Beyond Optimal Fault-Tolerance

Andrew Lewis-Pye
London Schoool of Economics (LSE), UK

Tim Roughgarden
Columbia University, New York, NY, USA
a16z Crypto Research, New York, NY, USA

Abstract

One of the most basic properties of a consensus protocol is its fault-tolerance – the maximum fraction of faulty participants that the protocol can tolerate without losing fundamental guarantees such as safety and liveness. Because of its importance, the optimal fault-tolerance achievable by any protocol has been characterized in a wide range of settings. For example, for state machine replication (SMR) protocols operating in the partially synchronous setting, it is possible to simultaneously guarantee consistency against α-bounded adversaries (i.e., adversaries that control less than an α fraction of the participants) and liveness against β-bounded adversaries if and only if $\alpha + 2\beta \leq 1$.

This paper characterizes to what extent "better-than-optimal" fault-tolerance guarantees are possible for SMR protocols when the standard consistency requirement is relaxed to allow a bounded number r of consistency violations, each potentially leading to the rollback of recently finalized transactions. We prove that bounded rollback is impossible without additional timing assumptions and investigate protocols that tolerate and recover from consistency violations whenever message delays around the time of an attack are bounded by a parameter Δ^* (which may be arbitrarily larger than the parameter Δ that bounds post-GST message delays in the partially synchronous model). Here, a protocol's fault-tolerance can be a non-constant function of r, and we prove, for each r, matching upper and lower bounds on the optimal "recoverable fault-tolerance" achievable by any SMR protocol. For example, for protocols that guarantee liveness against 1/3-bounded adversaries in the partially synchronous setting, a 5/9-bounded adversary can always cause one consistency violation but not two, and a 2/3-bounded adversary can always cause two consistency violations but not three. Our positive results are achieved through a generic "recovery procedure" that can be grafted on to any accountable SMR protocol and restores consistency following a violation while rolling back only transactions that were finalized in the previous $2\Delta^*$ timesteps.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed computing, consensus, recovery

Digital Object Identifier 10.4230/LIPIcs.AFT.2025.15

Related Version *See full version for appendix*: `https://eprint.iacr.org/2025/070`

Funding *Tim Roughgarden*: The research of Tim Roughgarden at Columbia University was supported in part by NSF award CNS-2212745.

1 Introduction

We consider protocols for the state machine replication (SMR) problem, in which processes receive transactions from an environment and are responsible for finalizing a common sequence of transactions. We focus on the partially synchronous setting [14], in which message delays are bounded by a known parameter Δ following an unknown "global stabilization time" GST (and unbounded until that point).

The two most basic requirements of an SMR protocol are consistency, meaning that no two processes should finalize incompatible sequences of transactions (one should be a prefix of the other), and liveness, which stipulates that valid transactions should eventually be finalized (ideally, following GST, within an amount of time proportional to Δ). Guaranteeing

consistency and liveness becomes impossible if too many of the processes are faulty (i.e., deviate from the intended behavior of a protocol). For the SMR problem in partial synchrony, it is possible to simultaneously guarantee consistency against α-bounded adversaries (i.e., adversaries that control less than an α fraction of the participants) and liveness against β-bounded adversaries if and only if $\alpha + 2\beta \leq 1$ [19, 14].

The focus of this paper is consistency violations – the type of violation that enables, for example, double-spending a cryptocurrency native to a blockchain protocol. What can be said about a protocol when the adversary is large enough to cause a consistency violation? For example, is it already in a position to cause an unbounded number of consistency violations (as opposed to just one), or could the protocol "fight back" in some way?

To make sense of this question and the idea of multiple consistency violations, we must formalize a sense in which a protocol might restore consistency following a violation, necessarily by rolling back transactions that had been viewed as finalized by some non-faulty processes. One key parameter is then the *recovery time* d, meaning the number of timesteps after a violation before a protocol returns to healthy operation. A second is the *rollback* h, meaning that the recovery process "unfinalizes" only transactions that have been finalized within the previous h time steps.

The natural wishlist for an SMR protocol in partial synchrony would then be:

(1) All of the "usual" guarantees, such as optimal fault-tolerance (i.e., consistency with respect to α-bounded adversaries and liveness with respect to β-bounded adversaries for some $\alpha, \beta > 0$ with $\alpha + 2\beta = 1$).

(2) Automatic recovery from a consistency violation with the worst-case recovery time d and worst-case rollback h as small as possible (if nothing else, independent of the specific execution).

(3) Never suffers more than r consistency violations overall, where r is as small as possible.

To what extent are these properties simultaneously achievable?

This paper provides a thorough investigation of this question. To expose the richness of the answer, we work with a timing model that can be viewed as an interpolation between the synchronous and partially synchronous settings. In addition to the usual parameters Δ and GST (known and unknown, respectively) of the partially synchronous model, we allow for a known parameter $\Delta^* \geq \Delta$ which may or may not bound message delays prior to GST. Canonically, Δ^* should be thought of as orders of magnitude larger than Δ, with Δ indicating the speed of communication between processes when all is well (no network issues, no attacks) and Δ^* a safe (and possibly large) upper-bound on the speed of (possibly out-of-band) communication around the time of an attack.[1] We will be interested in protocols that always satisfy all the "usual" guarantees (1) and that finalize transactions in time $O(\Delta)$ (rather than $O(\Delta^*)$) after GST, whether or not pre-GST message delays are bounded by Δ^*, and also satisfy the additional recovery guarantees (2) and (3) in the event that pre-GST message delays are in fact bounded by Δ^*.[2]

Our main positive result, stated formally in Theorem 1 and proved in Section 7, shows that such protocols do indeed exist. For example, we show that there is a protocol that satisfies:

[1] Indeed, for our positive results, message delays must be bounded by Δ^* for the duration of our recovery procedure, but not otherwise.

[2] In particular, a synchronous protocol with respect to the parameter Δ^* will not generally satisfy consistency and liveness if message delays do not happen to bounded above by Δ^*.

- $\frac{1}{3}$-resilience in partial synchrony (independent of whether Δ^* bounds pre-GST message delays), with expected latency $O(\Delta)$ after GST;
- should pre-GST message delays be bounded by Δ^*, recovers from consistency violations in expected time $O(\Delta^*)$ and with rollback $2\Delta^*$; and
- should pre-GST message delays be bounded by Δ^*, never suffers from more than one consistency violation with a $\frac{5}{9}$-bounded attacker, and never suffers from more than two consistency violations with a $\frac{2}{3}$-bounded attacker.

We achieve this result by designing a generic "recovery procedure" that can be grafted on to any accountable SMR protocol, including protocols with asymmetric fault-tolerance with respect to consistency and liveness attacks. Sections 4 and 5 give informal and formal, respectively, descriptions of this procedure.

Our results are tight in several senses. For example, we prove in Theorem 2 that recovery from a consistency violation necessarily requires a rollback proportional to the parameter Δ^*. In particular, in the pure partially synchronous model ($\Delta^* = +\infty$, in effect), recovery from consistency violations with bounded rollback is impossible. Theorems 3 and 4 show that the bounds we obtain on adversary size (as a function of the number r of consistency violations) are optimal. For example, in the symmetric case above, an attacker controlling five-ninths of the processes can always force two consistency violations, and one controlling two-thirds of the processes can cause unbounded rollback.

A comment on the benefits of automated recovery. It is sometimes assumed in the literature (e.g. [26]) that, in the event of a consistency violation, an "administrator" will somehow (through out-of-protocol means) remove perpetrators from the system and co-ordinate an appropriate "reboot". Our automated recovery procedure is "in-protocol," and therefore has the significant benefits that it does not rely on a centralized entity, has no single point of failure, and formalizes a process by which one can guarantee bounded rollback (a principal focus of this paper).

A comment on the (Δ, Δ^*) timing assumptions. In practical settings, partial synchrony may hold with respect to Δ of the order of a few hundred milliseconds. It may also be reasonable to suppose that synchrony will hold with respect to some much larger bound Δ^*, but making direct use of this larger bound to run a synchronous protocol (and increase resilience beyond 1/3) would result in an impractically slow protocol. Our use of the two bounds Δ and Δ^* reflects our interest in considering protocols that give *all* the usual guarantees of protocols for partial synchrony (with no requirement that the larger bound Δ^* should hold), but which *also* give recovery guarantees in the case that the larger bound for synchrony should hold.

2 The setup

We consider a set $\Pi = \{p_1, \ldots, p_n\}$ of n processes. Each process p_i is told its "name" i as part of its input. We focus on the case of a static adversary, which chooses a set of processes to corrupt at the start of the protocol execution.[3] We call a process corrupted by the adversary

[3] All results in this paper hold more generally for adaptive adversaries (with essentially identical proofs), with the exception of the bound on the expected termination time for the recovery procedure asserted in part (iii) of Theorem 1 (which requires that a random permutation of the processes be chosen subsequent to the adversary deciding which processes to corrupt).

faulty. Faulty processes may behave arbitrarily (i.e., we consider *Byzantine* faults), subject to our cryptographic assumptions (stated below). Processes that are not faulty are *correct*. The adversary is *ρ-bounded* if it corrupts less than a ρ fraction of the n processes.

Cryptographic assumptions. We assume that processes communicate by point-to-point authenticated channels and that a public key infrastructure (PKI) is available for generating and validating signatures. For simplicity of presentation (e.g., to avoid the analysis of negligible error probabilities), we work with ideal versions of these primitives (i.e., we assume that faulty processes cannot forge signatures). We also assume that all processes have access to a random permutation of Π, denoted $\Pi^* : [1, n] \to \Pi$, which is sampled after the adversary chooses which processes to corrupt.

Message delays. We consider a discrete sequence of timeslots $t \in \mathbb{N}_{\geq 0}$. As discussed in the introduction, we consider protocols that operate in partial synchrony (with some parameter Δ, perhaps in the order of seconds or milliseconds) and satisfy additional recovery properties should synchrony hold (with some different parameter Δ^*, which may be set much larger than Δ) while running the "recovery procedure".

Synchrony. In the synchronous setting, a message sent at time t must arrive by time $t + \Delta^*$, where Δ^* is known to the protocol.

Partial synchrony. In the partially synchronous setting, a message sent at time t must arrive by time $\max\{\text{GST}, t\} + \Delta$. While Δ is known, the value of GST is unknown to the protocol. The adversary chooses GST and also message delivery times, subject to the constraints already specified.

