Permissionless Clock Synchronization with Public Setup

Juan Garay*[1], Aggelos Kiayias[2], and Yu Shen†[3]

[1]Texas A&M University, garay@cse.tamu.edu
[2]University of Edinburgh and IOHK, aggelos.kiayias@ed.ac.uk
[3]University of Edinburgh, yu.shen@ed.ac.uk

Abstract

The permissionless clock synchronization problem asks how it is possible for a population of parties to maintain a system-wide synchronized clock, while their participation rate fluctuates —possibly very widely— over time. The underlying assumption is that parties experience the passage of time with roughly the same speed, but however they may disengage and engage with the protocol following arbitrary (and even chosen adversarially) participation patterns. This (classical) problem has received renewed attention due to the advent of blockchain protocols, and recently it has been solved in the setting of proof of stake, i.e., when parties are assumed to have access to a trusted PKI setup [Badertscher *et al.*, Eurocrypt '21].

In this work, we present the first proof-of-work (PoW)-based permissionless clock synchronization protocol. Our construction assumes a public setup (e.g., a CRS) and relies on an honest majority of computational power that, for the first time, is described in a fine-grain timing model that does not utilize a global clock that exports the current time to all parties. As a secondary result of independent interest, our protocol gives rise to the first PoW-based ledger consensus protocol that does not rely on an external clock for the time-stamping of transactions and adjustment of the PoW difficulty.

*Research supported by NSF grants no. 2001082 and 2055694.
†Work supported by Input Output – IOHK through their funding of the Edinburgh Blockchain Technology Lab.

Contents

1 Introduction

In the classical clock synchronization problem, thoroughly studied over the past four decades by the distributed computing community—non-exhaustively, [Lam78, LMS84, HSSD84, DHS86, ST87, WL88, ADD$^+$19, LL22]—, a set of processors, each one possessing a timer that is within a bounded rate of drift from "nominal time" (the real time—called Newtonian time in [DHS86]), should realize logical clocks that are within a distance $\mathsf{Skew} \in \mathbb{N}$ of each other and within a linear envelope of nominal time. The typical threat model involves a subset of parties who deviate arbitrarily either from correct protocol execution or in terms of their clock speed and may as a result prevent synchronization from happening. A clock synchronization protocol has parties exchanging messages to suitably adjust their clocks so that the synchronization condition is achieved.

Up until the work of [BGK$^+$21] all prior work in clock synchronization assumed that the number of parties are known during the protocol execution (and available, unless they are assumed adversarial[1]). This standard assumption in Byzantine fault tolerance protocols was challenged first with the advent of the Bitcoin blockchain and related "permissionless" protocols. As exemplified in [GKL17, GKL20], the Bitcoin blockchain operates in a setting where the number of active parties may be unknown and continuously fluctuating throughout the protocol execution. While such results paved the way to rethink the problem of consensus in this setting (cf. [PS17b, GK20]), near perfectly synchronized clocks remained a central assumption in all previous security analyses of blockchain protocols (cf. [GKL15, BMTZ17, PSS17, GKL17, GKL20]).

In the setting where participation is dynamic and fluctuating over time, the adversary can introduce and remove honest parties at will without notifying the existing participants. As a result, existing clock synchronization algorithms (e.g., [ST87, ADD$^+$19, LL22]) do not directly translate to such permissionless setting because they fundamentally rely on the fact that the parties are aware of the number of parties as well as of the number of tolerated corruptions/faults—i.e., they are able to *count*—and a different protocol design technique is needed.

The main challenge in this transient participation setting shifts from correcting the bounded-rate drift occurring between the ever connected honest parties over time to the task of bringing up to sync freshly joining parties who start without any information about nominal time, while accommodating for the fact that a (possibly large) number of honest parties is no longer active. In [BGK$^+$21], assuming a so-called private-state setup [GK20] (specifically, a PKI), a protocol called "Ouroboros Chronos" is presented that can synthesize a notion of global time using a continuous flow of clock measurements that are provided by parties who only transiently participate in the protocol and their local clocks are assumed to be correct up to a bound. The level of participation fluctuates broadly with the only requirements that (i) it does not become negligible, and (ii) honest majority is preserved in terms of stake (all parties have a number of coins associated to their public keys that amount to their individual stake). Given this, their result leaves open the question of only utilizing a public(-state) setup.

To our knowledge, the only known result with a public setup in the permissionless setting, again from [BGK$^+$21], is that parties may use a Nakamoto-style longest chain blockchain without difficulty adjustment and use the block index to define a concept of global time. The obvious downside of this idea is that the protocol execution speeds up and slows down as participation fluctuates and, most importantly, it will be entirely insecure when there is a steady increase (or decrease) of participants, making the construction essentially only suitable for a static model where

[1]We note that the problem of joining parties in the context of clock synchronization was considered, but only conditionally on the new party agreed upon and approved by a sufficient number of participants; see [HSSD84].

the number of parties (i.e., the computational power invested in the system for proof of work) remains fixed.

This motivates the current work, where the following open question is being tackled:

> *Is it possible for a dynamically changing population of peers to synchronize their clocks utilizing only a public setup and assuming PoW?*

One apparent difficulty in answering this question is that using a blockchain protocol to derive consistency for clock adjustments runs into the complication that the blockchain protocol itself utilizes a clock to adjust the PoW difficulty at regular intervals. Indeed, the Bitcoin blockchain [Nak08] relies fundamentally on a global clock being available to all parties.[2] It follows that this observation suggests also a secondary open question that will be tackled as well:

> *Is there a blockchain protocol in the PoW setting that has no dependency on a publicly accessible global clock?*

1.1 Overview of Our Results

The clock synchronization problem asks parties to report clocks that satisfy two properties (cf. [DHS86]) (i) *bounded skew:* the parties maintain logical clocks whose difference is upper bounded, and (ii) *linear envelope synchronization:* the logical clock reported by a party is always within a *linear envelope* of the nominal time. Note that we are interested in a formulation of this problem in a very general setting where some parties are adversarial and hence deviate from the protocol arbitrarily, while honest parties may come and go following arbitrary participation patterns. Given this setting we formulate the *desideratum* of a synchronized clock only with respect to a class of parties we call *alert*, which are honest parties that have also been online for a sufficiently long time to catch up with all protocol messages. More formally, the clock synchronization problem is stated as follows.

Definition 1 (Clock Synchronization)**.** *There exist constants* $\mathsf{Skew} \in \mathbb{N}$*,* $\mathsf{shiftLB}, \mathsf{shiftUB} \in (0,1)$ *such that honest parties' logical clocks satisfy the following two properties:*

- ***Bounded skews.*** *Let* $\mathtt{r}_1, \mathtt{r}_2$ *be the reported logical clocks of two alert parties at any nominal time* r*. Then* $|\mathtt{r}_1 - \mathtt{r}_2| \leq \mathsf{Skew}$*.*
- ***Linear envelope synchronization.*** *Each alert party's logical clock stays in a* (U, L)*-linear envelope*[3] *with respect to the nominal time* r*, where* $U = 1/(1-\mathsf{shiftUB})$ *and* $L = 1/(1+\mathsf{shiftLB})$*.*

Solving the clock synchronization problem asks for a protocol that within a certain threat model achieves the two properties. This brings us to our first main result.

A model for permissionless clock synchronization in the PoW setting. Our model (Section 2) simultaneously facilitates (i) the dynamic participation of parties, (ii) imperfect local clocks, and (iii) resource bounding by restricting parties' queries to a random oracle (cf. [GKO$^+$20]). Specifically, we extend the previous model of the global imperfect clock of [BGK$^+$21] to the PoW setting by introducing a random oracle functionality that apportions random oracle queries per unit of time between the honest parties and the adversary in a manner consistent with an honest majority assumption in terms of computational power. The concept of time provided by the imperfect local clock functionality of [BGK$^+$21] enables parties to advance their local clocks and experience time

[2]The protocol implements such clock by having nodes querying other nodes in the network and possibly seeking user input — it has no way of deriving a clock from the protocol operation itself. See [GKL20] for more details.

[3]A function $f : \mathbb{R} \to \mathbb{R}$ is within a (U, L)-linear envelope if and only if it holds that $L \cdot x \leq f(x) \leq U \cdot x$, for all x.

at roughly the same speed (a maximum drift of Φ_{clock} is allowed). Note that the environment is allowed to introduce new parties and remove old parties at will, something that results in them being de-registered from the clock functionality; when this happens the clock functionality is not responsible for keeping them up to speed with the rest of the honest parties. In this way, parties can be seen as entirely transiently engaging with the protocol—each individual party may only engage for a small fraction of the total execution time as adaptively decided by the environment. Armed with our model, we then present our second main result.

A new protocol for permissionless clock synchronization in the PoW model. We describe our new PoW-based clock synchronization protocol Timekeeper in Section 3. The construction is based on three key ingredients: (i) A mechanism that repurposes the concept of 2-for-1 PoWs introduced in [GKL15] and subsequently used to achieve various properties such as fairness in [PS17a] and high throughput in [BKT$^+$19], to the setting of time-keeping by employing it to enable the collection of "timing beacons" from the active parties in a rolling window process; (ii) a PoW-based longest-chain type of blockchain that enables parties at regular intervals to reach consensus about the timing beacons that are shared and extract a suitable correction to their local clocks taking into account the arrivals of the beacons; and (iii) a novel target-recalculation function that can be thought of as the *reverse* of the one used in Bitcoin, that uses protocol recorded timestamps as a means to define the length of an epoch, and then uses the number of blocks produced in that period of time to adjust the PoW difficulty accordingly.

Putting these elements together, our clock synchronization protocol instructs parties when their local clock passes some specific moment (which happens periodically with respect to the interval length) to execute an adjustment on their local clock based on the median value of the beacon timestamps and their corresponding arrival time. Moreover, towards the goal of letting newly joining parties become synchronized with the protocol time, we present a joining procedure which requires the fresh parties to passively listen to the protocol execution for a while and then synchronize with other honest participants.

Based on the ledger consensus function offered by our protocol it is easy to derive also the following result.

A new PoW-based blockchain protocol without a global clock. Given that our protocol is a Nakamoto-style "most-difficult chain" type of protocol that faclitates clock synchronization, it is easy to transform it to a full-fledged blockchain protocol that admits transactions as in Bitcoin script or Ethereum smart contracts. The resulting blockchain has the novel property that it does not depend on accessing a globally available clock. Instead, parties utilize their local clocks which may be drifting or be out of sync, but thanks to the synchronization (sub-)protocol that is offered by our construction they can adjust their local clocks periodically. This eliminates time as an attack vector in the context of PoW-based blockchain protocols and demonstrates that it is possible to achieve ledger consensus using merely local clocks in a fully dynamic setting where parties may come and go adaptively per the adversary's instructions.

