Notes on modern consensus protocols: Tendermint, Casper FFG, Hotstuff etc

ANDREW LEWIS-PYE

The literature on consensus protocols can be somewhat impenetrable despite the fact that modern consensus protocols are quite simple. The aim of these notes is to give clear explanations of (sometimes simplified) versions of Tendermint, Hotstuff and other protocols. We aim to explain not just how these protocols work, but why they are defined the way they are.

1 REQUIRED BACKGROUND.

We'll assume the reader is familiar with the concept of State-Machine-Replication (SMR) protocols. These notes will also be easier to read given some familiarity with such things as the 'partially synchronous' setting, although we'll also define the necessary terms here.

2 THE SETTING

(1) We consider a set of n processors, all of whom are known to each other from the start of the protocol execution. In particular, this means that all processors know the value of n. We also assume that each processor has a public/private key pair and that all processors know the public keys of the other processors.
(2) We consider a setting in which communication is *partially synchronous*. This means that there is some known delay Δ and some unknown 'global stabilisation time' GST such that, if a message is sent at time t, then it is received by all processors by time $\max\{t, GST\} + \Delta$ (but may be received by different processors at different times). For simplicity, we'll start by assuming messages are 'broadcast to all', rather than sent to individual processors.
(3) In this area, one normally supposes that communication is point-to-point rather than 'broadcast to all', i.e. one supposes that each pair of processors has a private (authenticated) communication channel between them. This becomes relevant if one is concerned with things like message complexity. We'll begin by ignoring such issues, but will come to them later – at which time we'll also consider the use of point-to-point communication rather than broadcast.
(4) For the sake of simplicity, we'll also start by assuming that all processors have perfectly synchronised clocks.
(5) We suppose that some (unknown) subset of f many processors is faulty, and that faulty processors may behave arbitrarily (i.e. we assume 'Byzantine' failures). Dwork, Lynch and Stockmeyer showed that, if there are f faulty processors, then at least $3f + 1$ processors are required (in total) for a state-machine-replication (SMR) protocol to work in the partially synchronous setting. So, we'll assume $n \geq 3f + 1$. We'll also assume an ordering on the processors is known, which means that we can talk about the ith processor.

3 TENDERMINT

(1) There are a number of versions of Tendermint. Part of the reason for this is that the original formulation (in Ethan Buchman's thesis) had liveness issues. We'll present a simplified version here, without worrying about things like message complexity.
(2) Time is divided up into *epochs*. Each epoch has a different *leader*, whose job it is to suggest a new block of transactions. The leader for the ith epoch is processor i mod n.

Emails. Andrew Lewis-Pye: a.lewis7@lse.ac.uk, andy@lewis-pye.com.

(3) We also consider (signed) *votes* on blocks at various stages of the protocol. If $n-f$ processors produce votes for a particular block (and corresponding to a specific round of voting), we will call this set of votes a *quorum certificate* (QC). We only ask for $n-f$ votes, because faulty processors cannot be relied on to vote. Note that, if non-faulty processors only vote for one block in each round of voting, then two different blocks can't both receive QCs in the same round. This follows since:

> ($\dagger_0$) Any two sets of $n-f$ processors must have at one non-faulty processor in common .

To see that ($\dagger_0$) holds, note that any two sets of $n-f$ processors must have at least $n-2f$ processors in common. They must then have at least one non-faulty processor in common, since $n-2f > f$.

(4) A simple approach would be to proceed as follows:
   (a) A processor considers a block confirmed once they see a QC for that block.
   (b) Each epoch is of length 2Δ. If the leader for epoch i is non-faulty, then, when the epoch begins (at time t say), they produce and broadcast a new block of transactions. This block includes the epoch number and should extend the highest confirmed block (i.e. that corresponding to the highest epoch number, rather than the longest chain) amongst those they have seen.
   (c) At time $t+\Delta$, non-faulty processors each consider the first (signed) block[1] they receive from the epoch leader. If it extends the highest confirmed block they have seen, then they produce a vote for that block corresponding to epoch i.

(5) In the partially synchronous setting, however, it's especially easy to see that the method above does not produce consistency, i.e. incompatible blocks might become confirmed. A block B might be confirmed in epoch i, for example, but it might be the case that fewer than $f+1$ of the non-faulty processors see the QC . Then a block B', which is incompatible with B, might be proposed at epoch $i+1$ and (potentially with the help of faulty votes) might also become confirmed.