Clock synchronization. In the partially synchronous setting, we suppose all correct processes begin the protocol execution before GST. When considering the synchronous setting, we suppose all correct processes begin the protocol execution by time Δ^*. A correct process begins the protocol execution with its local clock set to 0; thus, we do not suppose that the clocks of correct processes are synchronized. For simplicity, we assume that the clocks of correct processes all proceed in real time, meaning that if $t' > t$ then the local clock of correct p at time t' is $t' - t$ in advance of its value at time t.[4]

Notation concerning executions and received messages. We use the following notation when discussing any execution of a protocol:

- $M_i(t)$ denotes the set of messages received by process p_i by timeslot t;
- $M_c(t)$ denotes the set of all messages received by any correct process by timeslot t;
- M_c denotes the set of all messages received by any correct process during the execution.

Transactions. Transactions are messages of a distinguished form, signed by the *environment*. Each timeslot, each process may receive some finite set of transactions directly from the environment.

Determined inputs. A value is *determined* if it known to all processes, and is otherwise *undetermined*. For example, Δ, Δ^* and Π are determined, while GST is undetermined.

[4] Using standard arguments, our protocol and analysis can easily be extended to the case in which there is a known upper bound on the difference between the clock speeds of correct processes.

State machine replication. Informally, a protocol for state machine replication (SMR) must cause correct processes to finalize *logs* (sequences of transactions) that are live and consistent with each other, and must also produce "certificates" of finalization that can be presented to clients who may not always be online and observing the protocol execution. Formally, if σ and τ are sequences, we write $\sigma \preceq \tau$ to denote that σ is a prefix of τ. We say σ and τ are *compatible* if $\sigma \preceq \tau$ or $\tau \preceq \sigma$. If two sequences are not compatible, they are *incompatible*. If σ is a sequence of transactions, we write $\mathtt{tr} \in \sigma$ to denote that the transaction $\mathtt{tr}$ belongs to the sequence σ.

Fix a process set Π and *genesis log*, denoted $\mathtt{log}_G$. If $\mathcal{P}$ is a protocol for SMR, then it must specify a function $\mathcal{F}$, which may depend on Π and $\mathtt{log}_G$, that maps any set of messages to a sequence of transactions extending $\mathtt{log}_G$. We require the following conditions to hold in every execution (for any M_1, M_2, p_i, p_j and $\mathtt{tr}$):

Consistency. If $M_1 \subseteq M_2 \subseteq M_c$, then $\mathcal{F}(M_1) \preceq \mathcal{F}(M_2)$.[5]

Liveness. If p_i and p_j are correct and if p_i receives the transaction $\mathtt{tr}$ then, for some t, $\mathtt{tr} \in \mathcal{F}(M_j(t))$.

This definition of consistency ensures that correct processes never finalize incompatible logs: for any sets $M_1, M_2 \subseteq M_c$ that two such processes might have received, $\mathcal{F}(M_1) \preceq \mathcal{F}(M_1 \cup M_2)$ and $\mathcal{F}(M_2) \preceq \mathcal{F}(M_1 \cup M_2)$. We say a set of messages M is a *certificate* for a sequence of transactions σ if $\mathcal{F}(M) \succeq \sigma$: intuitively, any process can present the set of (potentially signed) messages M to a "client" that has not been observing the protocol execution as proof that it has finalized a sequence of transactions extending σ. If we wish to make $\mathcal{F}$ explicit, we may also say that M is an $\mathcal{F}$-certificate for σ.

SMR (informal discussion). The selection $\mathcal{F}$ of finalized transactions by a correct process depends only on the set of messages it has received, and not on the times at which these messages were received. The motivation for this restriction is to formalize the requirement of SMR [22] that "clients" wishing to verify the finality of transactions without observing the entire execution of the protocol should be able to do so (via a suitable certificate). As discussed in [25], this crucial distinction between SMR and Total-Order-Broadcast is sometimes overlooked in the literature: In the context of an honest majority, any report of finality by a majority of processes constitutes proof of finality, so a protocol for Total-Order-Broadcast directly gives a protocol for SMR. However, the distinction becomes non-trivial in the context of player reconfiguration, or if there is no guarantee of honest majority (such as in this paper). In partial synchrony, certificates are anyways required for guaranteed consistency and liveness [16], so our approach is general for the context considered here.

The liveness parameter. If there exists some fixed ℓ that is a function of determined inputs[6] other than Δ^* and such that the following holds in all executions of $\mathcal{P}$, we say $\mathcal{P}$ has *liveness parameter* ℓ: If p_i and p_j are correct and if p_i receives the transaction $\mathtt{tr}$ at time t then, for $t' = \max\{t, \mathrm{GST}\} + \ell$, $\mathtt{tr} \in \mathcal{F}(M_j(t'))$.

[5] This is equivalent to the seemingly stronger condition in which M_c is replaced by the set of messages received by any process (correct or otherwise), as faulty processes always have the option of echoing any messages they receive to correct processes.

[6] The requirement that ℓ is not a function of Δ^* (while Δ^* is not necessarily $O(\Delta)$) means that having liveness parameter ℓ may require finalization of transactions in time less than Δ^* after GST.

Liveness and consistency resilience. Recall that $n = |\Pi|$. When the protocol $\mathcal{P}$ is clear from context, we write ρ_C to denote the consistency resilience of $\mathcal{P}$, which is the largest ρ such that, for all n, the protocol satisfies consistency so long as the adversary is ρ-bounded. We write ρ_L to denote the liveness resilience, which is the largest ρ such that, for all n, the protocol satisfies liveness so long as the adversary is ρ-bounded. It is well known that $\rho_C + 2\rho_L \leq 1$ in the partially synchronous setting [14] and that $\rho_C + \rho_L \leq 1$ in the synchronous setting [19].

The number of consistency violations. When $\mathcal{F}$ is clear from context, we say the set of messages M *has* r *consistency violations* if there exist $M_0 \subset M_1 \subset \cdots \subset M_r \subseteq M$ such that, for each $s \in \{0, 1, \ldots, r-1\}$, $\mathcal{F}(M_s) \not\preceq \mathcal{F}(M_{s+1})$. We also say M *has a consistency violation* (w.r.t. $\mathcal{F}$) if it has at least one consistency violation. An execution has r consistency violations if M_c has r consistency violations.

Accountable protocols (informal discussion). Informally, a protocol is *accountable* if it produces *proofs of guilt* for some faulty processes in the event of a consistency violation. We cannot generally require proofs of guilt for a fraction $\lambda > \rho_C$ of processes, since consistency violations may occur when less than a fraction λ of processes are faulty. On the other hand, all standard protocols that provide accountability produce proofs of guilt for a ρ_C fraction of processes in the event of a consistency violation [24].

Accountable protocols (formal definition). Consider an SMR protocol $\mathcal{P}$:

- We say the set of messages M is a *proof of guilt* for $p \in \Pi$ if there does not exist any execution of $\mathcal{P}$ in which p is correct and for which $M \subseteq M_c$.[7]
- We say $\mathcal{P}$ is λ-*accountable* if the following holds at every timeslot t of any execution of $\mathcal{P}$: if $M_c(t)$ has a consistency violation, then $M_c(t)$ is a proof of guilt for at least a λ fraction of processes in Π.

Given that all standard protocols that are λ-accountable for any $\lambda > 0$ are also ρ_C-accountable, we will say that a protocol is *accountable* to mean that it is ρ_C-accountable. It is important to note that, while an accountable protocol ensures the existence of proofs of guilt for a ρ_C fraction of processes in the event of a consistency violation, it *does not* automatically ensure *consensus* between correct processes as to a set of faulty processes for which a proof of guilt exists. One role of the *recovery* procedure (as specified in Section 5) will be to ensure such consensus.

Message gossiping. In our recovery procedure, it will be convenient to assume that correct processes gossip all messages received. Then, if synchrony does hold with respect to Δ^*, any message received by correct p at some timeslot t is received by all correct processes by time $t + \Delta^*$. It will not generally be necessary to gossip all messages; for example, for standard quorum-based protocols, it will suffice to gossip blocks that have received quorum certificates (QCs) along with those QCs.

A comment on setup assumptions. Given an accountable SMR protocol $\mathcal{P}$ and a process set Π, our wrapper will initiate a sequence of executions of $\mathcal{P}$, with process sets that are progressively smaller subsets of Π. Of course, a PKI for Π suffices to provide a PKI for each

[7] If we wish to make $\mathcal{P}$, $\mathtt{log}_G$, and Π explicit, we may also say that M is a $(\mathcal{P}, \Pi, \mathtt{log}_G)$-proof of guilt.

subset of Π and a random permutation of Π naturally induces a random permutation of each subset. Moreover, the maximum number of executions of $\mathcal{P}$ initiated by the wrapper will be small and the size of the process set of each is known ahead of time. (For example, if $\rho_C = 1/3$ and the adversary is 5/9-bounded, the wrapper will initiate at most two executions of $\mathcal{P}$; if $\rho_C = 1/3$ and the adversary is 2/3-bounded, at most three.) Thus, for setup assumptions such as threshold signatures, one can simply run each required setup in advance, before executing the wrapper.

3 Recovery metrics

In this section, we introduce definitions to quantify how well a protocol recovers from consistency violations.

Generalizing resilience to take recovery into account. Is a protocol vulnerable to one consistency violation inexorably doomed to an unbounded number of them? Or could a protocol achieve strictly higher levels of resilience by tolerating (and recovering from) a bounded number of consistency violations? The following definitions generalize consistency and liveness resilience to account for the possibility of recovery from consistency violations.

- *Recoverable consistency resilience.* Consider a function $g : \mathbb{N}_{\geq 0} \to [0, 1]$. We say a protocol $\mathcal{P}$ has recoverable consistency resilience g if the following holds for each $r \in \mathbb{N}_{\geq 0}$: $g(r)$ is the largest ρ such that, for all n, provided the adversary is ρ-bounded, executions of $\mathcal{P}$ have at most r consistency violations.
- *Recoverable liveness resilience.* Consider a function $g : \mathbb{N}_{\geq 0} \to [0, 1]$. We say a protocol $\mathcal{P}$ has recoverable liveness resilience g if the following holds for each $r \in \mathbb{N}_{\geq 0}$: $g(r)$ is the largest ρ such that, for all n, provided the adversary is ρ-bounded, liveness holds in all executions with precisely r consistency violations.[8]

Suppose $\mathcal{P}$ has consistency resilience ρ_C and recoverable consistency resilience g. Note that $g(0) = \rho_C$. Also, g is nondecreasing (i.e., $g(s) \geq g(r)$ for $s > r$): if executions of $\mathcal{P}$ have at most r consistency violations when the adversary is ρ-bounded, then this is also true of all $s > r$. If $g(r + 1) > g(r)$, the protocol effectively has increased consistency resilience after r consistency violations.

