Design and Implementation of Distributed Applications Project Report

Carlos Costa
105768
Instituto Superior Técnico
carlos.d.costa@tecnico.ulisboa.pt

Francisco Fiães
105776
Instituto Superior Técnico
franciscofiaes@tecnico.ulisboa.pt

Duarte Costa
105787
Instituto Superior Técnico
duarte.ferreira.costa@tecnico.ulisboa.pt

Abstract

The goal of this project was to design and implement DADTKV, a distributed transactional key-value store to manage data objects (each data object being a <key, value> pair). In this approach, key-value stores are located in various Transaction Managers (TM) that operate over the data objects according to the result of the decision of the Leases Managers (LM), whose decisions relating to the order of the leases over the keys will be done through Paxos Consensus algorithm [1].

1. Introduction

Our solution allows users to submit transactions (read or write) to a distributed key-value store. Strong Consistency must be achieved since there are replicas that store the key-values. This was achieved by using a consensus algorithm [1] that determines the order of operations, and by using a replication mechanism between replicas. The logic implemented was to use leases that grant access to keys. Those leases are determined by the Lease Managers using PAXOS algorithm [1] in a deterministic way. Those leases are requested and consumed by the Transaction Managers in order to responds to client submitted transactions.

2. Architecture

In this solution, we implemented a Control program that spawns all processes according to the configuration settings present in the file ”configuration.txt”. This Control will spawn a given number of Clients, TMs and LMs. The Clients will initialize and create a *GRPC* channel to communicate with a TM, using a round-robin mechanism. After that, they are ready to submit transactions to their TM. They also have a channel with every TM and LM for the purpose of requesting the status of the machine and data objects.

The TMs will create channels with all LMs, to send the Lease Requests containing the transactions they received from clients, and with all the TMs, so that write operations can be replicated via broadcast, .

The LMs will also have a channel with all TMs, to send the result of the Paxos Consensus, so they can start the read and write operations over the data objects, and with all LMs, to send the messages of the Paxos Consensus.

2.1. Control program

The Control program will read the Configuration File (”configuration.txt”) and spawn all the processes that are described in this file. From the configuration file, the Control will read the number of time slots, the time slot duration and ignores the timestamp in which the program is supposed to start. Instead the programs' start time is hard coded as 15 seconds from the start time of the Control process. Then it will read every process in the file (ignoring the ones that start with '#'), reading the type of process (client, TM or LM), the name, and the server address. In the end it reads the states and suspicions of every process (TM, LM) in each time slot. After it finishes reading the configuration file it spawns all the processes, with the given configurations, and a global start time.

2.2. Data

We created a library *DADTKV* with an object of the type *DadInt*, which is a pair with a *string* key and *integer* value. This library has a getter, setter and a toString method that aids in the representation of these data objects to the client. These DadInt are the objects that are stored in the key-value stores of every TM.

Clients send transactions to TMs that are composed of read and write operations over these objects in the DADTKV servers (TMs/LMs).

3. Client Behavior

Clients are responsible for creating Transactions, which are read from their client script, with read and write operations on certain keys with given values, and sending them to TMs to be executed in their key-value store according to the order decided in the Paxos Consensus algorithm that the LMs perform. They may also send a Status Request to all processes (TMs, LMs) to know the state of the machine and their key-value store.

3.1. Submit Transaction

When the Client is initialized, it creates a *GRPC* channel, to use the *TransactionService* service, to a TM using a round-robin mechanism, then it reads of their text file the read and write operations (i.e. $T(\text{"}a-key-name\text{"},\text{"}another-key-name\text{"})(<\text{"}name1\text{"},10>,<\text{"}name2\text{"},20>)$), status requests (S) and possibly waits for x seconds (i.e. W 10000).

Then in the main loop, the client will wait the remaining seconds until the start time is reached. When that start time is reached, it will make a transaction out of the read and write operations of a line from the text file and send a *TransactionSubmitRequest* with these read and write operations to their assigned TM.

