DADTKV - A Distributed Transactional Key-Value Store

André Torres - 99053
andre.torres@tecnico.ulisboa.pt

Gonçalo Nunes - 99074
goncaloinunes@tecnico.ulisboa.pt

Pedro Lobo - 99115
pedro.lobo@tecnico.ulisboa.pt

1. Introduction

In this project we implemented DADTKV, a distributed transactional key-value store. The developed system is composed by multiple Transaction Managers and Lease Managers that work together to ensure linearizability while tolerating faults in a minority of replicas.

To achieve this, our solution relies on the Paxos [2] consensus algorithm to order requests for every time slot.

We assume a fail-stop failure model, where nodes can only crash silently. Byzantine failures are not tolerated by the system.

This project was implement in C# using asynchronous programming. The communication between processes was achieved using gRPC and the communication protocol was established using Protocol Buffers [1].

2. Lease Managers

The lease managers are responsible for receiving lease requests from the transaction managers and totally ordering those requests. This total order is achieved by running Paxos. Leases represents a lock on a set of key-value pairs. These act as exclusive write locks and are acquired by the transaction managers to access the set of data items they represent.

2.1. Time Slots

Time slots are used by the lease managers as a simplified way to simulate network partitions (i.e., process A might suspect process B has crashed, ignoring its messages and not sending messages to it).

Paxos is run each time slot, processing the requests received during the previous time slot. The algorithm outputs an attribution of leases to the transaction managers.

2.2. Paxos

Paxos, described by Leslie Lamport [2], is a consensus algorithm for solving consensus in an asynchronous system. It ensures safety. However it doesn't guarantee liveness.

The lease managers are ordered by their identifiers, establishing an order of leaders/proposers. All the lease managers play the role of acceptors. The learners of the system are the lease managers apart from the proposer and the transaction managers.

2.2.1 Prepare and Promise

When a new time slot start, the proposer proposes a new value for the lease attributions, based on the lease requests received during the last time slot. It then sends a *Prepare* message to all acceptors. These nodes reply with a *Promise* message.

In case the timestamp of the request isn't larger than the acceptor's read timestamp, but it has already accepted a proposal in the past, the promise reply includes the previously accepted value. If the timestamp in the request is not larger than the acceptor's read timestamp, the acceptors sends a NACK, as a way for the proposer to know to give up. This constitutes an optimization, so the proposer does not send any more messages through the network.

2.2.2 Accept and Accepted

Upon gathering a majority of promises, the proposer sends accept messages to all acceptors. These will then reply with an accepted message to the proposer and every other learner, which include the transaction managers. The learners learn the chosen value upon receiving a majority of accepted messages.

2.3. Fault Tolerance

When starting a new time slot, in the case of a lease managers suspecting the current leader is crashed, it tries to become the new leader by proposing a new value. This might lead to two (or more) concurrent leaders. However, this is handle by the Paxos algorithm, which is know to guarantee safety even in this type of situations.

In the case of failure of one of the processes, the next time another node tries to contact it, it is locally marked as faulty by the node, which will no longer send messages to it.

2.4. Design Choices

2.4.1 Propapagatting Leases to the Transaction Managers

For transmitting the result of the consensus to the transaction managers, there are two main approaches:

1. Waiting for the proposer to receive a majority of accept messages and then send the chosen value to the transaction managers.
2. Including the transaction managers in the set of learners.

We opted for the second approach as a failure of the proposer, after having received a majority of accepted messages, effectively learning the value, and before transmitting this decision to the transaction managers would render a consensus instance unknown to the transaction managers, possibly skipping the lease requests made in the previous time slot.

The second solution seems more plausible as a failure in the proposer will not imply that the consensus outcome is not known to the transaction managers.

2.4.2 Negative Acknowledgments

Our algorithm doesn't rely on negative acknowledgments, as if a process doesn't manage to get a majority of promises after a defined timeout, it gives up. However, we opted to introduce the timeout as well as to introduce negative acknowledgments in prepare messages so as to reduce the number of messages exchanged throughout the network.

Notice that if the node becomes partitioned, the timeout will kick in and safety will still be guaranteed.