Recoverable resilience for our wrapper. Suppose $\mathcal{P}$ is accountable and has consistency resilience ρ_C and liveness resilience ρ_L for partial synchrony with $\rho_C + 2\rho_L = 1$. If we identify some fraction x of the processes in Π as faulty and then run an execution of $\mathcal{P}$ using the remaining processes, there will be no consistency violation so long as less than a fraction $x + \rho_C(1 - x)$ of the processes in Π are faulty. Given this, let us define a sequence $\{x_r\}_{r \in \mathbb{N}_{\geq 0}}$ by recursion:

$$x_0 = 0, \quad x_{r+1} = x_r + \rho_C(1 - x_r).$$

Define:

$$g_1(r) = \min\{x_{r+1}, 1 - \rho_L\}, \quad g_2(r) = \min\{x_r + \rho_L(1 - x_r), 1 - \rho_L\}. \tag{1}$$

[8] Prior to the rth consistency violation, a sufficiently large adversary may still be in a position to cause a liveness violation.

Given $\mathcal{P}$ as input, our wrapper produces an SMR protocol with recoverable consistency resilience g_1 and recoverable liveness resilience g_2 as in (1). For example, if $\rho_C = \rho_L = \frac{1}{3}$, then $g_1(0) = g_2(0) = \frac{1}{3}$, $g_1(1) = g_2(1) = \frac{5}{9}$, and $g_1(r) = g_2(r) = \frac{2}{3}$ for all $r \geq 2$.

Specifying the recovery time. Next, we provide a definition that captures the time required by a protocol to recover from consistency violations. Suppose $\mathcal{P}$ has recoverable liveness resilience g. We say $\mathcal{P}$ has *recovery time d with liveness parameter ℓ* if the following holds for all executions $\mathcal{E}$ of $\mathcal{P}$:

($\dagger_{d,\ell}$) If there exists r such that $\mathcal{E}$ has precisely r consistency violations, let t be least such that $M_c(t)$ has r consistency violations (otherwise set $t = \infty$). If correct p_i receives the transaction $\mathtt{tr}$ at any timeslot t' then, for every correct p_j and for $t'' = \max\{t + d, \mathrm{GST}, t'\} + \ell$, $\mathtt{tr} \in \mathtt{log}_j(t'')$.

In the above, d should be thought of as a "grace period" after consistency violations, after which liveness with parameter ℓ must hold. In our construction, d is governed by the length of time it takes to run our recovery procedure.

Probabilistic recovery time. Our recovery procedure uses the random permutation Π^* – chosen after the adversary chooses which processes to corrupt – to select "leaders," and as such it will run for a random duration. To analyze this, we allow the grace period parameter d in the definition above to depend on an error probability $\varepsilon \in [0, 1]$ and sometimes write d_ε to emphasize this dependence. We then make the following definitions:

- We say that ($\dagger_{d,\ell}$) *is ensured with probability at least p* if, for every choice of corrupted processes (consistent with a static ρ-bounded adversary), with probability at least p over the choice of Π^* (sampled from the uniform distribution), ($\dagger_{d,\ell}$) holds in every execution consistent with these choices and with the setting.
- We say that $\mathcal{P}$ has *probabilisitic recovery time d_ε with liveness parameter ℓ* if it holds for every $\varepsilon \in [0, 1]$ that ($\dagger_{d_\varepsilon,\ell}$) is ensured with probability at least $1 - \varepsilon$.

Recovery time for our wrapper. Given $\mathcal{P}$ with liveness parameter ℓ as input, our wrapper will produce an SMR protocol with (worst-case) recovery time $O(\Delta^* \cdot f_a)$, probabilisitic recovery time $O(\Delta^* \cdot \log \frac{1}{\varepsilon})$, and liveness parameter ℓ, where f_a denotes the actual (undetermined) number of faulty processes.

Bounding rollback. We say that a protocol has *rollback bounded by h* if the following holds for every execution consistent with the setting and every correct $p_i, p_j \in \Pi$: if there exists an interval $I = [t, t + h]$ such that $\sigma \preceq \mathtt{log}_i(t')$ for all $t' \in I$, then $\sigma \preceq \mathtt{log}_j(t')$ for all sufficiently large t'. That is, consistency violations can "unfinalize" only transactions that have been finalized recently, within the previous h time steps. Here, h can be any value that depends only on determined inputs.

Bounding rollback for our wrapper. Given an SMR protocol $\mathcal{P}$ with liveness resilience ρ_L as input, our wrapper will produce an SMR protocol with rollback bounded by $h = 2\Delta^*$ so long as synchrony holds for Δ^* and the adversary is $(1 - \rho_L)$-bounded. In fact, while the recovery procedure described in Section 5 requires a common choice for Δ^*, rollback can be bounded on an individual basis, with each correct process making their own personal choice of message delay bound $\leq \Delta^*$. rollback will be bounded by twice their personal choice of bound, so long as that bound on message delay holds.[9]

[9] The requirement that the choice be $\leq \Delta^*$ stems from the fact that the recovery procedure requires delays to be bounded by Δ^* to function correctly.

4 The intuition behind the wrapper

We describe a wrapper, which takes an accountable and optimally resilient SMR protocol $\mathcal{P}$ as input, and which runs an execution of $\mathcal{P}$ until a consistency violation occurs.[10] Once this happens, the wrapper triggers a "recovery procedure", which achieves consensus on a set of faulty processes F for which a proof of guilt exists, together with a long initial segment of the log produced by $\mathcal{P}$ below which no consistency violation has occurred. The wrapper then initiates another execution of $\mathcal{P}$ that takes this log as its genesis log, with the players in F removed from the process set. This next execution is run until another consistency violation occurs, and so on.

Specifying $\mathtt{log}_i(t)$ and $\mathcal{F}$. While the formal definition of SMR in Section 2 requires us to specify the finalization rule $\mathcal{F}$ (from which the transactions $\mathtt{log}_i(t)$ finalized by p_i can then be defined as $\mathcal{F}(M_i(t))$), it will be more natural when defining our wrapper to specify $\mathtt{log}_i(t)$ directly, and then later to define $\mathcal{F}$ such that $\mathtt{log}_i(t) = \mathcal{F}(M_i(t))$. Recall that the given protocol $\mathcal{P}$ satisfies consistency and liveness with respect to a function that may depend on the process set and the genesis log. We write $\mathcal{F}(\Pi, \mathtt{log}_G)$ to denote this function.

The structure of this section. In Section 4.1, we describe the intuition behind a feature of the wrapper which allows us to ensure rollback bounded by $2\Delta^*$. We stress that *bounding rollback is non-trivial*: this is the requirement on the recovery procedure that requires the most delicate analysis. Section 4.2 then describes the intuition behind the recovery procedure.

In what follows, we use the variable $\mathcal{E}$ to denote an execution of the wrapper (with process set Π and $\mathtt{log}_G$ as the genesis log), which initiates successive executions $\mathcal{E}_1, \mathcal{E}_2, \ldots$ of $\mathcal{P}$, where $\mathcal{E}_r$ has process set Π_r and $\mathtt{log}_{G_r}$ as the genesis log. Process p_i maintains local variables $\mathtt{M}_i$ and $\mathtt{M}_{i,r}$ for each $r \geq 1$.[11] The former records all messages so far received in execution $\mathcal{E}$, while the latter records all messages so far received in execution $\mathcal{E}_r$. We suppose messages have tags identifying the execution in which they are sent, and that $\mathtt{M}_{i,r} \subseteq \mathtt{M}_i$ at every timeslot, for all correct p_i and all r.

4.1 Ensuring bounded rollback

In what follows, we write ρ_C and ρ_L to denote the consistency and liveness resilience of $\mathcal{P}$. Each process p_i executing the wrapper maintains a value $\mathtt{log}_i$. Suppose the currently running execution of $\mathcal{P}$ is $\mathcal{E}_r$. To ensure rollback bounded by $2\Delta^*$, p_i proceeds as follows:

- While running the execution $\mathcal{E}_r$ of $\mathcal{P}$, and when p_i finds that some subset of $\mathtt{M}_{i,r}$ is an $\mathcal{F}(\Pi_r, \mathtt{log}_{G_r})$-certificate for σ properly extending $\mathtt{log}_i$, it will set $\mathtt{log}_i$ to extend σ.
- Process p_i will only *strongly finalize* σ, however, once $\mathtt{log}_i$ has extended σ for an interval of length $2\Delta^*$.
- Upon finding that $\mathtt{M}_{i,r}$ has a consistency violation w.r.t. $\mathcal{F}(\Pi_r, \mathtt{log}_{G_r})$, p_i will:
  - Send a signed *r-genesis* message $(\text{gen}, \mathtt{log}_i, r)$ to all processes (motivation below);
  - Temporarily set $\mathtt{log}_i$ to be $\mathtt{log}_{G_r}$;
  - Stop running $\mathcal{E}_r$, and;
  - Begin the recovery procedure.

[10] By "optimally resilient," we mean that the protocol's consistency resilience ρ_C and liveness resilience λ_L in partial synchrony are both positive and satisfy $\rho_C + 2\rho_L = 1$ (as is the case for all of the "usual" SMR protocols designed for the partially synchronous setting). This assumption is merely to simplify the presentation. For a non-optimally resilient protocol, the "$1 - \rho_L$" term in (1) should be replaced by "$(1 + \rho_C)/2$".

[11] We use $\mathtt{M}_i$ when specifying the pseudocode, rather than $M_i(t)$, since p_i only has access to its local clock and does not know the "global" value of t.

To see what this achieves (modulo complications that may later be introduced by the recovery procedure), suppose that synchrony holds for Δ^*. Then, due to our assumptions on message gossiping described in Section 2, $2\Delta^*$ bounds the round-trip time between any two correct processes. In particular, suppose that p_i finalizes σ at t because there exists $M \subseteq \mathtt{M}_{i,r}$ which is an $\mathcal{F}(\Pi_r, \mathtt{log}_{G_r})$-certificate for σ. Then every correct process p_j will receive the messages in M by $t + \Delta^*$, and will then finalize σ (never to subsequently finalize anything incompatible with σ), unless $\mathtt{M}_{j,r}$ has a consistency violation (w.r.t. $\mathcal{F}(\Pi_r, \mathtt{log}_{G_r})$) by that time. In the latter case, p_i will begin the recovery procedure by timeslot $t + 2\Delta^*$ and will not strongly finalize σ. So far, this approach is similar to the "safety-favoring" construction of [25].

