Correctness of Tendermint-core Blockchains

Y. Amoussou-Guenou^,* A. Del Pozzo^, M. Potop-Butucaru*,
S. Tucci-Piergiovanni^*

^ Institut LIST, CEA, Université Paris-Saclay
* Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6
**BLOCKCHAIN**

- Potentially unbounded set of processes that communicate in a network through message passing

- **Distributed ledger**, ledger replicated by each process

- **Tamper-resistant**, by cryptographic mechanism

- Build in an **append only** manner

- A sequence of blocks, each block containing transactions

- Each block contains the hash of the prior block in the chain
• When adding a block in the Blockchain, others processes
  • Should be aware of it
  • Should add the block in their local copy of the blockchain

• The presence of such a structure can be harmful to the system, and the goal is to avoid it
OUTLINE
CONTRIBUTIONS

- The use of Consensus to build a Blockchain, e.g. Tendermint
- Formalization of Tendermint
- Conditions under which the protocol works
- Proofs of correctness of Tendermint
A process is correct if it follows the given protocol.

- **Termination**
  Every correct process eventually decides some value.

- **Integrity**
  No correct process decides twice.

- **Agreement**
  If there is a correct process that decides a value B, then eventually all the correct processes decide B.

- **Validity**
  A decided value is valid, it satisfies the predefined predicate.

WHAT IS TENDERMINT?

- Tendermint is a blockchain used in different applications.
- Tendermint is the first proposed blockchain to claim solving the Consensus, but has never been formalized.

HOW DOES IT WORK?

The blockchain network is composed of an unknown number $n$ of processes.

To append a new block, a committee of processes of fixed size $N$ is deterministically selected, and known by every process.

That committee runs a one-shot consensus protocol to decide on the next block.

The decision of the committee is sent to all processes, and is the next block to be appended.

The next committee rewards the previous one.
REPEATED CONSENSUS

• Every process produces a sequence of value/decision. We call that sequence the **output** of the process

• **Properties**:
  • **Termination**
    
    *Every correct process has an infinite output*
  
  • **Agreement**
    
    *For all k, the k\textsuperscript{th} value of any two correct processes is the same*
  
  • **Validity**
    
    *Each value in the output of any correct process is valid, it satisfies a predefined predicate*

---


The total number of processes by committee is $N = 3f+1$

- $f$ is the maximum number of Byzantine process

- The communication **is eventually synchronous**

- Messages are **signed** and signatures cannot be forged

- Broadcast
  - Gossip
  - Best effort broadcast

- **Finite** arrival model

---


When a process delivers a message, it broadcasts it.
if \( p_i == proposer(H, r) \) then
  if \( LLR_i \neq -1 \) then \( PoLCR_i = LLR_i \); \( B \leftarrow lockedBlock_i \);
  else \( B \leftarrow createNewBlock(signature) \);
  endif
  trigger broadcast \((PROPOSE, (B, H, r, PoLCR_i))\);
else
  set timerProposer to TimeOutPropose;
  wait until \(((timerProposer expired) \lor (proposalReceived^{H,r}(\_H, \_r) \neq \bot))\);
  if \(((timerProposer expired) \land (proposalReceived^{H,r}(\_H, \_r) = \bot))\) then
    TimeOutPropose \leftarrow TimeOutPropose + 1;
  endif
endif
• \( p_1 \) is not locked
• \( p_2 \) is not locked
• \( p_3 \) locks on B at round 1
• \( p_4 \) locks on C at round 2
if \((\exists B': (is23Ma(B', prevotesReceived_i^{H,r}))))\) then 
\[\text{lockedBlock}_i \leftarrow B';\]
\[\text{trigger}\ broadcast\ \langle\ \text{PRECOMMIT}, (B', H, r)\rangle;\]
\[\text{LLR}_i \leftarrow r;\]
else if \((is23Ma(nil, prevotesReceived_i^{H,r}))\) then 
\[\text{lockedBlock}_i \leftarrow nil; \text{LLR}_i \leftarrow -1;\]
\[\text{trigger}\ broadcast\ \langle\ \text{PRECOMMIT}, (nil, H, r)\rangle;\]
endif 
else trigger broadcast \(\langle\ \text{PRECOMMIT}, (nil, H, r)\rangle\); 
endif 
wait until \((is23Ma(nil, prevotesReceived_i^{H,r})\lor (|precommitsReceived_i^{H,r}| > 2/3))\)
EXAMPLE OF EXECUTION

\textbf{Propose} | \textbf{Prevote} | \textbf{Precommit}

\begin{align*}
\text{when} \ (\exists B' : \text{is23Maj}(B', \text{precommitsReceived}^H_{i, r'})) : \\
\text{return} B' ; \% \text{Terminate the consensus for the super-round } H \text{ by deciding } B' \%
\end{align*}
The live lock occurs because processes do not have the same view at the end of each round.

