Mechanized safety of Jolteon consensus in Agda

Orestis Melkonian, Mauro Jaskelioff, and James Chapman

Input Output, Global (IOG)

Introduction. Consensus protocols for distributed systems ensure that all participants agree on some state, which is often realised as a common order of blocks on a chain. They are at the core of blockchain technology, where any mistake in the protocol design or implementation could result in huge economic losses. Therefore, it is of utmost importance to provide strong guarantees of its correctness.

At the same time, the area of consensus protocols is rapidly moving, as evidenced by the amount of new and improved protocols that have appeared in the last few years [4, 9, 6, 3, 1, 5, to cite a few]. Hence, it might not be possible to aim both for a complete formally proven implementation and stay at the forefront of technological development. A good compromise is to formally verify the design of the protocol, and use this formalization as an oracle for testing. The formalization becomes the ground truth, and all implementations should follow it. However, this approach entails an additional requirement: the formalization should be readable by engineers which might not be well versed in the formalization language or its abstractions.

We present a formalization of the Jolteon consensus protocol [6], a modern consensus protocol of the BFT (Byzantine Fault Tolerance) family [2], and we mechanize its proof of safety. Safety for a consensus protocol means that consistency is always maintained (*i.e.* there are no diverging chains). The formalization is written with readability in mind, and aims to stay close to the paper description. All of our results are mechanized in the Agda proof assistant [7].

A formal definition of the Jolteon protocol. We present a readable Agda specification of the Jolteon consensus protocol [6], with the aim of mechanizing its pen-and-paper **proofs** of important properties such as safety, as well as having a rigid ground truth against which we can **test** actual implementations.

First, we assume the usual cryptographic primitives (hash functions, signatures) and the BFT-specific setup of a fixed number `n` of replicas/participants which contains an *honest majority* [2]. Our further formal development is parameterised over these assumptions.

Each participant, honest or not, has an identifier `Pid = Fin n`.

The description of the protocol takes the form of a binary step relation that formally expresses valid transitions between states of the **global** system. This global level mostly deals with generic scaffolding that every BFT consensus protocol would have to provide, such as the message-passing `Deliver` rule that enables communication between participants, the `WaitUntil` rule that advances time (as long as it does not break the assumption of the underlying network model of message delays having an upper bound Δ), and the `DishonestStep` rule which gives dishonest participants freedom to send any message as long as it doesn't involve forging signatures.

```
data _—→_ (s : GlobalState) : GlobalState → Type where
```

```
Deliver : ∀ {tm} →
  tm ∈ s .networkBuffer
  ──────────────────────
  s —→ deliverMsg s tm

DishonestStep : ∀ m →
  • NoSignatureForging m s
  ──────────────────────
  s —→ broadcast m s

WaitUntil : ∀ t →
  • All (λ (t' , _) → t ≤ t' + Δ) (s .networkBuffer)
  • currentTime s < t
  ──────────────────────────────
  s —→ record s { currentTime = t }
```

...

Apart from handling the message-passing aspect of the network, the global state also keeps track of each honest replica's **local** state, which contains all protocol-specific information such as the current round, the most recent certificate the replica has seen, etc. The behaviour of honest replicas is modelled with another step relation where the specific rules of the protocol manifest, such as how each epoch's honest leader *proposes* a block, or under which conditions a replica considers a chain to be *final*:

```
data _⦂_⊢_–_–→_ (p : Pid) (t : Time) (ls : LocalState) : Maybe Message → LocalState → Type where
  ProposeBlock : ∀ txs →
    let L = roundLeader (ls .r-cur)
        b = mkBlockForState ls txs
        m = Propose (sign L b)
    in
    • p ≡ L
    ──────────────────────────
    p ⦂ t ⊢ ls – just m —→ ls

  Commit : ∀ b b' ch →
    • b  -certified-∈- ls .db
    • b' -certified-∈- ls .db
    • (b' :: b :: ch) •∈ ls .db
    • length ch > length (ls .final)
    • b' .round ≡ 1 + b .round
    ──────────────────────────────────────────
    p ⦂ t ⊢ ls – nothing —→ record ls {final = b :: ch}
```

...