Complications introduced by the recovery procedure. Our recovery procedure introduces the complication that there is not necessarily consensus on which logs have been strongly finalized by some correct process. If a single correct process has strongly finalized σ when the recovery procedure is triggered, and if the procedure determines that a log $\sigma' \not\succeq \sigma$ should be used as the genesis log in the next execution of $\mathcal{P}$, then this may violate the condition that the protocol has rollback bounded by $2\Delta^*$. We must therefore ensure that the recovery procedure reaches consensus on a log that extends all logs strongly finalized by correct processes. As in explained in Section 4.2, the r-genesis messages sent by processes before entering the recovery procedure will be used to achieve this.

4.2 The intuition behind the recovery procedure

Recall that ρ_C (ρ_L) is the consistency (liveness) resilience of $\mathcal{P}$ in partial synchrony (with parameter Δ), and that the wrapper aims to deliver extra functionality in the case that synchrony happens to hold with respect to the (possibly large) bound Δ^*, and so long as the adversary is $(1 - \rho_L)$-bounded. So, suppose these conditions hold.

As noted in Section 4.1, while running execution $\mathcal{E}_r$ of $\mathcal{P}$, process p_i will enter the recovery procedure upon finding that $\mathtt{M}_{i,r}$ has a consistency violation. Given our gossip assumption, described in Section 2, this means that correct processes will begin the recovery procedure within time Δ^* of each other. The key observation behind the recovery procedure is that, if one has a proof of guilt for processes in some set F, where $|F| \geq \rho_C n$, then the fact that the adversary is $(1 - \rho_L)$-bounded (and $2\rho_L + \rho_C = 1$) means that the adversary controls less than half the processes in $\Pi - F$. This follows since:

$$1 - \rho_L - \rho_C = \rho_L, \text{ and so } (1 - \rho_L - \rho_C)/(1 - \rho_C) = \rho_L/2\rho_L.$$

As a consequence, we can run a modified version of a standard ($\frac{1}{2}$-resilient) SMR protocol for synchrony (our protocol is most similar to [1]), in which the instructions are divided into views, each with a distinct leader. In each view, the leader makes a proposal for the set of processes F that should be removed from Π_r to form Π_{r+1}, and *the processes outside F then vote on that proposal.*

Ensuring an appropriate value for $\mathtt{log}_{G_{r+1}}$. As well as proposing F, the leader p_i must also suggest a sequence σ to be used as $\mathtt{log}_{G_{r+1}}$ and this sequence must extend all logs strongly finalized by correct processes. To achieve this (while keeping the probabilistic recovery time small), we run a short sub-procedure at the beginning of the recovery procedure, before leaders start proposing values. We proceed as follows:

- Each correct p_j waits time $2\Delta^*$ upon beginning the recovery procedure and then sets $P_j(r)$ to be the set of processes in Π_r from which it has received an r-genesis message.
- Process p_j then enters view $(r, 1)$ (the 1^{st} view of the r^{th} execution of the recovery procedure).

To form an appropriate proposal σ for $\mathtt{log}_{G_{r+1}}$ while in view (r, v), the leader p_i of the view waits for $2\Delta^*$ after entering the view (to accommodate possible lags between the progress of and information received by different correct processes), and then proceeds as follows. If M is the set of r-genesis messages that p_i has received by that time and which are signed by processes in $\Pi_r - F$, then let M' be a maximal subset of M that contains at most one message signed by each process. We say σ is extended by the r-genesis message (gen, σ', r) if $\sigma \preceq \sigma'$. Process p_i then sets σ so that the following condition is satisfied:

$\dagger(M', \sigma)$: σ is the longest sequence extended by more than $\frac{1}{2}|\Pi_r - F|$ elements of M'.

Process p_i then sends M' along with σ as a *justification* for its proposal. A correct process p_j will be prepared to vote on the proposal if $\dagger(M', \sigma)$ is satisfied and M' includes messages from every member of $P_j(r)$.

To see that this achieves the desired outcome, note that if p_j is correct and M' includes messages from every member of $P_j(r)$, then it must contain a message from every correct process. If σ' has been strongly finalized by some correct process, then every correct process must have finalized σ' before entering the recovery procedure, and cannot have subsequently finalized any value incompatible with σ'. So, for each r-genesis message $(\text{gen}, \sigma'', r)$ sent by a correct process, σ'' must extend σ'. It therefore holds that σ' is extended by more than $\frac{1}{2}|\Pi_r - F|$ elements of M', so that, if $\dagger(M', \sigma)$ is satisfied, σ must extend σ'.

5 The formal specification of the wrapper

In what follows, we suppose that, when a correct process sends a message to "all processes", it regards that message as immediately received by itself. The pseudocode uses a number of inputs, local variables, functions and procedures, detailed below.

Inputs. The wrapper takes as input an SMR protocol $\mathcal{P}$, a process set Π, a random permutation Π^* of Π, a value $\mathtt{log}_G$, and message delay bounds Δ^* and Δ. The consistency resilience ρ_C of $\mathcal{P}$ is also given as input. Recall that the given protocol $\mathcal{P}$ satisfies consistency and liveness with respect to a finalization function that may depend on the process set Π' and the value $\mathtt{log}'_G$ for the genesis log. (For example, signatures from a certain fraction of the processes in Π' may be required for transaction finalization.) We write $\mathcal{F}(\Pi', \mathtt{log}'_G)$ to denote this function, and suppose also that this function is known to the protocol.

Permutations and the variables Π_r. Process p_i maintains a variable Π_r for each $r \in \mathbb{N}_{\geq 1}$. Π_1 is initially set to Π, while each Π_r for $r > 1$ is initially undefined.[12] Once Π_r is defined, Π^*_r is the permutation of Π_r induced by Π^*.

Views and leaders. Views are indexed by ordered pairs and ordered lexicographically: one should think of view (r, v) as the v^{th} view in the r^{th} execution of the recovery procedure. For $r, v \in \mathbb{N}_{\geq 1}$, we set $\mathtt{lead}(r, v) = p_i$, where $p_i = \Pi^*_r(v)$; this function is used to specify the leader of each view.[13]

[12] We write $x \uparrow$ to indicate that the variable x is undefined, and $x \downarrow$ to indicate that x is defined.

[13] We can write $p_i = \Pi^*_r(v)$ because the number of views in the r^{th} execution of the recovery procedure will be bounded by $|\Pi_r|$.

Received messages and executions. We let $\mathtt{M}_i$ be a local variable that specifies the set of all messages so far received by p_i. The wrapper will also initiate executions $\mathcal{E}_1, \mathcal{E}_2, \ldots$ of $\mathcal{P}$: for each $r \geq 1$, $\mathtt{M}_{i,r}$ specifies all messages so far received by p_i in execution $\mathcal{E}_r$. We suppose that messages have tags identifying the execution in which they are sent, and that all messages received by p_i in $\mathcal{E}_r$ are also received by p_i in the present execution of the wrapper, so that $\mathtt{M}_{i,r} \subseteq \mathtt{M}_i$ for all r.

The variables $\mathtt{log}_{G_r}$. Process p_i maintains a variable $\mathtt{log}_{G_r}$ for each $r \in \mathbb{N}_{\geq 1}$. Initially, $\mathtt{log}_{G_1}$ is set to $\mathtt{log}_G$, while each $\mathtt{log}_{G_r}$ for $r > 1$ is undefined. If the execution $\mathcal{E}_r$ of $\mathcal{P}$ is initiated by the wrapper, then this will be an execution with $\mathtt{log}_{G_r}$ as the genesis log and with process set Π_r.

Logs. Process p_i maintains two variables $\mathtt{log}_i$ and $\mathtt{log}_i^*$. The former should be thought of as the sequence of transactions that p_i has finalized, while the latter is the sequence that p_i has strongly finalized.

Signatures. We write m_{p_i} to denote the message m signed by p_i.

***r*-*genesis* messages.** An *r-genesis* message is a message of the form $(\text{gen}, \sigma, r)_{p_j}$, where σ is a sequence of transactions and $p_j \in \Pi$. These are used during the r^{th} execution of the recovery procedure to help reach consensus on an appropriate value for $\mathtt{log}_{G_{r+1}}$. We say σ' is extended by the r-genesis message $(\text{gen}, \sigma, r)_{p_j}$ if $\sigma' \preceq \sigma$.

***r*-proposals.** An *r-proposal* is a tuple $P = (F, \sigma, M, r)$, where $F \subset \Pi$, σ is a sequence of transactions, M is a set of r-genesis messages, and $r \in \mathbb{N}_{\geq 1}$. The last entry r indicates that this is a proposal corresponding to the r^{th} execution of the recovery procedure. One should think of F as a suggestion for $\Pi_r - \Pi_{r+1}$, while σ is a suggestion for $\mathtt{log}_{G_{r+1}}$ and M is a *justification* for σ.

(r, v)-proposals. An (r, v)-proposal is a message of the form $R = (P, v, Q)_{p_j}$, where P is an r-proposal, $p_j \in \Pi$, and either $Q = \perp$ or else Q is a QC (as specified below) for some (r, v')-proposal with $v' < v$.

Votes. A *vote* for the (r, v)-proposal $R = (P, v, Q)_{p_j}$, where $P = (F, \sigma, M, r)$, is a message of the form $V = R_{p_k}$, where $p_k \in \Pi$. We also say V is a *vote by* p_k. At timeslot t, p_i will regard V as *valid* if it is signed by one of the processes that, from p_i's perspective, remains in the active process set – i.e., if Π_r is defined and $p_k \in \Pi_r - F$.

QCs. A QC for an (r, v)-proposal $R = (P, v, Q')_{p_j}$, where $P = (F, \sigma, M, r)$, is a set Q of votes for R. At timeslot t, p_i will regard Q as valid if every vote in Q is valid and Q contains more than $\frac{1}{2}|\Pi_r - F|$ votes, each by a different process. If Q is a QC for an (r, v)-proposal $R = (P, v, Q')_{p_j}$, we set $\mathtt{view}(Q) = (r, v)$ and $\mathtt{P}(Q) = P$, and we may also just refer to Q as a QC.

Locks. Each process p_i maintains a value Q_i^+, which is initially undefined. This variable should be thought of as playing the same role as locks in Tendermint. The variable Q_i^+ may be set to a valid QC for an (r, v)-proposal during view (r, v).

The variables $P_i(r)$ and t_0. Process p_i maintains a local variable $P_i(r)$ for each $r \geq 1$, initially undefined. Upon halting execution $\mathcal{E}_r$ and entering the recovery procedure at timeslot t (according to its local clock), p_i will set $t_0 := t$, wait $2\Delta^*$, and then set $P_i(r)$ to be the set of processes in Π_r from which it has received signed r-genesis messages.

