Implementing the HotStuff consensus algorithm

Marc Harvey-Hill

1 1 2023

1 Algorithm description

1.1 Introduction

HotStuff is a Byzantine fault-tolerant consensus algorithm. It allows a group of parties to agree on some piece of information under adverse conditions where some messages can be lost and some parties are controlled by a malicious adversary. The main application of HotStuff is in permissioned blockchains. For example, one could create a cryptocurrency by using HotStuff to reach consensus on a log of transactions like "Account X transfers account Y £10". By allowing a large amount of devices to reach consensus on a ledger, one can develop a decentralised payment system. In general Blockchains can be applied in situations where there is some central authority (a bank, DNS server, government, etc.) to instead create a distributed and decentralised system that requires less trust from participants.

The protocol can be viewed as a solution to the Byzantine generals problem. In this problem a group of generals must all agree to siege a castle at the same time, as a single army would be defeated on its own. The problem is that the generals can only communicate via messengers that take some time to arrive and can be captured en-route. Additionally up to a third of the generals may be malicious, and try to prevent the other generals from reaching consensus on a time to attack. By following the HotStuff protocol the generals can ensure that they all attack together. In contrast to the generals problem, HotStuff is able to agree multiple values instead of just one, so instead of deciding a single value like "Attack at dawn", HotStuff can agree on a log of multiple values. The key is that once a value is decided and appended to the log it can never be modified or erased, the log can only ever be extended.

These conditions are described by the system model: partially synchronous, Byzantine, with reliable, authenticated, point-to-point delivery. This means that messages sent by one party will always be delivered to another within some bounded amount of time after GST has been reached and a message source cannot be spoofed. The Byzantine assumption means a maximum of n faulty nodes may be controlled by an adversary that is actively trying to prevent us from correctly reaching consensus, where $n = 3f + 1$ and n is the total number of parties.

HotStuff can be used in 'permissioned' blockchains, meaning that we must agree on the set of participants in advance. This is in contrast to permissionless blockchains like Bitcoin or Ethereum, where participants can freely leave and join the consensus protocol. This is because HotStuff is vulnerable to a 'Sybil' attack - by adding a large amount of Byzantine nodes to the network an adversary can bypass our assumption that only a third of nodes are Byzantine. To prevent this kind of attack one could extend HotStuff with a system like Proof of Work or Proof of Stake. One of the extensions to this project is to design and implement a way to reconfigure the network, allowing the set of nodes participating in the system to be changed - this has not been achieved before.

1.2 Non-Byzantine case

We will start by describing an algorithm to solve the simpler problem of reaching consensus with the stronger assumption of a crash-stop model instead of a Byzantine one. Examples of similar algorithms include Raft and multi-shot Paxos. Such an algorithm must ensure that once a value is decided and appended to the log, it cannot be modified or erased. In each view a leader proposes some log which it aims to decide upon with a group of replicas. Our implementation assigns leaders to views using a round robin system.

Each view can be broken into two phases; phase 1 allows the leader to learn of previously decided values, and in phase 2 it decides on a value. Phase 1 is initiated by the leader broadcasting the current view number to the replicas, that respond by sending their longest accepted log (the one with the highest corresponding view number). Once the leader has a quorum of responses, it initiates phase 2: it selects the longest log that has been sent to it and broadcasts it to the replicas. The leader may also extend the log at this point with its own values, or create a new log if it did not receive anything. Finally the replica updates its log to the value sent by the leader, and sends an acknowledgement. Once the leader receives a quorum of acknowledgements it can commit the new log. This algorithm satisfies our requirement that a committed log can only be extended.

We will refer to phase 1 as the 'new view' stage, and phase 2 as the 'propose' phase (there is no standard terminology but we find this to be more clear than the terminology used in the HotStuff paper). These are followed by the 'commit' phase, when the leader sends a commit message to replicas. Once they receive this message they can consider the log decided, and execute the new commands which have been added to the log.

Example:

1. New view:
   (a) leader → replicas: "view = 3"
   (b) replicas → leader: "view = 2, log = ['hello', 'world']", ...
2. Proposal:
   (a) leader → replicas: "log = ['hello', 'world', '!']"
   (b) replicas → leader: "ack", ...
3. Commit:
   (a) leader → replicas: "commit"
   (b) replicas → leader: "ack", ...

1.3 Byzantine case

In order to extend our algorithm to achieve consensus under a Byzantine threat model we must handle three threats that we will deal with in turn. In order to do this we must first introduce the concept of a 'threshold signature'. Acknowledgements from replicas all contain a signature over the message to prove that they were actually sent by the correct node. The leader can create a threshold signature by combining $n - f$ ack messages' signatures to prove that they really received a quorum of acknowledgements. A collection of a quorum of votes with a threshold signature is known as a 'quorum certificate'.