Security analysis. We present the full security analysis of Timekeeper in Section 4. As a high-level overview, we proceed to adapt the analytical toolset from [GKL17, GKL20] to the imperfect-local-clock model. Notably, we modify the concept of target recalculation epoch boundaries (from "point" to "zone") and the concept of isolated successes (which addresses the question of under what circumstances can a hash success guarantee the increase of accumulated difficulty). As an intermediate step, we study several predicates aiming at providing the "good" properties of an execution starting from the onset and until a given nominal round.

Our inductive-style proof works in the following manner. We prove that if at the onset, the PoW difficulty is appropriately set and the steady block-generation rate lasts during the whole clock synchronization interval, parties can maintain good skews after they enter the next interval and the shift value they compute to adjust their clocks is properly bounded. In addition, if good skews and certain time adjustment calculations are maintained during a target recalculation epoch, the block production rate will be properly controlled in the next epoch. To sum up, this guarantees that "good" properties can be achieved during the whole execution given a "safe" start and a bounded change in the number of parties (which can nevertheless still be exponential). We also provide an analysis of the joining procedure showing that joining parties starting with no *a-priori* knowledge of global time, can listen in and bootstrap their logical clock, turning themselves into *alert* parties being capable of fully engaging with the protocol.

In summary, Timekeeper solves the clock synchronization problem as defined above as follows.

Theorem (Theorem 18, informal). *Let Φ_{clock} be the maximum drift allowed on parties' local clocks and Δ the maximum (and unknown) message transmission delay. Then* Timekeeper *solves the clock synchronization problem assuming bounded dynamic participation and an honest majority in terms of random oracle queries, with parameter values*

$$\mathsf{Skew} = 2\Phi, \quad \mathsf{shiftLB} = 3\Phi/\mathrm{R}, \quad \mathsf{shiftUB} = 2\Phi/\mathrm{R},$$

where $\Phi = \Delta + \Phi_{\text{clock}}$ and $\mathrm{R} \in \mathbb{N}^+$ is a parameter chosen sufficiently large w.r.t. the security parameter and reflects the time required for an honest party to become alert.

Organization of the paper. The rest of the paper is organized as follows. In Section 2 we present our model, relevant definitions and building blocks. We describe our Timekeeper protocol in Section 3 and present the full analysis in Section 4. Detailed description of protocols, functionalities, and some of the proofs can be found in the appendix.

2 Model and Building Blocks

In this paper we adapt the timing and networking model of [BGK$^+$21] to the setting of proof of work, obviating the requirement for a PKI as a setup assumption. In more detail, in the model there is an upper bound Δ in message transmission (cf. [DLS88, PSS17, BMTZ17, GKL20]), and parties do not have access to a global clock, but instead rely on their local clocks, whose drift is assumed to be upper-bounded by Φ_{clock}. What complicates matters is that the model supports dynamic participation where parties may join and leave during the protocol execution without warning (it is worth noting here that this is where the difficulty of our setting is derived from: indeed if all honest parties were online throughout then it would be trivial to implement a logical clock by incrementing a counter). For succinctness, we choose to express primitives and building blocks (see below) in our execution model utilizing the ideal functionality language of [Can01], but we do not pursue a composability analysis for our security properties, which are expressed in a game based manner as in [GKL15, PSS17].

2.1 Imperfect Local Clocks

As in [BGK$^+$21], and as mentioned above, in this paper we remove the assumption that parties have access to a global clock, as in [GKL15, GKL17, PSS17, BMTZ17, GKL20], and instead assume *imperfect local clocks*. In a nutshell, every honest party maintains a local clock variable by communicating with an imperfect local clock functionality $\mathcal{F}_{\text{ILC}\textsc{lock}}$. In contrast to the global-setup clock functionality in [KMTZ13], where parties learn the exact global time and thus strong synchrony is guaranteed, parties registered with $\mathcal{F}_{\text{ILC}\textsc{lock}}$ will only receive "ticks" from the functionality to indicate that they should update their own clocks. In addition, $\mathcal{F}_{\text{ILC}\textsc{lock}}$ issues "imperfect" ticks, i.e., the adversary is allowed to set a bounded drift to each party by manipulating its corresponding status variable in $\mathcal{F}_{\text{ILC}\textsc{lock}}$. $\mathcal{F}_{\text{ILC}\textsc{lock}}$ can be viewed as a variant from [BGK$^+$21]'s with adaptations to provide a more natural clock model with real-word resources and in the proof-of-work setting.

For a detailed description of the functionality, see Appendix B. Here we just elaborate on the "imperfect" aspect of the clock and on the adversarial manipulation of clock drifts. Specifically, we allow the adversary to set some drifts to parties' local clocks, which will accelerate or stall their progress; such values are globally bounded by Φ_{clock}. This assumption allows local clocks to proceed at "roughly" the same speed.

Further, the adversary $\mathcal{A}$ can adaptively manipulate the drift of honest parties' clocks by sending CLOCK-FORWARD and CLOCK-BACKWARD messages to the functionality[4] after they conclude the current round. If $\mathcal{A}$ issues CLOCK-FORWARD for party P, it will enter a new local round before $\mathcal{F}_{\text{ILC}\textsc{lock}}$ updates the nominal time, and this can be repeated as long as P's drift is not Φ_{clock} rounds larger than other honest parties. On the other hand, if $\mathcal{A}$ issues CLOCK-BACKWARD, it will set P's budget to a negative value, thus preventing $\mathcal{F}_{\text{ILC}\textsc{lock}}$ from updating d_{P} at the end of the nominal round (d_{P} is the functionality variable that captures whether the party P has made its move for the clock tick). I.e., P will still be in the same logical round during these two nominal rounds. Again, this process can be repeated by $\mathcal{A}$ as long as the drift on P is not Φ_{clock} rounds smaller than others. As a consequence, the targeted party's local clock may remain static for several nominal rounds.

2.2 Other Core Functionalities

Common Reference String. We model a public-state setup by the CRS functionality $\mathcal{F}_{\text{CRS}}^{\mathcal{D}}$. The functionality is parameterized with some distribution $\mathcal{D}$ with sufficiently high entropy. Once $\mathcal{F}_{\text{CRS}}^{\mathcal{D}}$ receives ($\textsc{Retrieve}$, sid) from either the adversary $\mathcal{A}$ or a party P for the first time, it generates a string $d \leftarrow \mathcal{D}$ as the common reference string. In addition, $\mathcal{F}_{\text{CRS}}^{\mathcal{D}}$ will immediately send a message ($\textsc{Retrieved}$, sid) to functionality $\mathcal{W}(\mathcal{F}_{\text{RO}})$ (described next) to indicate that $\mathcal{W}(\mathcal{F}_{\text{RO}})$ should start to limit the adversarial RO queries. For all subsequent activations, $\mathcal{F}_{\text{CRS}}^{\mathcal{D}}$ simply returns d to the requester.

(Wrapped) Random Oracle. By convention, we model parties' calls to the hash function used to generated proofs of work as assuming access to a random oracle; this is captured by the functionality $\mathcal{F}_{\text{RO}}$. Notice that with regards to bounding access to real-world resources, functionality $\mathcal{F}_{\text{RO}}$ as defined fails to limit the adversary on making a certain number of queries per round. Hence, we adopt a functionality wrapper [BMTZ17, GKO$^+$20] $\mathcal{W}(\mathcal{F}_{\text{RO}})$ that wraps the corresponding resource to capture such restrictions. We highlight that our wrapper $\mathcal{W}(\mathcal{F}_{\text{RO}})$ improves on previous wrappers in two aspects, in order to provide a more natural model of the real world: (1) We capture the pre-mining stage by letting the adversary query the RO with no restrictions (albeit polynomially bounded) before the CRS is released; (2) The wrapper limits adversarial access per nominal round by bounding the total number of queries that $\mathcal{A}$ can make. The second aspect allows us to dispose

[4]As such, our clock functionality is a more natural model of the real world compared to [BGK$^+$21]'s, as it allows $\mathcal{A}$ to manipulate the clock in both directions, backward, and forward; in [BGK$^+$21], only forward manipulation is allowed. Nonetheless, this does not result in a more powerful adversary.

the "flat" computational model and define the computational power in terms of the number of RO queries per round, which makes it possible to further refine the notion of a "respecting environment" (see below) that is suited for imperfect local clocks.

Diffusion. We adopt the peer-to-peer communication functionality $\mathcal{F}_{\text{Diffuse}}^{\Delta}$ (cf. [BGK+21]), which guarantees that an honestly sent message will be delivered to all the protocol participants within Δ rounds. Moreover, for those adversarially generated messages, $\mathcal{F}_{\text{Diffuse}}^{\Delta}$ forces them to be delivered to all the honest parties within Δ rounds after they are learnt by at least one honest participant. This captures the natural behavior of honest parties that they will forward all the messages that they have not yet seen to their peers.

We refer to Appendix B for a detailed description of the above functionalities.

2.3 Dynamic Participation

The notion of a "respecting environment" was introduced in [GKL17] to model the varying number of participants in a protocol execution. In [BGK+18, BGK+21], the notion of *dynamic participation* was introduced aiming at describing the protocol execution in a more realistic fashion. Here we present a further refined classification of possible *types* of honest parties. See Table 1.

	Basic types of *honest* parties	
Resource	**Resource unavailable**	**Resource available**
random oracle $\mathcal{F}_{\text{RO}}$	*stalled*	*operational*
network $\mathcal{F}_{\text{Diffuse}}^{\Delta}$	*offline*	*online*
clock $\mathcal{F}_{\text{ILClock}}$	*time-unaware*	*time-aware*
synchronized state	*desynchronized*	*synchronized*

Table 1: A classification of protocol participants.

Consider an honest party P at a given point of the protocol execution. We say P is *operational* if P is registered with the random oracle $\mathcal{F}_{\text{RO}}$; otherwise, we say it is *stalled.* We say P is *online* if P is registered with the network; *offline* otherwise. We say P is *time-aware* if P is registered with the *imperfect* clock functionality $\mathcal{F}_{\text{ILClock}}$, and *time-unaware* otherwise.

Further, we say P is *synchronized* if P has been participanting in the protocol for sufficiently long time and achieves a "synchronized state" as well as a "synchronized clock." "Synchronized clock" means P holds a chain that shares a common prefix (cf. [GKL15]) with other *synchronized* parties; "synchronized clock" refers to that P maintains a local clock with time close to other *synchronized* parties. Otherwise, P is *desynchronized.* Additionally, P is aware of whether it is synchronized or not, and maintains a local variable `isSync` serving as an indicator for other actions.

Based on the above classification, we now define the notion of *alert* parties:

$$\textit{alert} \overset{\text{def}}{=} \textit{operational} \wedge \textit{online} \wedge \textit{time-aware} \wedge \textit{synchronized}.$$

In short, alert parties are those who have access to all the resources and are synchronized; this requires them to join the protocol execution passively for some period of time. They constitute the core set of parties that carry out the protocol.