The time for each view. Each view is of length $8\Delta^*$. Having set t_0 upon halting execution $\mathcal{E}_r$ and entering the recovery procedure, p_i will start view (r, v) (for $v \geq 1$) at time $t_0 + 2\Delta^* + 8(v-1)\Delta^*$.

Detecting equivocation. At timeslot t, we say p_i *detects equivocation in view* (r, v) if $\mathtt{M}_i$ contains at least two distinct (r, v)-proposals signed by $\mathtt{lead}(r, v)$.[14]

Valid (r, v)-proposals. Consider an (r, v)-proposal $R = (P, v, Q)_{p_j}$, where $P = (F, \sigma, M, r)$. At timeslot t (according to p_i's local clock), process p_i will regard R as valid if:

- **(i)** Π_r and $\mathtt{log}_{G_r}$ are defined;
- **(ii)** $F \subset \Pi_r$, and $|F| \geq \rho_C |\Pi_r|$;
- **(iii)** $\mathtt{M}_i$ is a $(\mathcal{P}, \Pi_r, \mathtt{log}_{G_r})$-proof of guilt for every process in F;
- **(iv)** M is a set of r-genesis messages, each signed by a different process in $\Pi_r - F$;
- **(v)** For each $p_k \in P_i(r)$, there exists an r-genesis message signed by p_k in M;
- **(vi)** σ is the longest sequence extended by more than $\frac{1}{2}|\Pi_r - F|$ elements of M;
- **(vii)** $p_j = \mathtt{lead}(r, v)$;
- **(viii)** Q_i^+ is undefined, or Q is a valid QC with (a) $\mathtt{view}(Q) \geq \mathtt{view}(Q_i^+)$, and (b) $\mathtt{P}(Q) = P$, and;
- **(ix)** p_i does not detect equivocation in view (r, v).

The local variables `voted` and `lockset`. For each (r, v), $\mathtt{voted}(r, v)$ and $\mathtt{lockset}(r, v)$ are initially set to 0. These values are used to indicate whether p_i has yet voted or set its lock during view (r, v).

***r*-finish votes and QCs.** An *r-finish vote for P* is a message of the form P_{p_j}, where $P = (F, \sigma, M, r)$ is an r-proposal and $p_j \in \Pi$. At timeslot t, p_i will regard the r-finish vote as valid if Π_r is defined and $p_j \in \Pi_r - F$. A *valid finish-QC* for P is a set of more than $\frac{1}{2}|\Pi_r - F|$ valid r-finish votes for P, each signed by a different process.

The procedure `Makeproposal`. If $p_i = \mathtt{lead}(r, v)$, then it will run this procedure during view (r, v). To carry out the procedure, p_i checks to see whether there exists some greatest $v' < v$ such that it has received a valid QC, Q say, with $\mathtt{view}(Q) = (r, v')$. If so, then p_i sends the (r, v)-proposal $R = (\mathtt{P}(Q), v, Q)_{p_i}$ to all processes. If not, then it sets F to be the set of all processes $p_j \in \Pi_r$ such that $\mathtt{M}_i$ is a $(\mathcal{P}, \Pi_r, \mathtt{log}_{G_r})$-proof of guilt for p_j. Let M be the set of r-genesis messages that p_i has received and which are signed by processes in $\Pi_r - F$, and let M' be a maximal subset of M that contains at most one message signed by each process. Process p_i then sets σ to be the longest sequence extended by more than $\frac{1}{2}|\Pi_r - F|$ elements of M' and sends to all processes the (r, v)-proposal $R = (P, v, \perp)_{p_i}$, where $P = (F, \sigma, M', r)$.

Message gossiping. We adopt the message gossiping conventions described in Section 2.

[14] If $\mathtt{M}_i$ contains a vote for an (r, v)-proposal, we consider it as also containing that (r, v)-proposal.

The function $\mathcal{F}$. While the function $\mathcal{F}$ is not explicitly used in the pseudocode, we will show in Section 7 that, at every t, $\texttt{log}_i = \mathcal{F}(\texttt{M}_i)$ (where $\texttt{log}_i$ and $\texttt{M}_i$ are as locally defined for p_i at t). The function $\mathcal{F}$ is specified in Algorithm 2.

Pseudocode walk-through. The pseudocode appears in Algorithm 1. Below, we summarise the function of each section of code.

Line 14. This line starts the execution of the wrapper by initiating $\mathcal{E}_1$, the first execution of $\mathcal{P}$, which has process set $\Pi_1 = \Pi$ and $\texttt{log}_{G_1} = \texttt{log}_G$ as the genesis log.

Lines 16 – 19. During the r^{th} execution of $\mathcal{P}$, these lines check whether the recovery procedure should be triggered. If so, then p_i disseminates an r-genesis message, temporarily resets its log, and starts the recovery procedure.

Lines 21 – 25. During the r^{th} execution of $\mathcal{P}$, these lines check whether p_i should extend its finalized and strongly finalized logs.

Lines 28 – 30. These lines initialize the r^{th} execution of the recovery procedure by setting t_0 and $P_i(r)$.

Lines 32 – 41. These lines specify the instructions for view (r, v). Initially, the leader waits $2\Delta^*$ and then makes an (r, v)-proposal. Processes vote upon receiving a first valid (r, v)-proposal. Upon receiving a first valid QC for an (r, v)-proposal, Q say, p_i sets its lock to Q and then waits $2\Delta^*$. If, at this time, it still does not detect equivocation in view (r, v), then it sends a finish vote for $\texttt{P}(Q)$.

Lines 43 – 47. These lines determine when p_i stops carrying out the r^{th} execution of the recovery procedure. This happens when p_i receives a valid finish-QC for some r-proposal P. The r-proposal P then specifies Π_{r+1} and $\texttt{log}_{G_{r+1}}$.

Informal discussion: how does the recovery procedure ensure consensus? To establish that at most one r-proposal can receive a valid finish-QC, suppose that some correct p_i sends a finish vote for the r-proposal P during view (r, v). In this case, p_i must set its lock to some valid QC, Q say, at some timeslot t while in view v. Suppose that Q is a QC for the (r, v)-proposal R, and note that $\texttt{P}(Q) = P$. We will observe that:

1. All correct processes set their locks to some valid QC for R while in view v.
2. No (r, v)-proposal other than R can receive a QC that is regarded as valid by any correct process.

From (1) and (2) it will be easy to argue by induction on $v' > v$ that no correct process votes for any proposal $R' = (P', v', Q')_{p_j}$ such that $P' \neq P$, since their locks will forever prevent voting for such proposals. It follows that if any correct p_k sends a finish vote for an r-proposal P' during some view $v' \geq v$, then $P = P'$. We conclude that, assuming (1) and (2), at most one r-proposal can receive a valid finish-QC.

To see that (1) holds, note that all correct processes will be in view (r, v) at $t + \Delta^*$ and will have received Q by this time. They will therefore set their lock to be some QC for R, unless they have already received a valid QC for some (r, v)-proposal $R' \neq R$. The latter case is not possible, since then p_i would detect equivocation in view (r, v) by $t + 2\Delta^*$, and so would not send the finish vote for P.

Algorithm 1 The instructions for p_i.

Local variables
$\mathtt{r}$, initially 1. ▷ Number of executions of $\mathcal{P}$ initiated
$\mathtt{rec}$, initially 0. ▷ 1 if carrying out recovery
$\mathtt{log}_i$, $\mathtt{log}_i^*$, initially set to $\mathtt{log}_G$ ▷ Finalized and strongly finalized transactions
Π_r, $r \geq 1$. Initially, $\Pi_1 = \Pi$, while $\Pi_r \uparrow$ for $r > 1$. ▷ Process set for $\mathcal{E}_r$
$\mathtt{log}_{G_r}$. Initially, $\mathtt{log}_{G_1} = \mathtt{log}_G$, while $\mathtt{log}_{G_r} \uparrow$ for $r > 1$. ▷ Genesis log for $\mathcal{E}_r$
$\mathtt{M}_i$, $\mathtt{M}_{i,r}$, initially empty. ▷ As specified in Section 5
Q_i^+, initially undefined. ▷ The lock
t_0, initially undefined. ▷ Timeslot at start of recovery
$P_i(r)$, initially undefined. ▷ A set of processes
$\mathtt{voted}(r,v)$, $\mathtt{lockset}(r,v)$ $(r,v \geq 1)$, initially 0. ▷ As specified in Section 5

At timeslot t**:**
  If $t = 0$, start execution $\mathcal{E}_1$ of $\mathcal{P}$, with process set Π_1 and with $\mathtt{log}_{G_1}$ as genesis log;

  If $\mathtt{rec} = 0$:
    If $\mathtt{M}_{i,\mathtt{r}}$ has a consistency violation w.r.t. $\mathcal{F}(\Pi_\mathtt{r}, \mathtt{log}_{G_\mathtt{r}})$:
      Send $(\mathrm{gen}, \mathtt{log}_i, \mathtt{r})_{p_i}$ to all processes; ▷ Send $\mathtt{r}$-genesis message
      Set $\mathtt{log}_{p_i} := \mathtt{log}_{G_\mathtt{r}}$; Stop running $\mathcal{E}_\mathtt{r}$; Set $\mathtt{rec} := 1$; ▷ Start recovery

  If $\mathtt{rec} = 0$:
    If $\exists \sigma, M$ s.t. $\sigma \succ \mathtt{log}_i$ and $M \subseteq \mathtt{M}_{i,\mathtt{r}}$ is an $\mathcal{F}(\Pi_\mathtt{r}, \mathtt{log}_{G_\mathtt{r}})$-certificate for σ;
      Let σ be the longest such; Set $\mathtt{log}_i := \sigma$; ▷ Extend log
    If there exists a longest $\sigma \succ \mathtt{log}_i^*$ s.t. $\mathtt{log}_i$ has extended σ for time $2\Delta^*$:
      Set $\mathtt{log}_i^* := \sigma$; ▷ Extend strongly finalized log

  If $\mathtt{rec} = 1$:
    If $t_0 \uparrow$, set $t_0 := t$; ▷ Set t_0 upon entering recovery
    If $t = t_0 + 2\Delta^*$: ▷ Set $P_i(\mathtt{r})$
      Set $P_i(\mathtt{r}) := \{p_j \in \Pi_\mathtt{r} :$ $\mathtt{M}_i$ contains an $\mathtt{r}$-genesis message signed by $p_j\}$;