2.5 Possible Optimizations

Multi-Paxos could be implemented to reduce the message complexity of the standard Paxos algorithm. If the proposer doesn't change from one time slot to the next, there is no need to run the read phase of Paxos, reducing the message delay from 4 message to 2 per Paxos execution.

3. Transaction Managers

The transaction managers serve as intermediaries between client applications and the stored data objects. They receive requests from clients, which they execute as transactions. The approach used is state machine replication, where the state is replicated throughout the transactions managers, allowing the system to tolerate faults in a minority of nodes.

3.1. Time Slots

A new time slot starts for the transaction managers, after a determined period of time, just like in the lease managers.

3.2 Leases

3.2.1 Lease Requests

In order to execute an update, a transaction manager needs to acquire the corresponding leases for the set of accessed keys. Leases act as exclusive write locks, so the locks are also necessary for reading operations.

Upon receiving a request, the transaction managers checks if it holds the lease for the accessed data items. If it doesn't it requests it to the lease managers.

3.2.2 Lease Management

Upon receiving the leases for the new time slot, each transaction manager adds the newly decided leases to its local queue of leases. This can be achieved with a HashMap, where the keys are the DADTKV object keys and the corresponding values are queues of transaction managers. If a transaction manager is the first element of a queue, it holds the lease for that data object. It is important to mention that all transaction managers keep the same queue state, which is updated when the lease managers finish the round of Paxos and broadcast the results to the transaction managers.

After completing a client request, the owner of the lock checks if there are other transaction managers in the queue. If there are, then the owner of the lease removes itself from its local queue of keys and sends a lease release message containing its id and the lease it is releasing to all other transaction managers. Upon receiving this message, all other transaction managers will update their local lease queue to reflect the changes described in the request.

Keys	Transaction Managers Queue
"Key-1"	TM1, TM3
"Key-2"	TM3, TM2
"Key-n"	TM4

Table 1. Example of Transaction Managers State

In the example described in Table 1, TM1 holds the lock for "Key-1", which will then be transferred to TM3 once TM1 broadcasts a release message to all transaction managers.

3.3. Uniform Reliable Broadcast

In order to disseminate the updates to other replicas, a transaction manager that receives a client's write request will broadcast its update to all other transaction managers. It waits for a majority of replies before executing it and replying to the client.

3.4. Design Choices

We opted for uniform reliable broadcast in order to disseminate updates instead of another approach like two-phase-commit because transactions in this system cannot abort. As URB has a better message complexity than 2PC, we can achieve higher throughput by using URB.

We also opted to only disseminate write updates as read operations do not alter state and don't need to be propagated.

As updates are idempotent, we don't need to uniquely identity them. There's no problem if a message is duplicated and a transaction manager ends up executing the same update twice.

3.5 Potential Problems

One problem may arise if a transaction manager crashes while holding a lease. This will block the system if clients submit requests which depend on the key whose lease is being held by the crashed transaction manager.

To solve this problem, a transaction manager which needs the lease being held by a suspected transaction manager, may send a message to all other transaction managers suggesting that each one of them updates its local lease queue, so as to take the lease away from the suspected transaction manager. If the update is successfully acknowledged by a majority of transaction managers, the system can now proceed.

4. Clients

The clients are individual processes that send requests and manipulate data from the DADTKV servers. Each client receives a script with the operations they should demand and this script is run in a loop.

Each client has a unique id that is hashed and used to decide which transaction manager it should contact. This way we can distribute the clients throughout the transaction managers nodes, achieving a greater parallelization than otherwise. If a client tries to contact a transaction manager that is down it skips it and retries with the next transaction manager.

Client applications interact with transaction managers using a set of two methods. The "TxSubmit" method, to request read and write operations to the DADTKV system, and the "Status" method, to request the output of each node's current state.

5. Launcher

This project can be executed using an automated launcher, which, when provided with a configuration file as a command line argument, initiates all the necessary processes with the desired simulation parameters.

Optionally, the *-o* flag can be used to overwrite the configuration file's wall time. The wall time indicates when all processes should start running the simulation. When this flag is used, the wall time is overwritten to the next 30 seconds, so that all processes have time to initialize.

References

[1] Protocol buffers overview. https://protobuf.dev/overview/. Accessed: 2023-10-28.

[2] L. L. Author. Paxos made simple. (1), November 2001.