1. Threat: equivocation - a faulty leader broadcasts one value to some replicas and a different value to others. For example in the case of a cryptocurrency, this could result in a malicious actor (controlling account X) carrying out a double spend attack, sending "Account X transfers account Y £10" to some nodes, and "Account X transfers account Z £10" to other nodes, even if Account X only contains £10.

   Solution: Add a new stage 'pre-propose' which happens just before the 'propose' phase. In this phase the leader again chooses the longest log it received in the 'new view' phase to send in the 'pre-propose' phase, this may also be extended with the leader's new values. Once we receive a quorum of acknowledgements, we begin the 'propose' phase. The difference is that this time we include a quorum certificate over the quorum of pre-propose acks, which proves that we pre-proposed the value to at least $n - f$ nodes and received their acks, so we are not proposing one value to some node and another value to others.

2. Threat: A faulty leader pre-proposes a log that conflicts with one that has already been committed.

   Solution: Replicas must lock on a value once it is proposed and not accept a pre-proposal from a leader that contradicts that. They will store the quorum certificate that they receive during the 'propose' phase and will only accept a new pre-proposal if it extends from the node stored in the certificate.

3. Threat: A faulty replica sends a the leader a fake log in its new-view message that was never actually proposed. Note that this does not break safety as a pre-proposal for the fake log would not be accepted since the protocol is safe. However, this breaks the liveness property of the protocol as a non-faulty leader could be prevented from making progress by faulty replicas sending fake logs.

   Solution:

Example:

1. New view:
   (a) leader → replicas: "view = 3"
   (b) replicas → leader: "view = 2, log = ['hello', 'world'], qc = (pre-proposal acks from view 2)", ...
2. Pre-proposal:
   (a) leader → replicas: "view = 2, log = ['hello', 'world']"
   (b) replicas → leader: "log = ['hello', 'world'] ack", ...
3. Proposal:
   (a) leader → replicas: "log = ['hello', 'world', '!']", qc = (pre-proposal acks from previous stage)"
   (b) replicas → leader: "ack", ...
4. Commit:
   (a) leader → replicas: "commit", qc = (proposal acks from previous stage)"
   (b) replicas → leader: "ack", ...

1.4 Optimistic responsiveness

Consider again the pre-proposal phase. In this phase the leader selects the quorum certificate from the highest view that it hears about from the new-view messages. However, it is possible that there is some honest replica that we do not hear from (perhaps their message was lost) that is locked on a higher view proposal than the one that we choose to send. When this replica receives our pre-proposal, it will reject it as it is locked on a higher view proposal. This means that we could be prevented from making progress in this view by missing one replica in the new-view phase.

This means that our system doesn't have the 'liveness' property, which means that it will make progress under synchronous conditions when a non-faulty leader is elected. It is possible that we repeatedly don't hear from this one honest replica and fail to make progress indefinitely. One solution to this problem is to introduce a timeout Δ, once this timeout is elapsed we will give up on this view and start a new one. This solution has the disadvantage that our system is not 'responsive', which means that the system can make progress as fast as network conditions allow when we have a faulty leader, and does not depend on Δ.

In order to achieve responsiveness we can modify our algorithm by adding a 'key' phase in between pre-propose and propose. This ensures that if some honest replica becomes locked on a value in the propose phase, then there are at least $f + 1$ honest nodes that have a 'key' for that value. More specifically, they have a 'key-proof' composed of a quorum certificate over pre-proposal acks. Replicas then send this key-proof with their new-view message when a new view begins. The new leader selects the key-proof with the highest view proposal, and sends the key-proof along with their pre-proposal. Even if the leader does not receive a new-view message from the replica which is locked on the highest view proposal, they must have received the key to this proposal from some honest replica, so their pre-proposal will make progress.

1. New view:
   (a) leader → replicas: "view = 3"
   (b) replicas → leader: "qc = (key-proof for view = 2, log = ['hello', 'world'])", ...

2. Pre-proposal:
   (a) leader → replicas: “view = 3, log = [‘hello’, ‘world’, ‘!’], qc = (key-proof for view = 2, log = [‘hello’, ‘world’])”
   (b) replicas → leader: “log = [‘hello’, ‘world’, ‘!’] ack”, ...
3. Key:
   (a) leader → replicas: “qc = (pre-proposal acks from previous stage)”
   (b) replicas → leader: “log = [‘hello’, ‘world’, ‘!’] ack”, ...
4. Proposal:
   (a) leader → replicas: “qc = (key acks / propose-proof from previous stage)”
   (b) replicas → leader: “ack”, ...
5. Commit:
   (a) leader → replicas: “commit”, qc = (proposal acks from previous stage)”
   (b) replicas → leader: “ack”, ...