    If $t = t_0 + 4\Delta^* + 8(v-1)\Delta^*$ (for some $v \in \mathbb{N}_{\geq 1}$) **and** $p_i = \mathtt{lead}(\mathtt{r}, v)$:
      $\mathtt{Makeproposal}$; ▷ Leader makes new proposal $2\Delta^*$ after starting view
    If $t \in [t_0 + 2\Delta^* + 8(v-1)\Delta^*, t_0 + 2\Delta^* + 8v\Delta^*)$ (for some $v \in \mathbb{N}_{\geq 1}$):
      If $\mathtt{voted}(\mathtt{r}, v) = 0$ **and** $\mathtt{M}_i$ contains a valid $(\mathtt{r}, v)$-proposal R:
        Send R_{p_i} to all processes; Set $\mathtt{voted}(\mathtt{r}, v) := 1$; ▷ Vote
      If $\mathtt{lockset}(\mathtt{r}, v) = 0$ **and** $\mathtt{M}_i$ contains a valid QC for an $(\mathtt{r}, v)$-proposal, Q say:
        Set $Q_i^+ := Q$, $\mathtt{lockset}(\mathtt{r}, v) := 1$; ▷ Set lock
        Set the $(\mathtt{r}, v)$-timer to expire in time $2\Delta^*$;
      If $(\mathtt{r}, v)$-timer expires and p_i does not detect equivocation in view $(\mathtt{r}, v)$:
        Send $\mathtt{P}(Q_i^+)_{p_i}$ to all processes; ▷ Send finish vote

    If $\mathtt{M}_i$ contains a valid finish-QC for some $P = (F, \sigma, M, \mathtt{r})$:
      Set $\Pi_{\mathtt{r}+1} := \Pi_\mathtt{r} - F$, $\mathtt{log}_{G_{\mathtt{r}+1}} := \sigma$; ▷ Start new execution of $\mathcal{P}$
      Set $\mathtt{r} := \mathtt{r} + 1$ and make t_0 and Q_i^+ undefined;
      Start execution $\mathcal{E}_\mathtt{r}$ of $\mathcal{P}$, with process set $\Pi_\mathtt{r}$ and with $\mathtt{log}_{G_\mathtt{r}}$ as genesis log;
      Set $\mathtt{rec} := 0$; Set $\mathtt{log}_i := \mathtt{log}_{G_\mathtt{r}}$;

Algorithm 2 The function $\mathcal{F}$.

Inputs
M ▷ A set of messages
Π, $\mathtt{log}_G$ ▷ Process set and genesis log
$\mathcal{F}(\Pi', \mathtt{log}'_G)$ ▷ A function for each possible Π' and $\mathtt{log}'_G$
Local variables
$\mathtt{r}$, initially 1.
Π_r, $r \geq 1$. Initially, $\Pi_1 = \Pi$, while $\Pi_r \uparrow$ for $r > 1$.
$\mathtt{log}_{G_r}$. Initially, $\mathtt{log}_{G_1} = \mathtt{log}_G$, while $\mathtt{log}_{G_r} \uparrow$ for $r > 1$.
$\mathtt{end}$, initially 0

While $\mathtt{end} = 0$ **do**:
  If M does not have a consistency violation w.r.t. $\mathcal{F}(\Pi_{\mathtt{r}}, \mathtt{log}_{G_{\mathtt{r}}})$:
    Let σ be longest such that M is an $\mathcal{F}(\Pi_{\mathtt{r}}, \mathtt{log}_{G_{\mathtt{r}}})$-certificate for σ;
    Return σ; Set $\mathtt{end} := 1$;
  Else if there does not exist a unique $\mathtt{r}$-proposal with a valid finish-QC in M:
    Return $\mathtt{log}_{G_{\mathtt{r}}}$; Set $\mathtt{end} := 1$;
  Else if there exists a unique $\mathtt{r}$-proposal $P = (F, \sigma, M', \mathtt{r})$ with a valid finish-QC in M:
    Set $\Pi_{\mathtt{r}+1} := \Pi_{\mathtt{r}} - F$, $\mathtt{log}_{G_{\mathtt{r}+1}} = \sigma$;
    Set $\mathtt{r} := \mathtt{r} + 1$;

To see that (2) holds, the argument is similar. All correct processes will be in view (r, v) at $t + \Delta^*$ and will have received R by this time. Item (ix) in the validity conditions for (r, v)-proposals prevents correct processes from voting for (r, v)-proposals $R' \neq R$ at later timeslots, and correct processes cannot vote for such proposals at any timeslot $\leq t + \Delta^*$ because p_i would detect equivocation in view (r, v) in this case.

Having established that at most one r-proposal can receive a valid finish-QC, suppose now, towards a contradiction, that no r-proposal ever receives a valid finish-QC. Let v be the least such that $\mathtt{lead}(r, v)$ is correct and let i be such that $p_i = \mathtt{lead}(r, v)$. Since p_i waits $2\Delta^*$, until some timeslot t say, before disseminating an (r, v)-proposal $R = (P, v, Q)_{p_i}$, it will have seen all locks held by correct processes by this time, and will have received r-genesis messages from all processes in any set $P_j(r)$ for correct p_j. At t, p_i will disseminate an (r, v)-proposal which all correct processes regard as valid by timeslot $t + \Delta^*$. All correct processes will therefore vote for the proposal by this time and will receive a valid QC for the proposal by time $t + 2\Delta^*$. All correct processes will then set their locks. They will still be in view (r, v) by time $t + 4\Delta^*$ (since they enter the view within time Δ^* of each other) and will send r-finish votes for P by this time.

6 The theorem statements

Given functions $g, g' : \mathbb{N} \to \mathbb{R}$, we say $g \leq g'$ if $g(r) \leq g'(r)$ for all $r \in \mathbb{N}$. If $x \in \mathbb{R}$, we say $g \leq x$ if $g(r) \leq x$ for all $r \in \mathbb{N}$. We say $g < g'$ if $g \leq g'$ and $g(r) < g'(r)$ for some n.

We begin with our main positive result, which states the guarantees our wrapper achieves for recoverable consistency and liveness, worst-case and probabilistic recovery time, and rollback. The proof of Theorem 1 is given in Section 7.

▶ **Theorem 1.** *Suppose the wrapper is given an accountable SMR protocol $\mathcal{P}$ as input, where $\mathcal{P}$ has consistency resilience ρ_C and liveness resilience ρ_L in partial synchrony, such that $\rho_C + 2\rho_L = 1$. Let g_1 and g_2 be as defined in expression (1) in Section 3. If $\mathcal{P}$ has liveness parameter ℓ and is accountable for $(1-\rho_L)$-bounded adversaries, then the wrapper produces a protocol with the same consistency and liveness resilience as $\mathcal{P}$ in partial synchrony, and with the following properties for $(1-\rho_L)$-bounded adversaries when message delays are bounded by Δ^*:*

- **(i)** *Recoverable consistency resilience $\geq g_1$ and recoverable liveness resilience $\geq g_2$.*
- **(ii)** *Recovery time $O(f_a\Delta^*)$ with liveness parameter ℓ, where f_a is the actual (unknown) number of faulty processes.*
- **(iii)** *Probabilistic recovery time $O(\Delta^* log\frac{1}{\varepsilon})$ with liveness parameter ℓ.*
- **(iv)** *Rollback bounded by $2\Delta^*$.*

The next three results describe senses in which Theorem 1 is tight. We say a protocol has *bounded rollback* if there exists some h such that the protocol has rollback bounded by h. Our first impossibility result states that the rollback of a protocol must scale with Δ^*, and hence bounded rollback in the partially synchronous setting is impossible.

▶ **Theorem 2** (Impossibility result 1)**.** *Suppose partial synchrony holds w.r.t. Δ and synchrony holds w.r.t. Δ^*. Suppose $\mathcal{P}$ is a protocol for SMR with liveness resilience ρ_L, consistency resilience $\rho_C \geq \rho_L$, liveness parameter ℓ, and with rollback bounded by h. If we are given only that the adversary is ρ-bounded for $\rho > 1 - 2\rho_L$, then $h = \Omega(\Delta^*)$. In particular, $\mathcal{P}$ does not have bounded rollback in the pure partially synchronous setting.*

The proof of Theorem 2 is given in the full online version of the paper. Our second impossibility result justifies our restriction to $(1-\rho_L)$-bounded adversaries: with a larger adversary, bounded rollback is impossible (even in the synchronous setting).

▶ **Theorem 3** (Impossibility result 2)**.** *Consider the synchronous setting and suppose $\mathcal{P}$ is a protocol for SMR with liveness resilience ρ_L and consistency resilience $\rho_C \geq \rho_L$. If we are given only that the adversary is ρ-bounded for $\rho > 1 - \rho_L$, then $\mathcal{P}$ does not have bounded rollback. (The same result also holds in partial synchrony.)*

The proof of Theorem 3 is given in the full online version of the paper. Our final impossibility result shows that the recoverable consistency and liveness functions g_1 and g_2 in Theorem 1 cannot be improved upon, giving an analog of the "$\rho_C + 2\rho_L \leq 1$" constraint for all positive values of r.

▶ **Theorem 4** (Impossibility result 3)**.** *Given ρ_C and ρ_L such that $\rho_C + 2\rho_L = 1$, let g_1 and g_2 be as defined in Section 3. Suppose $g_1', g_2' \leq 1 - \rho_L$ and that $\mathcal{P}$ is an SMR protocol for partial synchrony with recoverable consistency resilience $\geq g_1'$ and recoverable liveness resilience $\geq g_2'$ when message delays are bounded by Δ^*. Suppose that, for some d and ℓ, $\mathcal{P}$ has recovery time d with liveness parameter ℓ when the adversary is $1-\rho_L$-bounded. Then:*

1. *If $g_2' \geq g_2$, then $g_1' \leq g_1$, and;*
2. *If $g_2' > g_2$, then $g_1' < g_1$.*

The proof of Theorem 4 is given in the full online version of the paper.

7 The proof of Theorem 1

We assume throughout this section that the adversary is $(1-\rho_L)$-bounded and that message delays are bounded by Δ^*.

Some further terminology. We make the following definitions:

- Process p_i begins the r^{th} execution of the recovery procedure at the first timeslot at which $\mathtt{r} = r$ and $\mathtt{rec} = 1$ (where those values are as locally defined for p_i).
- The r^{th} execution of the recovery procedure begins at the first timeslot at which some correct process begins the r^{th} execution of the recovery procedure.
- Execution $\mathcal{E}_r$ begins at the first timeslot at which some correct process begins execution $\mathcal{E}_r$. If $r > 1$, then the $(r-1)^{\text{th}}$ execution of the recovery procedure also ends at this timeslot.
- If a QC/finish-QC is regarded as valid by all correct processes, we refer to it as a valid QC/finish-QC.

