DADTKV - Group 4

Guilherme Luís Francisco Soares - 96392
Miguel Agostinho da Silva Rocha - 110916
Charalampos Spyridon Botsas - 108794

Instituto Superior Técnico (IST)
Lisbon, Portugal

Abstract

This paper describes the design and implementation of DADTKV, a distributed transactional key-value store to manage data objects (each data object being a <key, value> pair) called DadInt, that reside in server memory and can be accessed concurrently by transactional programs that execute in different machines. This access to data will be regulated by using a Lease system. This paper describes the design of the final solution and the multiple components necessary for the correct implementation of this system.

1. Introduction

The final solution is divided into the following subprojects:

- Management Console;
- Client;
- Transaction Manager;
- Lease Manager.

The Management Console is used as the starting point to run the system. From there, all processes (Client, Transaction Manager and Lease Manager) are launched.

2. Management Console

This process has the task of starting all other processes in the system. Like in the Client program, this will first iterate over the configuration file, and store everything in a local structure. Then, after knowing every process that is going to be launched, system parameters (ex: time slots, starts, etc), and fault tolerance, it starts launching the processes one by one.

Depending on the process, he needs to know several things. For example, the Lease Manager processes need to know the set of Lease Manager nodes and also the set of Transaction Manager nodes. One possibility is to just start the process and let him read the configuration file. Instead, we decided to pass that information into the arguments of each process. This solution has the advantage that the file will only be parsed once. Every process will receive only the needed information to run. Each process won't have the overhead of having to parse the complete file, just to get the specific information that is relevant to him.

The Management Console is also the one that decides what Transaction Manager, each Client, will contact. To do this, it selects a random Transaction Manager, between all of them, and assigns it to the Client.

3. Client

When the Client starts it parses its configuration file given by the Management Console and stores the read commands on a structure, in order to execute them in a continuous loop. The Client is able to do 3 commands: *Wait*, *Transaction*, *Status*. The last 2 are commands issued to the Transaction Manager assigned to the Client by the Management console. The transaction is composed of a list of reads and writes to a set of DadInts. When issuing a transaction the Client waits for its conclusion and prints the result.

4. Transaction Manager

The Transaction Manager is responsible for executing Client transactions and maintaining the results of the transactions in storage. There can be multiple TM nodes executing concurrently, so there is a need to synchronize transactions that modify the same DadInt object.

4.1 Leases

When a Client wants to execute a transaction with a set of writes, the Transaction Manager that receives it will try to obtain a Lease for the keys modified by the transaction (if he doesn't have them already – explained later). The Lease has the following format:

```
Lease {
    TmId: string,
    SequenceNumber: int,
    EpochNumber: int,
    Keys: string[]
}
```

With the combination of the first 2 fields, we can identify the Lease. Each Transaction Manager will use a different sequence number every time they request a Lease. The addition of the Epoch number is to prevent Lease Managers from reaching a consensus on the same Lease twice without them needing to keep a state of previous consensus.

4.2 Execution of Transactions

Every TM node keeps a queue of Leases for every Key in the DADTKV data storage. When the Transaction Manager *learns* the result of the Lease Manager consensus for the next expected epoch, (received a Quorum of messages with the same ordered list of leases from $n/2 + 1$ different nodes, where n is the number of LM nodes), he can append the new leases to the end of the queues. To emphasize, the Lease Manager consensus is added **in ascending order** to the queues, from the consensus with lower epoch number to higher.

