Revisiting Tendermint: Design Tradeoffs, Accountability, and Practical Use

Ethan Buchman
Informal Systems
ethan@informal.systems

Rachid Guerraoui
École Polytechnique Fédérale de Lausanne
rachid.guerraoui@epfl.ch

Jovan Komatovic
École Polytechnique Fédérale de Lausanne
jovan.komatovic@epfl.ch

Zarko Milosevic
Informal Systems
zarko@informal.systems

Dragos-Adrian Seredinschi
Informal Systems
adi@informal.systems

Josef Widder
Informal Systems
josef@informal.systems

***Abstract*—Tendermint is a deterministic consensus protocol and is one of the most mature implementations of its kind. This implementation is used as the core for building State Machine Replication (SMR) platforms with Byzantine fault-tolerant (BFT) guarantees. A noteworthy deployment of Tendermint has been in continuous operation since 2019 within a blockchain called Cosmos Hub. The Cosmos Hub supports the development of decentralized applications, and stands as one of the largest and most stable ongoing deployments of a BFT SMR platform.**

While successful in practice, the Tendermint consensus protocol has no definitive description in the literature. It is not clear what makes this protocol unique or how it fits into a blockchain protocol stack. In this short paper, we revisit Tendermint. We contrast Tendermint with other major consensus algorithms, examining its unique design choices. We also focus on the requirements which dictated Tendermint's design. Lastly, we briefly analyze the accountability support which Tendermint provides.

I. Introduction

Current blockchain systems, such as Bitcoin [20] or Diem [24], are modern distributed systems that solve the *state machine replication* problem [18]. This problem corresponds to the theoretical concept of atomic broadcast [19]. Blockchains typically take the approach of solving atomic broadcast by doing iterated invocations to a fault-tolerant consensus algorithm, where each consensus instance decides on the next block, and each block commonly contains application-level transactions.

Some of the most prominent use-cases of blockchains consist of financial applications [4], [20]. Participants in the consensus protocol may therefore have financial incentives to deviate from protocol in an adversarial manner, e.g., by generating a diverging history of transactions in an attempt to send the same money to two distinct addresses (double-spending). The classic notion of a Byzantine fault [22] captures this kind of behavior abstractly. It is well-understood [22] that tolerating Byzantine faults requires certain resilience conditions, usually that less than a third of the processes can be faulty. The crucial problem of Byzantine fault-tolerant applications in environments that are ruled by financial incentives is how one can ensure that two thirds of the participants follow the protocol, that is, that they behave correctly, although they can benefit by deviating.

Proof-of-stake platforms address the incentive problem with a threat, informally speaking: Any financial gains that a participant may extract by misbehaving can result in a subsequent financial loss upon being caught. In the Tendermint protocol, for instance, the consensus participants are called *validators*, and they have to put money on the line, i.e., they *stake*. Once misbehavior is detected, proof of the misbehavior is submitted, and (a fraction of) tokens owned by the guilty validator(s) are burned.

Tendermint was among the first deterministic consensus protocols introduced specifically for use in a blockchain context [16]. It is for this reason that Tendermint includes an accountability mechanism [17] and relies inherently on a *gossip* building block – unlike most classic BFT SMR protocols that build on point-to-point links [3].

Discussions of Tendermint in the literature are scarce, yet the protocol adoption in practice is very successful. One of the most important deployments of Tendermint, for example, has been in continuous operation since 2019, executing on 125 nodes, within a blockchain called Cosmos Hub, and has a market cap of 8 *bil*. US$.[1] Motivated by its success, as well as by the gap in the literature, we revisit the Tendermint protocol in this paper. Our contributions are: (1) we provide a clear picture of the design tradeoffs that set Tendermint apart from alternative consensus algorithms; (2) we briefly discuss the accountability support of Tendermint; and (3) we clarify how Tendermint is used in practice.

Roadmap. We first provide an overview of Tendermint (§II), and then examine the algorithmic tradeoffs and uniqueness, as well as the accountability support of this protocol (§III). Next, we briefly discuss how Tendermint is used in practice (§IV). Finally, we conclude in §V.