▶ **Lemma 5.** *If the r^{th} execution of the recovery procedure begins at t_0, then:*

(i) *All correct processes begin the r^{th} execution of the recovery procedure by time $t_0 + \Delta^*$.*

(ii) *There exists a unique r-proposal, P say, that receives a finish-QC that is regarded as valid by some correct process.*

(iii) *If v_0 is least such that $\mathtt{lead}(r, v_0)$ is correct, all correct processes receive a valid finish-QC for P by time $t_0 + 2\Delta^* + 8v_0\Delta^*$.*

(iv) *All correct processes begin execution $\mathcal{E}_{r+1}$ within time Δ^* of each other and with the same local values for Π_{r+1} and $\mathtt{log}_{G_{r+1}}$.*

The proof of Lemma 5 is given in the full online version of the paper.

Further notation. Given statement (iv) of Lemma 5, each value Π_r or $\mathtt{log}_{G_r}$ is either undefined at all timeslots for all correct processes, or else is eventually defined and takes the same value for each correct process. We may therefore write Π_r and $\mathtt{log}_{G_r}$ to denote these globally agreed values.

▶ **Lemma 6.** *If p_i is correct, then, at the end of every timeslot, $\mathtt{log}_i = \mathcal{F}(\mathtt{M}_i)$.*

Proof. Let $\mathtt{rec}$, $\mathtt{r}$, $\mathtt{M}_i$, $\mathtt{M}_{i,r}$ and $\mathtt{log}_i$ be as locally defined for p_i. Consider first the case that $\mathtt{rec} = 0$ at the end of timeslot t. In this case, $\mathtt{M}_i$ has a consistency violation with respect to $\mathcal{F}(\Pi_r, \mathtt{log}_{G_r})$ for each $r < \mathtt{r}$ but does not have a consistency violation with respect to $\mathcal{F}(\Pi_\mathtt{r}, \mathtt{log}_{G_\mathtt{r}})$. Also, $\mathtt{M}_i$ contains a valid finish-QC for some r-proposal for each $r < \mathtt{r}$, which must be unique by Lemma 5. At the end of timeslot t, $\mathtt{log}_i$ is the longest string σ such that $\mathtt{M}_i$ (and $\mathtt{M}_{i,\mathtt{r}}$) is an $\mathcal{F}(\Pi_\mathtt{r}, \mathtt{log}_{G_\mathtt{r}})$-certificate for σ. The iteration defining $\mathcal{F}$ in Algorithm 2 will not return a value until it has defined all values Π_r and $\mathtt{log}_{G_r}$ for $r \leq \mathtt{r}$. Upon discovering that $\mathtt{M}_i$ does not have a consistency violation with respect to $\mathcal{F}(\Pi_\mathtt{r}, \mathtt{log}_{G_\mathtt{r}})$, it will return the same value σ, as the longest string for which $\mathtt{M}_i$ is an $\mathcal{F}(\Pi_\mathtt{r}, \mathtt{log}_{G_\mathtt{r}})$-certificate.

Next, consider the case that $\mathtt{rec} = 1$ at the end of timeslot t. In this case, $\mathtt{M}_i$ has a consistency violation with respect to $\mathcal{F}(\Pi_r, \mathtt{log}_{G_r})$ for each $r \leq \mathtt{r}$, and also contains a valid finish-QC for some r-proposal for each $r < \mathtt{r}$, which must be unique by Lemma 5. However, $\mathtt{M}_i$ does not contain a valid finish-QC for any $\mathtt{r}$-proposal. At the end of timeslot t, $\mathtt{log}_i = \mathtt{log}_{G_\mathtt{r}}$. The iteration defining $\mathcal{F}$ will not return a value until it has defined all values Π_r and $\mathtt{log}_{G_r}$ for $r \leq \mathtt{r}$, and will then also return $\mathtt{log}_{G_\mathtt{r}}$. ◀

▶ **Lemma 7.** *The wrapper has rollback bounded by $2\Delta^*$. Also, $\mathtt{log}_{G_{r+1}} \succeq \mathtt{log}_{G_r}$ whenever $\mathtt{log}_{G_{r+1}} \downarrow$.*

Proof. We say p_i finalizes σ if it sets $\mathtt{log}_i$ to extend σ and that p_i strongly finalizes σ if it sets $\mathtt{log}_i^*$ to extend σ. Suppose p_i finalizes σ while running $\mathcal{E}_r$ at t because there exists $M \subseteq \mathtt{M}_{i,r}$ which is an $\mathcal{F}(\Pi_r, \mathtt{log}_{G_r})$-certificate for σ. By (iv) of Lemma 5, every correct process p_j will begin $\mathcal{E}_r$ by $t + \Delta^*$, and will receive the messages in M by that time. This means p_j will finalize σ, never to subsequently finalize any sequence incompatible with σ

while running $\mathcal{E}_r$, unless $\mathtt{M}_{j,r}$ has a consistency violation w.r.t. $\mathcal{F}(\Pi_r, \mathtt{log}_{G_r})$ by $t + \Delta^*$. In the latter case, p_i will begin the recovery procedure by timeslot $t + 2\Delta^*$ and will not strongly finalize σ at that time. We conclude that, if p_i strongly finalizes σ while running $\mathcal{E}_r$, then all correct processes finalize σ while running $\mathcal{E}_r$.

If p_i strongly finalizes σ while running $\mathcal{E}_r$ and if the r^{th} execution of the recovery procedure does not begin at any timeslot, it follows that, for all correct p_j, $\sigma \preceq \mathtt{log}_j$ thereafter. So, suppose that the r^{th} execution of the recovery procedure begins at some timeslot t_0. Note that, if p_j is correct, then it waits $2\Delta^*$ after beginning the r^{th} execution of the recovery procedure before defining $P_j(r)$. By Lemma 5, all correct processes begin the r^{th} execution of the recovery procedure within time Δ^* of each other. Since correct processes send r-genesis messages immediately upon beginning the recovery procedure, it follows that $P_j(r)$ includes all correct processes.

By Lemma 5, there exists a unique r-proposal, $P = (F, \sigma', M', r)$ say, that receives a valid finish-QC. For this to occur, there must exist v and an (r, v)-proposal $R = (P, v, \perp)$ (signed by $\mathtt{lead}(r, v)$) which receives a valid QC. This QC must include at least one vote by a correct process, p_j say. It follows that M' must contain r-genesis messages from every member of $P_j(r)$, and so from every correct process. As we noted previously, if p_i strongly finalizes σ while running $\mathcal{E}_r$, then every correct process must finalize σ before beginning the r^{th} execution of the recovery procedure. So, for each r-genesis message $(\text{gen}, \sigma'', r)$ sent by a correct process, σ'' must extend σ, and also extends $\mathtt{log}_{G_r}$. It therefore holds that σ (and $\mathtt{log}_{G_r}$) is extended by more than $\frac{1}{2}|\Pi_r - F|$ elements of M'. Recall that σ' is as specified by P. No correct process will vote for R unless σ' is the longest sequence extended by more than $\frac{1}{2}|\Pi_r - F|$ elements of M, meaning that σ' must extend σ and $\mathtt{log}_{G_r}$. So far, we conclude that $\mathtt{log}_{G_{r+1}}$ extends σ and $\mathtt{log}_{G_r}$. Since it follows by the same argument that $\mathtt{log}_{G_s}$ extends σ for all $s > r$, the claim of the lemma holds. ◀

▶ **Lemma 8.** *If an execution of the wrapper has r consistency violations, then the r^{th} execution of the recovery procedure must begin at some timeslot (and so, by Lemma 5 must also end at some timeslot).*

The proof of Lemma 8 is given in the full online version of the paper.

▶ **Lemma 9.** *The wrapper has recoverable consistency resilience $\geq g_1$ and also recoverable liveness resilience $\geq g_2$.*

Proof. Recall that, in Section 3, we set $x_0 = 0$ and $x_{r+1} = x_r + \rho_C(1 - x_r)$, and then defined:

$$g_1(r) = \min\{x_{r+1}, 1 - \rho_L\}, \quad g_2(r) = \min\{x_r + \rho_L(1 - x_r), 1 - \rho_L\}.$$

Note that x_r lower bounds the fraction of the processes removed to form Π_{r+1}, i.e. $|\Pi - \Pi_{r+1}| \geq x_r n$. By Lemma, 8, if there exist r consistency violations, then the r^{th} execution of the recovery procedure must end at some timeslot. If the adversary is $g_2(k)$-bounded, and since $\mathcal{P}$ has liveness resilience ρ_L, it follows that liveness must hold. If the adversary is $g_1(r)$-bounded, then since $\mathcal{P}$ has consistency resilience ρ_C, the $(r+1)^{\text{th}}$ execution of the recovery procedure cannot begin at any timeslot. From Lemma 8, it follows that there are at most r consistency violations. ◀

▶ **Lemma 10.** *The wrapper has recovery time $O(f_a\Delta^*)$ with liveness parameter ℓ, where f_a is the actual (unknown) number of faulty processes. It also has probabilistic recovery time $O(\Delta^* log\frac{1}{\varepsilon})$ with liveness parameter ℓ.*

Proof. The fact that the wrapper has recovery time $O(f_a\Delta^*)$ with liveness parameter ℓ follows directly from (iii) of Lemma 5, since views are of length $O(\Delta^*)$. To establish the claim regarding probabilistic recovery time, note that we required $\rho_C > 0$ in the definition of optimal resilience. Some finite power of $(1 - \rho_C)$ is therefore less than ρ_L, so there exists r such that any execution in which the adversary is $1 - \rho_L$-bounded can have most r consistency violations. If the adversary is ρ-bounded, then the probability that, for one of the (at most r) executions of the recovery procedure, the first d views all have faulty leaders is $O(r\rho^d) = O(\rho^d)$ for fixed ρ_C. Since each view is of length $O(\Delta^*)$, it follows from (iii) of Lemma 5 that the wrapper therefore has probabilistic recovery time $O(\Delta^* \log \frac{1}{\varepsilon})$ with liveness parameter ℓ, as claimed. ◀

8 Related Work

Positive results. A sequence of papers, including Buterin and Griffith [7], Civit et al. [10], and Shamis et al. [23], describe protocols that satisfy accountability. Sheng et al. [24] analyze accountability for well-known permissioned protocols such as HotStuff [29], PBFT [8], Tendermint [4, 5], and Algorand [9]. Civit et al. [12, 11] describe generic transformations that take any permissioned protocol designed for the partially synchronous setting and provide a corresponding accountable version. These papers do not describe how to reach consensus on which guilty parties to remove in the event of a consistency violation (i.e. how to achieve "recovery"), and thus fall short of our goals here. One exception to this point is the ZLB protocol of Ranchal-Pedrosa and Gramoli [21], but the ZLB protocol only achieves recovery if the adversary controls less than a 5/9 fraction of participants, and does not achieve bounded rollback. Freitas de Souza et al. [13] also describe a process for removing guilty parties in a protocol for lattice agreement (this abstraction is weaker than SMR/consensus and can be implemented in an asynchronous system), but their protocol assumes an honest majority and the paper does not consider bounded rollback. Sridhar et al. [26] specify a "gadget" that can be applied to blockchain protocols operating in the synchronous setting to reboot and maintain consistency after an attack, but they do not describe how to implement recovery and assume that an honest majority is somehow reestablished out-of-protocol.