Proving safety. Taking the reflexive-transitive closure of the global step relation —→ above leads to a notion of execution trace for such a protocol. That way, we can prove state invariants that hold for every *reachable* state in those traces, that is all states for which there is a valid sequence of steps from the initial state.

Once we have proven several state invariants, *e.g.* expressing a certain connection between different pieces of the state, we were able to faithfully transcribe the original paper proof of `safety` (otherwise known as *consistency*) in Agda. Safety states that two honest participants never have diverging chains that they both consider final:

```
safety : ∀ s b p b' p' →
  • Reachable s
  • b  ∈ (s @ p)  .final
  • b' ∈ (s @ p') .final
  ──────────────────────
  (b ←–* b') ⊎ (b' ←–* b)
```

Testing. We further prove several decidability results in Agda about all the logical constructions we used thus far to define the protocol, such as the chain finalization conditions.

The purpose of these proofs is two-fold. First, they allow us to construct and type-check example traces without manually providing proofs for each rule hypothesis, since our model is *computable* and therefore one can leverage *proof-by-computation* [8] to automate proofs on closed examples that contain no variables.

Moreover, once extracted/compiled to a target general-purpose language, these proofs of decidability become *decision procedures* that can be utilised by a **conformance testing** pipeline that aims to ensure that a given replica's protocol implementation conforms to the formal specification.

Future work. We plan to proceed with also proving **liveness**, a crucial property to ensure the protocol makes progress in a timely fashion. Apart from having to model some additional aspects that relate to the notion of *time*, we expect the methodology we previously employed for safety to also extend adequately to liveness.

Finally, we plan to investigate different testing approaches (*e.g.* testing a local honest replica against the local step relation *versus* testing the whole testing environment against the global step relation), as well as variants of the protocol that have more optimal characteristics which we would prove "equivalent" to the original protocol (*i.e.* the original properties of safety and liveness still hold in the optimized protocol).

References

[1] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. 10 2017. doi:10.48550/arXiv.1710.09437.

[2] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In Margo I. Seltzer and Paul J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186. USENIX Association, 1999. URL: https://dl.acm.org/citation.cfm?id=296824.

[3] Benjamin Y. Chan and Rafael Pass. Simplex consensus: A simple and fast consensus protocol. In Guy N. Rothblum and Hoeteck Wee, editors, *Theory of Cryptography - 21st International Conference, TCC 2023, Taipei, Taiwan, November 29 - December 2, 2023, Proceedings, Part IV*, volume 14372 of *Lecture Notes in Computer Science*, pages 452–479. Springer, 2023. doi:10.1007/978-3-031-48624-1_17.

[4] T-H. Hubert Chan, Rafael Pass, and Elaine Shi. Pala: A simple partially synchronous blockchain. *IACR Cryptol. ePrint Arch.*, 2018:981, 2018. URL: https://api.semanticscholar.org/CorpusID:53238268.

[5] Isaac Doidge, Raghavendra Ramesh, Nibesh Shrestha, and Joshua Tobkin. Moonshot: Optimizing chain-based rotating leader bft via optimistic proposals, 2024. URL: https://arxiv.org/abs/2401.01791, arXiv:2401.01791.

[6] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In Ittay Eyal and Juan A. Garay, editors, *Financial Cryptography and Data Security - 26th International Conference, FC 2022, Grenada, May 2-6, 2022, Revised Selected Papers*, volume 13411 of *Lecture Notes in Computer Science*, pages 296–315. Springer, 2022. doi:10.1007/978-3-031-18283-9_14.

[7] Ulf Norell. Dependently typed programming in Agda. In *International School on Advanced Functional Programming*, pages 230–266. Springer, 2008.

[8] Paul Van Der Walt and Wouter Swierstra. Engineering proof by reflection in Agda. In *Symposium on Implementation and Application of Functional Languages*, pages 157–173. Springer, 2012.

[9] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 347–356, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331591.