When the *TransactionSubmitRequest* is sent to the TM, client will now wait for the reply that contains the DadInt pairs (key, value) that correspond to the result of the read operation executed by the TMs.

3.2. Status

The status request is used for a client to have some information about the general status of each node in the system. It is directly broadcast to every node in the system (TMs and LMs) through direct channels created specifically for this purpose, and these nodes respond directly to the client with information such as the node's name, its current time slot, state in the current time slot (crashed or normal), list of suspected nodes in the current time slot, current state of key-value store, current epoch of Paxos, number and what are the current lease requests, and its type (LM or TM). This information depends, of course, on the type of the node (i.e. LM doesn't maintain the *KeyValueStore* state, etc...).

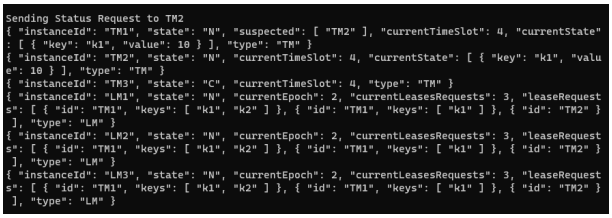

Figure 1. Example of Status console print

3.3. Wait

This is one of the possible operations that could be in the client script. The client will actively wait for x milliseconds (the value present in the file) on each iteration of the client's *MainLoop*, through the use of *Thread.Sleep* in the client. This could be useful to help test and simulate the behaviors and timings of the whole system.

3.4. Fault Tolerance

Clients are tolerant to faults in TMs, as when a *TransactionSubmitRequest* is sent to the TM, if there is no *TransactionSubmitReply* within the 15 seconds deadline, a *GRPC* Exception will be thrown and the client will reconnect with another random TM, except for the one it was just connected to. This mechanism means that the transactions that were sent to the TM and not yet executed are lost, although the client is aware of their non-execution as he receives no reply from the crashed TM and is notified of the change in connection to another TM.

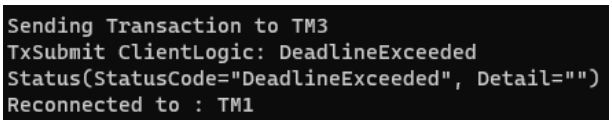

Figure 2. Client reconnecting to another Transaction Manager.

4. Transaction Manager

The Transaction Manager is responsible for handling the *TransactionSubmitRequest* sent by the clients, and execute read and write operations over their key-value store.
Some write operations in the *TransactionSubmitRequest*

can be executed immediately as long as this TM holds the lease for that key from the previous Epoch. The rest of the write operations and the read operations are broadcast to all LMs in a *LeaseBroadCastRequest*. When they execute the Paxos Consensus algorithm and decide the leases for that epoch, the TMs will receive the *RequestDecidedLeases* and are able to perform the write operations in their key-value store according to the order in the decided leases.

4.1. Initialization

When the Control program spawns a TM, it is initialized with some values that are passed through the Control, such as the TM's name, the starting time and the TM's server address. Then, other important variables are initialized, such as the crashed by time slot dictionary which indicates the time slots where this TM should behave as crashed, the suspected by time slot dictionary that dictates which TMs this TM will suspect in each time slot. TMs also get instances of important singletons, like *TimeStepInfo*, which keeps track of the time slot the system is currently in, is used to store the suspected by time slot and the crashed by time slot dictionaries, and the singleton *TransactionManager*, where the key-value store exists.

For clarification, i.e. when *TM1* suspects *TM2*, *TM1* doesn't send messages to *TM2* as he believes it is crashed. The Server is also initialized, and its ready to receive requests, with the following binded services: *TransactionService* (used to handle Client's transactions), *StatusService* (used to reply with its status to the client), *LeaseAssignService* (used to receive the decided leases by the Paxos Consensus of the LMs), *ReplicationService* (used for the replication following write operations in the key-value store) and the *RecoverService* (used by TMs who were crashed, become alive again and need to restore their key-value store state).

