DIDA-GSTORE

Lucas Lobo, 86464; João Freitas, 87671; Bruno Freitas, 98678
Instituto Superior Técnico
Design and Implementation of Distributed Applications
Group 7

1. Introduction

In this project we developed two versions of a geo-replicated storage system. A base version with a strong consistency but low availability and an advanced version with eventual consistency and high availability.

The base version takes the master-slave approach and offers linearizability by using distributed locking. In the advanced version all servers have the same responsibility. This version offers high availability and eventual consistency by using object versioning.

This project was implemented using C# with an async/await programming style, and all methods that can run asynchronously will do so to avoid using resources while waiting. For this reason, no explicit thread pool or request queue was implemented. In the case of the server, when a request arrives, a task is automatically created by gRPC/C#.

Servers are only created after the first non configuration command, meaning that the server is not created as soon as the server command is executed. It is advised to run a wait X command, which will trigger the instantiation. The replication-factor command is ignored, and it is expected for the partition commands to have the semantically and syntactically correct values.

2. Base Version

2.1. Protocol

2.1.1 Read Requests

Read requests may be sent to any server of the partition. When a server receives a read request, it acquires a local shared lock on the object, which means other read requests can be processed concurrently. Once the request is finished, it releases the lock. A single read request takes one round trip to complete plus processing and blocking time. Blocking time is time the request must wait before acquiring the lock.

2.1.2 Write Requests

Write requests must be sent to the master. When a master receives a write request, it acquires an exclusive lock on the object. Then, the master sends lock requests to each replica, which also acquire exclusive locks for that object. After getting the responses from the replicas, the master writes the new value to the object, releases the lock, and sends the new value to the remaining servers. The replicas then update the object, release the locks, and return to the master. Finally, the master returns to the client. A single write request from a client has a duration of 3 RTT (between client-master and master-replicas) plus processing time and blocking time.

2.1.3 Blocking Time

The blocking time can be a substantial factor in the time it takes for one client request to be processed. Since each write request locks an object for around 1 RTT plus processing time, multiple writes from different clients to the same object will cause a cascading effect in delays. In case a read operation follows the write operations, the same issue occurs.

2.1.4 Linearizability Issue

After further analysis on the protocol, it was noticed that it contains a linearizability problem regarding the write operation that could be easily fixed. The current protocol says that after the master receives the lock confirmations from the replicas, it can write to the object and unlock it. In reality, this means that if the master unlocks the object, a client reads the new value and the master crashes before it sends the new value to the replicas, the system ends up in an illegal state. To fix this, the master must first send the requests to the

replicas and only then write and unlock it. Additionally, at this point the master may return immediately to the client saving up to one additional round trip to the replicas, for a total of 2 RTT instead of 3. This solves the linearizability issue because once the write requests are sent to the replicas it is guaranteed that they receive it. The master may also return to the client immediately after writing and unlocking because waiting for the replicas provides no benefits. Once the object is unlocked, any other client can access it, which means the client responsible for the write request does not need to wait any longer.

This modification has no impact on the blocking time from section 2.1.3, since the object is unlocked as soon as the requests are sent.

2.1.5 Crash Recovery in Servers

When a server crashes, the remaining servers need to be reconfigured for the system to recover from the crash.

If the crashed server is a replica, the recovery is simple, as the master simply needs to stop sending further requests to this replica. Other replicas do not need to be aware of this crash since they do not communicate between themselves. No linearizability issues occur from this situation, as replicas do not control changes in data.

If the crashed server is a master, it is necessary to reconfigure the partition to elect a new master. After the replicas detect the crash of a server, they all choose the new master. This is enough to guarantee linearizability, as all previously written data is still accessible in the replicas. If the new master is also dead due to an inconsistent view of the aliveness of the servers, a second new master will be picked, and so on. This can occur when a replica crashes and other replicas are not informed. If the master crashes and the dead replica is elected as a master, they will quickly realize this and choose another master.

When a master crashes during a write request from the client, the replicas might be left with acquired locks on objects. At this point, there are three possible states for each object in the system:

- All replicas have the object unlocked, meaning that the write to the master was successful and any read to the replicas will return the new value.
- All replicas have the object locked, meaning that the write operation was never executed and no further operations will be possible for this object.
- Some replicas have the object locked and some have the object unlocked. Replicas with the object locked will not allow further operations on the object while replicas with the object unlocked will.

