Byzantine Consensus Under Dynamic Participation with a Well-Behaved Majority

Eli Gafni
University of California, Los Angeles, USA

Giuliano Losa
Stellar Development Foundation, USA

Abstract

In a permissionless system like Ethereum, participation may fluctuate dynamically as some participants unpredictably go offline and some others come back online. In such an environment, traditional Byzantine fault-tolerant consensus algorithms may stall – even in the absence of failures – because they rely on the availability of fixed-sized quorums.

The sleepy model formally captures the main requirements for solving consensus under dynamic participation, and several algorithms solve consensus with probabilistic safety in this model assuming that, at any time, more than half of the online participants are well behaved. However, whether safety can be ensured deterministically under these assumptions, especially with constant latency, remained an open question.

Assuming a constant adversary, we answer in the positive by presenting a consensus algorithm that achieves deterministic safety and constant latency in expectation. In the full version of this paper, we also present a second algorithm which obtains both deterministic safety and liveness, but is likely only of theoretical interest because of its high round and message complexity. Both algorithms are striking in their simplicity.

2012 ACM Subject Classification Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Consensus, Sleepy Model, Dynamic Participation, Byzantine Failures

Related Version *Full Version*: `https://arxiv.org/abs/2301.04817`

Supplementary Material *Formal specifications*: `https://github.com/nano-o/dynamic-participation-supplemental`

1 Introduction

In a permissionless system like Ethereum, the parties running the consensus algorithm are known and fixed (up to reconfigurations) but they may unpredictably go offline or come back online. We say that participation is dynamic.

Ideally, we would like to solve consensus even if the set of online participants fluctuates unpredictably, as long as a sufficient fraction of those who are online are well-behaved. BFT consensus algorithms like PBFT [?], Algorand Agreement [?] or Tendermint [?] do not work in this setting because they rely on the availability of fixed-size quorums to make progress, and thus they stall – even in the absence of failures – if too many participants go offline.

In this paper, we consider the consensus problem in a synchronous system with dynamic participation similar to the sleepy model [?]. We assume that a fixed set of processes execute a sequence of rounds where, each round, an adversary partitions the processes into three sets: offline processes, online and faulty (i.e. controlled by the adversary) processes, and online and well-behaved processes. Communication is synchronous, which means that a message sent in round r is guaranteed to be received by all processes that are online in round $r + 1$. Importantly, in each round, the processes do not know a priori who is online and who is not, nor who is well behaved and who is faulty.

Several algorithms solve consensus in variants of this setting [?, ?, ?, ?, ?, ?, ?, ?]. However, one question that remained open is whether we can solve consensus under dynamic participation with deterministic safety and constant latency in expectation, assuming that: a) each round, a strict majority of the online participants are well behaved, b) a PKI and verifiable random functions [?] (VRF) are available, and c) the faulty set is constant from one round to the next. In this paper, we present such a consensus algorithm.

Solving consensus under those constraints is not easy. On the one hand, we could imagine using a variant of the Dolev-Strong algorithm [?]. However, this algorithm takes a number of rounds linear in the number of failures, while we aim at constant latency. On the other hand, it seems that techniques based on intersecting quorums will not work: In a given round, it is possible that an online, well-behaved process p receives a message m from a strict majority of the processes it hears of, while another online, well-behaved process p' receives a message $m' \neq m$ from a strict majority of the processes it hears of.

We start by observing that we can prevent faulty processes from equivocating and witholding messages to some processes, using a simple two-round algorithm. This allows us to simulate what we call the no-equivocation model, in which it is no longer possible for two well-behaved processes to hear of conflicting strict majorities. Next, taking inspiration from Gafni and Zikas [?], we solve the commit-adopt problem [?] (a graded agreement [?] with two grades) deterministically and in exactly 2 rounds of the no-equivocation model (for a total of 4 rounds of the base model).

