linear in the # replicas

Hotstuff

→ leader-based Byzantine fault-tolerant replication protocol for the partially SYNCHRONOUS model

→ 3-phases → allow leader to pick the highest QC it knows of.

→ supports frequent succession of leaders

linear view-change
+
optimistic responsiveness

Algorithm 1 Utilities (for replica r).

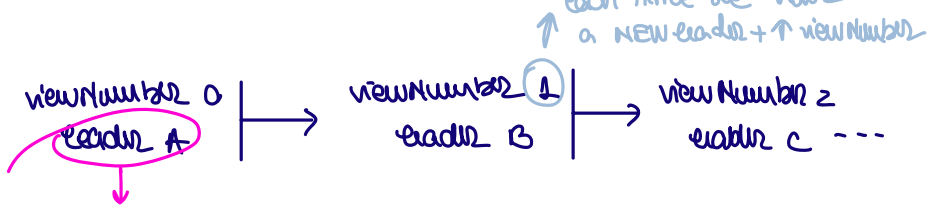
```
1:  function Msg(type, node, qc)
2:      m.type ← type
3:      m.viewNumber ← curView
4:      m.node ← node
5:      m.justify ← qc
6:      return m
7:  function voteMsg(type, node, qc)
8:      m ← Msg(type, node, qc)
9:      m.partialSig ← tsign_r((m.type, m.viewNumber, m.node))
10:     return m
11: procedure createLeaf(parent, cmd)
12:     b.parent ← parent
13:     b.cmd ← cmd
14:     return b
15: function QC(V)
16:     qc.type ← m.type : m ∈ V
17:     qc.viewNumber ← m.viewNumber : m ∈ V
18:     qc.node ← m.node : m ∈ V
19:     qc.sig ← tcombine((qc.type, qc.viewNumber, qc.node),
                  {m.partialSig | m ∈ V})
20:     return qc
21: function matchingMsg(m, t, v)
22:     return (m.type = t) ∧ (m.viewNumber = v)
23: function matchingQC(qc, t, v)
24:     return (qc.type = t) ∧ (qc.viewNumber = v)
25: function safeNode(node, qc)
26:     return (node extends from lockedQC.node) ∨   // safety rule
27:     (qc.viewNumber > lockedQC.viewNumber)   // liveness rule
```

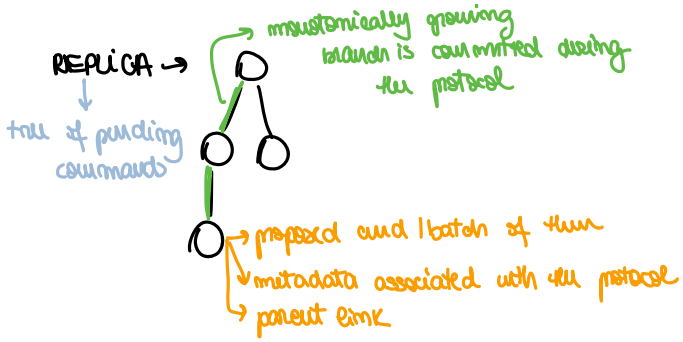
Algorithm 2 Basic HotStuff protocol (for replica r).

```
1:  for curView ← 1, 2, 3, ... do
    ▷ PREPARE phase
2:      as a leader  // r = leader(curView)
        // we assume special new-view messages from view 0
3:          wait for (n − f) new-view messages:
                M ← {m | matchingMsg(m, new-view, curView−1)}
4:          highQC ← (arg max_{m∈M} {m.justify.viewNumber}).justify
5:          curProposal ← createLeaf(highQC.node,
                                     client's command)
6:          broadcast Msg(prepare, curProposal, highQC)
7:      as a replica
8:          wait for message m from leader(curView)
                m : matchingMsg(m, prepare, curView)
9:          if m.node extends from m.justify.node ∧
                        safeNode(m.node, m.justify) then
10:             send voteMsg(prepare, m.node, ⊥) to leader(curView)

    ▷ PRE-COMMIT phase
11:     as a leader
12:         wait for (n − f) votes:
                V ← {v | matchingMsg(v, prepare, curView)}
13:         prepareQC ← QC(V)
14:         broadcast Msg(pre-commit, ⊥, prepareQC)
15:     as a replica
16:         wait for message m from leader(curView)
                m : matchingQC(m.justify, prepare, curView)
17:         prepareQC ← m.justify
18:         send to leader(curView)
                voteMsg(pre-commit, m.justify.node, ⊥)
    ▷ COMMIT phase
19:     as a leader
20:         wait for (n − f) votes:
                V ← {v | matchingMsg(v, pre-commit, curView)}
21:         precommitQC ← QC(V)
22:         broadcast Msg(commit, ⊥, precommitQC)
23:     as a replica
24:         wait for message m from leader(curView)
                m : matchingQC(m.justify, pre-commit, curView)
25:         lockedQC ← m.justify
26:         send to leader(curView)
                voteMsg(commit, m.justify.node, ⊥)
    ▷ DECIDE phase
27:     as a leader
28:         wait for (n − f) votes:
                V ← {v | matchingMsg(v, commit, curView)}
29:         commitQC ← QC(V)
30:         broadcast Msg(decide, ⊥, commitQC)
31:     as a replica
32:         wait for message m from leader(curView)
                m : matchingQC(m.justify, commit, curView)
33:         execute new commands through m.justify.node,
            respond to clients
    ▷ Finally
34:     nextView interrupt: goto this line if nextView(curView) is
        called during "wait for" in any phase
35:     send Msg(new-view, ⊥, prepareQC) to leader(curView + 1)
```

→ successive views with monotonically increasing numbers





HOTSTUFF 2→ 2 phases instead of 3