consensus - 2

leader election

- time-out failure detector,

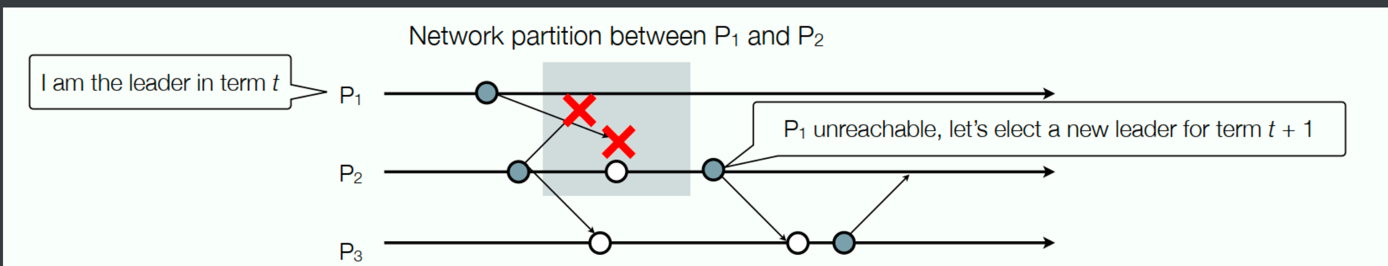
  on suspected leader crash, elect a new one
- prevent 2 leaders at the same time
- **term**: incremented once a leader election is started

  one node can only vote once per term

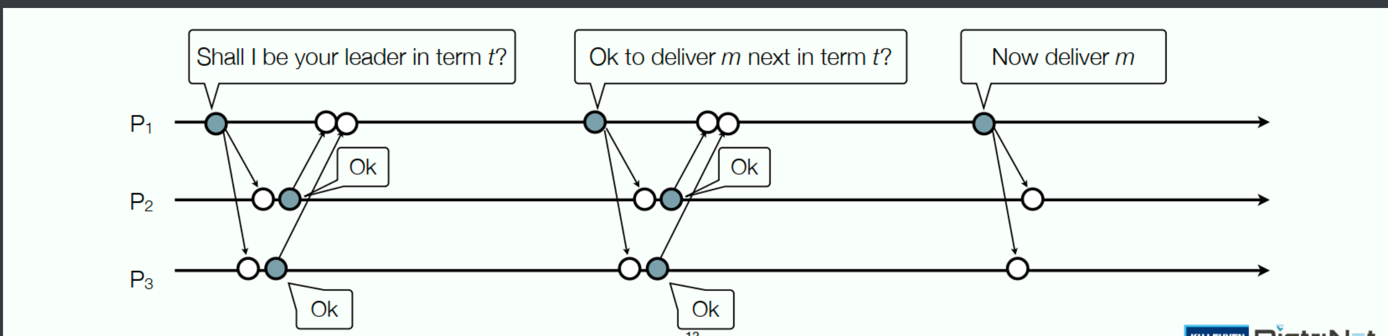
  require a quorum of nodes to elect a leader in a term

one-leader guarantee

如果leader失联，则elect一个新leader：



old leader无法越权，因为消息发送前需要获得一个quora统一，而old leader至慢也会在第二轮ack中得知自己不再是leader



consensus algorithms

- viewstamped replication
- paxos

agreement on a single value (multi-paxos is for a sequence of value)

