Randomized Consensus Project Report

Bogdan Suvar, Daniel Bali, Seckin Savasci
Instituto Superior Técnico
{bogdan.suvar,janos.bali,seckin.savasci}@tecnico.ulisboa.pt

Abstract

The FLP impossibility states that no deterministic algorithm can solve consensus in an asynchronous system where at least one process can crash. To circumvent this, we can relax the conditions and make a probabilistic algorithm for consensus. This report will give details on our implementation of Randomized Consensus.

1. Stack

Two channels are required according to the Randomized Consensus algorithm. For the first phase of the protocol, it is sufficient to use *Best-effort Broadcast*. For the final part of the second phase, *Reliable Broadcast* is required. We ended up using the same stack, but only special messages (decision) are passing through the full reliable broadcast stack. This can be seen on figure 1.

For the reliable channel, we had to use Lazy Reliable Broadcast on top of a TCP-based Perfect Failure Detector. The best-effor channel is much simpler, only requiring a basic broadcast layer.

The implementation of the randomized consensus algorithm is in the *tfsd.consensus* package.

2. Code

This section contains a brief explanation of the code and its connection to the randomized consensus algorithm.

Figure 2 shows the basic outline of the architecture for our solution. The *ConsensusApp* sends SendableEvents to the ConsensusSession, where they are converted to ProposeEvents and broadcasted. When the algorithm reaches the end of phase 2, this session notifies the ConsensusApp, which in turn sends a SendableEvent to ReliableConsensusApp. This session will then reliably broadcast the request to start the decision process.

The main part of the algorithm is implemented in ConsensusSession. *handleSendable* is the function responsible for converting SendableEvents into ProposeEvents which are then broadcasted.

After a ProposeEvent is received, its parameters are observed. It has three main parameters: phase, value and timestamp. The propose event is then processed according to its phase.

In phase one, the algorithm waits for a majority. This is handled by *handleProposePhase1*. When a majority is reached, we check if the values are the same. If they are, we enter phase two with this value - entering means we send a new ProposeEvent with the phase set to two. When the values are not the same, we enter phase two with the *bottom value*.

Phase two waits for all replies, minus the number of tolerated failures. When a quorum is reached, it tries to find f+1 occurences of the same v* value. When this is found, it issues a decide.

When a decision could not be made, we restart the algorithm from phase one, with a random

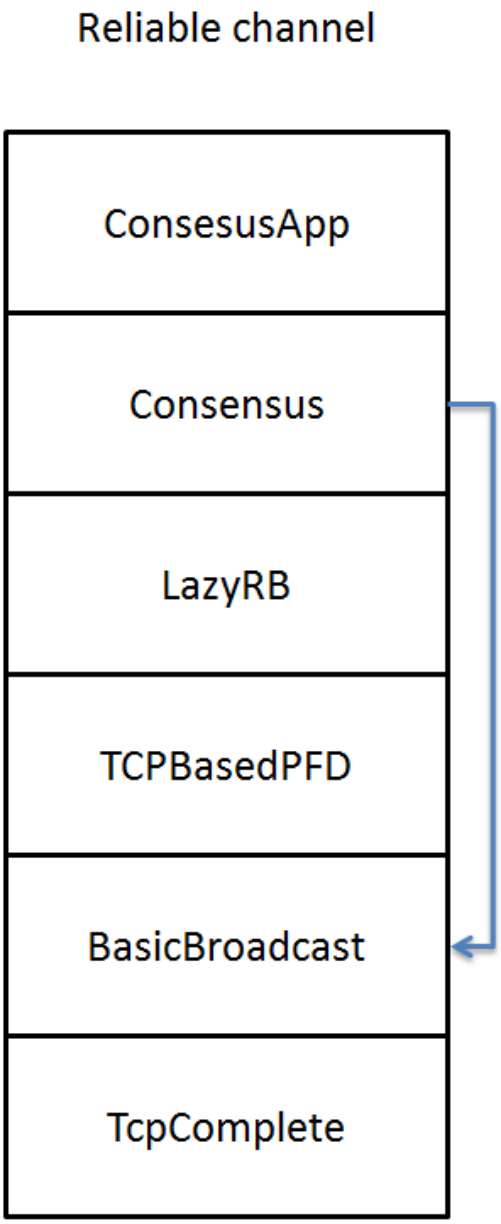

Figure 1: Applia layer stack

value from the previously seen proposals. After the restart, the queues of the quorums are emptied.

Timestamps are used to differentiate between different proposals. There is a started and a decided timestamp. When we see a proposal with a timestamp that is less-than or equal to our decided timestamp, we can discard it.

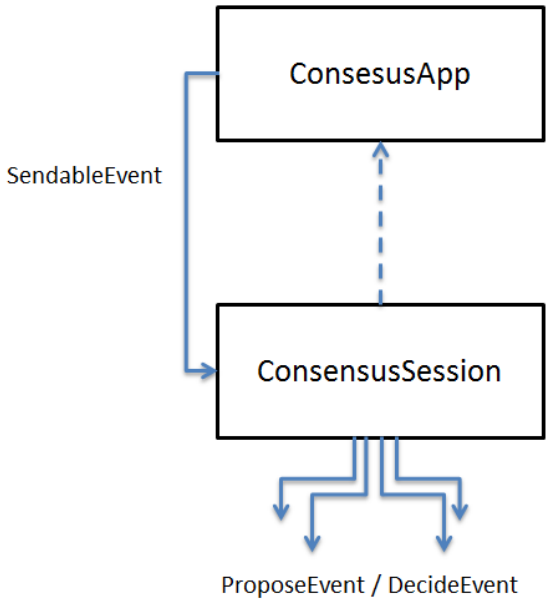

Figure 2: Architecture of the application

A possible feature is to store all the proposals that have a timestamp higher or equal.

3. Implementation & Testing

3.1 Consensus Logic

3.2 Sample Application

To test our implementation, we have developed a rapidly-exploring random tree generator. Rapidly-exploring random trees[1] enables searching n dimensional spaces efficiently. They grow through unexplored directions, and mainly used in automated robotic path planning. Related to our topic, distributing the same n dimensional path tree to efficiently can be done by reaching a consensus on each iteration to pick the destination point.

Rapidly-exploring random tree generator is a suitable application for testing. It creates random trees based on the random values for each iteration. To get the same tree in every process, the only way is reach consensus for each iterations

Figure 3: Visualization of RRT in 2D space

random value. So in each iteration, every process proposes its own value. Deciding on each iteration lets our consensus to run many times in quick succession which makes it possible to expose errors that are not visible by longer time windows between each decision.

4. Evaluation

The current implementation of the consensus, with the demo application reaches consensus 2000 times in 2 minutes. In each process, resulting trees are identical which verifies the correctness of our implementation for consensus.

References

[1] S. M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, 1998.