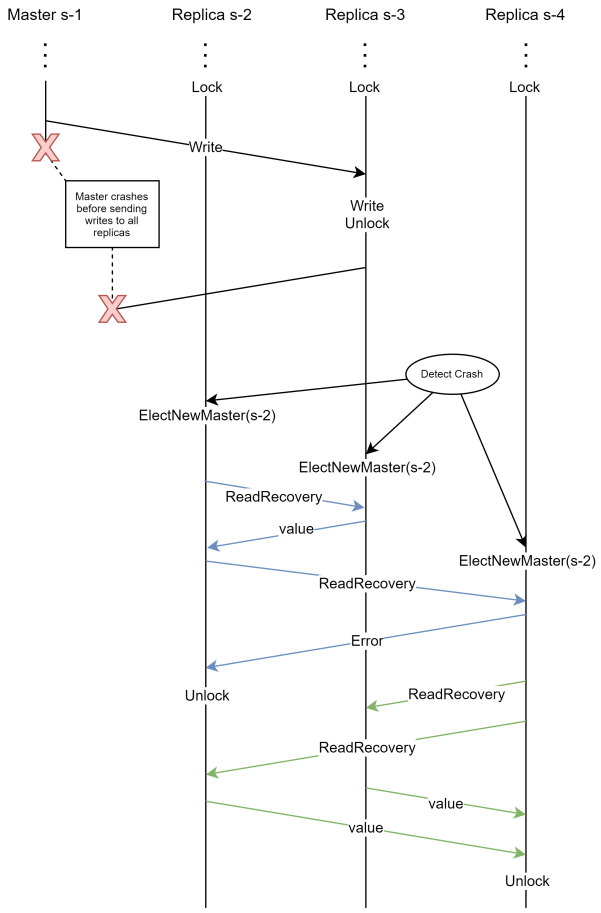

Figure 1. Recovery of replicas

All three states still offer linearizability, since it is impossible for an old value to be read after a new value has been successfully written. This assumes the modifications of section 2.1.4.

Replicas that have the object unlocked do not require further action since they contain the most recent value. Replicas that have the object locked must send Read Recovery operations to all other replicas in the partition. Replicas that have that object locked will reply with an error, meaning that they themselves do not know the most recent value. Replicas that have that object unlocked will reply with the most recent value. In the end, all replicas of the partition will be aware of the most recent value if there is one. If there isn't, all replicas will have the old value. After the Read Recovery operation is complete and the value updated, the replica will unlock the object.

In the end, the system will be consistent and all replicas will be available, as all replicas contain the ob-

ject unlocked exclusively with the new or the old value, but never a combination of both. Figure 1 illustrates the algorithm.

The recovery time of a partition depends on the time it takes for replicas to detect a crashed master. Objects that require recovery will take an additional round trip to the replicas to recover.

2.1.6 Crash Recovery in Clients

When a client detects a crashed server it simply declares the server as dead and stops sending further requests to that server. If the server is a master of some partition, the client will send a Get Master request to some replica of that partition. The client will repeat this operation periodically until it gets a new master, since the replica might not have yet reconfigured itself after the crash. Whenever an operation fails due to a crashed server, the clients will retry the operation on an appropriate server. If it is a read operation, the client will resend the request to another random server of the partition. If it is a write operation, the client will get the master and retry the operation.

2.2. Implementation

2.2.1 Master Election

The selected local criteria is based on the lexicographic order of the server id, which allows for master election without communication between replicas. Another option such as a hash function could have been used to avoid giving priority to the same servers.

2.2.2 Locking Mechanism

When a master sends a lock request to a replica, the replica must acquire an exclusive lock on the object. Since the new value for that object is only received on the replica on a follow-up request, the locking and writing operations occur on different tasks and potentially threads. This means that the locking mechanism must not be thread-affine. The locking mechanism ReaderWriterLockSlim from C# does not meet this requirement.

The naive approach to solve this problem would be to keep some sort of variable that keeps the locking state of the object. The problem with this approach is that it does not ensure atomicity and it does not offer any event-like interface which notifies the waiters.