If the TM has the necessary Lease to execute its transaction, he will propagate the transaction that he wants to execute to the other Transaction Managers. There are multiple problems that can happen in this communication, so to ensure that all TMs end up with the same state we used an abstraction of the *Uniform Reliable Broadcast*. The necessary guarantee that the *Uniform Reliable Broadcast* gives us that a simpler *Reliable Broadcast* doesn't is:

Uniform Agreement: For any message m, if a process delivers m, then every correct process delivers m

In our system, it means that a faulty Transaction Manager node will not report to a client that he executed a transaction without being sure that every correct node will *deliver/execute* the same transaction. To implement this property every Transaction Manager node has to wait for a Quorum of the same transaction in order to execute them. Now this Quorum will be $n/2 + 1$, where n is the number of TM nodes. The other guarantees implemented are:

Validity: If p1 and p2 are correct processes, then any broadcast by p1 is eventually delivered by p2

No creation: No message delivered unless broadcast

No duplication: No message is delivered more than once.

All of these properties were provided by our Perfect Link implementation.

When propagating a transaction a Transaction Manager verifies if there are any Leases conflicting with his Lease. If there are he appends a boolean indicating that the other TM nodes can release the Lease associated with his transaction after executing it. This means that a TM can keep its Lease and execute other client requests that involve the same DadInt keys if there are no conflicts. In order to do that the Transaction and Lease need to be identified further with a Transaction Counter:

```
Transaction {
    TmId: string,
    SequenceNumber: int,
    TransactionCounter: int,
    Writes: DadInt[]
}
```

Every time the Transaction Managers execute a transaction that doesn't have the boolean release set to true, they increment the Lease Transaction Counter on the Lease associated with that transaction.

4.3 Liveness

In order to guarantee *liveness* in the system we need to make sure that a Lease can be released if a Transaction Manager crashes, is delayed due to network problems or simply didn't release its lease due to not being aware of Lease conflicts. Enter the Release Lease process, which will happen if a TM node *suspects* other TM holding a Lease or if he detects that there are other pending conflicting Leases after executing a transaction that doesn't release the associated Lease. Similar to the Propagate Transaction process, we can leverage the *Uniform Agreement* property of the *Uniform Broadcast* to ensure that if a process releases a Lease then every correct process will eventually release the Lease too. When broadcasting the Release Lease message, the TM sends along with the Lease to be released the last write executed with that Lease. This ensures that when the Quorum of Release Lease messages are delivered to every correct TM, every one of them will have the most recent transaction executed with that Lease, so the eventual consistency of the system is ensured. Is worth noting that as soon a TM node receives a Release Lease request he stops accepting any more transactions associated with that Lease. When a

TM node gets its Lease released without being able to execute its transaction it returns a single DadInt with an abort key.

5. Lease Manager

The Lease Managers are responsible for executing a consensus algorithm in order to decide on a value. This value will be an ordered list of Leases, that will be delivered to the Transaction Managers.

5.1. Lease Requests

As mentioned before a Lease is identified by an Epoch Number to prevent being added to multiple instances of consensus. So when a Lease request is received, the Lease Manager checks if the epoch in the request is decided, and discards it if it is.

5.2. Paxos: Consensus Algorithm

To achieve consensus the algorithm chosen was Paxos. Every time a timeslot passes, a new instance of consensus is launched with an epoch number associated. This doesn't mean that the previous consensus instances are aborted. In fact, multiple instances of Paxos might run at the same time. This can happen if the system experiences network latency during some time periods, resulting in messages involved in a given epoch being delayed so the consensus doesn't terminate until the next time slot. The Paxos algorithm has 3 types of processes: *Proposers*, *Acceptors*, *Learners*. In this implementation, each LM node will have the 3 roles. Additionally, the Transaction Manager nodes will be *Learners* of the consensus too.

5.2.1 Prepare

When a node thinks he is the leader, he broadcasts a Prepare message to all other Lease Manager nodes. The Prepare message has this format:

```
Prepare {
    ID: int,
    EpochNumber: int
}
```

The ID is a unique identifier that will be equal to the node index number (assigned at the start by the Management Console) plus a multiple of the number of Lease Manager nodes to ensure that will always be unique ($ID = nodeIndex + n * nodesSize$). The epoch number is needed to prevent conflicts between concurrent epochs.