II. Overview

As a consensus protocol, Tendermint draws inspiration from two landmark results in the field: the work of Dwork,

[1] https://coinmarketcap.com/currencies/cosmos/

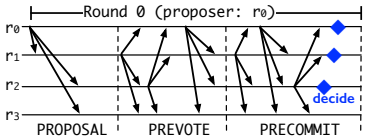

Fig. 1: Tendermint consensus protocol pattern. An important part of the story of Tendermint consensus algorithm is that there is no view-change sub-protocol or additional communication – *these three steps is all there is.* Additionally, protocol messages pass over a gossip overlay, not pairwise (point-to-point) links.

Lynch, and Stockmeyer (DLS) [12], and the PBFT protocol of Castro and Liskov [6]. Tendermint adopts from DLS the insight of rotating processes in the role of proposer as part of common-case processing. This enables the protocol to account for stakes and achieve more equity, or fairness. Proposer selection is a deterministic process in Tendermint. Essentially, there is a predefined function that ensures round-robin rotation of proposers, similar to earlier work on BFT consensus [6], [12], but with the important addition that processes are weighted in proportion to their stake. One potential drawback of this approach is the determinism—proposers are known well in advance, so an attacker can easily target them (e.g., through DDoS). Such an attack poses a significant threat; there is an ongoing effort to make proposer selection randomized in Tendermint, and a promising direction relies on verifiable random functions [15].

While DLS calls a single communication step a (communication-closed) round, in Tendermint a round consists of a sequence of *three* protocol steps having a certain process as the *proposer*. The three protocol steps in Tendermint are *proposal*, *prevote*, and *precommit*. Figure 1 provides an overview of the algorithm, describing a round (with process r_0 as proposer), and the three steps within that round leading up to a decision. Note that process r_3 is slow (or faulty), thereby not sending any message. This communication pattern resembles the well-known PBFT algorithm, which means that, under favorable timing conditions, Tendermint terminates in three message delays.

Akin to DLS and PBFT, Tendermint offers deterministic guarantees and assumes a partially synchronous network [12]. In contrast to Bitcoin, whose set of participants is unknown and open [20] (i.e., *permissionless*), Tendermint is a *permissioned* protocol, in so far as all of the participating processes know each other at the consensus layer (this mechanism is further discussed in §IV).

III. Design & Tradeoffs in Tendermint

In this section, we examine protocol tradeoffs that Tendermint exhibits, and analyze the uniqueness of this protocol. We focus on three aspects: communication layer (§III-A), termination mechanism and how this differs from other consensus protocols (§III-B), and accountability (§III-C).

A. Communication Layer

As already mentioned, a distinguishing feature of the Tendermint consensus algorithm is that processes communicate via a gossip-based communication layer. This stands in sharp contrast with the standard assumption in BFT protocols, namely, that processes exchange messages via pairwise links [7], [12]. Gossip has been explored in the context of consensus algorithms from a theoretical standpoint [8], and we also observe that more recent work on consensus systems is starting to rely on gossip [2]. However, to the best of our knowledge, Tendermint innovated as the first real-world BFT SMR system to build and deploy on a gossip-based communication layer.

Building a consensus algorithm on top of gossip has two significant consequences. First, a gossip layer is a higher-level primitive than pairwise links. This layer exposes a broadcast/deliver interface (instead of send/deliver), and implements a *reliable broadcast* abstraction [5].[2] Building on such an abstraction represents a stronger assumption; yet, this allows for a simpler logic at the higher layer (i.e., consensus). For instance, the consensus algorithm in Tendermint does not require a separate view-change (i.e., recovery) sub-protocol, being substantially easier to understand and more concise than similar algorithms. Second, gossip-based communication methods are well-known for their flexibility and scalability, which helps Tendermint accommodate growth and flexibility in the connectivity of its participating processes.

B. Termination Mechanism Tradeoffs

To understand precisely how Tendermint is innovative in its termination mechanism, we contrast this protocol with PBFT [7] and HotStuff [25]. We discuss the three protocols from the point of view of a single consensus instance.