To fix both these issues, an alternative version of the original locking mechanism from C#, called ReaderWriterLockEnhancedSlim, was implemented. The core difference is that it returns a lockId when a writer lock is acquired, which can be used to reacquire the lock in another thread. This way, the first request a replica receives can acquire a lock and return the id to the master, which can then use this id to reacquire the lock when updating the value in the next request. Additionally, the master can use await within a lock while waiting for replicas to reply, giving control back to the CPU and not wasting a thread in a blocked state.

Exclusive locks have priority over shared locks, which results in write operations having priority over read operations for the same object. In theory, a read request could get blocked for a long time if a series of write requests arrive at the server, but since requests are not dropped, at some point the request is processed.

2.2.3 Crash Detection in Servers

During normal operation, replicas do not initiate contact with the master. For this reason, replicas have no way of knowing when a master crashes. To handle this issue, replicas will periodically send heartbeats to their masters. If a heartbeat fails or takes too long, the replica will assume the master as dead and initiate the recovery process as explained in section 2.1.5. Each server that is a replica for some master will execute this algorithm. Servers never communicate to inform about crashes, meaning that each server finds the crash by itself.

The master has two ways of detecting the crash of one of its replicas. During normal operation, when a master sends a request to a replica and the request fails out times out, the master will declare the replica as dead and initiate the recovery process. Additionally, to speed up the crash detection process, the master will keep track of the heartbeats sent by the replicas using a watchdog. In case the watchdog expires, the replica is also declared dead. Both the heartbeats and the watchdogs are only initiated after a grace period to avoid detecting a crash when the server is still initiating.

Since servers may be replicas and masters at the same time for different partitions, it is possible that unnecessary requests are sent. For example, if server A is a master for server B in one partition, and server B is master for server A in another partition, both servers will send heartbeats to each other and keep track of each other's heartbeats using watchdogs. This is unnecessary, but due to the added complexity of computing the minimum required heartbeats between the servers the fix was not considered worth it.

2.2.4 Crash Detection in Clients

Clients detect server failures during normal operation. Every time a client sends a gRPC request to a server, the request might fail due to a crash. If this is the case, the client will reconfigure itself as explained in section 2.1.6.

2.2.5 Remarks

Since servers are only aware of crashes if they have a master-replica relationship, and clients are only aware of crashes if they try to communicate, the status command might show a "wrong" view of the aliveness of the servers.

3. Advanced Version

3.1. Protocol

3.1.1 Versioning

When an object is created it is assigned the version 1, this version is incremented when the object is updated. This version is used by the servers that receive the broadcast requests to decide which version to maintain if they already have a previous copy of that object stored. If the version that the server receives is more recent than the one that is stored, it replaces the object value with the new one, which represents a last-writer-wins approach.

3.1.2 Read Requests

Read requests work the same way as in the base version and also take 1 RTT to complete. The only difference is in the return value due to the more relaxed consistency model, which might return an older outdated value.

3.1.3 Write Requests

Write requests take 1 RTT and may be sent to any server of the partition. When a server receives a write request, it acquires an exclusive local lock on the object, updates the object, releases the lock, broadcasts the update to the other servers in the partition and returns to the client without waiting for the confirmation of the servers. This is possible by leveraging one assumption of the project: there are no network failures, meaning that all the other alive servers in the partition will receive this update. The other important part of the write operation is the versioning property that helps to decide on which to discard or maintain an object value. These properties combined ensure the eventual consistency. The only case where an issue might arise is when there are concurrent updates in the same object which result in the same version, leading to a tie. If there is a tie and the server that sent the update request has a smaller id than the id of the receiver, the update is accepted, otherwise it is rejected. For this we also send the server id in the update request.

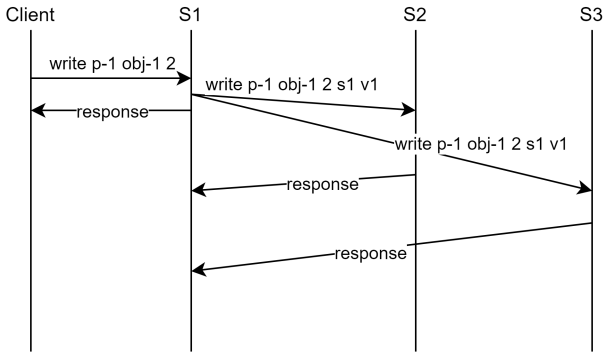

Figure 2. Write Request in advanced version

Figure 2, shows an example of a write request of a new object.