4.2. TransactionService

This Service is used to handle transactions sent by clients. Upon arrival, the read operations are executed and inserted in the reply request. Afterwards, the write operations are filtered in the sense that write operations over keys that this TM still holds the lease for since the last Epoch, can be executed immediately. The rest of the write operations need to be inserted into the *LeaseBroadCastRequest* that will be broadcast to all LMs.

The preparation and delivery of this *LeaseBroadCastRequest* is done in the *TransactionManagerClientLogic*, where the keys of this write operations are inserted in this request.

4.3. StatusService

This was the service created to answer *StatusRequests* sent by the client, and reply with the status of the TM node, informing the client of the following items: the TM's name, its current state (crashed or normal), the current timeslot, the list of suspected TMs, its current set of DadInts, and its type (TM).

4.4. LeaseAssignService

In this service, the TM will receive the decision of the Paxos Consensus at every Epoch from the LMs. The decided leases are sent in the *RequestDecidedLeases*. On arrival, the TM will check if it has received a request from a majority of LMs. If this majority was achieved, it updates the decided leases and is ready to begin the write operations that are ordered according to these decided leases. Before starting with the loop responsible for the operations, each TM creates a waiting list according to their position in each key in the decided leases.

This loop, responsible for the execution of the write operations in each TM is called *proceedOperations()*. Here, firstly the TM checks whether or not it is his turn to write in any key, by checking if its name is the first in that key of the waiting list. If it is, it writes the respective value in that key and replicates via broadcast to all TMs using the 4.5 *ReplicationService*.

If it cannot write in any key, it will check its waiting list. If its turn is approaching, meaning that there is only one TM before its turn to write in a certain key, it will create a timer to wait for this TM's replication. In case this timer expires (timeout) it means that something went wrong. To avoid waiting indefinitely when the timer activates, a *RecoverRequest* is broadcast to all TMs before writing to update the value of that key, in case the TM that supposedly crashed only had a problem mid replication. This only happens upon a quorum of replies to the recover request.

While the TM is "idle" it can receive a *ReplicateRequest*, meaning that another TM finished its writing operations on a key and replicated. When this happens, the TM that receives the request will change the values of its key-value store according to the values of the request, remove this TM's name from the waiting list of the keys that it wrote, and will return to *proceedOperations()* loop to check if it can write on any key this time. This implementation allows parallel writes in different keys, as there is a waiting list for every key decided in the Paxos Consensus.

4.5. ReplicationService

The service used to replicate sends *ReplicateRequest* requests via broadcast with the value of a given key, after a

write operation on the key-value store. The TM that broadcasts the request checks if a majority of replies occur, and sets this replication as successful accordingly. It is after this majority of replications (successful) that the TM is able to respond to the requesting client with their read operations.

4.6. RecoverService

When a TM is crashed for a duration of x time slots, it is losing out on replication requests, and therefore its key-value store is outdated. If it recovers to a normal state of functioning, it needs to recover the state of the key-value store. This service will be used to broadcast *CrashRecoverRequest* to all TMs so they reply to this TM with *UpdateStateRequest* requests, enabling it to update its key-value store state, once it achieves a quorum of *UpdateStateRequest* replies.

5. Lease Managers

The Leases Managers are responsible for handling the *LeaseRequests* sent from the TMs, and use the Paxos Consensus algorithm [1] to decide the next epoch's leases over keys. These leases also ensure the systems' strong consistency, as they create a global order of operations throughout all TMs, by ordering the write operations deterministically throughout all replicas. This is achieved, as if there are conflicts in TMs needing to write the same key, they are ordered by the same hierarchy of the creation of their processes in the configuration file.

5.1. Initialization

When the Control program spawns a LM, similarly to TMs, it is initialized with some values that are passed through the Control, such as the LMs name, the starting time and the LM's server address. Then, other important variables are initialized, such as the crashed by time slot dictionary which indicates the time slots where this LM should be crashed, and the suspected by time slot dictionary, which dictates which LMs this LM will suspect in each time slot. Once again, it gets instances of important singletons, like *TimeStepInfo*, which keeps track of the time slot the system is currently in and is used to store the suspected by time slot, as well as the crashed by time slot dictionaries.

