Distributed Ledger Implementation

Sahil Jain - 2016BSY7510,Vamsi - 2015MCS2358

May 2017

Distributed Ledger Design

1 Introduction

The problem of Distributed Ledger involves doing successful transactions between 3 nodes using 2-phase commit protocol. Since we are designing a sytem where nodes can die and recover(fail-stop model), we need to ensure the nodes involved in the transaction have a virtual synchrony among them.A successful transaction needs to be atomically broadcasted/multicasted such that every node in the system that is working correctly should process the transaction in the same order.

1.1 Atomic Broadcast Properties

Validity: if a correct participant broadcasts a message, then all correct participants will eventually deliver it.
Uniform Agreement: if one correct participant delivers a message, then all correct participants will eventually deliver that message.
Uniform Integrity: a message is delivered by each participant at most once, and only if it was previously broadcast.
Uniform Total Order: the messages are totally ordered in the mathematical sense; that is, if any correct participant delivers message 1 first and message 2 second, then every other correct participant must deliver message 1 before message 2.

1.2 Virtual Synchrony Properties

Virtual Synchrony implies letting every node invlved in the tx have the same view about other nodes.Nodes can fail and recover and it is important that all nodes involved in the transaction have a view of the nodes which are active and nodes which are dead or appear dead.

Virtual synchrony gives us a model for managing a group of replicated processes (aka state machines) and coordinating communication with that group. With virtual synchrony, a process within a group (one of the replicated servers) can join or leave a group – or be evicted from the group. Any process can send a messag to the group and the virtual synchrony software will implement an atomic multicast to the group.An atomic multicast is the strongest form of reliable delivery, ensuring that a message is either delivered to all processes in the group or to none. This all-or-nothing property is central to keeping all members of the group synchronized. Message ordering requirements can often be specified by the programmer but we nominally expect causal ordering of any messages that effect changes so that we can ensure consistency among the replicas.

2 Design Implementation

2.1 2-Phase Commit Design

A node maintains a txMap and a txList . The role of the txMap is to log all the ongoing transactions given node is involved in and to maintain the state of the transaction. "txList" is the list of all the transactions committed by the node.

Every node runs in the infinite for loop and can initiate transaction only once with any 2 other nodes picked using random number generation. However, a node can be part of more than one transaction and there is no upper limit as such. Using some random number generation , a node is chosen to initiate the transaction with a unique transaction id generated by using some combination of node.id and node's current timestamp.

A node receives the messages on 3 channels dedicated for diff types of messages. TwoPhaseMsgCh for receiving 2-Phase Transaction messages and a BroadcastCh for receiving broadcast messages.

Everytime a node recives a txMsg it looks for the txObject in the txMap maintained for each transaction the node is part of and if found it updates the txState, if not it implies a newTx and it creates a tx Object in the map.

A node can die with certainly probability after writing the tx state in the log and if it doesnt it sends the 2 phase message to the coordinator who initiated the tx.

2.2 Virtual Synchrony Design

Virtual synchrony defines a group view as the set of processes that are currently in the group. These are live processes that are capable of communication. Each group multicast message is associated with a group view and every process in the system has the same group view. There will never be the case where one process will see a different set of group members than another. Group membership and view changes

2.2.1 Group membership and view changes

Whenever a process joins or leaves a group – or is forced out of a group, the group view changes. This change information has to be propagated to all group members. After all, we need to ensure that everyone has the same group view. Group view information is shared with a view change message. This is a multicast message that announces a process joining or leaving a group.

Since we cannot reliably detect process failure, we rely on timeouts to assume that a process is dead. If any process times out waiting for a response from a process, a group membership change takes place and that non-responding process is removed from the group. Since it is no longer in the group, it will not receive any messages sent to the group.Timeout is implemented using ping channel. If the message is not received from a particular node,the node is pinged to ping channel to see if its dead or alive.

2.2.2 Group membership events

Group members may receive any of three types of events:

Regular message. This message is simply treated as an input to the program (the state machine), although its delivery to the application may be delayed based on message ordering policy and our ability to be sure that other group members have received the message.

View change. A view change is a special message that informs the process of a group membership change. This will affect any multicasts that are generated by the process from this point forward. We will discuss the view change in more detail shortly.

Checkpoint request. If a process joins a group, it needs to bring its state (internal state as well as stored data) up to date so that it contains the latest version and is synchronized with the other replicas. To do this, a process may send a checkpoint request message to any other process in the group. That process will send its current state to the requesting process.

2.3 Atomic Broadcast Implementation

2.3.1 Sending node:

1. When a process wants to send a broadcast message, the message is sent to all receivers. It is given a timestamp that is higher than all (by the sender) known timestamps. This timestamp together with the sender's node's unique name will be the unique identifier,id ,for the message.The message is inserted into the local message queue of not delivered messages in timestamp order (this means that it must be put at the end!). The message is marked pending.

2. When the node has got an acknowledgement with a global timestamp proposal for a given message from all the other nodes the maximum of those

is chosen as global timestamp for the message.In the local queue the message's timestamp is updat ed to the chosen timestamp, the message is marked ready and the queue is sorted in timestamp order. If two messages have the same timestamp they will be ordered according to the senders identifier. A confirmation message giving information on the global timestamp is sent to all other nodes.

2.3.2 Receiving node:

3. When receiving a broadcast message it is given a timestamp that is higher than any in the nodes message queue and not lower than the timestamp give n by the sender.Then it is put last in the local broadcast message queue marked pending.Then an acknowledgement message with the proposed timestamp is sent to the originator of the message.

4. When receiving a confirmation message giving info confirmation on the global timestamp.

5. When the first message in the queue is marked ready it should be delivered to the receiving process and then erased from the queue.

6. If the first message in the queue is marked pending , no messages should be delivered even if they are marked ready.

2.4 Node Thread Implementation

–In go Thread is implemented as a goroutine.
– For Every node a goroutine nodeRun is executed concurrently each running in infinite loop trying to initiate a transaction and reading the messages from channels in a non-blocking manner.
– A go-routine for a node which is alive terminates if the number of transactions that it has added onto its global txList becomes equal to the maximum transactions of the system.

2.5 Code Termination

–a sync variable is used to wait till all the go routines terminate.
–When all the threads terminate or go-routines return the list of transactions made by all the nodes is dumped.