Budish et al. [6] consider "slashing" in proof-of-stake protocols in the "quasi-permissionless" setting. Their main positive result is a protocol that, in the same timing model considered in this paper (with additional guarantees provided pre-GST message delays are bounded by a known parameter Δ^*), guarantees what they call the "EAAC property" – honest players never have their stake slashed, and some Byzantine stake is guaranteed to be slashed following a consistency violation. Budish et al. [6] do not contemplate repeated consistency violations, a prerequisite to the notions of recoverable consistency and liveness that are central to this paper. To the extent that it makes sense to compare their "recovery procedure" with our "wrapper," our protocol is superior in several respects, with worst-case recovery time $O(n\Delta^*)$ (as opposed to $O(n^2\Delta^*)$); probabilistic recovery time $O(\Delta^* \log \frac{1}{\varepsilon})$, where ε is an error-probability bound (as opposed to $O(n\Delta^* \log \frac{1}{\varepsilon})$); and rollback $2\Delta^*$ (as opposed to unbounded rollback).

Gong et al. [15] consider the task of recovery and achieve results that are incomparable to those in this paper since they consider different timing and fault models: they restrict to the case of "ABC faults"[15] but consider full partial synchrony, meaning that it is not possible to

[15] Processes subject to "ABC faults" can only display faulty behaviour so as to cause consistency failures, and not to threaten liveness.

achieve bounded rollback. Given this setup, their protocol satisfies the condition that after a single execution of the recovery procedure, a 5/9-bounded adversary is unable to cause further consistency and liveness violations, and after two executions of the recovery procedure a 2/3-bounded adversary is unable to cause further consistency and liveness violations. The authors also describe sufficient conditions for a general BFT SMR protocol to allow for "complete and sound fault detection" in the case of consistency violations, even when the actual (unknown) number of faulty processes is as large as $n - 2$.

Distinct from our aims here, i.e., the *removal* of guilty parties so as to achieve recovery in the event of a consistency violation, a number of papers consider the ability of protocols to recover from temporary dishonest majorities (without consideration of any mechanism to ensure that the dishonest majority is temporary). Avarikioti et al. [2] establish a formal sense in which Bitcoin is secure under temporary dishonest majority. We note that, since Bitcoin is dynamically available [17], it cannot be accountable [20], meaning that there can be no mechanism to remove guilty parties (and only guilty parties) in the event of a consistency violation. Badertscher et al. [3] extend this analysis to consider dynamically available proof-of-work and proof-of-stake protocols more generally and also establish negative results for BFT-style protocols that do not make use of accountability to remove guilty parties (as we do here).

Prior to the study of accountability, Li and Mazieres [18] considered how to design BFT protocols that still offer certain guarantees when more than f failures occur. The describe a protocol called BFT2F which has the same liveness and consistency guarantees as PBFT when no more than $f < n/3$ players fail; with more than f but no more than $2f$ failures, BFT2F prohibits malicious players from making up operations that clients have never issued and prevents certain kinds of consistency violations.

Negative results. There are a number of papers that describe negative results relating to accountability and the ability to punish guilty parties in the "permissionless setting" (for a definition of the permissionless setting see [17]). Neu et al. [20] prove that no protocol operating in the "dynamically available" setting (where the number of "active" parties is unknown) can provide accountability. The authors then provide an approach to addressing this limitation by describing a "gadget" that checkpoints a longest-chain protocol. The "full ledger" is then live in the dynamically available setting, while the checkpointed prefix ledger provides accountability. Tas et al. [28, 27] and Budish et al. [6] also prove negative results regarding the possibility of punishing guilty participants of proof-of-stake protocols before they are able to cash out of their position.

9 Final Comments

For the sake of simplicity, we have presented our wrapper in the "permissioned" setting (with a fixed and known set of always active participants). However, since the procedure produces certificates that suffice to verify each new genesis log and the set of processes removed after each consistency violation, it can also be applied directly to proof-of-stake protocols in the quasi-permissionless setting [17]. Specifically, our procedure can be used to give a practical replacement for the (proof-of-concept) recovery procedure of Budish, Lewis-Pye and Roughgarden [6] within the formal setup they consider.

While Theorems 2-3 show senses in which Theorem 1 is tight, a number of natural questions remain. For example, our recovery procedure implements a synchronous protocol and has recovery time $O(f_a\Delta^*)$. While Theorem 2 establishes that some bound on message delays is required if we are to achieve bounded rollback, one might still make use of a

recovery procedure that does not require synchrony: could such a procedure achieve recovery time $O(f_a\Delta)$ after GST? Also, while our recovery procedure has rollback bounded by $2\Delta^*$, Theorem 2 only establishes a lower bound of Δ^*. Is this lower bound tight, or is $2\Delta^*$ optimal?

References

1. Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 106–118. IEEE, 2020. doi:10.1109/SP40000.2020.00044.
2. Georgia Avarikioti, Lukas Käppeli, Yuyi Wang, and Roger Wattenhofer. Bitcoin security under temporary dishonest majority. In *International Conference on Financial Cryptography and Data Security*, pages 466–483. Springer, 2019. doi:10.1007/978-3-030-32101-7_28.
3. Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Consensus redux: distributed ledgers in the face of adversarial supremacy. In *2024 IEEE 37th Computer Security Foundations Symposium (CSF)*, pages 143–158. IEEE, 2024.
4. Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, University of Guelph, 2016.
5. Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus. *arXiv preprint*, 2018. arXiv:1807.04938.
6. Eric Budish, Andrew Lewis-Pye, and Tim Roughgarden. The economic limits of permissionless consensus. In *Proceedings of the 25th ACM Conference on Economics and Computation*, pages 704–731, 2024. doi:10.1145/3670865.3673548.
7. Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017. arXiv:1710.09437.
8. Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation OsDI*, pages 173–186, 1999.
9. Jing Chen, Sergey Gorbunov, Silvio Micali, and Georgios Vlachos. Algorand agreement: Super fast and partition resilient byzantine agreement. *IACR Cryptol. ePrint Arch.*, 2018:377, 2018.
10. Pierre Civit, Seth Gilbert, and Vincent Gramoli. Polygraph: Accountable byzantine agreement. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 403–413. IEEE, 2021. doi:10.1109/ICDCS51616.2021.00046.
11. Pierre Civit, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, and Jovan Komatovic. As easy as abc: Optimal (a) ccountable (b) yzantine (c) onsensus is easy! *Journal of Parallel and Distributed Computing*, 181:104743, 2023. doi:10.1016/J.JPDC.2023.104743.
12. Pierre Civit, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, Zarko Milosevic, and Adi Seredinschi. Crime and punishment in distributed byzantine decision tasks. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 34–44. IEEE, 2022. doi:10.1109/ICDCS54860.2022.00013.
13. Luciano Freitas de Souza, Petr Kuznetsov, Thibault Rieutord, and Sara Tucci-Piergiovanni. Accountability and reconfiguration: Self-healing lattice agreement. *arXiv preprint*, 2021. arXiv:2105.04909.
14. Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988. doi:10.1145/42282.42283.
15. Tiantian Gong, Gustavo Franco Camilo, Kartik Nayak, Andrew Lewis-Pye, and Aniket Kate. Recovery from excessive faults in partially synchronous bft smr, 2025. URL: https://eprint.iacr.org/2025/083.
16. Andrew Lewis-Pye and Tim Roughgarden. How does blockchain security dictate blockchain implementation? In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1006–1019, 2021. doi:10.1145/3460120.3484752.
17. Andrew Lewis-Pye and Tim Roughgarden. Permissionless consensus. *arXiv preprint arXiv:2304.14701*, 2023. doi:10.48550/arXiv.2304.14701.

18 Jinyuan Li and David Mazieres. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *NSDI*, pages 10–10, 2007.

19 Atsuki Momose and Ling Ren. Multi-threshold byzantine fault tolerance. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1686–1699, 2021. doi:10.1145/3460120.3484554.

20 Joachim Neu, Ertem Nusret Tas, and David Tse. The availability-accountability dilemma and its resolution via accountability gadgets. In *International Conference on Financial Cryptography and Data Security*, pages 541–559. Springer, 2022. doi:10.1007/978-3-031-18283-9_27.

21 Alejandro Ranchal-Pedrosa and Vincent Gramoli. Zlb: A blockchain to tolerate colluding majorities. *arXiv preprint*, 2020. arXiv:2007.10541.

22 Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *Acm Computing Surveys (CSUR)*, 22(4):299–319, 1990. doi:10.1145/98163.98167.

23 Alex Shamis, Peter Pietzuch, Burcu Canakci, Miguel Castro, Cédric Fournet, Edward Ashton, Amaury Chamayou, Sylvan Clebsch, Antoine Delignat-Lavaud, Matthew Kerner, et al. {IA-CCF}: Individual accountability for permissioned ledgers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 467–491, 2022.

24 Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. Bft protocol forensics. In *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, pages 1722–1743, 2021. doi:10.1145/3460120.3484566.

25 Srivatsan Sridhar, Ertem Nusret Tas, Joachim Neu, Dionysis Zindros, and David Tse. Consensus under adversary majority done right. *arXiv preprint*, 2024. arXiv:2411.01689.

26 Srivatsan Sridhar, Dionysis Zindros, and David Tse. Better safe than sorry: Recovering after adversarial majority. *arXiv preprint*, 2023. arXiv:2310.06338.

27 Ertem Nusret Tas, David Tse, Fangyu Gai, Sreeram Kannan, Mohammad Ali Maddah-Ali, and Fisher Yu. Bitcoin-enhanced proof-of-stake security: Possibilities and impossibilities. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 126–145. IEEE, 2023. doi:10.1109/SP46215.2023.10179426.

28 Ertem Nusret Tas, David Tse, Fisher Yu, and Sreeram Kannan. Babylon: Reusing bitcoin mining to enhance proof-of-stake security. *arXiv preprint*, 2022. arXiv:2201.07946.

29 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*, pages 347–356, 2019. doi:10.1145/3293611.3331591.