Finally, using commit-adopt, we propose a consensus algorithm reminiscent of the phase-king algorithm [?, ?], where we use a probabilistically-elected leader (which can be done using VRFs) instead of selecting a king round-robin. The algorithm ensures safety deterministically, terminates in 18 rounds in expectation, and, in an execution with at most m online participants, each process broadcasts at most m messages each round. Whether the termination bound and message complexity can be improved is left for future work.

In the full version of this paper, we generalize the model to a setting where the set of processes has unknown cardinality and failures are governed by a more powerful mobile message adversary. We then show with detailed proofs that the consensus algorithm described in the present paper works in this model, and we present another consensus algorithm, inspired by the Dolev-Strong algorithm [**?**], that achieves both deterministic safety and liveness, although with a linear latency in the number of failures (even if the set of processes is unknown) as long as there is a strict subset of the processes that contains all the faulty sets.

2 The model

We consider a finite, nonempty set P of processes running an algorithm in a synchronous, message-passing system with point-to-point links between every pair of processes. An unknown subset $F \subset P$ of the processes is faulty.

The set P is publicly known and every process p has a key pair K_p whose public component is also known to all (this is an abstraction of a PKI). We write $K_p(m)$ for the message m signed with the private component of K_p.

The system executes an infinite sequence of rounds numbered $1, 2, 3, \ldots$. Each round r, an adversary partitions the processes into a nonempty set O_r of online processes and a set $P \setminus O_r$ of offline processes such that a) all the faulty processes are online (i.e. $F \subseteq O_r$) and b) the faulty processes consist of a strict minority of the online processes (i.e. $2|F| < |O_r|$). The sets O_r and F are a priori unknown to the processes.

Operationally, execution proceeds as follows. Initially, each process receives an external input. Then, each round r consists of a send phase followed by a receive phase. In the send phase, each online, well-behaved process sends a set of messages that, if $r = 1$, is a function of its input, and that otherwise is a function of the set of messages it received in the preceding round and of the output of a leader-election oracle described below. In both cases, the algorithm determines the function. Faulty processes are controlled by the adversary and may deviate by sending arbitrary messages, except that they cannot forge signatures. In the receive phase, each process, online or not, receives all the messages sent to it in the round[1].

The leader-election oracle gives an output to each online process at the beginning of every round $r > 1$, and it ensures with probability $1/2$ that every process that is online and well-behaved in round r receives the identity of a unique process that is online and well-behaved in round $r - 1$. In practice, the oracle can be implemented using VRFs by selecting the process that has the highest VRF output.

Note that, as mentioned in the introduction, a difficulty in this model is that two well-behaved processes may witness two conflicting strict majorities in the same round. For example, take 5 processes p_1 to p_5 and consider a round r where all processes participate and only p_4 and p_5 are faulty. Let p_1 and p_2 broadcast message m while p_3 broadcasts message $m' \neq m$. Moreover, let p_4 and p_5 send message m' to p_1 while they do not send any messages to p_2. Observe that p_1 hears of 5 processes and receives m' from 3 processes, and so it witnesses a strict majority for m'. However, p_2 hears of 3 processes and receives m from 2 processes, so it witnesses a strict majority for m.

3 Implementing commit-adopt

The most important building block of the consensus algorithm we present in the next section is a solution to the commit-adopt problem. In the commit-adopt problem, each online process initially receives an arbitrary input. After a fixed number of rounds R, each online process

[1] Assuming that all processes receive messages is convenient, and it does not make things easier since only the processes that are online in round $r + 1$ can use the messages they received in round r.

must produce an output either of the form $\langle \text{commit}, v\rangle$ for some v, in which case we say the process commits v, or of the form $\langle \text{adopt}, v\rangle$ for some v, in which case we say that the process adopts v. The outputs must satisfy the following properties:

Agreement If a well-behaved process commits a value v, then every process that is online and well-behaved in round R must either commit or adopt v.

Validity If all well-behaved processes that initially receive an input receive the same value v as input, then all well-behaved processes that are online in round R commit v.

Next, we present an algorithm that implements commit-adopt in $R = 4$ rounds. We first simulate a model that we call the no-equivocation model, and then implement commit-adopt in the no-equivocation model.