Google Chubby

- raft

  specifically for log replication

  supports sequences of values

the Raft consensus algorithm

system model

- partially synchronous, crash-recovery

  uses clocks only for **liveness**
- processes may fail, network may fail
- **cooperative processes**, does not deal with malicious ("byzantine") processes

basic idea

- term is an integer counter
- **consistent logs**

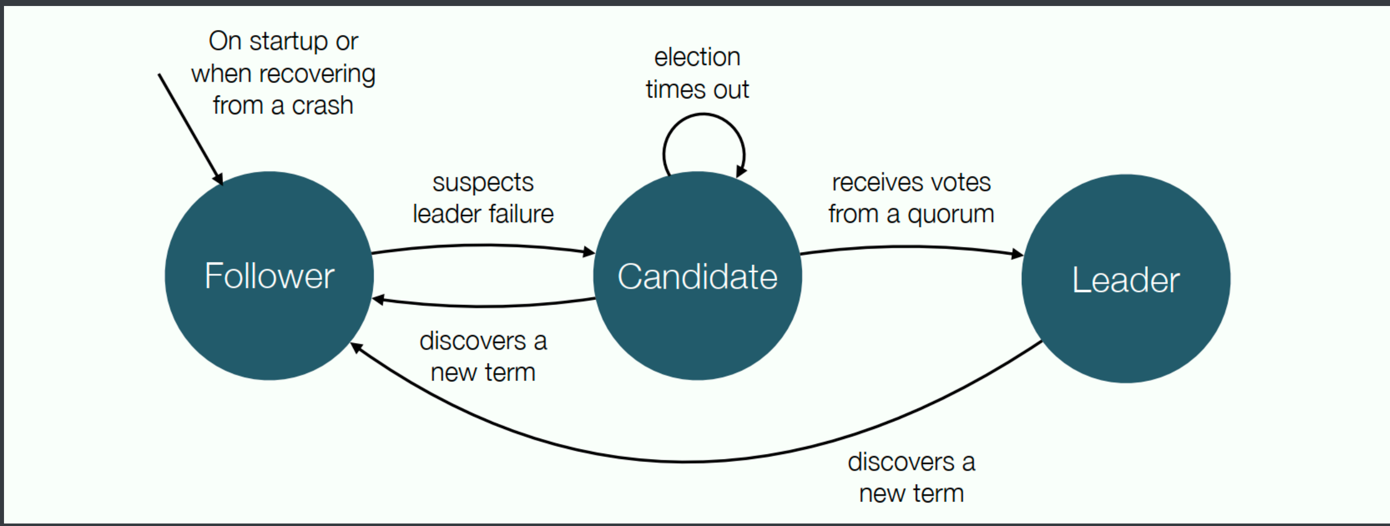
  one leader per term, all other nodes are followers

  ordered entries, messages + terms

  only the leader can append new entries to the log

  (*log is append-only !!! grow only at the end*)

algorithm



- initialisation

```
on initialisation do
    currentTerm := 0; votedFor := null
    log := ⟨⟩; commitLength := 0
    currentRole := follower; currentLeader := null
    votesReceived := {}; sentLength := ⟨⟩; ackedLength := ⟨⟩
end on

on recovery from crash do
    currentRole := follower; currentLeader := null
    votesReceived := {}; sentLength := ⟨⟩; ackedLength := ⟨⟩
end on
```

  - currentTerm, votedFor, log, commitLength are stored in persistent memory
  - currentRole, currentLeader, votesReceived, sentLength, ackedLength can be stored in volatile memory, which could be reset during crash-recovery

- starting a leader election

```
on node nodeId suspects leader has failed, or on election timeout do
    currentTerm := currentTerm + 1; currentRole := candidate
    votedFor := nodeId; votesReceived := {nodeId}; lastTerm := 0
    if log.length > 0 then lastTerm := log[log.length − 1].term; end if
    msg := (VoteRequest, nodeId, currentTerm, log.length, lastTerm)
    for each node ∈ nodes: send msg to node
    start election timer
end on
```

votes for itself,

sends a VoteRequest msg to each other node

sets a timer, if times out then steps are repeated

- voting on a new leader

```
on receiving (VoteRequest, cId, cTerm, cLogLength, cLogTerm)
          at node nodeId do
    if cTerm > currentTerm then
        currentTerm := cTerm; currentRole := follower
        votedFor := null
    end if
    lastTerm := 0
    if log.length > 0 then lastTerm := log[log.length − 1].term; end if
    logOk := (cLogTerm > lastTerm) ∨
           (cLogTerm = lastTerm ∧ cLogLength ≥ log.length)

    if cTerm = currentTerm ∧ logOk ∧ votedFor ∈ {cId, null} then
        votedFor := cId
        send (VoteResponse, nodeId, currentTerm, true) to node cId
    else
        send (VoteResponse, nodeId, currentTerm, false) to node cId
    end if
end on
```

