DADTKV: A Distributed Transactional Key-Value Store for Managing Data Objects

Alexei Calugari, Franscisco António, Rodrigo Liu
Design and Implementation of Distributed Applications
Instituto Superior Técnico

Abstract

DADTKV is a three-tier distributed transactional key-value store. The architecture comprises a client tier, that sends transaction requests to the second tier. These Transaction Manager servers offer transactional access to the stored data while replicating the client's requests to ensure consensus and long-term persistence of the information. Finally, the third tier, the Lease Manager servers, employs Paxos to determine which transaction manager a lease is assigned to, controlling simultaneous access to shared data by different Transaction Manager servers. All processes are started by the Process Manager and are run locally, meaning all sent messages are eventually delivered. To simulate server crashes, DADTKV runs on a fixed slot-based timer, with each slot determining the server's state.

1. Introduction

The main objective behind the creation of DADTKV is to develop a straightforward yet realistic distributed application. Making use of gRPC for communication between processes, a client initiates a request, it can be received by any transaction manager. Subsequently, the transaction manager will broadcast the request to the other servers within the same tier. This approach ensures that all transaction managers can apply the modifications made by the request, having all the same set of information. Transactions can only be initiated if the server holds the appropriate lease. If not, a lease request is broadcast to the lease managers. These requests will queue in the third-tier, which, at the start of each time slot, will organize all the leases and inform all transaction managers of the new set of leases. The Lease Managers maintain consistency among all replicas for the received requests in every round through the use of the Paxos protocol. They achieve consensus by electing a leader when there isn't one or when the current leader is suspected to have crashed. As a result, all lease managers maintain the same set of information. By utilizing a Process Manager, and time slots, we can simulate real-life scenarios, such as server crashes, message delays, and facilitate easier setup and testing in general. The manager instantiates all servers, ensuring that all processes start running simultaneously, progressing to the following slots concurrently.

2. Process Manager

The purpose of the Process Manager is to instantiate all processes.

It is also where the terminals are managed so that it is possible to observe the behavior of each element of the system.

2.1. Configuration File

The Configuration File can be accessed by all processes, it contains all the information for a simulation of the application. It determines the general aspects of the simulation: the time that all the processes should start running, how many server processes are instantiated in each tier, and their respective names, identifiers, and IP addresses; how many slots the simulation will have, and its duration. It also states the individual configuration for each server, for every slot, which includes: the identifier, the state of the server for that particular slot (normal or crashed), and the sequence of pairs identifying suspected processes, each pair including the IDs of the suspecting process and the suspected process (simulating, messages that take a long time to reach its destination).

3. Client

The client is a part of the application whose sole purpose is to repeatedly send messages to the Transactions Managers in order to make the whole system operate and it has three possible types of operations.

1. Submitting a new transaction to one of the Transaction Managers;

2. Requesting that the entire System show its Status;
3. Wait until being able to do a new action.

3.1. Client Configuration File

The Client Configuration File is akin to the Configuration File, as it simulates client actions, each line can be: a request to the Transaction Manager tier, this request being a set of objects that the client wants to read, and a set of objects to be written; a Wait command, in which the client waits for a duration stated; and finally the Status command, sending a request for all servers to print out their status, at the given time.

4. Transaction Managers

These sets of servers store a collection of data objects (DadInts) in memory, replicated across all Transaction Managers. Clients can read and write on DadInts, by making a request to any of the servers, in this tier, by utilizing a specific service ran on these servers. When one of the servers receives a request, it will check if it holds the lease for the DadInts requested it employs its own service to propagate the clients' request, among all servers from this tier.

4.1. Consensus

When Transaction Managers receive a client request, they first check if they hold the appropriate lease to process it. If they do, they initiate a 2-phase commit by broadcasting an update request to all transaction managers. This step is taken to verify the availability of the majority of servers. If a quorum is achieved during this update, the initiating server proceeds to issue a commit, including the set of objects to be written. This ensures that the majority of servers maintain the same set of information.

4.2. Forced lease releases

For the lease management, the leases are stored in order that was given by the Lease Manager, the leases are granted in order but can't be used if they hold the same DadInt. To achieve this all the DadInts that are granted by the lease, are used as indexes for the "waiting list" which is composed of the Transaction Manager id that requested the lease. For example, if TxM1 requested a lease for DadA and DadB, and after that TxM2 asked for a lease for DadB and DadC, there will be a "waiting list" for each one of the DadInts requested (in this example 3) in this case TxM1 was first, this id will be in first place in the list for the DadA and DadB, so if TxM1 receives a request for DadA and/or DadB, since it is at the front of the line it will be able to use the lease. On the other side TxM2 will be in second for the DadB and first for the DadC, so it won't be able to respond to DadB because TxM1 still holds the lease, TxM2 will have to wait for TxM1 to release its lease. So the Transaction Managers make sure that only 1 of them use the lease for a certain DadInt at a time.