We start by observing a commonality among these algorithms: they all incorporate a synchronous delay, or *timeout* T, towards overcoming non-graceful periods (e.g., due to malicious proposers or asynchrony). A synchrony assumption (i.e., timeout) is a classic way to circumvent the impossibility of progressing under asynchrony [13]. Intuitively, if T time elapses without progress, each protocol interprets this as a trigger to change the proposer; consequently, some process r_0 becomes the new proposer and processes transition from some round *i* into round *i+1*. We use the term *recovery phase* to denote the protocol steps having r_0 as a newly elected proposer; the critical difference between the protocols lies exactly in this phase.

The recovery phase in PBFT consists of the view-change sub-protocol, having a bit complexity of $O(n^3)$ [7]. Essentially, the new proposer r_0 has to collect messages from other processes to rebuild the state of round *i* and determine which value to propose in round *i+1*. Each of the messages received by r_0 contains a "proof of locking" of size $O(n)$ bits as it contains signatures from a quorum of processes. Moreover, the proposer r_0, along with the selected value, includes a vector of received proofs of locking, implying that the proposal message contains $O(n^2)$ bits. Since the proposal

[2]Note that the gossip layer does not ensure that all correct processes receive the identical message from a faulty sender, i.e., it does not ensure consistency.

message is broadcast to all processes, we reach the cubic bit complexity of the view-change sub-protocol of PBFT.

The recovery phase in HotStuff embodies the same pattern as PBFT, and calls this sequence of two steps a *prepare* phase [25, §4.1]. In HotStuff (as in PBFT), each round has a different proposer; thus, recovery happens at the beginning of every round (hence the name *prepare*). The upside is that HotStuff is optimized to change proposers regularly, and does so more efficiently than PBFT, since recovery has $O(n)$ bit complexity (due to the use of threshold signatures and the fact that a proof of locking for *only* the selected value is included in the leader's message) and two one-way message delays. The first downside is that HotStuff has latency of 7 message delays since it relies exclusively on all-to-leader and leader-to-all communication pattern in order to achieve linear bit complexity. Another drawback is that the cost of recovery is paid with *every* round. Importantly, both HotStuff and PBFT guarantee *responsiveness*: Once recovery begins, the new proposer r_0 of round *i+1* never waits on any timeout to guarantee progress (assuming that synchrony holds).

The recovery phase in Tendermint is entirely comprised of local computation, requiring no message exchanges. Once in round *i+1*, proposer r_0 can propose a value without coordinating with the others. This is because every process, during round *i*, stores in their local state the most up-to-date value that could be decided; doing so is a part of the common-case protocol. Upon becoming proposer in the succeeding round *i+1*, this is the value which process r_0 proposes.

To obtain this zero-complexity recovery, Tendermint builds on the following insight: To transition correctly between rounds *i* and *i+1* implies a timeout T, as we established earlier; this time window T allows processes to witness messages circulating in the system towards determining in round *i+1* the value to propose. Intuitively, the gossip layer guarantees that processes witness a common view of the system in round *i* while waiting to transition into *i+1*. This minimalist approach does not come for free: Tendermint sacrifices responsiveness, since waiting is on the critical path of r_0 before this process becomes a proposer and can effectively propose a value.

It is important to note that PBFT, HotStuff, and Tendermint all make identical synchrony assumptions. PBFT is perhaps the most optimized to extract performance from a stable proposer; HotStuff is primed to execute frequent recovery and to switch proposers by paying in modest complexity regularly; Tendermint optimizes for minimality of coordination, and has zero-complexity recovery.

C. Accountability in Tendermint

As already mentioned, Tendermint is a variant of the seminal algorithm by Dwork, Lynch and Stockmeyer [12]. It shares the property that, if less than a third of the processes are faulty, agreement is guaranteed. If there are more than two thirds of faulty processes, they have control over the system. The question we are interested in is whether in the area between, while we cannot prevent disagreement [9], we can ensure to collect evidence of misbehavior of (some) culprits if a disagreement occurs.

