Tolerância a Faltas Distribuídas

Streamlet Final Implementation

Bruno Cotrim (54406)

[fc54406@alunos.fc.ul.pt](mailto:fc54406@alunos.fc.ul.pt)

Professor: Alysson Bessani

1. **Introduction**

This project consists on implementing Streamlet, a consensus algorithm that can be used to implement a Distributed Ledger, this algorithm, opposed to permissionless consensus using Proof-Of-Work such as Nakamoto’s Bitcoin, has permissioned consensus as a building block, it achieves this by using Proof-Of-Stake, where there is a set of known participants and each one has voting power proportional to their stake in the system. One of the key features of Streamlet is that when compared to classical algorithms such as PBFT or PAXOS it is extremely simple. In this implementation I will be focusing on the crash-fault implementation, however, very few modifications are needed to make it Byzantine faults.

Here is a resumed way of the requirements and how the protocol works:

* It is assumed that participants have synchronized clocks (needed for liveness), and the protocol runs in synchronized epochs (property of the system).
* In this protocol each epoch has a leader that is defined by a publicly known hash function, so it must be known for all participants in the system. Each sequence of blocks is “chained” by a hash function where each block contains the previous block hash.
* The best part of this protocol is that it is simply a propose-vote mechanism, for each epoch the following must take place:
  + The epoch´s leader proposes a new block extending from the longest notarized chain it has seen, if there are multiple, it should be arbitrarily untied.
  + Every player votes for the first proposed block they see from the epoch´s leader, only and only if it extends from one of the longest notarized chains the voter has seen.
  + When a block receives more than n/2 votes it becomes notarized. A chain is notarized if all blocks it contains are notarized.
* **Finalization**: Whenever a chain contains 3 epoch consecutive blocks it means it can be finalized up to the second of the three last blocks. Meaning that transactions inside it are confirmed.
* **Echoing**: Each message received by a node is broadcasted to all other nodes once.

Uma imagem com captura de ecrã, texto, Tipo de letra, diagrama

Descrição gerada automaticamente

This was just a small introduction to the project, from now one I will be focusing more on my implementation aspects.

1. **Implementation**

**Data Structures**

**Message**:

* content(object): Message or Block.
* type (enum):
  + Propose: Used for proposing, content is a block.
  + Vote: Used for Voting, content is a block that is being voted.
  + Echo: Used for echoing messages, content is the message being echoed.
  + Alive: Used for starting the algorithm and knowing which nodes are alive.
  + Transaction: Used for adding transactions to the system.
  + Reconnect: Used when a node crashes and reconnects to the system needing for synchronization.
* sender(int): the ID of the sender node.
* sequence(long): sequence number of the message used for FIFO-Broadcast.
* additionalInfo(Object): Used for storing additional info such as timestamps.
* isReconnected(boolean): Used for messages related to reconnections.

**Transaction**:

* sender(int): The sender of the transaction.
* receiver(int): The receiver of the transaction.
* id(int): Random nonce number ID.
* amount(int): Amount being transfered.

**Block**:

* previousHash(byte[]): 32 byte Hash of the previous block.
* epoch(int): epoch number.
* length(int): length of the block in the chain.
* transactions(List<Transaction>): List of the transactions that the block contains and is trying to validate.
* votes(Queue<Integer>): contains the votes of each block.

**ReconnectMessage**(used for recovery):

* node(int): Node that is requesting to reconnect.
* epochRandom(Random): RNG State of the Sync. Node epoch
* epochLeaders(Map<Integer, Integer>): Leaders previously calculated
* messageHistory(Map<Integer, Message>): FIFO-Map that stores the last message received for each node.
* epoch(AtomicInteger): Current epoch number.
* nextEpoch(Instant): Instant when the next epoch starts
* epochVotes(Map<String, List<Integer>>): unprocessed votes that are still missing a block.

**Code Aspects**

**Node Start:** When a node starts, it reads the system properties, such as addresses, epoch from a property file and reads ID and isRestart from command line arguments.

Then for each node that the system contains the given Node will create N+1 Virtual Threads (Virtual Threads work good in the presence of blocking IO such as sockets, this way improving performance), one for a socket that listen requests, and the other N are related a Node.java class that contains the network code responsible to pulling messages from a local node queue and FIFO-Broadcast it to all the other given nodes.