The Server is also initialized, and it is ready to receive requests, with the following binded services: *LeaseRequestService* (used to broadcast the Decided Leases after the Paxos Consensus), *StatusService* (used to reply the its status to the client) and the *PaxosServices* (used to exchange messages between LMs to perform the Paxos Consensus).

5.2. StatusService

This was the service created to answer *StatusRequests* requests sent by the client, and reply with the status of the Lease Manager node, informing the client of the following items: the LM's name, its current state (crashed or normal), the current time slot, the current Paxos epoch, the list of suspected LMs, its current Lease Requests, and its type (LM).

5.3. Paxos Consensus

5.3.1 Leader Election

In every LM there is a round-robin mechanism to determine who is the Leader in this time slot. In every time slot the Leader will create a Proposal composed with a *MapField<string, List<string>>* where the keys of the map are the keys the TMs intend to execute write operations on and the values are the TMs that desire to write in that key. This is the value that is proposed in the Paxos Consensus amongst all LMs.

5.3.2 Prepare and Promise

To begin the Paxos Consensus, the Leader will firstly broadcast to all LMs a *PrepareRequest* with its id and his newly incremented read timestamp. The other LMs will compare the read timestamp in the *PrepareRequest* with their own read timestamp, only updating their read timestamp with the received read timestamp if it is higher. These LMs will then respond to the Leader with a *PromiseRequest* containing their id, their write timestamp and their accepted value.

5.3.3 Accept

As the Leader receives these *PromiseRequests*, it checks if a quorum was achieved. When it is achieved, the Leader can proceed to the next step which is checking the *PromiseRequests*. Here if the request's write timestamp is greater than the leader's read timestamp (which means that the leader was outdated) the Leader updates it's values with the request's. When this is verified, a *AcceptRequest* is created with the Leader's id, read timestamp and the value to be accepted by the other LMs (these values could have been updated by the *PromiseRequests* in the last step), then this request is broadcast to the other LMs.

5.3.4 Accepted

The other LMs, once they receive the *AcceptRequest* will update their read and write timestamp, and accepted value with the request's values. An *AcceptedRequest* containing the id, the write timestamp and the accepted value is created, and sent to the Leader. At this stage, this LM has decided and therefore broadcasts his decision to all TMs.

5.3.5 Leader Decide

Once the Leader achieves a quorum of *AcceptedRequests* received, it is ready to decide and broadcast his decision to all TMs. With this, the current epoch of the Paxos Consensus algorithm is concluded and all LMs are ready for the next Epoch.

5.4. LeaseRequestService

When an LM decides the leases on a new epoch, it broadcasts to all TMs a *RequestDecidedLeases* request so that they can proceed with their write operations according to the ordering of the decided leases of the new epoch. This Request contains the LM's name, the epoch, the decided leases (*map<string, TransactionManagers>*) and the transactions, that contain write operations and are meant to be executed.

5.5. Fault Tolerance

The LM's Paxos Consensus tolerates some faults from other LMs as long as a majority of LMs is alive. This happens because in each time slot of the Paxos, the Leader accounts for a quorum of requests from the other nodes.

6. Conclusions

With our solution we guarantee that all the client transactions that are sent to the TMs, are encapsulated in Lease Requests and broadcast to all LMs. These time slot's Leader will aggregate all Lease Requests and the keys they demand, create a proposal and initiate the Paxos Consensus algorithm resulting in a majority agreement between all the LMs about the order of writes for every key, which is broadcast to TMs. These TMs will execute their write operations on certain keys according to the ordering of the decided leases in Paxos.

We also guarantee Fault tolerance throughout the system, recover state mechanisms and replication services to guarantee consistency of the data objects in each key-value store in the TMs.

References

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