In addition, we define *active* parties to include all parties that are alert, adversarial, and time-unaware.

$$\textit{active} \overset{\text{def}}{=} \textit{alert} \vee \textit{adversarial} \vee \textit{time-unaware}.$$

Respecting environment in terms of computational power. Next, we provide the following generalization of "respecting environment" to relate it to computational power as opposed to number of parties. Our assumption is that during the whole protocol execution, the honest computational power is higher than the adversarial one (cf. the "honest majority" condition in [GKL15] and follow-ups). The computational power is captured by counting the number of RO (hash) queries that parties make in each round. Further, we restrict the environment to fluctuate the number of such queries in a certain limited fashion.

Definition 2. *For $\gamma \in \mathbb{R}^+$ we call the sequence $(h_r)_{r\in[0,B)}$, where $B \in \mathbb{N}$, (γ, s)-respecting if for any set $S \subseteq [0, B)$ of at most s consecutive integers, $\max_{r\in S} h_r \leq \gamma \cdot \min_{r\in S} h_r$.*

We say that *environment $\mathcal{Z}$ is (γ, s)-respecting* if for all $\mathcal{A}$ and coins for $\mathcal{Z}$ and $\mathcal{A}$ the sequence of honest hash queries (h_r) is (γ, s)-respecting.

Note that the notion of respecting environment here is different from the "flat" model adopted in [GKL15, GKL17, GKL20, BMTZ17]. In a flat model, honest parties are assumed to have the same computational power, hence the total number of RO queries is a direct 1-to-1 map from the number of parties. The new respecting environment allows some subset of the honest parties to query the RO multiple times or stay stalled during a nominal round and hence it adapts to the "imperfect local clock" model used in this paper.

3 The Clock Synchronization Protocol with Public Setup

In this section we present the general approach and the various core building blocks of the new clock synchronization protocol—Timekeeper. For a complete description, refer to Appendix C. At a high level, Timekeeper is a Nakamoto-style PoW-based blockchain protocol together with time synchronization functionalities. Readers can think of it as a Bitcoin protocol with the following modifications:

- It replaces Bitcoin's original clock maintenance solution[5] with a new clock synchronization scheme, which requires parties to use 2-for-1 PoWs [GKL15] to mine and emit clock synchronization beacons and include them in an upcoming block. Furthermore, protocol participants will periodically adjust their local clock values based on the beacons collected in the blockchain and their (local) receiving time.
- Events are triggered by counting the number of local rounds (which is different from the convention that events in PoW-based blockchains are triggered by the arrival of blocks). In other words, the protocol has a clock synchronization *interval* of length R and a target recalculation *epoch* of length M that are defined in terms of the number of rounds; in addition, M is a multiple of R. Both of these values are hardcoded in the protocol. More precisely, parties will call the SyncProc functionality (see Appendix C.10) when their local clock enters round $\langle \texttt{itvl}, \texttt{itvl} \cdot \mathrm{R} \rangle$ (this represents the last round in interval `itvl`; see below for details on the round structure); and for target in the next epoch they will call UpdateMiningTarget (Appendix C.6) when their local clock enters $\langle \texttt{itvl}, \texttt{itvl} \cdot \mathrm{R} \rangle$, where ($\texttt{itvl} \mod (\mathrm{M}/\mathrm{R})) = 0$ (i.e., at the boundary of every (M/R) synchronization intervals).

See Figure 1 for an illustration of the protocol execution.

Next, we present the basic components that are employed in Timekeeper.

[5]In Bitcoin's original implementation, miners will adjust their time based on three different sources: (1) their local system clock; (2) the median of clock values from peers; (3) the human operator (if the first two disagrees).

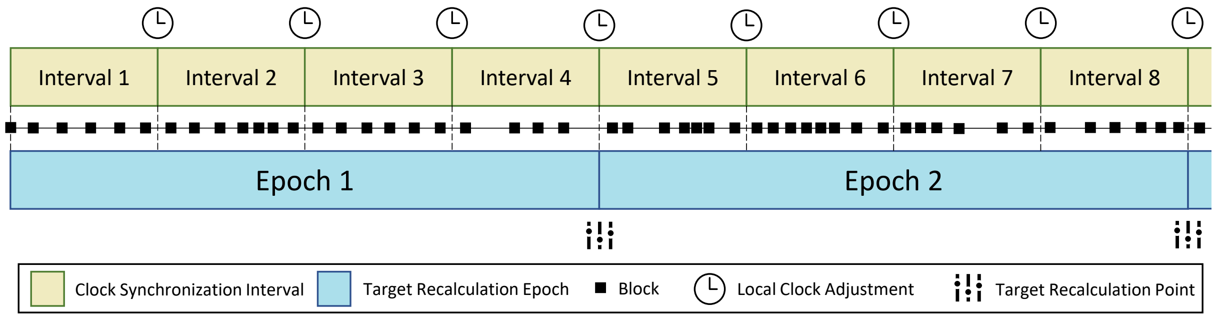

Figure 1: *An illustration of the clock synchronization protocol execution with one target recalculation epoch consisting of four clock synchronization intervals.*

3.1 Timekeeper Timestamps

As opposed to the conventional approach where blocks' timestamps are integer values, timestamps (both blocks' and beacon values) in Timekeeper are represented by a pair of values interval number and round number $\langle \mathtt{itvl}, \mathtt{r} \rangle \in \langle \mathbb{N}^+, \mathbb{N}^+ \rangle$. Note that (ideally) one synchronization interval would last for R rounds (i.e., rounds $((i-1)\cdot \mathrm{R}, i \cdot \mathrm{R}]$ would belong to the i-th interval). However, in Timekeeper we let the lower bound be 0, which means that timestamps with a somewhat small round number are still valid. Specifically, a timestamp $\langle \mathtt{itvl}, \mathtt{r} \rangle$ is considered valid if and only if it satisfies the predicate $\mathsf{validTimestamp}(\mathtt{itvl}, \mathtt{r}) \triangleq \mathtt{r} \leq \mathtt{itvl} \cdot \mathrm{R}$. We note that this new treatment allows for some small distortion at the end of each interval—i.e., the round number of a few blocks at the beginning of the next interval may be smaller than the last block of the previous interval (we call these "retorted" timestamps); see Figure 2.

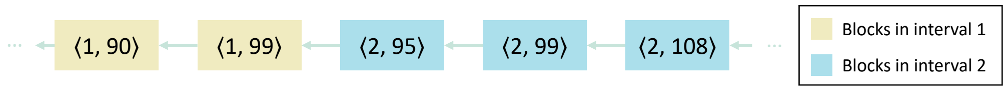

Figure 2: *An illustration of a segment of the blockchain with synchronization interval length* $\mathrm{R} = 100$. *Blocks can have timestamp values equal to blocks in the previous interval.*

Consider a chain of blocks in Timekeeper. Their timestamps should increase monotonically in terms of their interval number, and the round number in a single interval should also increase monotonically. More specifically, given two timestamps $\langle \mathtt{itvl}_i, \mathtt{r}_i \rangle, \langle \mathtt{itvl}_j, \mathtt{r}_j \rangle$ of two blocks $\mathcal{B}_i, \mathcal{B}_j$ respectively, if $\mathcal{B}_i$ is an ancestor block of $\mathcal{B}_j$, they should satisfy the following predicate:

$$\mathsf{validTimestampOrder}(\langle \mathtt{itvl}_i, \mathtt{r}_i \rangle, \langle \mathtt{itvl}_j, \mathtt{r}_j \rangle) \triangleq \left\{ \begin{array}{c} \mathsf{validTimestamp}(\mathtt{itvl}_i, \mathtt{r}_i) \wedge \mathsf{validTimestamp}(\mathtt{itvl}_i, \mathtt{r}_i) \\ \wedge [(\mathtt{itvl}_i \leq \mathtt{itvl}_j) \vee (\mathtt{itvl}_i = \mathtt{itvl}_j \wedge \mathtt{r}_i < \mathtt{r}_j)] \end{array} \right\}$$

Furthermore, we will overload the notation of comparison operators based on the valid order of timestamps. E.g., "=" will denote that two timestamps are identical, and $\langle \mathtt{itvl}_1, \mathtt{r}_1 \rangle < \langle \mathtt{itvl}_2, \mathtt{r}_2 \rangle$ if and only if $\mathsf{validTimestampOrder}(\langle \mathtt{itvl}_1, \mathtt{r}_1 \rangle, \langle \mathtt{itvl}_2, \mathtt{r}_2 \rangle)$ holds. Other operators $>, \leq, \geq, \neq$ are defined similarly.

We also redefine "$+, -$" to describe the timestamp that is $k \in \mathbb{N}$ rounds before (resp., after) $\langle \mathtt{itvl}, \mathtt{r} \rangle$. Regarding addition, $\langle \mathtt{itvl}, \mathtt{r} \rangle + k = \langle \max\{\mathtt{itvl}, \lceil (\mathtt{r} + k)/\mathrm{R} \rceil\}, \mathtt{r} + k \rangle$. Intuitively, the additive operation simply adds k to $\mathtt{r}$, and only increments $\mathtt{itvl}$ when it is going to become invalid. For subtraction, $\langle \mathtt{itvl}, \mathtt{r} \rangle - k = \langle \max\{1, \lceil (\mathtt{r} - k)/\mathrm{R} \rceil\}, \max\{1, \mathtt{r} - k\} \rangle$. In other words, regarding the subtraction operation, we only apply the operation on round, and the interval number is derived from the round after calculation. It does not output timestamps that are not "normally" belong to an interval. In case we do the subtraction operation for $k \geq \mathtt{r}$, it will return $\langle 1, 1 \rangle$.

Timekeeper's new approach to timestamps raises questions regarding the "trimming" of blockchains by counting the number of rounds. Recall that in [GKL15] the notation $\mathcal{C}^{\lceil k}$ represents the chain that results from removing the k rightmost blocks. In this paper, we overload this notation to denote the chain that results from removing blocks with timestamps in the last k rounds with respect to the current time. Specifically, for $\mathcal{C} = \mathcal{B}_1 \mathcal{B}_2 \dots \mathcal{B}_n$ and local time $\langle \mathtt{itvl}, \mathtt{r} \rangle$, $\mathcal{C}^{\lceil k} = \mathcal{B}_1 \mathcal{B}_2 \dots \mathcal{B}_m$ is the longest chain such that $\forall \mathcal{B} \in \mathcal{C}^{\lceil k}, \mathsf{Timestamp}(\mathcal{B}) < \langle \mathtt{itvl}, \mathtt{r} \rangle - k$. In other words, $\mathcal{B}_{m+1}$ is the first block (if it exists) such that $\mathsf{Timestamp}(\mathcal{B}_{m+1}) \geq \langle \mathtt{itvl}, \mathtt{r} \rangle - k$ holds.

3.2 2-for-1 Proofs of Work and Synchronization Beacons

2-for-1 PoW is a technique that allows protocols to utilize a single random oracle $H(\cdot)$ to compose two separate PoW sub-procedures involving two distinct and independent random oracles $H_0(\cdot), H_1(\cdot)$. It was first proposed in [GKL15] in order to achieve a better/optimal corruption threshold (from one-third to one-half) for the solution of the traditional consensus problem using a blockchain.