3.1 Simulating the no-equivocation model

The no-equivocation model is similar to the base model of the preceding section except that, each round r, each online process broadcasts a single message, and faulty processes may deviate only by omitting to send their message to some processes (they cannot equivocate, i.e. send different messages to different processes). Moreover, when a faulty process q deviates in round r, then the online, well-behaved processes of the next round $r + 1$ get a failure notification for q, noted λ, and such that one of the following three cases hold in round $r + 1$:

1. All online, well-behaved processes receive the same message m from q, or
2. Some online, well-behaved processes receive the same message m from q while all the others receive the failure notification λ for q, or
3. Some online, well-behaved processes receive the failure notification λ for q while all the others do not hear of q.

Note that the no-equivocation model does not allow for two different online, well-behaved processes to witness two conflicting strict majorities of messages in the same round. Moreover, the failure notification λ limits the ability of faulty processes to make other processes witness different levels of participation.

We now simulate each no-equivocation round in two rounds of the base model:

1. In the first round, each process p must broadcast the tuple $\langle m, K_p(m)\rangle$, where m is the message to simulate the broadcasting of.
2. In the second round, each process p must broadcast $\langle \text{heard-of}, m, K_p(m)\rangle$ for each message $\langle m, K_p(m)\rangle$ that it received in the first round. Finally, at the end of the second round, for each process q such that p receives $\langle \text{heard-of}, m_q, K_q(*)\rangle$ for some message $*$:
   a. If there is a message m' such that p receives $\langle \text{heard-of}, m', K_q(m')\rangle$ from a strict majority of the processes it hears of and p does not receive $\langle \text{heard-of}, m'', K_q(m'')\rangle$ for any $m'' \neq m'$, then p must simulate receiving m' for q.
   b. Otherwise, p must simulate receiving the failure notification λ for q.

Essentially, processes broadcast their message and then tell each other what messages they received in order to detect equivocations or missing messages. An excerpt from the formal specification of the simulation algorithm, written in PlusCal/TLA+ [?], appears in ??. The full specification appears in the supplemental material [?].

Note that the simulated received messages satisfy the minority-failure requirement because the set of faulty processes remains constant.

3.2 Implementing commit-adopt in the no-equivocation model

We now implement commit-adopt in 2 rounds of the no-equivocation model (which amounts to 4 rounds of the base model) as follows.

1. In the first no-equivocation round, each process must broadcast its input in_p.
2. In the second no-equivocation round, each process must propose to commit v, by broadcasting v, where v is the value received from a strict majority of the processes it hears of, if any, and otherwise it must broadcast $\langle\text{no-commit}\rangle$.
3. Finally, at the end of the second no-equivocation round, for each process p:
   a. If p receives v from a strict majority of the processes it hears of, then it must commit v.
   b. Else, if there is a value v such that p receives v more often than it receives v' for any other value v', then p must adopt v.
   c. Else, p can adopt an arbitrary value.

An excerpt from the formal specification of the commit-adopt algorithm, written in PlusCal/TLA+ [?], appears in ??. The full specification appears in the supplemental material [?].

Let us sketch why the commit-adopt algorithm satisfies the agreement property. Assume that a well-behaved process p commits a value v. Note that it suffices to show that, for every $v' \neq v$, no well-behaved process p' receives $\langle\text{propose-commit}, v'\rangle$ more often than $\langle\text{propose-commit}, v\rangle$. For every process q and value w, let $\text{count}(q, w)$ be the number of times q receives the message $\langle\text{propose-commit}, w\rangle$. With this notation, what we would like to show is that $\text{count}(p', v) - \text{count}(p', v') > 0$.