(6) What happens if we consider two rounds of voting on the block instead? Let us suppose that, if they see a QC on the block B produced in 'stage 1' (say) of epoch i, processors then produce a 'stage 2' vote, and it is a QC in stage 2 that means confirmation. The problem with the protocol from (4) was that a block might be confirmed, but non-faulty processors might not see the relevant QC. With our new approach, it is at least the case that if B is confirmed then at least $n-2f$ ($\geq f+1$) non-faulty processors must have seen the stage 1 QC for the block (because we need at least $n-2f$ non-faulty processors to vote in the second round to produce confirmation). Suppose we now instruct processors who have seen the stage 1 QC to 'lock' on the block B, which means that they don't yet consider it confirmed, but now will not vote for blocks incompatible with B (while the lock is in place). It's easy to see that this new rule produces consistency:

> ($\dagger_1$) If the block B is confirmed at epoch i, then at least $n-2f$ non-faulty processors become locked on B. Since at least $n-2f$ non-faulty processors are required to produce any QC, this makes it impossible any block incompatible with B to receive a QC in subsequent epochs.

The only danger now is that we threaten liveness. When the $n-2f$ non-faulty processors become locked on B, it is possible that $\leq f$ non-faulty processors are already locked on a previous and incompatible (and necessarily unconfirmed) block B'. With the non-faulty processors split, we now can't ensure that any future blocks become confirmed. This doesn't

[1] We'll assume throughout that processors ignore messages that aren't properly formed, with the right epoch number etc.

seem like much of a problem, though, because, once the global-stabilisation-time (GST) comes, the processors that are locked on B' will see the stage 1 QC for B. So, we can just agree that, when a processor is locked on a specific block corresponding to a certain epoch, they will unlock if they subsequently see a QC corresponding to a greater epoch. Note that this doesn't harm the argument in (†) above in any way, so the consistency argument still holds.

(7) We therefore consider a protocol with the following instructions. All processors begin with their 'lock' set to be the genesis block, which corresponds to epoch 0. All blocks and votes contain their epoch and stage number. We consider the obvious ordering on QCs, in terms of epoch number and then stage number within the epoch.[2] The 'highest quorum' below a block B is the highest QC that corresponds to any block (including B) in the initial segment of that chain.

The instructions for epoch $i > 0$, starting at $t = 3i\Delta$.
Time t. The leader for epoch i proposes (produces and broadcasts) a block extending the block with the highest QC it has seen (or the genesis block, if no QC has been seen).
Stage 1. Time $t + \Delta$. Each processor considers the first block B they receive for epoch i produced by the leader. If it extends their lock, or else if the highest QC below the block is as high or higher than they have previously seen (if it's higher they release their lock), they broadcast a stage 1 vote for B.
Stage 2. Time $t + 2\Delta$. If a processor has seen a stage 1 QC for some block B corresponding to epoch i, then they set B as their lock and broadcast a stage 2 vote for B.

(8) We've already proved consistency for this protocol. Liveness is not hard to prove either. Consider an epoch with non-faulty leader which begins at time t such that $t - \Delta$ is after the global-stabilisation-time (GST). Let B be the highest lock (i.e. with greatest epoch number) amongst all non-faulty processors at $t - \Delta$. The non-faulty leader will see the corresponding QC by time t, and will then propose B' which has highest QC below it at least that of B. All non-faulty nodes will produce votes for B' in stages 1 and 2 of the epoch, and the block will then be confirmed.

4 CHAINED TENDERMINT (BASICALLY CASPER FFG)

(1) If that protocol seemed simple, the next realisation is that we can use a trick to make it even simpler! In fact, we can get the extremely simple protocol from (4) in the previous section to work, if we just apply our new notion of confirmation together with locking.

(2) The basic idea is as follows. Rather than have two rounds in epoch i, we have a single round, but we have the votes for that round count in two ways. First of all, they count as a stage 1 vote for the block at epoch i. If that block extends an epoch $i - 1$ block with a QC, however, we *also* consider those votes as a stage 2 vote for that previous block. So now, when we see QCs corresponding to two compatible blocks at two successive epochs i and $i + 1$, we consider the block at epoch i confirmed.

(3) Here's the new protocol.

The instructions for epoch $i > 0$, starting at $t = 2i\Delta$.
Time t. The leader for epoch i broadcasts a block extending the block with the highest QC it has seen.
Time $t + \Delta$. Each processor considers the first block B they receive for epoch i produced by the leader. If B extends their lock, or else if the highest QC below the block is as high or

[2] It's worth stressing that it's epoch number rather than chain length that we care about here. Streamlet cares about chain length and it makes the proofs more fiddly.

higher than they have previously seen, they: (a) broadcast a vote for B, (b) set their lock to be the block corresponding to the highest QC below B.

