extremely high levels of availability and average read latencies are tens of milliseconds, depending on the amount of data, showing that most reads are local. Average write latencies are of 100–400 milliseconds, depending on the distance between datacenters, the size of the data being written, and the number of full replicas.

IV. DISCUSSION

Throughout the paper, the authors provide solutions or justifications for some shortcomings that their methods may encounter. For example, because Megastore uses an optimistic concurrency, a lot of concurrency conflicts can be met when a large number of writes are issued. The authors propose advisory locking and batching writes to reduce or avoid contention. They also say that in case a coordinator faces asymmetric network partitions, the operator could manually disable the coordinator because this condition is faced only a handful of times.

However, some shortcomings remain without solutions. During development, instead of fixing persistent underlying problems, the authors masked them to achieve the appearance of fault tolerance. Instead of actually addressing these problems, they band-aid fixed them. For us, it was interesting to read that even Google could encounter this kind of problem and respond with such "solution".

Another issue that Megastore can face is the issue of anomaly chain gang throttling encountered, when, if slowness is interpreted as a fault, and tolerated, the fastest majority of disparate machines processes requests at their own pace, reaching equilibrium only when slowed down by the load of the laggers struggling to catch up. Moreover, achieving good performance for more complex queries requires attention to the physical data layout in Bigtable.

We could note that at the beginning of the paper, the authors criticize master/slave approaches for data replication but later in the paper, they weren't able to circumvent the use of a master even though they slightly modify Paxos.

We could also note that the authors emphasize the importance of good partitioning, but give no instructions on how to appropriately partition the data. Rather, it seems to be application-specific, which means we have to study our expected usage patterns and make our best guess at it. This means it is very possible to make bad partitioning decisions and get poor performance.

[1] James C. Corbett JJ Furman Andrey Khorlin James Larson Jean-Michel Leon Yawei Li Alexander Lloyd Vadim Yushprakh Jason Baker, Chris Bond. Megastore: Providing scalable, highly available storage for interactive services, 2011.

[2] L. Lamport. The part-time parliament, 1998.

[3] R. Griesemer T. D. Chandra and J. Redstone. Paxos made live: an engineering perspective, 2007.

[4] P. Helland. Life beyond distributed transactions: an apostate's opinion. CIDR.

[5] S. Ghemawat W. C. Hsieh D. A. Wallach M. Burrows-T. Chandra A. Fikes F. Chang, J. Dean and R. E. Gruber. Bigtable: A distributed storage system for structured data. 26(2):1–26, 2008.

[6] L. Lamport. Google protocol buffers: Google's data interchange format, 2008.

[7] M. Burrows. The chubby lock service for loosely-coupled distributed systems. *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, page 335–350.