DADTKV - Distributed Transactional Key-Value Store Group 11

Beatriz Militão
Instituto Superior Técnico
beatriz.militao@tecnico.ulisboa.pt
99183

Carlos Vaz
Instituto Superior Técnico
carlosvaz@tecnico.ulisboa.pt
99188

Pedro Neves
Instituto Superior Técnico
pedro.santos.neves@tecnico.ulisboa.pt
99305

Abstract

This report describes DADTKV, a distributed transactional key-value store whose objects can be accessed concurrently by multiple clients. The system's fault-tolerance is achieved by the use of the Paxos consensus algorithm.

The system is implemented in C# and uses the gRPC framework for communication between the involved processes.

1. Introduction

DADTKV is a multi-tiered system, composed of Clients, Transaction Managers, and Lease Managers.

Transaction Managers are in charge of storing and modifying data objects, as requested by the Clients.

Lease Managers are responsible for agreeing upon which Transactions Managers have leases and in what order. These leases are required for manipulating data objects.

This report looks to clarify our design decisions regarding the system implementation.

2. Structure

The system is implemented in C# and uses the gRPC framework for communication between the involved processes. It consists of a solution with 6 projects:

- `Client`: contains the code for the Client process.
- `ConfigParser`: contains the code for the configuration file parser. This project is also responsible for launching the processes as described in the configuration file.
- `DADTKV Client Lib`: contains the code for the Client library.
- `DADTKV_LM`: contains the code for the Lease Manager process.
- `DADTKV_TM`: contains the code for the Transaction Manager process.
- `Parse_Lib`: contains the auxiliary functions for parsing text input.

3. Lease Managers

A Lease Manager runs 2 server gRPC services: one for communication with Transaction Managers (used to receive lease requests), and one for communication with other Lease Managers (used for the Paxos algorithm).

3.1. Paxos

We started by implementing a sequential Paxos where all Lease Managers are Learners, Proposers and Acceptors and where the LM that was created first is the leader.

Every LM keeps track of its Paxos value, a list of the requests sent by the Transaction Managers, internal variables like `write_ts`, `read_ts`, `round_id`, and epoch (the last epoch started in case you're the leader or the epoch of the last message received, otherwise). The leader keeps track of its proposed value, `my_value`, `other_ts` (the highest `write_ts` received from the other LMs), `other_value` (the value associated with the highest `write_ts`).

3.1.1 Paxos Instance

When a new Epoch/Paxos Instance starts, the leader sees if it has received any requests from the TMs. If it has received requests, it creates its value to propose by ordering the requests by their TM names and sends the Prepare Message to the other LMs. If not, it waits until it receives at least one request.

When the leader gets a majority of promises, it sends a value for every LM to accept. This value is its own value if its `write_ts` is higher or equal to the `other_ts`, if not, it sends the `other_value`.

A Paxos instance only ends when the leader gets a majority of LMs to accept its proposed value. When this happens, all the LMs that have accepted the value proposed by that instance, send the result to all the alive/correct TMs.

3.1.2 From Paxos to Multi-Paxos

Switching from a sequential Paxos to a multi-threaded Paxos was relatively simple. We only had to change all the values that Paxos stores to Dictionaries (where the epoch is the key), and added the epoch in all the messages exchanged by the Paxos algorithm.

3.1.3 Paxos Leader

The leader executes a predefined number of Paxos instances, as defined in the configuration file. If the leader crashes or finishes that number of instances, the next LM currently alive will be the new leader and will also execute the same number of Paxos instances.

3.1.4 Leader Change

The leader change is done by the LM that is supposed to be the next leader. It periodically sends a message to the current leader to see if it has crashed. If it has, then the LM that sent the message becomes the new leader. If not, nothing happens because the current leader is still alive.

All the learners also keep the ID of the possible leader. The possible leader is the ID of the LM that has sent the prepare with the largest roundId.

3.2. Crashes

An LM crashes as dictated by the configuration file. We use an internal bitmap to keep track of which LMs and TMs are still alive. LMs can also know when other LMs and TMs crash, by sending a message to them and checking if they receive an exception. We only communicate with LMs and TMs that are alive (as dictated by the bitmap). To implement this, we also had to keep all the names of the LMs and TMs and send them in all the gRPC messages.

3.3. Suspicions

The LMs receive all of their suspicions from the configuration file. Before starting a new epoch, they verify if they suspect anyone in that new epoch. If they do, they send a message to the suspected server. If the server responds with an ACK, we ignore the suspicion. If the server responds with an exception, we send a message to all the other servers to treat the suspected server as crashed (to ensure a consistent view of crashed servers).

4. Transaction Managers

A Transaction Manager runs 3 server gRPC services: one for communication with Clients (used to serve the Clients' requests, for transactions or status), one for communication with Lease Managers (used to request leases), and one for communication with other Transaction Managers (used for broadcasts of writes or cleaning leases).

4.1. Receiving Transaction Requests

When the Client starts a transaction, the Transaction Manager thread receiving the request will try to find a lease that can be used. The TM will then either find existing leases that can be used, or request new leases from the Lease Manager.

When looking for existing leases, the TM will first check if they do not conflict with newer leases and if they do not intersect with newer requests (between the last transaction using the lease and this transaction).

When no existing leases can be used, the TM will request a new lease from the LM, only if it doesn't intersect with newer requests (between the last transaction using the lease and this transaction). The request is sent to all alive/correct LMs and, if the TM doesn't receive ACKs from all of the alive LMs, the request won't be added to the request buffer and will immediately return with an error to the Client, asking it to try again later.