- collecting votes

```
on receiving (VoteResponse, voterId, term, granted) at nodeId do
    if currentRole = candidate ∧ term = currentTerm ∧ granted then
        votesReceived := votesReceived ∪ {voterId}
        if |votesReceived| ≥ ⌈(|nodes| + 1)/2⌉ then
            currentRole := leader; currentLeader := nodeId
            cancel election timer
            for each follower ∈ nodes \ {nodeId} do
                sentLength[follower] := log.length
                ackedLength[follower] := 0
                ReplicateLog(nodeId, follower)
            end for
        end if
    else if term > currentTerm then
        currentTerm := term
        currentRole := follower
        votedFor := null
        cancel election timer
    end if
end on
```

- broadcasting messages

```
on request to broadcast msg at node nodeId do
    if currentRole = leader then
        append the record (msg : msg, term : currentTerm) to log
        ackedLength[nodeId] := log.length
        for each follower ∈ nodes \ {nodeId} do
            ReplicateLog(nodeId, follower)
        end for
    else
        forward the request to currentLeader via a FIFO link
    end if
end on

periodically at node nodeId do
    if currentRole = leader then
        for each follower ∈ nodes \ {nodeId} do
            ReplicateLog(nodeId, follower)
        end for
    end if
end do
```

- replicating from leader to followers

```
function ReplicateLog(leaderId, followerId)
    prefixLen := sentLength[followerId]
    suffix := ⟨log[prefixLen], log[prefixLen + 1], ...,
                log[log.length − 1]⟩
    prefixTerm := 0
    if prefixLen > 0 then
        prefixTerm := log[prefixLen − 1].term
    end if
    send (LogRequest, leaderId, currentTerm, prefixLen,
            prefixTerm, commitLength, suffix) to followerId
end function
```

- followers receiving messages from a leader

```
on receiving (LogRequest, leaderId, term, prefixLen, prefixTerm,
              leaderCommit, suffix) at node nodeId do
    if term > currentTerm then
        currentTerm := term; votedFor := null
        cancel election timer
    end if
    if term = currentTerm then
        currentRole := follower; currentLeader := leaderId
    end if
    logOk := (log.length ≥ prefixLen) ∧
             (prefixLen = 0 ∨ log[prefixLen − 1].term = prefixTerm)
    if term = currentTerm ∧ logOk then
        AppendEntries(prefixLen, leaderCommit, suffix)
        ack := prefixLen + suffix.length
        send (LogResponse, nodeId, currentTerm, ack, true) to leaderId
    else
        send (LogResponse, nodeId, currentTerm, 0, false) to leaderId
    end if
end on
```

- updating the followers' logs

```
function AppendEntries(prefixLen, leaderCommit, suffix)
    if suffix.length > 0 ∧ log.length > prefixLen then
        index := min(log.length, prefixLen + suffix.length) − 1
        if log[index].term ≠ suffix[index − prefixLen].term then
            log := ⟨log[0], log[1], ..., log[prefixLen − 1]⟩
        end if
    end if
    if prefixLen + suffix.length > log.length then
        for i := log.length − prefixLen to suffix.length − 1 do
            append suffix[i] to log
        end for
    end if
    if leaderCommit > commitLength then
        for i := commitLength to leaderCommit − 1 do
            deliver log[i].msg to the application
        end for
        commitLength := leaderCommit
    end if
end function
```

