DIDA-TKVS - A Distributed Transactional Key-Value Store

André Matos - 92420
andre.matos@tecnico.ulisboa.pt

Diogo Ribeiro - 102484
diogo.l.ribeiro@tecnico.ulisboa.pt

Luís Calado - 103883
luis.maria.calado@tecnico.ulisboa.pt

Abstract

Distributed systems require robust mechanisms to handle failures and maintain consistency across multiple servers. This paper describes DIDA-TKV, a distributed transactional key-value store developed to meet these challenges. DIDA-TKV uses the Paxos consensus algorithm to manage client request ordering and ensure consistency. Key features include Multi-Paxos, which enhances throughput by enabling concurrent processing of multiple requests, and a reconfiguration mechanism that dynamically adjusts server roles, improving the system's flexibility and adaptability to changing conditions or potential failures.

1 Introduction

In this project, we implemented a distributed transactional key-value store, called DIDA-TKV, designed to ensure linearizability and tolerate faults in a minority of replicas. The system consists of multiple servers that collaborate to maintain consistency.

We assume that a server may crash or "freeze" (while in freeze mode, it will neither receive client requests nor participate in the consensus process to learn the agreed order from other servers). It also handles scenarios where servers may process requests in different orders due to network delays, but does not tolerate Byzantine failures.

To achieve this, we used the Paxos consensus algorithm to order requests received from clients across the system. This approach guarantees that all servers agree on the same order of requests, ensuring consistent state updates. The project was developed in Java with asynchronous programming. The communication between processes was achieved using gRPC and the communication protocol was established using Protocol Buffers.

2 Paxos

Paxos, as described by Leslie Lamport [1], is a consensus algorithm designed to solve consensus in asynchronous systems. It guarantees safety, ensuring that no two processes ever decide on different values. However, Paxos does not guarantee liveness, meaning that it may fail to reach consensus in some scenarios, especially in the presence of continuous failures or network partitions.

The Paxos algorithm consists of 2 phases, followed by a learn phase of the value agreed.

2.1 Roles

Paxos defines three roles in the system, which must cooperate to achieve consensus. These roles are: Proposer, Acceptor and Learner.

In our implementation, we have 5 servers, 3 of which act as proposers and simultaneously as acceptors, and all of the servers are learners.

2.2 Prepare and promise

The first phase of Paxos starts when some proposer receives a request from a client. The server sends a prepare message to all acceptors with its leader id / proposal number, then waits for a majority of responses. After that, if the majority accepts it as leader, the server becomes the leader and can proceed to the following phase.

The role of acceptors in this phase is to promise to the server that sent the prepare message, if the leader id is the same or bigger than the last they have seen. The promise message also lets the leader know if any previous leader has proposed a value for that instance, so that it can adopt the previous proposed value. So the server will propose that value again, ensuring continuity in the consensus process and preventing conflicting proposals.

There is the possibility that the server can be rejected if some acceptor already promised to a leader with higher id.

In this case, the server will increase its id and try again right after being rejected. Compared to timeout after rejection, this solution can reduce the overall time to reach consensus. If the conditions that led to the rejection have changed, the server may succeed without waiting for the timeout.

2.3 Accept and Accepted

After receiving a majority of promises from the acceptors, the proposer initiates the second phase of Paxos. In the second phase, the proposer sends an accept message to all the acceptors, with the value (requestId) it wants them to accept. The acceptors, upon receiving the accept message, check if the proposal number is the same as the one they promised to accept and, if so, they accept the value (sending an accepted message to the proposer) and sending a learn message to the learners.

The leader knows that the second phase is completed if the majority of acceptors accept the proposed value. In case of rejection, Paxos' first phase must be repeated before trying to propose the value again.

2.4 Learn After Consensus

The learners, upon receiving a majority of learn messages, know that consensus was reached, so they can execute the request on the key-value store in total order and reply to the client.

2.5 Design Choices

When the leader id in prepare/accept message is lower, instead of ignoring, the acceptor informs the server that it was rejected. One advantage of the approach is that the proposer doesn't have to wait for a timeout to infer that it was rejected, allowing it to retry phase one with a higher proposal number. However, this adds extra messages to the networks and could lead to higher network congestion and latency. Since rejections are not so frequent, then the impact on network performance may be minimal in most cases.

Phase one allows a proposer to efficiently increase its timestamp by receiving the highest timestamp that acceptor has promised to.

3 Reconfiguration

It's possible to change the servers that act as "proposers" and "acceptors" at a given time. For doing so, the consoleClient sends a commit message to a special key (the key 0). Upon receiving the commit message, the server can easily understand that this request is a reconfiguration request, given that only the consoleClient can write in the special key and that its client id is 0.

Following the different solutions that Lamport proposes [2], we opted out for implementing the **Stoppable Paxos**.

In this solution, it's possible to execute a special command, the **stop command**, that will stop the next requests from starting Paxos, until the reconfiguration is complete.

For the reconfiguration, the leader/proposer executes Paxos as usual, and after consensus is reached, will change the configuration and release the stop command. The other servers, if they accept the new configuration, will also proceed to change their own configuration, learning their new roles for the following instances. If the server that proposed the reconfiguration is rejected, it will also release the stop.

