DADTKV

Lucas Pinto
Instituto Superior Técnico,
Universidade de Lisboa
lucas.f.pinto@tecnico.ulisboa.pt

Rafael Santos
Instituto Superior Técnico,
Universidade de Lisboa
rafael.santos.alves@tecnico.ulisboa.pt

Abstract

*This project focuses on developing **DADTKV**, a distributed transactional key-value store. **DADTKV** enables concurrent data access through a multi-tier architecture. Clients submit transactions using a specialized library, and transactions are coordinated through leases and consensus algorithms. This project leverages C# and gRPC to create a robust and fault-tolerant distributed system for efficient data management.*

1. Introduction

This project introduces a three tier architecture, encompassing client applications, transaction managers, and lease manager servers. **DADTKV**'s primary data unit is the *DadInt* key-value pair, with a focus on strict serializability to ensure data consistency.

2. Manager

The Manager process is the main entry point for the system. It is responsible for reading the system configuration and starting the other processes.

Lease Managers are the first processes to be created and then the Transaction Managers. This, however, does not guarantee that the Lease Managers are ready before the Transaction Managers. Because of that, each process notifies the Manager, through a *gRPC* call, that it is ready to work.

After all Lease and Transaction Managers are ready, the Manager demands each one to start their operations through a serial broadcast –messages are blocking. After that, the Manager will start the Client processes.

Now, the Manager will only do two activities: receive *StatusHook* updates from each server process. This was not much explored, but allows for the System Administrator to have a better visualization of the system; and receive Status requests from Clients, which will then relay to each server process, so that the servers write their status on the terminals.

For the Status relaying, the Manager does a concurrent broadcast –the messages do not block.

3. Clients

Client processes are very simple, in the state that they cycle through the list of operations that they have to execute, over and over again.

They will just send requests to a Transaction Managers, predetermined by the Manager, and wait for the response.

In case the given Transaction Manager is suspected of being unavailable, the Client would try another Transaction Manager it thinks is correct. This would work by detecting if the *gRPC* call fails because the server is unable to accept it. However, due to time constraints, such feature was not implemented.

4. Transaction Managers

Transaction Managers are the most complex processes in the system. They are responsible for receiving transaction execution requests and make sure they are executed concurrently and correctly.

4.1. Transaction Requests Reception

Transaction Execution requests are received through a *gRPC* call from the clients. The request is put in a buffer to be executed sequentially by a single worker thread.

In order to maintain the client connection open, the receiving thread is blocked and a monitor will wait for the reply lock to be pulsed. This reply lock may be pulsed in two situations: when the transaction read state is populated, and when the transaction replication is successful. Regarding failure detection, like mentioned above, is not implemented. The consequences of this is that the client will be blocked until the transaction is successful. A timeout could have been implemented to release the client from the reply lock block.

4.2. Transaction Execution

The transaction starts its execution phase when the worker thread is available to execute it. It takes the first from the buffer queue (FIFO) and starts to operate on it.

If there are leases the Transaction Manager does not own, it will request to the Lease Managers, through a concurrent broadcast, and wait for the lease update to be received. Just like the client reply lock block, this is achieved by a monitor that waits for the lease update lock to be pulsed, and rechecks each time if all the required leases are now owned.

As also stated before, the reads are populated to the reply lock object (and it is pulsed to notify the waiting thread). The reads are made locally (because they are read-only), as there is no need to form a quorum to satisfy strict serializability.

4.3. Transaction Replication

However, for the writes, the Transaction Manager must form a write quorum to successfully replicate the transaction. This is achieved by running Uniform Reliable Broadcast [1] (URB): the coordinator sends a concurrent broadcast to all other processes, which will add the (message, sender) tuple to their pending list and re-broadcast the message. On every reception of a (message, sender), the receiver adds the tuple to its acks list. If the tuple was already pending, it will check if it can deliver by guaranteeing it was not delivered before and it has a majority.

Commiting (on URB deliver) a transaction will apply the value changes to the local state of the Transaction Manager. The values are only updated if they are more recent than the current value. This is done by passing the epoch of the transaction and the write version of the value (it is reset on each epoch change). In conclusion, last write wins.

4.4. Lease Updates

All Transaction Managers receive the same lease updates messages, which can be seen as a key-value pair between key (lease) and the list of processes that want to use it in the order established by the Lease Managers.

Because Consensus may take more time for a later instance than on an earlier instance, updates may be received out of order. Because of this, lease updates are also added to a buffer to be processed sequentially. Because Paxos guarantees termination, we will not block forever, but may still block for an arbitrary amount of time.

Each time a Transaction Manager applies lease updates, it will pulse the Leasing object responsible for holding all the leases, so that the waiting thread is notified and can recheck if it owns all the leases it needs.

4.5. Lease Freeing

Leases, by requirement, are to be kept indefinitely, and only released when the process notices it owns a lease that is now conflicting and must release it after a single transaction is executed.

For this, it was decided that the best course of action was implicitly freeing the leases when transactions are commited, instead of explicitly freeing them when they are no longer needed.

The freeing only happens when commiting a transaction, and each process will check if the transaction was commited with any conflicting lease. However, this is not enough, because the process may not have all the updates from the Lease Managers –TM1 may receive the updates and the transaction replication may arrive earlier than the updates to TM2, which could make TM2 decide there is no conflict when there actually is.

To solve this, for transaction replication to be accepted by a process, it must follow the same rules for executing a transaction, which is verifying the sender Transaction Manager owns all the leases it needs. By following the same blocking process, this guarantees that the lease freeing on transaction commit will see all the necessary updates.

What if a Transaction Manager owns a lease from epoch 1 and only on epoch 24 is it conflictuous? In this case, it will generate a "dummy"/no-op transaction to be replicated to all other processes, so that each one can free the lease. In this case, the lease freeing is "explicit", but maskered beneath a read-only transaction.

Freeing leases when a process crashes is not implemented. This, however, could be achieved by

5. Lease Managers

When the Lease Manager is ordered to start by the Manager, it will create the timer responsible for the creation of Consensus instances. This timer runs every *slotDuration* milliseconds.

5.1. Lease Requests Reception

Just like all the other request receptions, lease requests are stored in a *buffer queue*, so that no requests are between Consensus instances. Whenever a new instance is created, it will lock and fetch all requests from the *buffer*, clear, and unlock it for more requests to be added.

5.2. Consensus

Paxos [2] is extended to allow the Proposer to provide a *SHA256 hash* of the lease requests (value) it wishes to propose on the Prepare message. The Promise message comes as a response to the *gRPC* Prepare call, and will include the respondents' lease requests if the hashes are different –this allows for the Proposer to make a more complete proposal.

Even if there are no requests on the *buffer*, the *Paxos* instance will still be created, because it is possible for a process to have received no requests, but other processes have.

Then, Accept messages are sent by the Proposer to all Acceptors, and if the Acceptors accept the message, will concurrent broadcast an Accepted message (the Lease Update message) to the Learners (Transaction Managers). The reply to the Accept message call serves as the Accepted message.

Althought fault tolerance was not implemented, it is technically possible for other processes to takeover the Proposer role as each process checks if its a Proposer depending on the proposal number. Because the suspect flags are not implemented and updated, the latter is not seen happening.

The implementation does not follow *MultiPaxos* specification, specially the predetermined leader, in favor of a more correct and complete proposal. Still, the current implementation allows for multiple instances to run in parallel, identified by their slot number.

References

[1] B. A. JOHNSSON. Broadcast algorithms. Lecture slides. Distributed Algorithms, Lund University.

[2] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001.