It has been shown in [14] that, if there are between one third and two thirds of faulty processes, every attack on Tendermint consensus that leads to violation of agreement is either the "double-vote" attack or the "amnesia" attack. The double-vote attack happens if a process sends two conflicting messages (e.g., one voting for block a and one voting for block a') in the same round of a consensus instance. The amnesia is a violation of the locking mechanism introduced in [12]: a process locks a value in a round if the value is supported by more than two thirds of all processes. A process that has locked a value can only be convinced to release that lock if more than two thirds of the processes have a lock formed in a later round. In the case of less than a third faults, if a process decides value v in a round r, the algorithm ensures that more than two thirds have a lock on value v for that round. As a result, once a value is decided, no other value w will be supported by enough processes. However, if there are more than a third faults, adversarial processes may lock a value v and in a later round "forget" about that and support a different value.

Currently, Tendermint does not punish amnesia. The main reason for this is that the only way to identify the misbehaving processes is via a query-response protocol where processes that have changed their opinion from v to w are requested to prove that they did that following the protocol, that is, they need to present more than two thirds messages that supported w after v was decided. If a process can do so, the two thirds of the processes that supported w now need to be queried whether they changed their mind in a sound way, etc. The major drawbacks of this solution are that it requires synchrony and it does not "produce" proof of misbehavior of faulty processes. Indeed, a faulty process that invalidly unlocks its value might simply avoid answering the query.

Being inspired by [23], we observe that a slight modification of Tendermint allows it to detect (and punish) amnesia attacks. Namely, the idea is to ensure that every conducted amnesia attack implies sending of conflicting messages, thus "transforming" amnesia into double-vote attacks. Such a transformation demands that every prevote message incorporates some information about the received proposal message that triggered it. In that way, every amnesia attack would require faulty processes to send invalid prevote messages, which are self-evident proofs of their misbehavior. Obtaining accountability guarantees for Tendermint demands a similar transformation to the one presented in [23] and applied to HotStuff [25]; due to the lack of space, we leave the formal treatment of the accountability-enabling transformation of Tendermint as future work.

IV. Practice Meets Theory

As a consensus engine, Tendermint is designed to be a general core for implementing BFT SMR. The state machine, i.e., application, communicates with the consensus engine over the *application blockchain interface* (ABCI). Tendermint

can thus be used with different applications. An application running on top of Tendermint needs to provide some control information to the consensus engine. For instance, it needs to provide the *validator set*, that is, the participants in the next consensus instance, which are identified by their public key. Moreover, the application assigns to each validator a *voting power*, which is a positive integer, representing the amount of stake this validator possesses.

In classic consensus algorithms, rules are typically guarded by threshold expression, e.g., if n is the number of processes, one needs to receive more than $2n/3$ messages. In other words, a quorum commonly consists of a certain number of processes. Tendermint replaces the number of processes by their cumulative voting power. If all validators have voting power 1, then we fall back to the classic case. Additionally, in each consensus instance, the validators of the next consensus instance are decided. Thus, the validator set can change completely between two consensus instances (i.e., blocks).

In Tendermint, the mapping from a round to its proposer is done in a weighted round-robin fashion. Specifically, during a sequence of n rounds, each validator is a proposer in S rounds, where S represents the voting power of the validator.

One instance of Tendermint running in production is the Cosmos main chain called *Cosmos Hub*. Here, the application contains a staking module. Validators can transfer and stake (self-delegate) Atoms (the native currency) and any user can delegate its Atoms to a validator. The sum of the delegated Atoms represents the voting power of a validator. In each block, the staking module (currently) picks the top 125 validators according to voting power to constitute the next validator set. Therefore, (in practice) the validator set changes over time, but, due to this rule, only mildly, while the top validators are pretty stable as the cost of becoming a validator in top-125 is substantial.