After a request has a lease assigned to it, regardless of whether it has arrived or not, it's put in the request buffer (and if it's full, we wait until it's not). It then waits for the response to arrive in its results buffer.

4.2. Main Thread

The TM's main thread is in an infinite cycle, where it tries to execute all the transactions it can (sequentially) followed by trying to release all existing residual leases. This cycle randomly waits between iterations, to avoid using up resources by busy waiting. It also checks for crashed and suspected servers.

We consider a residual lease to be a lease that is ours, not assigned to any transaction and in conflict with some lease of some TM.

4.2.1 Executing Transactions

When executing a transaction, the TM checks to see if it has some request pending (in the request buffer) and, if not, it returns.

If it has at least some request pending (in the request buffer), the request is analyzed to see if its lease has arrived and is a complete lease (we consider a lease complete if it's the first lease of all of its keys) and, if not, it returns.

If it has a complete lease, the TM tries to execute the transaction and propagate the writes with a two-phase commit to all of the other alive/correct TMs. If the other TMs respond with ACK, we confirm the transaction to all of them and finish the execution of the transaction on our own TM. If the other TMs respond with NACK, we put the request in the results buffer with the error code.

When we finish the transaction, the results are put in a list for the thread that will respond to the Client to pick up. In case of an error (if it could not propagate, for instance), we return an error code and an empty list of results to the Client.

4.2.2 Residual Leases Deletion

When releasing residual leases, the TM goes through every lease it owns and sees if it's residual. If it is, it adds it to a list to then propagate the whole list to all alive/correct TMs, also with a two-phase commit, for them to approve the deletion of each of the leases.

After that, it sends the confirmation saying what leases we can really remove (the ones that received all ACKs) and removes those leases too.

While waiting for the replies for either of the propagations, we do randomly timed waits to prevent deadlocks in the system. If for some reason a deadlock occurs and if a transaction can not execute X times (this number is proportional to the Paxos time between rounds and is reset every time we receive external communcations, either from TMs or LMs), we drop that lease, remove all affected transactions, re-do the verification of which leases they will use and re-add them to the end of the transactions buffer, to be executed in the future (if a new lease is required but we couldn't contact the LMs, we place the transaction in the result buffer with an appropriate error code).

4.2.3 Crashed Servers

As defined in the configuration file, if it's our time to crash, we just end the program. We will have no problems with the consistency of the writes of our previous transactions and the other servers will ignore us even if they are mid propagation (if they still have a majority of both LM and TM alive nodes).

If a server has crashed, we set its entry to false on our internal bitmap that we use to keep track of which servers are alive (even if we didn't do this, we would notice the exception while trying to contact him and, if that happens, we also set its entry to false on the bitmap). After a server crash, the other servers will remove the leases of the crashed server.

If, at some point, we have less than the majority of servers (of TMs or LMs) alive, the remaining servers will crash automatically, so as to prevent possible erroneous propagations.

4.2.4 Suspicions

If a server suspects another server, it sends a ping to that server.

If the server responds with an ACK, we ignore the suspicion.

If the server responds with an exception, we send a message to all the other servers to treat the suspected server as crashed (to ensure a consistent view of crashed servers). If the suspected server did not really crash, it will still be treated as crashed by the other servers (including the LMs) and they will refuse to work with it, by sending exceptions when it tries to communicate. Internally, this will make the suspected server treat all other servers as crashed and will then crash itself, as it no longer perceives it has a majority of alive servers to communicate with.

When receiving an exception from some server, we set its entry to false on our internal crashed servers bitmap. We also answer with exceptions to any further requests from that server.

4.3. Incoming Broadcast of Writes

When we receive a broadcast of writes, we first check if we have the complete lease for these writes so that we can execute them. If we do, we send an ACK to the server that broadcasted the writes. When we receive the confirmation, we write the transaction.

If the writes we are trying to test for use a lease from an epoch we don't have yet, we wait until we receive the needed epoch.

If the broadcast comes from a crashed server (either truly crashed or a server that everyone agrees as being crashed), we ignore it.

4.4. Incoming Broadcast of Releasing Leases

When we receive a broadcast for residual leases removal, we first check if they all are complete leases. If they are, we send an ACK to the server that broadcasted the removal and the leases go to an internal removal batch while waiting for confirmation. When we receive the confirmation, we remove the leases and remove them from the internal waiting list.

If the leases we are trying to remove are from an epoch we don't have yet, we wait until we receive the needed epoch.

If the broadcast comes from a crashed server (either truly crashed or a server that everyone agrees as being crashed), we ignore it.

4.5. Incoming Broadcast of New Lease Epoch

We start by adding it to a waiting list. We then wait until we have a majority of LMs giving us the same lease epoch batch. When we do, we send it to the store to be added as a lease batch (removing it from the waitlist).

If the lease batch received by the store is not the epoch we were waiting for, we add it to another waiting list until the epoch we want arrives. When it does, we try to add all the other batches that we can (removing the added ones from the waiting list).

If the broadcast comes from a crashed server (either truly crashed or a server that everyone agrees as being crashed), we ignore it.

5. Conclusion

Implementing this system proved to be a challenging but valuable experience. Thanks to the Paxos algorithm, we were able to implement a fault-tolerant system that's able to main consistency even in the presence of server crashes.

References

[1] Wikipedia. Paxos (computer science) — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Paxos\%20(computer\%20science)&oldid=1170236310`, 2023. [Online; accessed 28-October-2023].