(4) As stated above, when we see QCs corresponding to two compatible blocks at two successive epochs i and $i+1$, we consider the block at epoch i confirmed (together with all earlier blocks in the same chain). The proofs for liveness and consistency are essentially the same as before, but now you need two non-faulty leaders in a row for the liveness proof.

5 HOTSTUFF

(1) A notable feature of the protocols we have described so far is that the speed at which they progress is governed by Δ rather than the *actual* network delay – if the network speeds up over time, or has sustained periods of smaller delivery time, the protocol has no way to take advantage. Roughly, protocols that *can* do so are called *optimistically responsive*. So, informally, optimistic responsiveness requires that, after GST, a non-faulty leader can drive the protocol to consensus in time depending only on the actual message delays, independent of any known upper bound on message delays.

(2) If we want our protocol to be optimistically responsive then, rather than have processors wait for intervals of length Δ between instructions, we could allow processors to progress to the next stage of their instructions when they receive sufficiently many messages from the previous stage. If we proceed in this way, then we'll still need some notion of a *timeout* (after which a processor will progress to the next instruction anyway), because we don't want to wait for ever for faulty leaders to propose new blocks. That's okay, because optimistic responsiveness only requires the protocol to progress at network speed when leaders are non-faulty.

(3) With this plan in mind, it seems natural to consider the following version of Tendermint. All processors keep a *timeout clock*, which is set to 0 whenever they enter a new epoch. When their timeout clock reaches $\texttt{timeout}(i)$, the processor sends a $\texttt{Start}\ i+1$ message. [3] Whenever a processor sends a $\texttt{Start}\ i+1$ message, it attaches to this message the highest QC they have seen,[4] together with the corresponding block. A processor enters epoch i if they are presently in a lower epoch and they see $\texttt{Start}\ i$ messages from $n-f$ distinct processors.

While a processor is in epoch i it carries out the following instructions (in addition to the timeout instructions above):
Block proposal. Upon entering epoch i, the leader proposes a block extending the block with the highest QC it has seen (or the genesis block, if no QC has been seen).
Stage 1 voting. Upon seeing the first block B corresponding to the epoch and produced by the leader, each processor proceeds as follows. If B extends their lock, or else if the highest QC below B is as high or higher than they have previously seen (if it's higher they release their lock), they broadcast a stage 1 vote for B.
Stage 2 voting. Upon seeing a block B for epoch i with a Stage 1 QC, processors set B as their lock and broadcast a Stage 2 vote for B.
The sending of $\texttt{Start}\ i+1$ messages. Upon seeing a block B for epoch i with a Stage 2 QC, processors send a $\texttt{Start}\ i+1$ message.

(4) For the protocol defined in (3), consistency follows just as before. It's not hard to see, though, that the protocol has liveness issues. To see this, we specify an infinite sequence of epochs after *GST* during which no new blocks become confirmed. We start each epoch $i+1$ with

[3] So $\texttt{timeout}$ is a function that tells the processor how long to wait before timeout in epoch i. We'll assume, for now, that all clocks run at the same speed.
[4] The reason for attaching this QC will become clear soon.

only a single non-faulty processor (p_i say) having seen the highest stage 1 QC and being locked on the corresponding block, B_i say. The new leader begins the epoch because it sees `Start` $i+1$ messages from $n-f$ processors that do not include p_i. The highest stage 1 QC attached to those `Start` $i+1$ messages is for epoch $i-1$ and is for a block B_{i-1} which is incompatible with B_i. The leader then proposes B_{i+1} which extends B_{i-1}, to which $n-f-1$ non-faulty processors respond with a stage 1 vote, but which p_i rejects it because it is locked on B_i. Eventually, all of the non-faulty processors but one give up and move to the next epoch. Just before one of the non-faulty processors (p_{i+1} say) times out, however, a faulty processor produces a stage 1 vote for B_{i+1} and shows that vote to p_{i+1}, who then becomes locked on that block, and so on.

(5) With the previous example in mind, it becomes clear that having intervals of length Δ between instructions played an important part in establishing liveness – the gap allowed the next leader to be made aware of the highest lock held by a non-faulty processor. Interestingly, the same sort of mechanism takes place when establishing liveness for longest chain protocols like Bitcoin: One requires an interval of a certain length in which no blocks are produced to ensure that all non-faulty processors have the same view as to the longest chain. It therefore becomes especially interesting to see how we can circumvent this need.

