Distributed Ledgers
(Advanced Distributed Systems)

by

Shruti Goel, Ruchir Dhiman
(2016MCS2657, 2016MCS2685)

MASTER OF TECHNOLOGY

in

Computer Science & Engineering

Introduction

Distributed systems often require nodes to agree upon one unanimous order of execution of the transactions executed, for example, in BitCoin, the order of the transactions by which bitcoins are shared among users has to be agreed upon by all nodes for the proper functioning of the protocol. This unanimous order is typically implemented using distributed ledgers, where all nodes decide upon a single order in a distributed manner, only via message passing.

Another typical decision which has to be made is whether the results of a transaction will be committed or not. This decision has to be mutually decided by all (or a subset of) nodes. One algorithm which allows this to happen is the 2-Phase algorithm. The owner/executor of a transaction takes on the role of a coordinator while the rest act as a participant. The transaction is fully committed if, and only if, all the participants tell the coordinator to commit.

The main complexity of such applications arises, because nodes are not reliable. They often stop execution for sometime, and so, fault tolerance is an essential part of distributed systems. Here, we try to simulate these in a multi-threaded environment using Google's Go language.

Design

Basic Node Structure

Each node, at any time, can perform two functions simultaneously: it can be the coordinator of a transaction as well as a participant for some other node (transaction). Nodes, with a specified probablity can choose to create a transaction; otherwise, they simply process the messages they receive.
Each node has one channel associated with it. All the other nodes have to use the node's associated channel, to communicate with a node.
To create a fail-stop model, nodes with a random probability can go down at any time. The nodes, in this case, will sleep for an indefinite amount of time till they are killed by the main thread.

2-Phase Commit

After creating a transaction, in the *create_txn* function, a node becomes the coordinator of the same, and performs all the necessary operations needed after that: sending a vote request to the two randomly selected nodes, receiving the replies and taking the necessary decision based on them. All these tasks, uptill broadcasting of the committed transaction, is done in the create_txn function. A failed transaction (one of the participants die) is not aborted; the process of the 2 phase commit is simply re-initiated for it.
The participant state is maintained in a dedicated structure in the node, and all the required processing is done via message processing. If the coordinator dies, the transaction is aborted.

Virtual Synchrony

Virtual Synchonry, here, means that if a transaction has been sent to even one of the alive nodes, all others must have it as well. This has been implemented by using a hold-back queue. Received transactions are only added to the ledger if all their predecessors have been received as well.Nodes request other nodes for a transaction they have not yet received.

Global Ordering

Global ordering has been done using a modified version of the Suzuki-Kasami algorithm for mutual exclusion.
At any given time, only one node can have the token, and only the node having the token can use the next global order number (in the token). It sets the global order number for its transaction equal to the current order number in the token, increments the token's number and sends the token to a node which has requested it.
Token forwarding is done using the same procedure as described in the algorithm. If a node having the token dies, a new special token is generated by one of the requesters, and normal execution is resumed.

Fault-Tolerance

Faults are tolerated using a simple ***wait-for-reply*** mechanism.
For each message, the wait time was originally calculated by running a no-failure version of the simulation. After sending a message, the nodes wait for the amount of time (slightly more) it took for the reply to arrive in the no-failure version. This waiting time has been revised for the updated models, but the idea is that once a timer expires, it is assumed that the node, for which we are waiting, is dead.
The necessary contingency measures (some of which are described above) are applied for each case, for example, expiry of the wait-for-token timer starts the execution of a procedure which releases a special token into the system.
Dead nodes are marked using a flag, and such nodes are not selected for the 2-phase protocol, and neither are broadcasts sent to them. Failure of nodes is permanent: they do not wake up, once they have failed.

Message Passing

As written previously, messages are sent to nodes by placing a message on their channel. The processing of the messages is received is done in the *process_message()* function.
This function also includes the required waiting mechanisms for the fault-tolerance implementation. Nodes instantly reply to the messages (which need replying), and all of this is done in this function itself. The final ledger is also created in this

function - entirely done using messages.
This function is repeatedly called during any wait stage. So, the message processing is entirely done via the same thread which owns the channel.

Execution

The main function takes, as input, the number of nodes which have to be run. It then spawns the threads, the starting funtion of which is the *start_node* function. This function contains an endless for-loop, which is used to determine the type of operation the node will do: create transaction or process messages.
The main then sleeps for a pre-set time. All processing done by the nodes occurs during this time period. After waking up, main prints the final results for each node and exits, killing all the spawned threads with itself, at which point the execution finally stops.