Note that, since processes may not witness conflicting strict majorities in the no-equivocation model, no well-behaved process broadcasts $\langle\text{propose-commit}, v'\rangle$ for $v' \neq v$. Additionally, remember that a faulty process q cannot send different messages to different processes and that, should it selectively send a message to only some processes but not others, the others receive the failure notification λ for q. From this, we get that $\text{count}(p', v) - \text{count}(p', v')$ is equal to $\text{count}(p, v)$ minus the number of processes q such that either a) p receives $\langle\text{propose-commit}, v\rangle$ from q and p' receives the failure notification λ for q, or b) p' receives $\langle\text{propose-commit}, v'\rangle$ from q and p did not receive $\langle\text{propose-commit}, v\rangle$ from q. In both cases, q must be a faulty process. Thus, we have $\text{count}(p', v) - \text{count}(p', v') \geq \text{count}(p, v) - |F_2|$, where F_2 is the set of faulty processes in the second no-equivocation round.

Finally, by our assumption that p commits v, we have $\text{count}(p, v) > |F_2|$.[2] We conclude that $\text{count}(p', v) - \text{count}(p', v') > 0$, i.e. p' receives $\langle\text{propose-commit}, v\rangle$ more often than $\langle\text{propose-commit}, v'\rangle$.

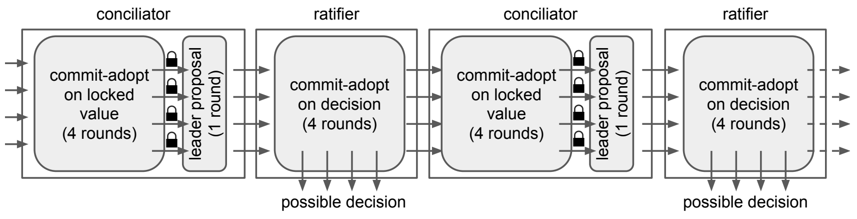
4 Consensus with deterministic safety and constant expected latency

In the consensus problem, each process that is initially online receives an input taken from a set of values V, and each process may produce an output called a decision subject to the following requirements:

Agreement	No two well-behaved processes produce different decisions.
Validity	If all processes receive the same input v, the no well-behaved process decides $v' \neq v$.
Liveness	With nonzero probability, in some round, all online, well-behaved processes decide.

[2] This is only true if all faulty processes make themselves heard. A complete proof appears in the full version of this paper.

Figure 1 Infinite alternating sequence of conciliators and ratifiers. Horizontal arrows represent processes locally taking their output from one block and using it as input to the next block, while vertical arrows represent processes possibly outputing a consensus decision.


conciliator
ratifier
conciliator
ratifier
commit-adopt on locked value (4 rounds)
leader proposal (1 round)
commit-adopt on decision (4 rounds)
commit-adopt on locked value (4 rounds)
leader proposal (1 round)
commit-adopt on decision (4 rounds)
possible decision
possible decision

To solve consensus, we use the construction shown in **??**. It consists of an infinite alternating sequence of conciliators and ratifiers (following the terminology of Aspnes [**?**]), starting with a conciliator. Informally, a ratifier tries to produce a consensus decision using commit-adopt, but since processes are not guaranteed to commit, it may fail to do so. Thus, the job of a conciliator is to try to make processes agree on their input to the next ratifier. We do this using a leader. However, to ensure that no leader overrides a previous decision, processes first negotiate, again using commit-adopt, on whether to listen to the leader or not.

More precisely, a ratifier simply consists of an instance of the commit-adopt algorithm in which processes try to commit a consensus decision. Each process p inputs $\langle \text{decide}, v_p \rangle$ to the commit-adopt, where v_p is p's output in the preceding conciliator, and each process that commits $\langle \text{decide}, v \rangle$ for some v decides v as a consensus decision.

In a conciliator, each process p first inputs $\langle \text{lock}, v_p \rangle$ into a commit-adopt instance to try to lock the value v_p that it gets as output in the preceding ratifier. Then follows an additional round, called the leader-proposal round, in which each process p broadcasts its commit-adopt output. At the end of the leader-proposal round, for each process p that receives $\langle \text{commit}, \langle \text{lock}, v \rangle \rangle$ for some v from a strict majority of the processes it hears of, p considers v locked and outputs v. Otherwise, p outputs the value v received from the process identified as leader by the leader-election oracle, if any, and else outputs an arbitrary value.