We refer to [GKL15] for more details, and here we present a simple implementation with the clock synchronization application in mind. In order to do the 2-for-1 mining, a party P prepares a composite input w that is a concatenation of two inputs w_0, w_1 of two different sub-procedures S_0, S_1, respectively. I.e., $w = w_0 \parallel w_1$. After selecting a nonce ctr, quering the random oracle with $ctr \parallel w$ and getting result u, P checks if $u < T$ which implies success in sub-procedure S_0; P also checks if $[u]^{\mathsf{R}} < T$ (where $[u]^{\mathsf{R}}$ denotes the reverse of a bitstring u) which indicates success in sub-procedure S_1. After successfully generating a PoW for S_0 (resp., S_1), in order to let parties others check validity, the proof will include the nonce and the entire composite input $ctr \parallel w$. Note that sub-procedure S_0 (resp., S_1) only cares about its corresponding part w_0 (resp., w_1), and treat the other part as dummy information.

The 2-for-1 PoW technique has several advantages when compared with the straightforward approach that would simply utilize two different random oracles. The most prominent advantage is that it prevents the adversary $\mathcal{A}$ from concentrating its computational power on one RO and thus gain advantage in the corresponding sub-procedure.

Synchronization beacons. In addition to the conventional blocks constituting the blockchain, protocol participants in Timekeeper also produce another type of "tiny" blocks using 2-for-1 PoWs. We call these blocks *clock synchronization beacons* ("beacons" for short) since they are used to report parties' local time and synchronize their clocks.

In more detail, one clock synchronization beacon $\mathtt{SB}$ is a tuple with the following structure.

$$\mathtt{SB} \triangleq \langle \langle \mathtt{itvl}, \mathtt{r} \rangle, \mathsf{P}, \eta_{\mathtt{itvl}}, ctr, blockLabel \rangle,$$

where $\langle \mathtt{itvl}, \mathtt{r} \rangle$ is the local time $\mathtt{SB}$ reports; P denotes the identity of its miner; $\eta_{\mathtt{itvl}}$ is some fresh randomness in the current interval; ctr represents the nonce of the PoW and $blockLabel$ is the associated block input. Note that $\mathtt{SB}$ must record the identity of its miner because there might be multiple beacons, mined by different parties, reporting the same timestamp as well as nonce value; otherwise, it would be impossible for the parties to distinguish such beacons. Worse still, other participants would not be able to distinguish the same beacon $\mathtt{SB}$ when they receive $\mathtt{SB}$ multiple times. Regarding $\eta_{\mathtt{itvl}}$, it is a string associated with interval $\mathtt{itvl}$ for the purpose of preventing the adversary $\mathcal{A}$ from mining beacons with *future* timestamps. In other words, protocol participants (including $\mathcal{A}$) can only learn $\eta_{\mathtt{itvl}}$ after they have (almost) finished interval $\mathtt{itvl} - 1$. We present the structure of intervals in detail and how we compute $\eta_{\mathtt{itvl}}$ in Section 3.3 and treat it as a communal bitstring here. We note that parties can learn $\eta_{\mathtt{itvl}}$ from their local chain, and indeed $\mathtt{SB}$ does not need to include $\eta_{\mathtt{itvl}}$ (P can prune those beacons that are invalid with $\eta_{\mathtt{itvl}}$ in their local view). We keep $\eta_{\mathtt{itvl}}$ in the description for convenience.

Regarding the structure of a blockchain block $\mathcal{B}$, we adopt the similar structure as inin [GKL17] (with the dummy information in the 2-for-1 PoWs):

$$\mathcal{B} \triangleq \langle h, \mathtt{st}, \langle \mathtt{itvl}, \mathtt{r} \rangle, ctr, txLabel \rangle,$$

where h is the reference to the previous block, $\mathtt{st}$ the Merkle root of the block content, $\langle \mathtt{itvl}, \mathtt{r} \rangle$ its timestamp, ctr the nonce of PoW, and $txLabel$ the binded beacon input.

We are now ready to describe how the parties in $\mathsf{Timekeeper}$ do the 2-for-1 PoW mining. The composite input prepared in $\mathsf{Timekeeper}$ is different from the trivial instance above, in that the term $\langle \mathtt{itvl}, \mathtt{r} \rangle$ appears in both blocks and beacons. Hence, simply concatenating two inputs introduces redundant information in the PoW. When a party P is ready to perform the mining procedure, P binds the nonce, the blocks' input and beacon input together as

$$\langle ctr, h, \mathtt{st}, \langle \mathtt{itvl}, \mathtt{r} \rangle, \mathsf{P}, \eta^{\mathcal{C}}_{\mathsf{ep}} \rangle$$

and hand them over to random oracle $\mathcal{F}_{\mathrm{RO}}$. Let u denote the result from $\mathcal{F}_{\mathrm{RO}}$. If $u < T$ (i.e., the block query succeeds), P finds a new block $\mathcal{B} = \langle h, \mathtt{st}, \langle \mathtt{itvl}, \mathtt{r} \rangle, ctr, txLabel \rangle$ where $txLabel := \langle \mathsf{P}, \eta^{\mathcal{C}}_{\mathsf{ep}} \rangle$; if $[u]^{\mathsf{R}} < T$ (the beacon query succeeds), P gets a new beacon $\mathtt{SB} = \langle \langle \mathtt{itvl}, \mathtt{r} \rangle, \mathsf{P}, ctr, blockLabel \rangle$, where $blockLabel := \langle h, \mathtt{st} \rangle$. Note that for the sake of presentation, we reorder the content of blocks and beacons so that they are inconsistent with the input to the PoW.

After receiving the result from $\mathcal{F}_{\mathrm{RO}}$, P checks if it was able to successfully generate a new block. In addition, P checks if he successfully produces a beacon but only when P's local clock stays in the *beacon mining and inclusion* phase. Namely, P reports a timestamp that satisfies a certain criterion (details in Section 3.3).

3.3 Clock Synchronization Intervals and the Synchronization Procedure

As mentioned earlier, $\mathsf{Timekeeper}$ participants will periodically adjust their local clock. We call the time interval between two adjustment points[6] a *clock synchronization interval* (or "interval" for short). Ideally, one interval will last for R rounds. The actual number of local rounds that parties observe may differ according to the shift computed in the previous interval (we will show later that the shift computed in every interval is well-bounded). When party P's local clock gets to the last round of an interval, it will call $\mathsf{SyncProc}$ (Appendix C.10), which adjusts its local clock and gets the fresh randomness to run the next interval.

[6]The first interval in particular lies between the beginning of the execution and the first time parties adjust their clock.

3.3.1 Interval Structure

Timekeeper divides one interval into three phases: (1) *view convergence*, (2) *beacon mining and inclusion* and (3) *beacon-set convergence*. The phase parties stay in depends on their local clocks. Furthermore, parties will keep track of the (local) arriving time of a synchronization beacon as long as it is online. In this section we describe these three phases as well as the bookkeeping function and explain the design intention behind them.

View convergence. When a party P's local clock reports a time $\langle \mathtt{itvl}, \mathtt{r}\rangle$ such that $\mathtt{r} < (\mathtt{itvl} - 1)\cdot \mathrm{R} + \mathrm{K}$, P is in the *view convergence* phase. Note that this also includes rounds with potentially retorted timestamps. In this phase, if P is alert, it will try to mine the next block with the 2-for-1 PoW technique (i.e., the input information that P forwards to the $\mathcal{F}_{\mathrm{RO}}$ functionality does not need to be changed); nonetheless, P will not check if he successfully mines a beacon after P acquires the output. This is because all the beacons obtained in this phase are *invalid* in that they report an undesirable timestamp.

The general motivation for introducing the view convergence phase and letting parties wait for some period of time at the beginning of an interval is that we would like parties to start mining beacons with a *consistent* view of the previous interval. Since K is larger than the common prefix parameter (we will quantify K in later, in Section 4.2), at the end of the view convergence phase of interval $\mathtt{itvl}+1$, alert parties will have a common view of interval $\mathtt{itvl}$. In other words, they will agree on all the blocks in interval $\mathtt{itvl}$, and the adversary $\mathcal{A}$ will not be able to apply any changes to these blocks. Hence, alert parties agree on the number of blocks in the previous interval, which decides the mining difficulty within the current interval. (This will used in our new target recalculation function, presented in Secion 3.4.) Parties will mine beacons with the same difficulty, and this simplifies the protocol description as well as its analysis. Furthermore, alert parties will compute the same fresh randomness as

$$\eta_{\mathtt{itvl}+1} \triangleq G(\eta_{\mathtt{itvl}} \parallel (\mathtt{itvl}+1) \parallel v), \tag{1}$$

where v is the concatenation of all block hashes in interval $\mathtt{itvl}$. Note that we adopt a different hash function $G(\cdot)$ (as opposed to $H(\cdot)$) to compute the next fresh randomness that is not used in the 2-for-1 PoW , which does not consume any queries to random oracle $\mathcal{F}_{\mathrm{RO}}$.

Recall that by assumption the adversary $\mathcal{A}$ has full knowledge of the network, and hence it can learn all honest blocks from the previous interval immediately and manipulate the chain at will for up to a number of rounds bounded by the common prefix parameter, allowing $\mathcal{A}$ to mine the synchronization beacons before the alert parties start to mine. We call this period where $\mathcal{A}$ starts ahead of time the *pre-mining* stage. Nonetheless, we will show later that there will be at least one block generated by an alert party near the end of interval $\mathtt{itvl}$, which prevents the adversary from pre-mining for too long a time.

Remark 1. *We note that, with some modifications, it is safe to get rid of the view convergence phase. The fresh randomness will still need to be extracted from the settled part of the chain, so we will replace it with the randomness generated in previous beacon mining and inclusion phase. This can be implemented by modifying some parameters and does not change the protocol framework. The main difference lies in how validity of a timestamp beacon is checked. At the beginning of a target recalculation epoch (which is also the beginning of an interval), parties may be mining on different chains, and hence mining beacons under different targets. In order to check if the beacon target is correctly computed, some additional information (e.g., the block height of the previous* *target recalculation epoch) should be included in the block header (and the height should not be less than the number of blocks in the settled part). Moreover, when parties run the synchronization procedure at the first interval in an epoch, they should use the "weighted" median timestamp instead of the plain median (and "weighted" means we would assign a weight to each timestamp based on its difficulty). Details on applying the median timestamp are given in Section 3.3.2.*

Beacon mining and inclusion. When a party P's local clock is in rounds $\langle \mathtt{itvl}, \mathtt{r} \rangle$ satisfying $(\mathtt{itvl} - 1) \cdot \mathrm{R} + \mathrm{K} \leq \mathtt{r} \leq \mathtt{itvl} \cdot \mathrm{R} - \mathrm{K}$, P is in the *beacon mining and inclusion* phase. Next, we define the predicate $I_{\text{sync}}(\mathtt{itvl})$ to extract the set of timestamps in this phase. Formally,

$$I_{\text{sync}}(\mathtt{itvl}) \triangleq \{(\mathtt{itvl} - 1) \cdot \mathrm{R} + \mathrm{K}, \ldots, \mathtt{itvl} \cdot R - \mathrm{K}\}. \tag{2}$$

For convenience, we slightly overload this predicate. When the input is a timestamp, $I_{\text{sync}}(\langle \mathtt{itvl}, \mathtt{r} \rangle)$ outputs whether $\langle \mathtt{itvl}, \mathtt{r} \rangle$ stays in a beacon mining and inclusion phase. I.e., $I_{\text{sync}}(\langle \mathtt{itvl}, \mathtt{r} \rangle) = \mathsf{true}$ if $\mathtt{r} \in I_{\text{sync}}(\mathtt{itvl})$, and false otherwise.