3.1.4 Crash Recovery

When a server or a client detects a crash in another server, it simply stops sending further requests to the server by locally declaring it dead. Since there are no masters and replicas, no further reconfiguration is required.

3.2. Implementation

3.2.1 Crash Detection in Servers

This version of the protocol does not contain masters or replicas, an explicit crash detection module such as the one used in the base version is not required. All servers communicate with all servers from the same partition, meaning that if a server crashes, all the servers that must be aware of this crash are eventually informed. Every time a server propagates a write to another server, the request might timeout or crash. In this case, the server will declare the crashed server as dead and initiate crash recovery.

3.2.2 Crash Detection in Clients

When a client sends an operation to a server, the operation might timeout or crash. In this case, the client will declare the server as dead and initiate the crash recovery. After this, it will retry the request on another server.

3.2.3 Remarks

In theory, the update requests to all replicas should be broadcasted to all the other servers. But gRPC doesn't allow that operation, so in the code they are sent one at the time, what potentially represents a failure in the protocol because if the server crashes before sending all the requests to the servers, some servers won't receive the most recent version of all the objects. This can cause consistency anomalies.

An optimization to avoid read inconsistencies (i.e., read an older version of an object than the one read before) could have been implemented in the following way: maintain a previously read object list that would be checked every time the read request returns. If the object version returned is older than the one in the list, it would give the value in the list.

3.3. Applications

Our advanced version provides high availability, low latency and eventual consistency. Consequently it wouldn't be good for applications or systems where a stronger consistency model is needed. Social apps like Facebook or Instagram can cope with more relaxed consistency models because it isn't crucial for a post or a like, for example, to be immediately available to everyone in the world. In addition, as they can benefit from a more relaxed consistency these systems can provide high availability and low latency.

A good application for the advanced DIDA-GStore is a high frequency sensor data logging where real-time accuracy is not a requirement. This would allow data to be stored on the system with the guarantee of durability while also allowing for a high throughput.

4. Comparison

4.1 Read Requests

In the base version, a read operation acquires a local shared lock on the object. This lock can only be acquired if no write operations are queued or in progress. Due to this reason, a read operation takes a minimum of one round trip to complete (i.e., between the client and the servers), but might extend to increasingly long times if a lot of write operations are queued. This is also the case for the advanced version, with the difference being that since write operations take longer to complete in the base versions, the respective blocking time will also be significantly longer.

4.2 Write Requests

In the base version, a write operation acquires a distributed exclusive lock on the object. In the advanced version, a write operation acquires a local exclusive lock on the object. This results in different blocking times. In the case of the base version, the blocking time is at least the duration of the round trip between the master and the replica plus processing time. In case of the advanced version, the blocking time is simply the processing time. Blocking time has serious impacts on both types of operations on the same object.

4.3 Crash Recovery

Since the advanced version does not require a special type of crash detection and recovery, the base version will always perform worse when recovering. First, it is necessary to detect the crash and elect a new master. Then, it is necessary to clean any wrong locking state and rewrite new values that were lost on crash.

5. Evaluation

The main difference between the base and the advanced version are the blocking times that occur during its operations and the number of round trips that must be completed to execute one request. For this reason, the evaluation was performed in a way that highlights these cases. All the tests were performed with one partition and three servers.

5.1. Remarks

All tests were executed on a system with an i7 3770k and 16GB of RAM. Due to testing limitations, servers and clients were all run on the same machine, which is not optimal. This means that the performance of the servers is not independent from each other and that the clients might influence the servers. It was noticed that at a higher number of clients (more than 30), the last client would be launched with some delay after the first. With more than 50 clients, this delay started being too long which meant test results were no longer useful. The reason is that the machine was busy launching clients and resources were being used for tasks other than processing requests.