The lock mechanism ensures that a value unanimously supported by online, well-behaved processes at the beginning of the conciliator cannot be overridden during the leader-proposal round. Thus, once a value is first decided, all online, well-behaved processes keep unanimously supporting that value in all subsequent rounds and the agreement property of consensus is guaranteed. The validity property holds for similar reasons.

Finally, for liveness, note that if the oracle outputs the same online, well-behaved leader to all, then all online, well-behaved processes output the same value from the conciliator (because if an online, well-behaved process considers a value locked, then the leader must have proposed that value). Since this happens with probability $1/2$, the liveness property holds. Moreover, given that a ratifier takes 4 rounds and a conciliator takes 5 rounds, it takes at best 9 rounds to decide and 18 rounds in expectation.

5 Related Work

Starting with Bitcoin's longest-chain protocol, a series of works solve consensus under dynamic-participation with probabilitic safety [**?**, **?**, **?**, **?**, **?**].

In this paper, we are interested in solving consensus with deterministic safety. Sandglass [**?**] achieves deterministic safety under a minority of benign failures even when the faulty set can grow from round to round. The algorithm we present also works in the Sandglass model (without Byzantine failures, we do need to assume a constant adversary). Gorilla [?] extends Sandglass to Byzantine failures using verifiable delay functions (VDF) [?]. We conjecture that, using VDFs to curb equivocation, the algorithm we present can be ported to the Gorilla model and achieve constant latency.

Momose and Ren [?] tolerate a minority of Byzantine failures under an eventual-stabilization assumption. Malkhi, Momose, and Ren [?] remove the stabilization assumption but tolerate only one-third failures. We improve on this result by tolerating a minority (i.e. less than 1/2) of failures under a constant adversary, albeit assuming a constant adversary.

Finally, we have used a simple model in order to focus on essential algorithmic issues. See Lewis-Pye and Roughgarden [?] for more holistic models of permissionless systems.

References

1 James Aspnes. A modular approach to shared-memory consensus, with applications to the probabilistic-write model. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, 2010. doi:10.1145/1835698.1835802.

2 Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros Genesis: Composable Proof-of-Stake Blockchains with Dynamic Availability. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018. doi:10.1145/3243734.3243848.

3 P. Berman, J. A. Garay, and K. J. Perry. Towards optimal distributed consensus. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, 1989. doi:10.1109/SFCS.1989.63511.

4 Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Bit Optimal Distributed Consensus. In *Computer Science: Research and Applications*. Springer US, 1992. doi:10.1007/978-1-4615-3422-8_27.

5 Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable Delay Functions. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, Lecture Notes in Computer Science, Cham. Springer International Publishing. doi:10.1007/978-3-319-96884-1_25.

6 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus, 2019. doi:10.48550/arXiv.1807.04938.

7 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20, 2002. doi:10.1145/571637.571640.

8 Jing Chen, Sergey Gorbunov, Silvio Micali, and Georgios Vlachos. ALGORAND AGREEMENT: Super Fast and Partition Resilient Byzantine Agreement, 2018. URL: https://eprint.iacr.org/2018/377.

9 Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science*, 777, 2019. doi:10.1016/j.tcs.2019.02.001.

10 Phil Daian, Rafael Pass, and Elaine Shi. Snow White: Robustly Reconfigurable Consensus and Applications to Provably Secure Proof of Stake. In *Financial Cryptography and Data Security*, 2019. doi:10.1007/978-3-030-32101-7_2.

11 Francesco D'Amato, Joachim Neu, Ertem Nusret Tas, and David Tse. No More Attacks on Proof-of-Stake Ethereum? *arXiv preprint arXiv:2209.03255*, 2022.

12 D. Dolev and H. R. Strong. Authenticated Algorithms for Byzantine Agreement. *SIAM Journal on Computing*, 12, 1983. doi:10.1137/0212045.

13 Eli Gafni. Round-by-round Fault Detectors (Extended Abstract): Unifying Synchrony and Asynchrony. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, 1998. doi:10.1145/277697.277724.