However, when a reconfiguration is in progress, processes cannot start executing instance **n+1** without knowing the accepted value of instance **n**. This dependency creates a sequential bottleneck, limiting the parallelism in the system. Without the ability to execute instances out of order, the system may experience delays. Even though this dependency decreases throughput and efficiency, it does not significantly affect the overall performance of the system in practice. This is because the sequential execution constraint primarily applies to configuration updates, which are typically infrequent. During normal operations, the system can continue processing without frequent interruptions for consensus on new configurations.

4 Multi-Paxos

For Multi-Paxos, a proposer can start a new instance of consensus while the others are still executing, allowing for multiple instances to be running at the same time, and consequently a more efficient system.

Another optimization of Multi-Paxos is that it allows prepare messages to be executed only once, executing this phase in a single message for multiple instances, [X, infinity]. Then, when the leader receives a new request, the proposer only needs to execute the second phase of Paxos, after the prepare was accepted for the first time.

4.1 Work done by previous leaders

Acceptors, when promising to the leader, also send all proposed values from the previous leader for each instance up to infinity. This allows the new leader to catch up on all instances that were proposed by the previous one.

In order to achieve this, all servers save information about each Paxos instance they participated in. Upon receiving the prepare message, and after checking the leader id, the server will get all the instances from the index in the prepare message and beyond, that it has information about. This way, the leader can continue the work done by previous leaders.

5 Implementation Details

In our implementation, each server has 2 main threads to process the requests:

- One of them is responsible for executing Paxos, to decide the order in which a given request will be executed. This thread is used by proposers to manage creation of Paxos instances.
- The other one, has the role of managing the execution of requests in the key-value store, and to provide a response to the client once the request has been processed. This ensures total order, given that the execution only takes place after the order was decided by Paxos.

5.1 Requests from a client

When a request is received from a client, the server saves its request id to be ordered using Paxos. To avoid letting the thread in a wait state until the consensus is reached and the order is decided, the server saves the remaining request data encapsulated as a list. Encapsulation allows more abstraction so we can use the same process for read and commit operations. The stream observer is also saved to further reply to client.

5.1.1 Requests from console client

Requests from a special and unique client (the consoleClient) can be sent to a specific server (for changing the leader and setting debug modes) not requiring Paxos for this type of requests. This is the only client that can change the configuration and this process needs to be replicated in all servers, thus needing to execute Paxos as for a normal request. There is only one difference: when reading to check the older configuration, it skips Paxos and reply directly to the client.

5.2 Management of requests to avoid duplication

To manage the requests executing Paxos, we use three lists: pending, ongoing, and ordered.

When a request is received by a server, it is added to a pending list to be ordered. Then, while proposing the value it is moved to another list saving the ongoing requests (request ids that are being proposed), avoiding proposing the same value in different instances.

After consensus we move the value to another list, which contains the requests that were already ordered using Paxos. When a request id is in this list, its order can't change again and the other 2 lists referred before will not contain this value anymore. The approach described ensures that the same value is not proposed twice.

Afterwards, the main thread responsible to execute the requests is notified and proceeds to execute the request and to send a response to the client.

5.3 Value change during instance

Before proposing a value, the leader reserves the next value that is pending to be ordered. This works like a lock of the value for that Paxos instance. However, this value could change given that there might be another server that also thinks is the leader, and might have proposed a different value for the same instance.

When this happens there is the need to make the value available again, so other instances can propose it. This way, the value that was proposed with the highest proposal number will be executed for this instance, while the other value will go back to the pending list, so that it may proposed in a later instance.

5.4 Receiving Paxos request in new configuration

Network delays are a possibility and could lead to a request that is supposed to be executed in the new configuration to arrive before the server knows that the configuration has changed. In this case, it will proceed with the configuration change, improving the performance and avoiding delays caused by learn messages.

6 Limitations

Our implementation does not support state-transfer. This fact leads to significant issues regarding consistency and latency. When a server crashes and loses request data, it can only use Paxos consensus to order subsequent requests, but it won't be able to correctly process the previous requests, leaving the key-value store unchanged. This breaks the consistency between servers. Another problem is that the server will not be able to reply to client's request. Which can increase the latency that the servers reply to client, for example if the affected server is the one closer to the client (consequently having lower latency).

Furthermore, the system only allows a basic transaction with a fixed format. As a result, whenever a client wants to write in a key, he also has to read from 2 keys. This raises some disadvantages, like increased latency and inefficient resource usage. For example, when the client only needs to write, he will also execute 2 reads, sending 2 more messages than what he needs. Those extra messages contribute to network congestion and client also needs to wait

for an answer before proceeding, increasing the overall time to process the transaction.

In a more flexible system it would be preferable to allow clients to easily define the number of reads/writes that they want to make in a transaction.

7 Conclusions

In order to further improve the DIDA-TKV project, we would like to explore strategies that allow servers that crash to ”catch up”, using state-transfer, given that is one of its limitations. We also would like to allow servers to to run in different machines, something that could easily achieved with a name server to dynamically obtain the IPs and ports of other servers. Clients would also use the name server to send their requests to available servers.

To conclude, the DIDA-TKV successfully implements a distributed transactional key-value store using Paxos consensus for consistency and fault tolerance. Multi-Paxos boosts throughput by allowing concurrent request processing, while reconfiguration enhances flexibility by adapting server roles. Although the system only supports a fixed number of servers this implementation can also be scaled to support multiple servers, that can dynamically be inserted or removed from the system.

References

[1] L. Lamport. Paxos made simple. November 2001.

[2] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. *ACM SIGACT News*, March 2010.