5.2. Processing Time

To highlight the issue of section 2.1.3, which happens to the cascading delay from accumulated writes

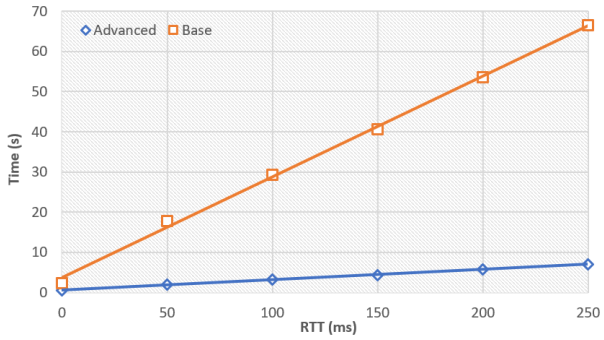

Figure 3. How RTT affects processing time

on the same objects, the test represented in figure 3 was performed. It shows how the processing time varies with the RTT.

In this test, there are 10 clients concurrently executing 25 writes each to the same object for a total for 250 writes. It is easy to see that the base version is largely affected by the RTT due effect it has on blocking time, since requests cannot execute while the object is locked, and the object cannot be unlocked until the round trip to the replicas is completed.

5.3. Latency

To measure the latency, tests were run using a varying number of clients and an RTT of 30ms on random objects. Since the objects are random, it effectively means that there are next to no blocking time issues. At each client, each request latency was logged and the results were averaged between the clients.

From figure 4 it is possible to note that in the base version each request has three times the latency of the advanced version, as is expected. Increasing the number of clients also increases the latency slightly, which is happening due to an increased load in the machine. As explained in section 5.1, the increased latency with a higher number of clients is partially related to the shared resources.

5.4. Throughput

Throughput was measured on the same tests as the latency. During each test, a total of 10k write requests to random objects were executed, distributed evenly across the clients. For example, with 1 client, the test had it execute 10k by itself. For 10 clients, each client executed 1k requests. The throughput of all clients was added for a final throughput of the system.

Due to the delay between the first and last client, at a higher number of clients the throughput starts to be

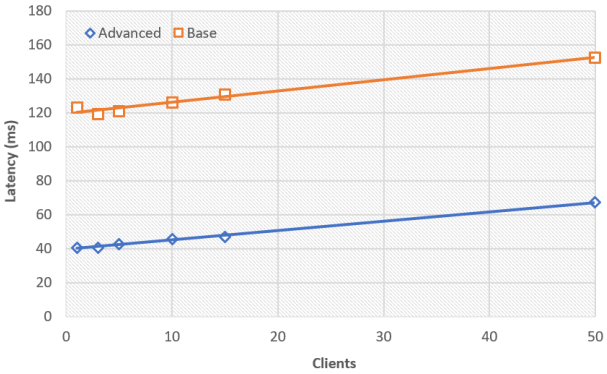

Figure 4. Latency

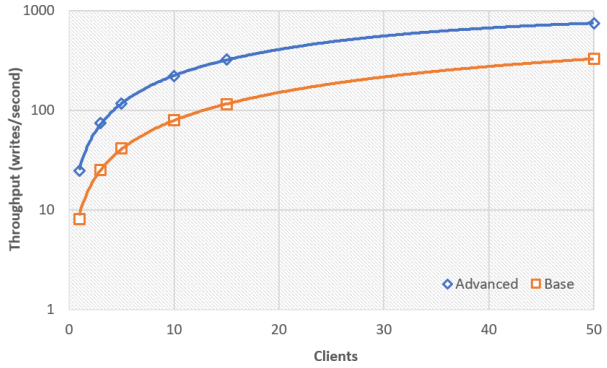

Figure 5. Throughput

negatively affected. In this test scenario, the advanced version could be plateauing before it should, and that it was not possible to test with a higher number of clients to find out the plateau of the base version. In any case, figure 5 allows us to see how much of an impact the blocking has. It uses a logarithmic scale, showing how the order of magnitude in the increased throughput stays the same across all numbers of clients. The advanced version has around three times the throughput of the base version.

6. Conclusion

The two protocols allowed us to understand the impact that the consistency guarantees have on performance. The base version uses distributed locking which results in slower writes but offers linearizability, while the advanced version has better performance but only offers eventually consistent data. Each version has its own advantages and choosing one will depend entirely on the use case.