Progress report - Implementing the HotStuff consensus algorithm
Marc Harvey-Hill (mnh33@cam.ac.uk)
Supervisor: Christopher Jensen
DoS: Timothy Jones
Overseers: Pietro Lio, Jeremy Yallop

My project is implementing the HotStuff consensus algorithm in Ocaml. To meet my success criteria I need to carry out more testing, but I will work on proving correctness first. The criteria should be met by the end of next week (starting 6/2). I am slightly ahead on performance tuning so I am roughly on schedule.

Work completed:

I implemented RPC calls between nodes using Capnproto. To do this I first learnt the Lwt library and monads - this is necessary for carrying out asynchronous tasks like sending a request and waiting for a response. I designed a message schema for nodes and clients to communicate and functions to convert between Capnproto API types and consensus state machine internal types. Additionally I implemented server code to wait for messages and advance the consensus state machine, and client code that carries out actions depending on the state machine output (eg. sending messages).

I implemented both the unchained and chained Hotstuff algorithm. The pseudocode for both algorithms left room for interpretation and I made some modifications in order to implement it in practice. I had to debug deadlocks by analysing the program trace in live tests.

I integrated cryptography (threshold signatures) using the Tezos crypto library. This allows nodes to sign messages and have them verified. It also allows nodes to create a quorum certificate by combining these partial signatures, this is also verified by the receiver. I did not implement key generation and distribution as this is outside the project scope.
I implemented a byzantine view change protocol. The pacemaker mechanism was not sufficiently specified in the paper so I based this on a talk by Ittai Abraham (St. Petersburg 2019). This involved a timer mechanism so a view times out when progress is not made and augmenting the consensus implementations with code to ‘complain’ to the next leader on a timeout, and that new leader to start a new view.

I tuned the consensus and RPC implementations to reduce memory usage (used a memory profiler to analyse the program). I did this by using the node offset instead of the node itself to prevent messages becoming very large or having recursive types - this dramatically reduced the memory footprint.

I wrote ‘expect’ tests to informally check the correctness of the consensus implementations for:

- Leader being able to commit in isolation
- Leader with replicas being able to commit several views
- Similar tests with cryptography enabled
- Client requests being committed and responded to
- Carries out a view change on a timeout event

I also implemented a live test with a client sending requests. This client is able to vary throughput and number of requests sent and output statistics like mean and standard deviation of latency.

Todo:

- Run a live test on the lab's Sofia server with 32 nodes
- Further local testing for correctness
- Analysis of throughput and latency
- Proof of correctness for modifications to the original algorithm
- Extension: further improve throughput and reduce latency
- Extension: implement horizontal reconfiguration of nodes participating