14 Eli Gafni and Vasileios Zikas. Synchrony/Asynchrony vs. Stationary/Mobile? The Latter is Superior...in Theory, 2023. doi:10.48550/arXiv.2302.05520.

15 Vipul Goyal, Hanjun Li, and Justin Raizes. Instant Block Confirmation in the Sleepy Model. In *Financial Cryptography and Data Security*, 2021. doi:10.1007/978-3-662-64331-0_4.

16 Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002.

17 Andrew Lewis-Pye and Tim Roughgarden. Permissionless Consensus, 2023. doi:10.48550/arXiv.2304.14701.

18 Giuliano Losa. Supplemental material. 2023. doi:10.5281/zenodo.8226250.

19 Dahlia Malkhi, Atsuki Momose, and Ling Ren. Byzantine Consensus under Fully Fluctuating Participation, 2022. URL: https://eprint.iacr.org/2022/1448.

20 S. Micali, M. Rabin, and S. Vadhan. Verifiable random functions. In *40th Annual Symposium on Foundations of Computer Science*, 1999. doi:10.1109/SFFCS.1999.814584.

21 Atsuki Momose and Ling Ren. Constant Latency in Sleepy Consensus. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022. doi:10.1145/3548606.3559347.

22 Rafael Pass and Elaine Shi. The Sleepy Model of Consensus. In *Advances in Cryptology – ASIACRYPT 2017*, 2017. doi:10.1007/978-3-319-70697-9_14.

23 Youer Pu, Lorenzo Alvisi, and Ittay Eyal. Safe Permissionless Consensus. In *36th International Symposium on Distributed Computing (DISC 2022)*, volume 246, 2022. doi:10.4230/LIPIcs.DISC.2022.33.

24 Youer Pu, Ali Farahbakhsh, Lorenzo Alvisi, and Ittay Eyal. Gorilla: Safe Permissionless Byzantine Consensus, 2023. doi:10.48550/arXiv.2308.04080.

```
--algorithm $NoEquivocation$ {
    variables
      $input \in [P \rightarrow V]$ ;
      $received = [p \in P \mapsto [q \in P \mapsto Bot]]$ ;   message received by p from q
      $rnd = 1$ ;   1..3
      $output = [p \in P \mapsto [q \in P \mapsto Bot]]$ ;   output[p][q] is the message that p simulates receiving from q
    define {
        $HeardOf(p) \triangleq \{q \in P : received[p][q] \neq Bot\}$   heard of in the current round
        $Minority(S) \triangleq \{M \in \text{SUBSET } S : 2 * Cardinality(M) < Cardinality(S)\}$
        $NumHeardOf(p1, p2) \triangleq$   number of processes that report to p1 hearing from p2:
            $Cardinality(\{q \in P : received[p1][q] \neq Bot \wedge received[p1][q][p2] \neq Bot\})$
        $NumHeardValue(p1, p2, v) \triangleq$   number of processes that report to p1 hearing v from p2:
            $Cardinality(\{q \in P : received[p1][q] \neq Bot \wedge received[p1][q][p2] = v\})$
        $ValidOutput(p1, p2, v) \triangleq$
            $\wedge\ 2 * NumHeardValue(p1, p2, v) > Cardinality(HeardOf(p1))$
            $\wedge\ \forall q \in P : received[p1][q] \neq Bot \wedge received[p1][q][p2] \neq Bot \Rightarrow received[p1][q][p2] = v$
        $Output(p1, p2) \triangleq$
            IF $\exists v \in V : ValidOutput(p1, p2, v)$   true for at most one value v
             THEN CHOOSE $v \in V : ValidOutput(p1, p2, v)$
             ELSE
                IF $\exists q \in P : received[p1][q] \neq Bot \wedge received[p1][q][p2] \neq Bot$
                 THEN $Lambda$
                 ELSE $Bot$
        $SimulatedParticipants \triangleq \{p \in P : \exists q \in P : output[q][p] \neq Bot\}$
        $CorrectSimulatedParticipants \triangleq participating[1] \setminus corrupted$
        Now we define the correctness properties of the algorithm:
        $NoEquivocation \triangleq \forall p1, p2, q \in P :$
            $output[p1][q] \in V \wedge pc[p2] =$ "Done" $\Rightarrow output[p2][q] \in \{output[p1][q], Lambda\}$
        $NoTampering \triangleq \forall p, q \in P :$
            $\wedge\ p \in CorrectSimulatedParticipants$
            $\wedge\ pc[q] =$ "Done"
            $\Rightarrow output[q][p] = input[p]$
        $MinorityCorruption \triangleq (\forall p \in P : pc[p] =$ "Done" $) \Rightarrow$
            $2 * Cardinality(CorrectSimulatedParticipants) > Cardinality(SimulatedParticipants)$ }
    We now specify the simulation algorithm:
    process ( $proc \in P$ ) {
 r1:    $broadcast(input[self])$ ;
 r2:    await $rnd = 2$ ;
        $broadcast(received[self])$ ;
 r3:    await $rnd = 3$ ;
        $output[self] := [p \in P \mapsto Output(self, p)]$ } }
```