After entering this phase, P will use a 2-for-1 PoW to mine both blocks and clock synchronization beacons. During interval $\mathtt{itvl}$, the output will be a beacon which indicates its local time and value $\mathtt{SB} \triangleq \langle \langle \mathtt{itvl}, \mathtt{r} \rangle, \mathsf{P}, \mathit{ctr}, \mathit{blockLabel} \rangle$. Regarding the mining difficulty, $\mathsf{Timekeeper}$ will set the same target value for blocks and beacons[7]. In other words, the expected number of blocks and of beacons in this phase are equal.

After a beacon is successfully generated, it will be diffused into the network via $\mathcal{F}_{\text{Diffuse}}^{\mathsf{sync}}$. P will include a beacon $\mathtt{SB}$ into the pending block content if $\mathtt{SB}$ is valid w.r.t. the current interval. Next, we describe how they check the validity of a beacon is checked. The format of a beacon $\mathtt{SB}$ with respect to interval $\mathtt{itvl}$ is correct if and only if it reports a timestamp $\langle \mathtt{itvl}, \mathtt{r} \rangle$ such that $\mathtt{r} \in I_{\text{sync}}(\mathtt{itvl})$. We say a beacon $\mathtt{SB}$ is *valid w.r.t. chain* $\mathcal{C}$ if and only if its format is correct and the hash value of $\mathtt{SB}$ (after concatenating with the fresh randomness in $\mathcal{C}$) is smaller than the corresponding mining target. P will try to include all the (valid) beacons mined in the current interval $\mathtt{itvl}$ with timestamps earlier than the current local time but which have not yet been included in the blockchain. Specifically, at round $\langle \mathtt{itvl}, \mathtt{r} \rangle$, all valid beacons recording timestamp $\langle \mathtt{itvl}, u \rangle$ with $u \leq \mathtt{r}$ will get into P's pending block content.

When P's local clock goes past the last round of beacon mining and inclusion phase, it stops checking the beacon hash output and it no longer includes beacons in the next block. Beacons that are generated and diffused right at the end of this phase get dropped.

Beacon-set convergence. The third and last phase—*beacon-set convergence*—consists of the last K rounds in an interval. In other words, a party P is in this phase when P reports a timestamp $\langle \mathtt{itvl}, \mathtt{r} \rangle$ with $\mathtt{r} > \mathtt{itvl} \cdot \mathrm{R} - \mathrm{K}$. During this phase, P behaves similar to the first phase. I.e., it will not check for the 2-for-1 PoW result to see if the beacon generation succeeds.

Parties have to wait for at least K rounds to ensure that they share a *consistent* view of the set of beacons included in the current interval (except with some negligible probability). This phase cannot be omitted (as opposed to the case of first phase mentioned in Remark 1) since only when parties agree on the same beacon set can the synchronization procedure maintain the protocol's security properties (Section 4).

[7]We will adopt the same target for simplicity. Indeed, maintaining a constant ratio between the difficulty level of blocks and that of beacons will work.

Beacon arrival booking. In order to adjust its clock, P also needs the local receiving time of all beacons that have been included in the chain. Hence, P will maintain a local registry that records the beacons it receives as well as their arrival time. More specifically, this local beacon ledger is an array of synchronization beacons. For each beacon $\mathtt{SB}$, a pair $(a, \mathsf{flag}) \in \langle \mathbb{N}^+, \mathbb{N}^+ \rangle \times \{\mathsf{final}, \mathsf{temp}\}$ is assigned to it. Consider a round $\langle \mathtt{itvl}, \mathtt{r} \rangle$ when P receives a beacon $\mathtt{SB}$ with $\mathsf{Timestamp}(\mathtt{SB}) = \langle \mathtt{itvl}', \mathtt{r}' \rangle$.

- If $\mathtt{itvl}' \leq \mathtt{itvl}$, which means the beacon $\mathtt{SB}$ is generated in the current or previous interval[8]. P will drop $\mathtt{SB}$ if it is not valid w.r.t. its localchain; otherwise, P will assign $(\langle \mathtt{itvl}, \mathtt{r} \rangle, \mathsf{final})$ to $\mathtt{SB}$. This means that all the information gathering regarding this beacon has been finalized and it is ready to be used.
- If $\mathtt{itvl}' > \mathtt{itvl}$, the beacon is generated in the future. P will assign $(\langle \mathtt{itvl}, \mathtt{r} \rangle, \mathsf{temp})$ to $\mathtt{SB}$, which indicates that modifications on the receiving time may be applied in the future. Note that parties may not know the fresh randomness in future intervals (for example, if they are newly joint parties and have not yet synchronized with the blockchain or they are alert but receive forthcoming beacons). Hence they cannot check the validity of beacons with temp flag. Nevertheless, invalid beacons would be excluded from the registry after P learns the upcoming fresh randomness.

If P receives multiple beacon messages with the same creator and time reported, P will adopt the first one it receives as its arrival time.

3.3.2 The Synchronization Procedure

At the end of an interval (i.e., when the local time reports $\mathtt{r} = \mathtt{itvl} \cdot \mathrm{R}$), parties will use the beacons information to compute a value shift that indicates how much the logical clock should be adjusted. (See Appendix C.10 for the complete specification.)

Adjusting the local clock. When a party P's local clock reaches round $\langle \mathtt{itvl}, \mathtt{itvl} \cdot \mathrm{R} \rangle$ and P has finished the round's regular mining procedure, P will adjust its local clock based on the beacons recorded on chain and their local receiving time. More specifically, P will extract all the beacons from the beacon mining and inclusion phase, and compute the differences between their timestamp and local receiving time $\mathsf{Timestamp}(\mathtt{SB}) - \mathsf{arrivalTime}(\mathtt{SB})$. Since the timestamp of $\mathtt{SB}$ and its arrival time share the same interval index, we only need to compute the difference between their round numbers. Subsequently, all the beacons will be ordered based on this difference and a shift will be computed by selecting the median difference therein. Formally,

$$\mathsf{shift}^{\mathsf{P}}_{\mathtt{itvl}} \triangleq \mathsf{med}\{\mathsf{Timestamp}(\mathtt{SB}) - \mathsf{arrivalTime}(\mathtt{SB}) \mid \mathtt{SB} \in \mathcal{S}^{\mathsf{P}}_{\mathtt{itvl}}\}. \tag{3}$$

In case there are two median beacons $\mathtt{SB}_1, \mathtt{SB}_2$, parties will adjust $\mathsf{shift}^{\mathsf{P}}_{\mathtt{itvl}} \triangleq \lceil (\mathsf{Timestamp}(\mathtt{SB}_1) - \mathsf{arrivalTime}(\mathtt{SB}_1) + \mathsf{Timestamp}(\mathtt{SB}_2) - \mathsf{arrivalTime}(\mathtt{SB}_2))/2 \rceil$. Afterwards, P will update its local clock to $\langle \mathtt{itvl} + 1, \mathtt{r} + \mathsf{shift} \rangle$. Later we show that this update strategy in the synchronization procedure allows parties' clocks to remain in a narrow interval and do not deviate too much from the nominal time.

Note that parties will enter local round $\langle \mathtt{itvl}, \mathtt{r} \rangle$ where $\mathtt{r} = \mathtt{itvl} \cdot \mathrm{R}$ only once. If they enter some time $\langle \mathtt{itvl}', \mathtt{r} \rangle$ in the future, we will get $\mathtt{itvl}' > \mathtt{itvl}$ and they will never revert back.

[8]Beacons generated in previous intervals are stale in that P has already passed the synchronization point associated with these beacons, and they will never be used in the future. We list them for completeness.

Mining with backward-set clocks. After the adjustment at the end of intervals, and shift is added to P's local clock, it may set its local time to values $\langle \mathtt{itvl}, \mathtt{r} \rangle$ such that $\mathtt{r} \leq (\mathtt{itvl} - 1) \cdot \mathrm{R}$ (i.e., the retortion effect that was mentioned earlier). Nonetheless, P can continue to mine blocks with this timestamp and its local clock will eventually proceed to a time value of regular format (i.e., $\mathtt{r} > (\mathtt{itvl} - 1) \cdot \mathrm{R}$).

We compare this treatment with the similar scenario in a PoS blockchain [BGK$^+$21]. In [BGK$^+$21], setting local clocks backward is never a problem since parties can keep silent during this period. Due to the nature of PoS-based blockchains, parties do not need to do anything if they are not assigned the leader slot. In our context, however, adopting the same 'silence' policy contradicts the basic nature of PoW-based blockchains as parties will forfeit the chance to extend their local chain. In other words, there is no point for an activate party to not make RO queries. This is taken care of by Timekeeper's timestamping scheme.

Updating the beacon arrival time registry. Notice that the beacon information stored in a party P's arrival time registry is closely related to which interval P stays in; after P enters the next interval, it needs to update the beacon bookkeeping. P will apply a shift computation for all beacons with flag temp. Furthermore, for those beacons that report a timestamp with interval equal to the incoming one, their flag will be set to final. In more detail, at the end of interval $\mathtt{itvl}$, for all eligible SB in the beacon registry, their associated pair $(\langle \mathtt{itvl}_{\mathsf{SB}}, \mathtt{r}_{\mathsf{SB}} \rangle, \mathsf{temp})$ will be updated to $(\langle \mathtt{itvl}_{\mathsf{SB}}, \mathtt{r}_{\mathsf{SB}} + \mathsf{shift} \rangle, \mathsf{final})$ if $\mathtt{itvl}_{\mathsf{SB}} = \mathtt{itvl} + 1$. Note that for those beacons whose flags are set to final, P will removed all invalid ones from the registry after the update.