- leader receiving log acknowledgements

```
on receiving (LogResponse, follower, term, ack, success) at nodeId do
    if term = currentTerm ∧ currentRole = leader then
        if success = true ∧ ack ≥ ackedLength[follower] then
            sentLength[follower] := ack
            ackedLength[follower] := ack
            CommitLogEntries()
        else if sentLength[follower] > 0 then
            sentLength[follower] := sentLength[follower] − 1
            ReplicateLog(nodeId, follower)
        end if
    else if term > currentTerm then
        currentTerm := term
        currentRole := follower
        votedFor := null
        cancel election timer
    end if
end on
```

- leader committing log entries

```
define acks(length) = |{n ∈ nodes | ackedLength[n] ≥ length}|

function CommitLogEntries
    minAcks := ⌈(|nodes| + 1)/2⌉
    ready := {len ∈ {1, . . . , log.length} | acks(len) ≥ minAcks}
    if ready ≠ {} ∧ max(ready) > commitLength ∧
            log[max(ready) − 1].term = currentTerm then
        for i := commitLength to max(ready) − 1 do
            deliver log[i].msg to the application
        end for
        commitLength := max(ready)
    end if
end function
```

evaluating Raft

- guarantees safety (agreement & validity): consistent committed log entries

  <= 1 leader per term & msg from older terms are ignored

- cannot guarantee liveness (all log entries will eventually be commited) <- FLP impossibility, but in practice ✅

  timeouts for failed leader detection, randomness in election, quorum

Byzantine fault tolerance

- CFT (crash fault-tolerant) / BFT (byzantine fault-tolerant)

  CFT: processes implement the consensus algorithm correctly,

    CFT tolerates up to (not including) 1/2 failing processes

    BFT tolerates up to (not including) 1/3 failing processes

- "byzantine failure":

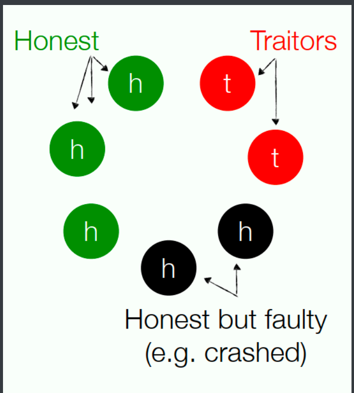
  totally arbitrary failures, including

  - Failing to respond to messages
  - Returning incorrect results from messages
  - Returning deliberately misleading results from messages
  - Returning a different result for the same request to different processes (!)

- in an **adversarial** context: attacks -> byzantine failures
- in **real-world deployments**: bugs -> byzantine failures

the byzantine generals problem

- N = 3f + 1



f honest but faulty processes, f traitors

then # available honest = N - 2f, need to outnumber the traitors

N - 2f > f <=> N > 3f

- $N = 3f + 1$ or $f = \text{floor}((N-1)/3)$

N	Honest quorum $f+1$	Faulty or traitors f
1	-	0
2	-	0
3	-	0
4	**2**	**1**
5	2	1
6	2	1
7	**3**	**2**
8	3	2

最少需要4 machines才能tolerate一个traitor

实际应用数量一般是7

（对比3和5，non-byzantine的结果）

- BFT consensus agorithms exist, **complexer** then CFT alg.

  <- digital signatures / cryptographic has functions,

  to ensure unforgeable & irrefutable

- **blockchains**: require byzantine consensus, **open and adversarial** environment, order of transactions

consensus vs. atomic commit protocols

atomic commit protocol: commit or abort (a transaction)?

Atomic Commit	Consensus
Every process votes whether to commit or abort	*Any* process may propose *any* value to agree on
Must commit if *all* processes vote to commit; must abort if *at least one* node votes to abort	Any one of the proposed values is agreed on (decided)
Must abort if *any* participating process crashes	Crashed processes can be tolerated, as long as a quorum (majority) has decided