The *Acceptors* will receive the Prepare message and reply with a message identified with an ID too. When the *Proposer* receives a Quorum ($n/2+1$ successful messages from different *Acceptors*) he can progress to the Accept phase. The ID present in the reply will be equal to the highest ID seen by the *Acceptor*, meaning that if it is different than the Prepare message ID then the Prepare was not successful. Although it is still possible to achieve a Quorum of successful replies, we would probably be rejected in the Accept phase where the ID is relevant too. So we have the option to try immediately by sending another Prepare message with a higher ID, however, in our implementation we add a delay before retrying. This is done to mitigate the problem where 2 "leader" nodes would block each other on the Prepare and Accept phase, by retrying with higher IDs every time they are rejected preventing the consensus from progressing. This way we give δ time for the other node to progress before retrying.

A successful reply can have an Accepted Value attached if the *Acceptor* already accepted a value before, that will be used by the *Proposer* in the Accept phase. This is done in order to ensure the property described on Paxos Made Simple [1]:

P2: If a proposal with value v is chosen, then every higher-numbered proposal that is chosen has value v.

5.2.2 Accept

When a Quorum of Prepare replies is reached, if no value was adopted from the majority of *Acceptors* that replied, a new Paxos value is created, consisting of the Lease Requests made by the Transaction Managers for this epoch ordered by arrival. Then the *Proposer*, broadcasts Accept messages to the *Acceptors* with its Paxos value. Because in our implementation the *Proposers* are *Acceptors* too, the *Proposer* accepts his own value, and, therefore, he counts this as an Accept message.

When a *Proposer* receives an Accept message reply, indicating that the Accept message ID is outdated (meaning that the *Acceptor* promised to another leader with a higher ID), it stops the broadcasting and will retry the Prepare with the same retry mechanism described in the Prepare phase.

When a *Acceptor* receives an Accept message, it checks if he promised to a *Proposer* with a higher message ID (rejecting if so), and if not, broadcasts the Accepted value to all the *Learners* (LM and TM nodes). He also updates his highest Accepted Value if the new value has a higher ID, in order to be used in the Prepare phase when piggybacking the Accepted value.

Note that this doesn't mean a new value was decided, since, as *Learners* the TM and LM nodes will still have to wait for a Quorum of Accepted Values from $n/2 + 1$ different nodes. Important to mention that this Quorum is not dependent on the ID attached to the Accepted value, in Paxos the decision is made, not on the Accepted value ID, but rather on the Paxos value[1].

5.2.3 Lease Managers as Learners

Despite the Lease Managers don't need to know which value was decided, it is convenient for them to know if an instance has already achieved consensus in order to stop more unnecessary retries. Therefore, whenever a request is sent to a Lease Manager node, related to an epoch that is already decided, he notifies the node that made the request that the epoch was ended.

5.2.4 Transaction Managers as Learners

For the Transaction Managers to know the decided value, there are two options. The first is for them to get notified by an LM with the decided Paxos value. Yet, if the Lease Managers can be learners, so the Transaction Managers can be. This way, the round of Accepted messages is not wasted.

6. Simulation Of Suspected Nodes And Server Crashes

When a node is suspecting or is suspected by another one, it means that they are not able to communicate with one another. Therefore, to simulate this situation, our system throws exceptions on incoming messages from suspected nodes. A crashed process does the same thing, whenever he receives a message he responds with an exception. So in reality, the node will continue to run but will ignore all requests related to him.

7. Conclusions

With this project, we managed to put into practice all the topics lectured in the theory classes. The most difficult task of this project was to handle all the possible unpredictable events that may happen during network communication, and still be able to ensure the correctness of the system. Also, besides applying the basic Paxos, we also considered some optimizations, that don't affect correctness, meaning they are not essential to the system: Delay on the retry to ensure a better chance of progression; the discard of message from done epochs (due to Lease Managers being *Learners* too) reducing processing time.

References

[1] L. Lamport. Paxos made simple. pages 1–11, November 2001.