Besides validators, any deployment of Tendermint (including the Cosmos Hub) presumes other kinds of additional processes. Most importantly, every validator is protected by multiple *sentry* processes [10]. A sentry acts as a proxy between the validator and the rest of the network, hence the rest of the Internet, protecting the validator from inbound traffic or DDoS attacks. The sentry-based architecture ties in with the choice of gossip as a building block, since the gossip network can accommodate an arbitrary amount of peers (e.g., sentry processes, or other kinds of processes [11]), which can join the network and observe all messages, but cannot produce consensus protocol messages since they are not validators. This architecture has two consequences: (1) the gossip network itself is permissionless, so any process may join as a non-validator, observe the network, replay the protocol, and report misbehavior if they detect it, and (2) the overall network is substantially more extensive than just the set of validators. For a concrete example, observations from Cosmos Hub reveal that there are roughly 570 peers that are discoverable by their IP, while the network is estimated to count around 900 total peers [1], [21].

V. Conclusion

We have reviewed Tendermint, a Byzantine-fault tolerant consensus algorithm that is the core of the Cosmos ecosystem of blockchains. The algorithm is designed for the wide area network with high number of mutually distrusted processes that communicate over a gossip-based peer-to-peer network. We contrasted Tendermint with two other major BFT consensus algorithms and highlighted its unique design tradeoffs. Moreover, we discussed the accountability support of Tendermint, as well as how Tendermint is used in practice.

References

[1] Atlas, C., 2021. https://atlas.cosmos.network/nodes.

[2] Baird, L. Hashgraph consensus: fair, fast, byzantine fault tolerance. *Swirlds Tech Report, Tech. Rep.* (2016).

[3] Buchman, E., Kwon, J., and Milosevic, Z. The latest gossip on BFT consensus. *arXiv preprint arXiv:1807.04938* (2018).

[4] Buterin, V. Ethereum: A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2014. Accessed: 2018-07-11.

[5] Cachin, C., Guerraoui, R., and Rodrigues, L. *Introduction to Reliable and Secure Distributed Programming.* Springer Science & Business Media, 2011.

[6] Castro, M., and Liskov, B. Practical Byzantine Fault Tolerance and Proactive Recovery. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation* (Feb. 1999).

[7] Castro, M., and Liskov, B. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACMTCS* (2002).

[8] Chlebus, B. S., and Kowalski, D. R. Gossiping to Reach Consensus. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (New York, NY, USA, 2002), SPAA '02, ACM, pp. 220–229.

[9] Civit, P., Gilbert, S., and Gramoli, V. Polygraph: Accountable Byzantine Agreement. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)* (2021), IEEE, pp. 403–413.

[10] Docs, T. C., 2021. https://docs.tendermint.com/master/nodes/validators.html.

[11] Docs, T. C., 2021. https://docs.tendermint.com/master/nodes/#node-types.

[12] Dwork, C., Lynch, N., and Stockmeyer, L. Consensus in the Presence of Partial Synchrony. *JACM* (1988).

[13] Fischer, M. J., Lynch, N. A., and Paterson, M. S. Impossibility of Distributed Consensus with one Faulty Process. *JACM* (1985).

[14] Galois, I. Tendermint Ivy Proofs. https://github.com/tendermint/spec/tree/master/ivy-proofs.

[15] Kim, S. Randomized Leader Election in Tendermint using VRF. https://research.codechain.io/t/randomized-leader-election-using-vrf/17.

[16] Kwon, J. Tendermint: Consensus without Mining. *Draft v. 0.6, fall 1* (2014), 11.

[17] Kwon, J., and Buchman, E. Cosmos: A network of distributed ledgers. *Whitepaper* (2018), 1–41.

[18] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* (1978).

[19] Lynch, N. *Distributed Algorithms.* Morgan Kaufman, 1996.

[20] Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System, 2009.

[21] Operator, C. H. V. private communication, 2021.

[22] Pease, M., Shostak, R., and Lamport, L. Reaching Agreement in the Presence of Faults. *Journal of the ACM 27*, 2 (1980), 228–234.

[23] Sheng, P., Wang, G., Nayak, K., Kannan, S., and Viswanath, P. BFT Protocol Forensics. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (2021), pp. 1722–1743.

[24] Team, T. D. DiemBFT v4: State Machine Replication in the Diem Blockchain.

[25] Yin, M., Malkhi, D., Reiter, M. K., Gueta, G. G., and Abraham, I. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2019), PODC '19, ACM, pp. 347–356.