Figure 2 Excerpt from the PlusCal/TLA+ specification of the simulation algorithm.

```
--algorithm $CommitAdopt${
    variables
        $input \in [P \to V]$ ;  the processors' inputs
         message received by p from q in the current round; Bot means no message received:
        $received = [p \in P \mapsto [q \in P \mapsto Bot]]$ ;
        $rnd = 1$ ;  the current round (1, 2, or 3); round 3 is when we produce outputs
        $output = [p \in P \mapsto Bot]$ ;
    define {
            the set of processors from which p received a message (i.e. heard of):
          $HeardOf(p) \triangleq \{q \in P : received[p][q] \neq Bot\}$
            the set of minority subsets of S:
          $Minority(S) \triangleq \{M \in \text{SUBSET } S : 2 * Cardinality(M) < Cardinality(S)\}$
            the number of votes for v that p received:
          $VoteCount(p, v) \triangleq Cardinality(\{q \in P : received[p][q] = v\})$
            the set of values v for which p received a strict majority of votes:
          $VotedByMajority(p) \triangleq$
              $\{v \in V : 2 * VoteCount(p, v) > Cardinality(HeardOf(p))\}$
            the set of values v that were voted for the most often according to p:
          $MostVotedFor(p) \triangleq$
              $\{v \in V : \forall w \in V \setminus \{v\} : VoteCount(p, v) \geq VoteCount(p, w)\}$
            the correctness properties:
          $Agreement \triangleq \forall p, q \in P :$
              $output[p] \neq Bot \land output[q] \neq Bot \land output[p][1] = \text{"commit"}$
               $\Rightarrow output[p][2] = output[q][2]$
          $Validity \triangleq \forall p \in P : \forall v \in V :$
              $pc[p] = \text{"Done"} \land (\forall q \in P : input[q] = v)$
               $\Rightarrow output[p] = \langle \text{"commit"}, v \rangle$ }
    process ( $proc \in P$ ) {
r1:       $broadcast(input[self])$ ;
r2:       await $rnd = 2$ ;
          if ( $VotedByMajority(self) \neq \{\}$ )
               with ( $v \in VotedByMajority(self)$ )
               $broadcast(v)$
          else $broadcast(NoCommit)$ ;
r3:       await $rnd = 3$ ;  in round 3 we just produce an output
          if ( $VotedByMajority(self) \neq \{\}$ )
               with ( $v \in VotedByMajority(self)$ )
               $output[self] := \langle \text{"commit"}, v \rangle$
          else if ( $MostVotedFor(self) \neq \{\}$ )
              with ( $v \in MostVotedFor(self)$ )
              $output[self] := \langle \text{"adopt"}, v \rangle$
          else
              with ( $v \in V$ )
              $output[self] := \langle \text{"adopt"}, v \rangle$ }  }
```

Figure 3 Excerpt from the PlusCal/TLA+ specification of the commit-adopt algorithm.