3.4 The Target Recalculation Function

If the mining target is not set appropriately ("appropriately" means that the block generation rate according to the current hashing power and target is somewhat steady; see [GKL17]), PoW-based blockchain protocols fail to maintain any of the security properties in a permissionless environment. In Bitcoin, the target is adjusted after receiving the last block of the current epoch (and an epoch consists of 2016 blocks). Based on the time elapsed to mine these blocks, a new target is set based on the previous target value and the variation is proportional to the time elapsed. Note that Bitcoin's target recalculation function is not the only way to adjust the difficulty level. A large number of other recalculation functions have been proposed in alternate blockchains (e.g., Ethereum, Bitcoin Cash, Litecoin), with their security asserted by either theoretical analysis or empirical data.

In Timekeeper, we propose a new target recalculation function that is suitable for the new setting. Intuitively, our function is a reversed version of Bitcoin's original function, namely, protocol participants wait for some fixed number of rounds M (in their local view) to update the difficulty level. We call such M number of rounds a *target recalculation epoch*. Moreover, Timekeeper sets M as a multiple of R, which makes the target recalculation epoch consist of several clock synchronization intervals, and the start and end point of an epoch coincide with the start and end of different synchronization intervals. Recall that in the Timekeeper timestamp scheme introduced in Section 3.1, the first term in $\langle \mathtt{itvl}, \mathtt{r} \rangle$ does not directly reflect which target recalculation epoch it is in. For simplicity, we introduce function TargetRecalcEpoch that maps the protocol timestamp to the target recalculation epoch it belongs to:

$$\mathsf{TargetRecalcEpoch}(\langle \mathtt{itvl}, \mathtt{r} \rangle) \triangleq \lceil \mathtt{itvl}/(\mathrm{M}/\mathrm{R}) \rceil.$$

In addition, we introduce a function EpochBlocks which extracts all the blocks in chain $\mathcal{C}$ that belong to target recalculation epoch ep. Formally, given $\mathsf{ep} \geq 1$,

$$\mathsf{EpochBlocks}(\mathcal{C}, \mathsf{ep}) \triangleq \{\mathcal{B} : \mathcal{B} \in \mathcal{C} \wedge \mathsf{TargetRecalcEpoch}(\mathsf{Timestamp}(\mathcal{B})) = \mathsf{ep}\}.$$

Also for convenience, we let $\mathsf{EpochBlockCount}$ be a function that returns the number of blocks in chain $\mathcal{C}$ that belong to epoch ep. We also extend the input domain of epoch numbers to 0 and let it output Λ_{epoch} (the ideal number of blocks) to capture the fact that the target at the beginning of an execution is set appropriately and hence maintains the ideal block generation rate. Formally,

$$\mathsf{EpochBlockCount}(\mathcal{C}, \mathsf{ep}) \triangleq \begin{cases} |\mathsf{EpochBlocks}(\mathcal{C}, \mathsf{ep})| & if\ \mathsf{ep} \geq 1 \\ \Lambda_{\mathsf{epoch}} & if\ \mathsf{ep} = 0 \end{cases} \tag{4}$$

Going back to the algorithm, for the first epoch ($\mathsf{ep} = 1$) parties will adopt the target value of the genesis block (T_0). I.e., $T_1 = T_0$. Regarding other epochs ($\mathsf{ep} > 1$), parties will figure out how many blocks are produced in the previous epoch, and set the next target based on the previous one. This variation is proportional to the ratio of expected number of blocks Λ_{epoch} and the actual number. I.e., for epoch $\mathsf{ep} + 1$,

$$T_{\mathsf{ep}+1} \triangleq \frac{\Lambda_{\mathsf{epoch}}}{\Lambda} \cdot T_{\mathsf{ep}}, \quad \mathsf{ep} \in \mathbb{N}^+, \tag{5}$$

where Λ is the number of blocks in epoch ep—in other words, the size of $\mathsf{EpochBlocks}(\mathcal{C}, \mathsf{ep})$.

In order to prevent the "raising difficulty attack" [Bah13], the maximal target variation in a single recalculation step still needs to be bounded (we denote this bound by τ). Specifically, if $\Lambda > \tau \cdot \Lambda_{\mathsf{epoch}}$, $T_{\mathsf{ep}+1}$ will be set as T_{ep}/τ; on the other hand, if $\Lambda < \Lambda_{\mathsf{epoch}}/\tau$, $T_{\mathsf{ep}+1}$ will be set as $\tau \cdot T_{\mathsf{ep}}$.

Remark 2. *We observe that, compared to the Bitcoin case, the adversary $\mathcal{A}$ in* Timekeeper *is in a much worse position to carry out the raising difficulty attack. This is because in Bitcoin, in order to significantly raise the difficulty in the next epoch, $\mathcal{A}$ only needs to mine 2016 blocks with close timestamps; in the case of* Timekeeper*, however, the adversary has to mine $\tau \cdot \Lambda_{\mathsf{epoch}}$ blocks (with fake timestamps) in order to raise the same level of difficulty. The number of blocks that $\mathcal{A}$ needs to prepare is τ times larger than that in Bitcoin (assuming both protocols share the same number of expected blocks in an epoch).*

3.5 Newly Joining Parties

Recall that Timekeeper runs in a permissionless environment where parties can join and leave at will. As such, it is essential that newly joining parties can learn the protocol time to become alert and participate in the core mining process. More specifically, after the joining procedure, newly joining party P's local clock should report a time in a sufficiently narrow interval with all other alert parties, at which point P can claim also being alert.

Based on the fine-grained classification of types of parties in our dyanmic participation model (Section 2.3), newly joining parties can be classified into two types: (1) parties that are temporarily de-registered from $\mathcal{F}_{\mathrm{RO}}$, and (2) parties that start with bootstrapping from the genesis block, or parties that temporarily lose the network connection (i.e., de-registered from $\mathcal{F}^{\Delta}_{\mathrm{Diffuse}}$), or parties that are temporarily de-registered from $\mathcal{F}_{\mathrm{ILC}\textsc{lock}}$.

For parties that are stalled for a while, since they do not miss any clock tick or other necessary information from the network, they can easily re-join by calling the procedure SimulateClockAdjustments (Section C.12). For the rest of newly joining parties, they will be classified as de-synchronized (note that parties are aware of their synchronization status), and will run the joining procedure JoinProc, which we now describe.

Procedure JoinProc. In order to synchronize its clock, a newly joining party P needs to "listen" to the protocol for sufficiently long time. We describe the joining process below, which is similar to that in [BGK⁺21]. The main difference is that we adopt the heaviest-chain selection rule in order to adapt to the PoW context. The complete specification of this protocol is presented in Appendix C.14, and the default parameters values are summarized in Table 2.

Parameter	Default	Phase
t_{off}	2K	B
t_{gather}	5R/2	C
t_{pre}	3K	D

Table 2: *Parameters of the joining procedure and their corresponding phases.*

- **Phase A (state reset).** When all resources are available to P, after resetting all its local variables, P invokes the main round procedure triggering the join procedure.
- **Phase B (chain convergence, with parameter t_{off}).** In the second activation upon a MAINTAIN-LEDGER command, the party will jump to phase B and stay in phase B for t_{off} rounds. During this phase, the party applies the *heaviest-chain selection rule* maxvalid to filter its incoming chains. The motivation behind Phase B is to let P build a chain that shares a sufficiently long common prefix with all alert parties. Note that since P has not yet learnt the protocol time, it cannot filter out chains that should be put aside in the `futureChains`. Hence, the chain held by P may still contain a long suffix built entirely by the adversary. However, it can be guaranteed (Lemma 15(a)) that this adversarial fork can happen for up to k rounds ahead. Thus, the beacons recorded before the fork can be used to compute the adjustment and their local arrival times will be reliable.
- **Phase C (beacon gathering, with parameter t_{gather}).** Once a party P has finished Phase B, it continues with Phase C, the beacon-gathering phase. During this phase, P continues to collect and filter chains as in Phase B. In addition, P now processes and bookkeeps the beacons received from $\mathcal{F}_{\text{Diffuse}}^{\text{sync}}$. At a high level, this phases' length parameter t_{gather} guarantees that: (1) enough beacons are recorded to compute a reliable time shift; (2) enough time has elapsed so that the blockchain reaches agreement on the set of (valid) beacons to use. At the end of Phase C, P is able to reliably judge valid arrival times.
- **Phase D (shift computation, with parameter t_{pre}).** Since party P has now built a blockchain sharing a common prefix with any alert party, and has bookkeeped synchronization beacons for a sufficiently long time, P starts from the earliest interval i^* such that (1) the arrival times of all beacons included in blocks within the beacon mining and inclusion phase of interval i^* have been locally bookkeeped, and (2) all of these beacons arrived sufficiently later than the start of Phase C (parameterized by t_{pre} rounds). Based on this information, P computes the shift value as alert parties do at the boundary of synchronization interval i^*. P concludes Phase D when the adjusted time is a valid timestamp in interval $i^* + 1$ (in other words, `r` does not exceed (i^*+1)R); otherwise, P updates the local arrival time of beacons with flag temp and repeats the above process with interval $i^* + 1$. We note that if Phase D involves

the computation w.r.t. multiple intervals, the local time may temporarily be set as an invalid timestamp. Nevertheless, eventually after P has passed $(2\mathrm{K} + 5\mathrm{R}/2)$ (local) rounds, P will end up with a valid timestamp that with overwhelming probability is close enough to those of all alert parties.

4 Protocol Analysis

Our ultimate goal is to show that, at any point of the protocol's execution, the timestamps reported by all alert protocol participants of $\mathsf{Timekeeper}$ will satisfy the properties defined in Definition 1. We start off with some additional definitions and preliminary results.

4.1 Notation, Definitions and Preliminary Propositions

We note that several of the analytical tools proposed in [GKL17, GKL20] do not directly apply in the environment (with $\mathcal{F}_{\mathrm{ILCLOCK}}$) where $\mathsf{Timekeeper}$ runs in. Therefore, we first extend and enhance these tools to adapt them to this new environment.

Our probability space is over all executions of length at most some polynomial in κ and λ; we use $\mathbf{Pr}$ to denote the probability measure of this space. Furthermore, let $\mathcal{E}$ be a random variable taking values on this space and with a distribution induced by the random coins of all entities (adversary, environment, parties) and the random oracle.

For the sake of convenience, we define a *nominal time* that coincides with the internal variable `time` in $\mathcal{F}_{\mathrm{ILCLOCK}}$. Recall that `time` aims at recording how many times the functionality sends clock ticks to all registered honest parties.

Definition 3 (Nominal Time)**.** *Given an execution of* $\mathsf{Timekeeper}$*, any prefix of the execution can be mapped deterministically to an integer r, which we call* nominal time*, as follows: r is the value of variable* `time` *in the clock functionality at the final step of the execution prefix which is obtained by parsing the prefix from the genesis block and keeping track of the honest party set registered with the clock functionality (bootstrapped with the set of inaugural alert parties). (In case no honest party exists in the execution, r is undefined).*

Note that we adopt r to denote the nominal time, which is different from the protocol timestamp $\langle \mathtt{itvl}, \mathtt{r} \rangle$.