(6) So, how can we modify the protocol to ensure liveness together with optimistic responsiveness? In Section 3, we solved our issues by adding a round of voting. In fact, the same thing will work again here. The basic idea is as follows. Let's suppose that we now have three rounds of voting. A processor becomes locked on B when they see a Stage 2 QC, while being confirmed means seeing a Stage 3 QC. Now suppose that a non-faulty leader enters a new epoch i sufficiently long after GST. If B has the highest lock when the leader enters the new epoch, then at least $n-2f$ of the non-faulty processors must have seen the Stage 1 QC for B. The leader enters the new epoch because it sees `Start` i messages from $n-f$ processors, at least $n-2f$ of which must be non-faulty. By ($\dagger_2$) below, this means that at least one of those $n-2f$ non-faulty processors must have seen the Stage 1 QC for B, and will have told the leader about it with their `Start` i message. The leader will then propose a block extending B, which will be confirmed by the end of epoch i. A precise proof will follow the precise description of the protocol below.

($\dagger_2$) Any two sets of $n-2f$ non-faulty processors must have at one processor in common .

To see that ($\dagger_2$) holds, note that any two sets of $n-2f$ non-faulty processors must have at least $n-3f$ (≥ 1) processors in common, since there are $n-f$ non-faulty processors in total.

(7) While a processor is in epoch i it carries out the following instructions (in addition to the same timeout instructions as previously):
Block proposal. Upon entering epoch i, the leader for epoch i proposes a block extending the block with the highest QC it has seen (or the genesis block, if no QC has been seen).
Stage 1 voting. Upon seeing the first block B corresponding to the epoch and produced by the leader, each processor proceeds as follows. If B extends their lock, or else if the highest QC below B is as high or higher than they have previously seen (if it's higher they release their lock), they broadcast a stage 1 vote for B.
Stage 2 voting. Upon seeing a block B for epoch i with a Stage 1 QC, processors broadcast a Stage 2 vote for B.
Stage 3 voting. Upon seeing a block B for epoch i with a Stage 2 QC, processors set B as their lock and broadcast a Stage 3 vote for B.

The sending of `Start` $i+1$ messages. Upon seeing a block B for epoch i with a Stage 3 QC, processors send a `Start` $i+1$ message.

(8) Consistency for the protocol given in (7) follows as before. To prove liveness, we need to specify the `timeout` function. In the original Hotstuff paper, the authors were not overly concerned with doing this in an efficient way, and the suggestion there was to use timeouts that increase exponentially with i. The following proof works when `timeout`(i) takes the constant value 4Δ.
- Suppose that, at time t such that $t - \Delta > GST$, the non-faulty leader p for epoch i enters epoch i. This means: ($\diamond$) No non-faulty processor can have entered any epoch $i' \geq i$ at any time prior to $t - \Delta$, since otherwise p would have seen those same `Start` i' messages strictly prior to time t.
- Suppose first that some non-faulty processor p' enters an epoch $i' > i$ strictly prior to time $t + 3\Delta$. Then p' enters epoch i' because it receives `Start` i' messages from $n - f$ distinct processors, at least $n - 2f$ of which must be non-faulty. Those non-faulty processors will not send `Start` i' messages due to timeout at any time prior to $t + 3\Delta$, because ($\diamond$) is satisfied, and because the `timeout` function takes the constant value 4Δ. Those $n - 2f$ non-faulty processors must therefore have seen a block corresponding to some epoch $\geq i$ being confirmed, which establishes liveness for this case.
- So suppose, instead, that no non-faulty processor enters an epoch $i' > i$ strictly prior to time $t + 3\Delta$. Since p enters epoch i at t, all non-faulty processors must enter epoch i by time $t + \Delta$. Let B have the highest lock held by any non-faulty processor upon entering epoch i (so that the epoch corresponding to this lock is $< i$). At least $n - 2f$ non-faulty processors must have seen a stage 1 QC for B and have produced stage 2 votes for B prior to entering epoch i. Note also that (the leader) p enters epoch i because it sees `Start` i messages from $n - f$ distinct processors, at least $n - 2f$ of which must be non-faulty. By ($\dagger_2$), at least one of those non-faulty processors must therefore have seen the stage 1 QC for B prior to entering epoch i. This means that the `Start` i message sent by the processor will include a QC at least as high as the stage 1 QC for B. The leader p for epoch i will then propose a block with highest QC below it at least that for B, which will receive a stage 3 QC by $t + 3\Delta$.

(9) Of course, Hotstuff can then be 'chained' in the same way as Tendermint.