The Transaction Managers releases its lease in 2 stages. First, when a TM is able to execute a client request, after it is done, the TM checks if there are any other Transaction Managers in line for the DadInts that were used to satisfy the client request, if there are any in line, then a release lease broadcast is executed, this broadcast removes the TM Id from the waiting list of all the DadInts that were used in that particular request, this way the next TM in line will be first and therefore be able to use the lease. However doing the release only in this stage may cause a lock of leases, in case that the first TM executes the request and then when checking for other TMs in line, if there are none the lease won't be released, only when the current TM will use the lease for a second time.

Second case, when the TM is given a request and is checking if it's able to execute the request, if it is not in a position to execute, then checks if it is first or second in line for all the DadInts that are necessary for the current client request, with the idea of sending a request to all Transaction Managers to release the leases that block the line. For this to happen first it sends a request to all of the TMs that are holding the DadInts(first in line), if all of them agree to it or don't respond within a time window(crashed) the waiting lists will be updated for all of the Transaction Managers and the TM will be able to use the lease to execute the request. Doing the release lease in this stage also ensures that when a TM has a lease and then crashes, it doesn't permanently lock the lease to their respective DadInts.

5. Lease Managers

Lease Managers are the set of servers responsible for receiving, ordering and assigning all lease requests. At the beginning of each Time Slot, they will collect all the lease requests received in the previous slot and try to reach a consensus.

5.1. Leases

Lease is the object used in this system to allow a Transaction Manager to access the necessary DadInts. This object is used for both read and write operations.

5.2. Consensus Process

When a new slot begins, the Lease Managers start a new process in order to achieve a new consensus. To achieve this, they work under Paxos logic, more specifically using Multi-Paxos, which makes the choice of leader a more important decision. Although it is more important, it is not critical since the basic Paxos algorithm is prepared to deal with races between leaders, it is only desirable in order to have an efficient system.

One of the reasons we chose to use Multi Paxos is that we can skip phase one of the algorithm, i.e. when a leader is decided by a transaction manager, it only needs to wait for its accept messages, which means that during periods when the system is stable, leases are delivered more efficiently.

5.3. Muti-Paxos Algorithm

Each Lease Manager implements a Paxos service, with a Proposer, a Learner and an Acceptor. In our Multi-Paxos implementation, there are three possible scenarios.

1. The Lease Manager knows who the current leader is and doesn't suspect it: At the start of a round, he will wait for an accept from the current leader or possibly a prepare from another proposer who is trying to become leader. It will do its job as normal, but it won't compete to become leader;

2. The Lease Manager knows who its leader is, but is suspicious: Using the Config File as a Failure Detector mechanism, when a proposer is suspicious of its current leader, it will check whether it is the server that should try to become leader and, if so, try to run for leader by increasing its proposal number and spreading its Prepare messages. In other words, since we want to maintain an orderly sequence of leaders, if there are servers between this and the current leader (who have not yet been leaders) it should wait for them to do so, until it suspects them too.

It is also important to notice that, in order to generate new proposal numbers, we decided that everyone should start by proposing their id, increasing it when necessary by always adding the number of existing servers.

This way we can guarantee that there will never be any repeated numbers, thus avoiding something that could be problematic for the algorithm.

After that, just like in Paxos, the Lease Manager who gets a majority of Promises can ask the others to Accept its value. If it also gets a majority of acceptances, then its value has been decided. The Way we've implemented it, the Lease Manager becomes the leader as soon as he gets a majority of acceptances and, from the point of view of the other members, it is recognized as the leader when an Acceptor replies with Accepted.

When everyone accepts the proposed value, the system reaches a consensus. All that remains is to inform the Learners. To do that, we decided that an Acceptor, as soon as it decides to respond with accepted to the proposer, sends the new value so that the Learner can learn it. To be sure of this value, the Learner will count these incoming messages and make sure that they arrive from a majority. If this is the case, they learn the value and share it with the Transaction Managers.

The latter don't need to count the messages again as they are sure that they are receiving a valid value for that consensus epoch.

6. Conclusion

In conclusion, with this project we implemented a working DADTKV system, however, some parts of the implementation are not fully implemented or working properly. Mainly, the logic stated in section 4.2, in the current version the lease is released after being used. If a transaction manager crashes, or its messages are delayed too long, other servers must reach consensus, and forcefully remove this lease from the server.