If at a nominal round r exactly h parties query the oracle with target T, the probability of at least one of them will succeed is

$$f(T, h) = 1 - (1 - pT)^h \leq pTh, \text{ where } p = 1/2^\kappa.$$

During nominal round r, alert parties might be querying the random oracle for various targets. We denote by $T_r^{\min}$ and $T_r^{\max}$ the minimum and maximum of those targets. Moreover, the initial target T_0 implies in our model an initial estimate of the number of honest RO queries h_0; specifically, $h_0 = 2^\kappa \Lambda_{\mathsf{epoch}}/(T_0 M)$, i.e., the number of parties it takes to produce Λ_{epoch} blocks of difficulty $1/T_0$ in time M. For convenience, we denote $f_0 = f(T_0, h_0)$ and simply refer to it as f. Also note that the ideal number of blocks $\Lambda_{\mathsf{epoch}} = \mathrm{M}f$, so in the analysis we will use $\mathrm{M}f$ to represent Λ_{epoch}.

"Good" properties. Next, we present some definitions which will allow us to introduce a few ("good") properties, serving as an intermediate step towards proving the desired clock properties.

Let us consider the boundary of two target recalculation epochs. Recall that Bitcoin's target recalculation algorithm defines epoch in terms of the number of blocks (m blocks forms an epoch). Thus, a block with block height a multiple of m is the last block of an epoch. While it might be manipulated, its timestamp naturally becomes the proof that miners have adjusted their difficulty and entered the next epoch (known as the *target recalculation point* [GKL17]). In contrast, Timekeeper adopts a new target recalculation function (see Section 3.4) that divides the epoch based on the parties' local view. While we can still define a target recalculation point based on one party's local view, parties can *never* agree on a point where they enter the next epoch based on nominal time.

In order to circumvent the above obstacle, we extend the notion of target recalculation point to target recalculation *zone*. See Figure 3 for an illustration. Intuitively, a "target recalculation zone" w.r.t. epoch $\mathtt{ep}$ is a sequence of consecutive nominal rounds such that during these nominal rounds, at least one alert party crosses its own target recalculation point w.r.t. epoch $\mathtt{ep}$. For convenience, we assume a "safe" start—i.e., the first epoch also has a target recalculation zone, and it naturally satisfies all good properties we will later define.

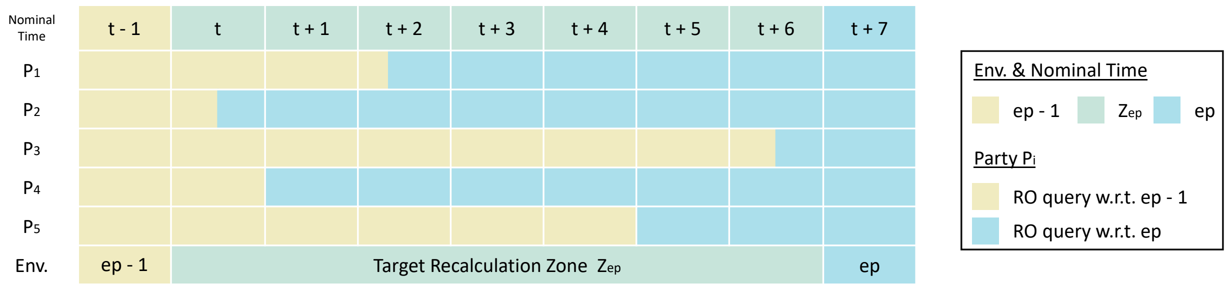

Figure 3: *An illustration of the target recalculation zone* $Z_{\mathtt{ep}} = \{t, \ldots, t+6\}$.

Definition 4.

- *Nominal time r is* good *if* $f/2\gamma^2 \leq ph_r T_r^{\min}$ *and* $ph_r T_r^{\max} \leq (1+\delta)\gamma^2 f$.
- *Round* $\langle \mathtt{itvl}, \mathtt{r}\rangle$ *is a* target-recalculation point w.r.t. epoch $\mathtt{ep}$ *if* $(\mathtt{r} = \mathtt{itvl} \cdot R) \wedge [\mathtt{itvl} \mod (M/R) = 0]$.
- *A sequence of consecutive nominal rounds* $Z_{\mathtt{ep}} = \{r\}$ *is a* target recalculation zone w.r.t. target recalculation epoch $\mathtt{ep}$ *if during* $Z_{\mathtt{ep}}$ *some subset of synchronized parties are in the logical round that is a target recalculation point w.r.t.* $\mathtt{ep} - 1$.
- *A target-recalculation zone* $Z_{\mathtt{ep}}$ *is* good *if for all* $h_r, r \in Z_{\mathtt{ep}}$ *the target* $T_{\mathtt{ep}}$ *satisfies* $f/2\gamma \leq ph_r T_{\mathtt{ep}} \leq (1+\delta)\gamma f$.
- *A chain is* good *if all its target-recalculation zones are good.*
- *A chain is* stale *if for some nominal time u it does not contain an honest block computed after nominal time* $u - \ell - 2\Delta - 2\Phi$.
- *The* blocklength *of an epoch* $\mathtt{ep}$ *on a chain* $\mathcal{C}$ *is the number of blocks in* $\mathcal{C}$ *with timestamp* $\langle \mathtt{itvl}, \cdot\rangle$ *such that* $\mathsf{TargetRecalcEpoch}(\langle \mathtt{itvl}, \mathtt{r}\rangle) = \mathtt{ep}$.

We would like to prove that, at a certain nominal round r of the protocol execution, alert parties enjoy good properties on their local chains and reported timestamps. Towards this goal, we extract all chains that either belong to alert parties at r or have accumulated sufficient difficulty and thus might be adopted in the future. We denote this chain set by $\mathcal{S}_r$:

$$\mathcal{S}_r \triangleq \left\{ \mathcal{C} \in E_r \;\middle|\; \begin{array}{l} \text{“}\mathcal{C}\text{ belongs to an alert party” or} \\ \text{“}\exists \mathcal{C}' \in E_r \text{ that belongs to an alert party and } \mathrm{diff}(\mathcal{C}) > \mathrm{diff}(\mathcal{C}')\text{” or} \\ \text{“}\exists \mathcal{C}' \in E_r \text{ that belongs to an alert party and } \mathrm{diff}(\mathcal{C}) = \mathrm{diff}(\mathcal{C}') \\ \qquad\qquad\qquad \text{and } \mathrm{head}(\mathcal{C}) \text{ was computed no later than } \mathrm{head}(\mathcal{C}')\text{”} \end{array} \right\}.$$

Next, we define a series of useful predicates with respect to the potential chain set $\mathcal{S}_r$ and parties' local clocks at nominal round r. Note that Φ is a constant that is the ideal maximal skew of all alert clocks, and $\Phi = \Delta + \Phi_{\text{clock}}$ where Δ is the network delay and Φ_{clock} is the maximal clock drift that $\mathcal{A}$ can set (see Section 2.1).

Definition 5. *For a nominal round r, let:*

- $\textsc{GoodChains}(r) \triangleq$ *“For all $u \le r$, every chain in $\mathcal{S}_u$ is good.”*
- $\textsc{GoodRound}(r) \triangleq$ *“All rounds $u \le r$ are good.”*
- $\textsc{NoStaleChains}(r) \triangleq$ *“For all $u \le r$, there are no stale chains in $\mathcal{S}_u$.”*
- $\textsc{CommonPrefix}(r) \triangleq$ *“For all $u \le r$ and $\mathcal{C}, \mathcal{C}' \in \mathcal{S}_r$, $\mathrm{head}(\mathcal{C} \cap \mathcal{C}')$ was created after nominal round $u - \ell - 2\Delta - 2\Phi$.”*
- $\textsc{BlockLength}(r) \triangleq$ *“For all $u < r$ and $\mathcal{C} \in \mathcal{S}_u$, the blocklength Λ of any epoch* $\texttt{ep}$ *in $\mathcal{C}$ satisfies $\frac{1}{2(1+\delta)\gamma^2} \cdot mf \le \Lambda \le 2(1+\delta)\gamma^2 \cdot mf$*
- $\textsc{GoodBeacons}(r) \triangleq$ *“For all $u < r$ and the beacon set $\mathcal{S}_{\texttt{itvl}}$ bookkeeped during any interval* $\texttt{itvl}$, *more than half of beacons within $\mathcal{S}_{\texttt{itvl}}$ are generated by honest parties”.*
- $\textsc{GoodShift}(r) \triangleq$ *“For all $u < r$, and the alert party P_i that adjusts its local clock at round u, P_i computes shift_i that $-2\Phi \le \mathsf{shift}_i \le \Phi$”.*
- $\textsc{GoodSkew}(r) \triangleq$ *“For all alert parties in nominal time r, their local time in this round differs by at most Φ if they are in the same interval or differs by at most 2Φ if they are in different intervals.” Formally,*

$$\textsc{GoodSkew}(r) :\Leftrightarrow \left(\forall \mathsf{P}_1, \mathsf{P}_2 \in \mathcal{P}_{\mathsf{alert}}[r] : \left| \begin{array}{ll} |\mathtt{r}_1 - \mathtt{r}_2| \le \Phi & \text{if } \mathtt{itvl}_1 = \mathtt{itvl}_2 \\ |\mathtt{r}_1 - \mathtt{r}_2| \le 2\Phi & \text{if } \mathtt{itvl}_1 \ne \mathtt{itvl}_2 \end{array} \right. \right)$$

where $\langle \mathtt{itvl}_1, \mathtt{r}_1 \rangle$ and $\langle \mathtt{itvl}_2, \mathtt{r}_2 \rangle$ are the timestamps that P_1 and P_2 reports during r[9].

Random variables and (Δ, Φ)-isolated success. Next, for the purpose of estimating the difficulty acquired by honest parties during a sequence of rounds, we define the following random variables w.r.t. nominal round r.

- D_r: the sum of the difficulties of all blocks computed by alert parties at nominal round r.
- Y_r: the maximum difficulty among all blocks computed by alert parties at nominal round r.
- Q_r: equal to Y_r when $D_u = 0$ for all $r < u < r + \Delta + \Phi$ and 0 otherwise.

We call a nominal round r such that $D_r > 0$ *successful* and one wherein $Q_r > 0$ *isolated successful.* An isolated successful round guarantees the irreversible progress of the honest parites.

We highlight that, under the imperfect local clock model $\mathcal{F}_{\textsc{ILClock}}$, the notion of an “isolated successful round” needs to be re-considered as parties' local clocks may span some consecutive

[9] If P passes multiple local rounds in nominal round r, we require that all of these timestamps should satisfy the predicate.

rounds. Assuming a Φ-drift is maintained during the sequence of rounds we are interested in, an irreversible contribution to the chain happens when the (nominal) distance between such success and the following success is at least $\Phi + \Delta$ rounds. This is because the block producer may have a local clock that is already Φ rounds behind other alert parties, and it takes Δ rounds to diffuse the block. This cancels out other parties' successes for up to $\Phi + \Delta$ rounds. As a result, we call such event a (Δ, Φ)*-isolated successful*, which is the for the new formulation of Q_r (cf. [GKL20]). Note that this (Δ, Φ)-isolated successful round is meaningful only when the protocol is able to maintain a Φ-bounded skew during the sequence of rounds we are considering.