Remark: When $f > 1$, the byzantine processes need to coordinate to make such attack.
TENDERMINT SYSTEM MODEL

• The total number of processes by committee is $n = 3f+1$
  $f$ is the maximum number of Byzantine process

• The communication is eventually synchronous

• Messages are signed and signatures cannot be forged

• Additional assumption: Eventually $2f+1$ processes will lock on the same proposed value
**Termination:** Every correct process eventually decides some value

- During the **synchronous period**, there is a time from which messages from correct processes are delivered in their corresponding step.

- When a correct process $p_i$ is the proposer, it proposal will be prevoted by processes whose locks are smaller than $p_i$’s.
  - Eventually a proposed value will be accepted by at least $2f+1$ processes.
  - There will be $2f+1$ processes that will prevote.
  - Eventually correct processes will deliver them, then will precommit, and decide.
PROOFS SKETCH: AGREEMENT

**Agreement:** If there is a correct process that decides a value B, then eventually all the correct processes decide B.
REPEATED CONSENSUS

- **Termination**
- **Agreement**
- **Validity**

```plaintext
Function repeatedConsensus(Π); %Repeated Consensus for the set Π of processes%
Init :
    H ← 1 %Height%; B ← ⊥; V ← ⊥ %Set of validators%; TimeOutCommit ← ∆Commit;
    commitsReceived_H ← ∅; toReward_H ← ∅ %Set of processes to reward%
while (true) do
    B ← ⊥;
    V ← validatorSet(H); %Application and blockchain dependant%
    if (p_i ∈ V) then
        B ← consensus(H, V, toReward_H-1); %Hth Consensus instance%
        trigger broadcast ⟨COMMIT, (B, H)⟩;
    else
        wait until (∃B' : |atLeastOneThird(B', commitsReceived_H)|); %Committing
        B ← B';
    endif
    set timerCommit to TimeOutCommit;
    wait until(timerCommit expired);
    trigger decide(B);
    H ← H + 1;
endwhile

upon event delivery ⟨COMMIT, (B', H')⟩:
    if (((B', H') ∉ commitsReceived_H) ∧ (p_i ∈ validatorSet(H'))) then
        commitsReceived_H ← commitsReceived_H ∪ (B', H'));
        toReward_H ← toReward_H ∪ p_i;
        trigger broadcast ⟨COMMIT, (B', H')⟩;
    endif
```

Committee 1  Committee 2
Tendermint:

- Complexity of $O(n^3)$
- Each round has an $O(n^2)$ message complexity and there can be $O(n)$ rounds
- Intuitively, there is a View Change each round without sending the whole messages of a round, thanks to the lock mechanism
- The cost is that process may wait for $2f+1$ rounds before deciding
- Called the Linear View Change in [2]

Classical algorithms such as PBFT [1]:

- Complexity of $O(n^4)$
- Each round has an $O(n^2)$ message complexity, a View-Change has a cost of $O(n)$, and the $f = O(n)$ first rounds may be faulty


CONCLUSIONS

• Formulate the version of Tendermint implemented.
  • Helps identify some bugs
  • Leads to a proposition of a new version which aims to solve the consensus without the assumption

• Capture in which model Tendermint works

• Proof of correctness
• Lower bounds on rounds with the lock mechanism

• Incentives
  • Study of a fair reward mechanism
  • Study of a fair selection mechanism
  • Rational vs Byzantine

```plaintext
Function repeatedConsensus(Π): %Repeated Consensus for the set Π of processes%
Init:
  H ← 1; %Height%; B ← ⊥; V ← ⊥; %Set of validators%; TimeoutCommit ← ΔCommit;
  commitsReceived[1] ← 0; toReward[1] ← 0; %Set of processes to reward%
while (true) do
  B ← ⊥;
  V ← validatorSet(H); %Application and blockchain dependant%
  if (p ∈ V) then
    B ← consensus(H, V, toReward[1]); %Hth Consensus instance%
    trigger broadcast (COMMIT,(B,H));
  else
    wait until (∃ B': |AtLeastOneThird(B', commitsReceived[1])|)
    B ← B';
  endif
  set timerCommit to TimeoutCommit;
  wait until (timerCommit expired);
  trigger decide(B);
  H ← H + 1;
endwhile
```

Thank You!