After this, each node starts Broadcasting Alive Messages to specify that they are up and running and ready to start the execution. To start the algorithm, I used the “useless” genesis block, the block 0 of the chain, and in the epoch 0 the leader sends the genesis block with the Instant when they should start epoching and executing the algorithm, so I used the genesis block almost as a signal to the Start the algorithm, it also could be done in any type of message but I liked this one since it starts the algorithm by “running” itself.

**FIFO-Broadcasting:** To make the implementation more memory efficient and take into account the order of the messages sent by the nodes, I implemented a form of FIFO-Broadcast, that runs on top of URB Broadcast, this was done by including the sequence number for messages of each node, and storing the last message received, and, if a message received at a certain time as sequence number equal to the expected(after the one received in last message), the message is processed, otherwise the message will try to be reprocessed later on, when the expected number matches the one present in the message, if the sequence number received is older, the message is simply ignored, this way we prevent reprocessing of messages, however this security measures are also present in each part of the Streamlet Algorithm.

**Leader Generation:** Each node starts with a RNG with the same seed, this way in every epoch, when nodes pull numbers from the RNG they will all pull the same leader for a given round, leaders of each epoch are stored in a Map this way it also helps verifying leaders of each round either for the future or for the past.

**Propose:**

* **Sender**: For each epoch, the expected leader will calculate one of the current longest notarized chains and will generate a Block that extends it. Including in the block, transactions that are still pending. This is done using a recursive algorithm that returns the list of the longest notarized chains and when they have the same length, the leader chooses one of the chains to extend arbitrarily. For this each node contains a structure of the type of Map<Integer, List<Block>>, where the keys are the “length” indexes of the chain and the Values are a list of blocks present at that level, that extend one of the blocks presents at the level before it. The algorithm used for finding the longest chain is an adaptation of depth first search on a tree.
* **Receiver**: When a block is received by a node, it verifies the integrity of the block and runs the same recursive algorithm as said before

**Voting:**

* **Sender**: When a node receives a propose, it will add it to the blockTree if and only if it is the first time receiving It and its length is greater than the lengthiest notarized chains that the node has seen so far. If these conditions are verified the block is added to the chain the node will broadcast its Vote containing the block being voted and its ID.
* **Receiver:** When a vote is received it is added to the respective block, only if that block doesn’t have that vote yet (no-double Voting), however votes can received before the block itself, so in case the block does not exist, the vote is added to a pendingVotes List, that is refreshed every time a block is received, time-to-time, when a vote is received, etc. When a block reaches more than n/2 votes it becomes notarized.

**Finalization:** Running the same recursive as before, the nodes verify if one of the longest notarized chains has 3 epoch consecutive blocks, and if it does it means that, due to the finalization rule, that chain is common and the longest among all nodes, so it finalizes it, confirming all transactions inside it and saving the finalized chain in disk memory, leaving only in the programs memory the last finalized block and the last notarized block of that chain.

**Crash Recovery:** To note that the implementation of this crash recovery was done in a rough manner, although everything seems to be running fine, code and more testing might be needed, given this, this Is how I implemented it, when a node is trying to reconnect it send a reconnect request sequentially for each note in the list, when a node attends to its request, it sends the current state of the system along with an estimate of the next epoch initiation, when the node trying to reconnect it sends a reconnect message to all specifying to clean its message sequence numbers in order to its messages to be processed again, and after this state transfer the node will start epoching at the estimated time, and everything will be running as normal. The state transfer is made by transferring Java Objects Memory.

**Transactions:** For each epoch the epoch leader will generate dummy transactions and broadcast them to all the other nodes, when a node is proposing a block, it includes all the unprocessed transactions inside it, and when forks happen and blocks are discarded, all failed transactions will return to the pool of transactions.

Uma imagem com texto, captura de ecrã, círculo, lua

Descrição gerada automaticamente

*Fig2: Visual Representation of the system that tries to represent the system*

**Testing and validation**

To test the program, we should run the following command in the root of the main folder:

mvn clean package exec:java -Dexec.mainClass="fcul.tdf.Streamlet" -Dexec.args="1 0"

The first argument is related to the node Id, and the second one refers to if a node is trying to reconnect or not.

To test forks, the utils class, has static variables related to the confusion properties, and they should be changed on demand, as requested in the project specification.

Due to my computer performance, for a high number of nodes it becomes hard to have epochs perfectly synchronized.

Uma imagem com texto, captura de ecrã, menu, Tipo de letra

Descrição gerada automaticamente

*Fig. 3: Example of forks that happened in the first block of the system.*