Recall that the total number of hash queries alert parties (resp., the adversary) can make during nominal round r is denoted by h_r (resp., t_r). For a sequence of rounds S we write $n(S) = \sum_{r \in S} n_r$ and similarly, $t(S), D(S), Q(S)$.

Regarding the adversary $\mathcal{A}$, while $\mathcal{A}$ may query the random oracle for an arbitrarily low target and obtain blocks with arbitrarily high difficulty, we wish to upper-bound the difficulty it can accrue during a set of J queries. Consider a set of consecutive adversarial queries J and associate it with the target of the first query (this target is denoted by $T(J)$). We define $A(J)$ and $B(J)$ to be equal to the sum of the difficulties of all blocks computed by the adversary during queries in J for target at least $T(J)/\tau$ and $T(J)$, respectively.

Let $\mathcal{E}_{r-1}$ fix the execution just before (nominal) round r. In particular, a value E_{r-1} of $\mathcal{E}_{r-1}$ determines the adversarial strategy and so determines the targets against which every party will query the oracle at round r and the number of parties h_r and t_r, but it does not determine D_r or Q_r. For an adversarial query j we will write $\mathcal{E}_{j-1}$ for the execution just before this query.

Blockchain properties. We use blockchain properties as formulated in [GKL15, GKL17] as an intermediate step towards proving the clock properties and achieve our blockchain synchronizer. Next, we briefly describe these properties: common prefix, chain growth, chain quality and existential chain quality.

Notably, we consider common prefix in terms of number of rounds. I.e., honest parties will agree on a settled part of the blockchain with timestamps at most a given number of rounds before their local time.[10] Let $\mathcal{C}^{\lceil k}$ denote the chain resulting from removing all rightmost blocks with timestamp larger than $\mathtt{r} - k$, where $\mathtt{r}$ is the current (local) time. We can now define common prefix as follows (we will quantify k in Corollary 6).

- **Common Prefix** (with parameter $k \in \mathbb{N}$). For any two alert parties $\mathsf{P}_1, \mathsf{P}_2$ holding chains $\mathcal{C}_1, \mathcal{C}_2$ at rounds r_1, r_2, with $r_1 \leq r_2$, it holds that $\mathcal{C}_1^{\lceil k} \preccurlyeq \mathcal{C}_2$.

Regarding chain growth, the lemma below provides a lower bound on the irreversible progress of achieved by the honest parties regardless of any adversarial behavior. This lemma has appeared in previous analyses under varying settings, evolving from the synchronous network and static environment ([GKL15]), to a dynamic environment ([GKL17]), and further to a bounded-delay network setting ([GKL20]). The next lemma extends the chain growth property to a Δ-bounded network delay, Φ-bounded clock drift and dynamic environment.

[10] While most of the previous work considers common prefix in terms of number of blocks, we note that these two definitions are equivalent. This is due to the fact that if the protocol guarantees security, then the block generation rate is somewhat steady (cf. [GKL17]) and thus the number of blocks generated during a period of time can be inferred from its length and the highest mining speed.

Lemma 1 (Chain Growth)**.** *Suppose that at nominal round u of an execution E an honest party diffuses a chain of difficulty d. Then, by (nominal) round v, every honest party has received a chain of difficulty at least $d + Q(S)$, where $S = [u + \varDelta + \varPhi, v - \varDelta - \varPhi]$.*

4.2 Protocol Parameters and Their Conditions

We summarize all Timekeeper parameters in Table 3 in Appendix E. It is worth noting that ϵ is a small constant regarding the quality of concentration of random variables (it will appear in the typical executions in Section 4.3). We introduce a parameter λ—which is related to the properties of the protocol—to simplify several expressions. Protocol parameter λ and the RO output length κ are the seucrity parameters of Timekeeper.

In order to get desired convergence and perform meaningful analysis, we consider a sufficiently long consecutive sequence of at least

$$\ell = \frac{4(1+3\epsilon)}{\epsilon^2 f[1-(1+\delta)\gamma^2 f]^{\varDelta+\varPhi+1}} \cdot \max\{\varDelta + \varPhi, \tau\} \cdot \gamma^3 \cdot \lambda \tag{6}$$

consecutive rounds.

We are now ready to discuss the conditions that protocol parameters should satisfy. We first quantify the length of a clock synchronization interval R, the length of a target recalculation interval M and the length of the convergence phase K. Specifically, we let one target recalculation epoch consists of 4 clock synchronization intervals, i.e., M = 4R; we set K = $\ell + 2\varDelta + 4\varPhi$ (this will coincide with our common prefix parameter and thus provide some desired properties; see Corollary 6).

Next, we will require that ℓ (defined in Equation (6)) is appropriately small compared to the length of an epoch and of an interval (note that M = 4R).

$$\ell + 2\varDelta + 7\varPhi \leq \epsilon \mathrm{M}/(4\gamma) = \epsilon \mathrm{R}/\gamma. \tag{C1}$$

Further, we require that the advantage of the honest parties is large enough to absorb the errors introduced by ϵ (from the concentration of random variables) and $[1-(1+\delta)\gamma^2 f]^{\varDelta+\varPhi}$ (from the network delay and clock skews).

$$[1-(1+\delta)\gamma^2 f]^{\varDelta+\varPhi} \geq 1 - \epsilon \text{ and } \epsilon \leq \delta/12 \leq 1/12. \tag{C2}$$

4.3 Typical Executions

We define the notion of *typical* executions following [GKL17, GKL20]. The idea here is that given a certain execution E, we compare the actual progress and the expected progress that parties will make under the success probabilities. If the difference and variance are reasonably small, and no bad events (see Definition 6) about the underlying hash function happen, we declare E *typical.*

Definition 6. *An insertion occurs when, given a chain $\mathcal{C}$ with two consecutive blocks $\mathcal{B}$ and $\mathcal{B}'$, a block $\mathcal{B}^*$ created after $\mathcal{B}'$ is such that $\mathcal{B}, \mathcal{B}^*, \mathcal{B}'$ form three consecutive blocks of a valid chain. A copy occurs if the same block exists in two different positions. A prediction occurs when a block extends one with later creation time.*

Note that in addition (compared to [GKL17, GKL20]), in Definition 7(a) we require that the difficulty of all blocks the alert parties can acquire during consecutive rounds S (i.e., $D(S)$) is well lower-bounded. This is because $D(S)$ also captures the beacon production process, where there is no loss incurred by the bounded-delay network as well as by skewed local clocks. Hence, a reasonably better lower-bound on $D(S)$ helps us get better results when arguing for the good properties of generated beacons by alert parties (Lemma 11).

Definition 7 (Typical Execution)**.** *An execution E is typical if the following hold.*

(a) For any set S of at least ℓ consecutive good rounds,

$$(1-\epsilon)[1-(1+\delta)\gamma^2 f]^{\Delta} ph(S) < Q(S) \leq D(S) < (1+\epsilon)ph(S) \text{ and } D(S) > (1-\epsilon)ph(S).$$

(b) For any set J of consecutive adversarial queries and $\alpha(J) = 2(\frac{1}{\epsilon}+\frac{1}{3})\lambda/T(J)$,

$$A(J) < p|J| + \max\{\epsilon p|J|, \tau\alpha(J)\} \text{ and } B(J) < p|J| + \max\{\epsilon p|J|, \alpha(J)\}.$$

(c) No insertions, no copies, and no predictions occurred in E.

In the next lemma, we establish the quantitative relation between honest and adversarial hashing power during consecutive rounds with length at least ℓ, as well as the relationship between the total difficulty acquired by all parties ($D(S) + A(J)$) and their hashing power.

Lemma 2. *Consider a typical execution in a (γ, s)-respecting environment. Let $S = \{r : u \leq r \leq v\}$ be a set of at least ℓ consecutive good rounds and J the set of adversarial queries in $U = \{r : u - \Delta - \Phi \leq r \leq v + \Delta + \Phi\}$. We have*

(a) $(1+\epsilon)p|J| \leq Q(S) \leq D(U) < (1+5\epsilon)Q(S)$.
(b) $T(J)A(J) < \epsilon \mathrm{M}/4(1+\delta)$ or $A(J) < (1+\epsilon)p|J|$; $\tau T(J)B(J) < \epsilon \mathrm{M}/4(1+\delta)$ or $B(J) < (1+\epsilon)p|J|$.
(c) If w is a good round such that $|w-r| \leq s$ for any $r \in S$, then $Q(S) > (1-\epsilon)[1-(1+\delta)\gamma^2 f]^{\Delta}|S|pn_w/\gamma$. If in addition $T(J) \geq T_w^{\min}$, then $A(J) < (1-\delta+3\epsilon)Q(S)$.
(d) If w is a good round such that $|w-r| \leq s$ for any $r \in S$ and $T(J) \geq T_w^{\min}$, then $D(S)+A(J') < (1+\epsilon)p(h(S)+|J'|)$ where J' denotes the set of adversarial queries in S.

We conclude that almost all executions (that are polynomially bounded by κ and λ) are typical.

Theorem 3. *Assuming the ITM system $(\mathcal{Z}, C)$ runs for L steps, the probability of the event "$\mathcal{E}$ is not typical" is bounded by $O(L^2)(e^{-\lambda} + 2^{-\kappa})$.*

4.4 Proof Roadmap

In the remainder of this section we present an overview of the analysis. Note that the predicates in Definition 5 are proved in an inductive way over the space of typical executions in a (γ, s)-respecting environment.

First, we focus on the steady block genetation rate. For a warm-up, we argue that an adversarial fork cannot happen too long ago and then extract the common prefix parameter. Equipped with this knowledge, we show that if good skews and certain time adjustment calculations are maintained during a target recalculation epoch, the block production rate will be properly controlled in the next epoch.

Lemma 4. $\textsc{GoodRound}(r-1) \implies \textsc{NoStaleChains}(r)$.

Lemma 5. $\textsc{GoodRound}(r-1) \wedge \textsc{GoodSkew}(r-1) \implies \textsc{CommonPrefix}(r)$.

Lemma 7. $\textsc{GoodRound}(r-1) \wedge \textsc{GoodChains}(r-1) \wedge \textsc{GoodSkew}(r-1) \wedge \textsc{GoodShift}(r-1) \implies \textsc{BlockLength}(r)$.

Lemma 8. $\textsc{GoodRound}(r-1) \implies \textsc{GoodChains}(r)$.

Corollary 9. $\textsc{GoodRound}(r-1) \implies \textsc{GoodRound}(r)$.

Next, we move to the properties w.r.t. clocks. We argue that if at the onset, the PoW difficult is appropriately set and the steady block generating rate lasts during the whole clock synchronization interval, the beacon set used by parties to update their clock will be identical and the majority of these beacons will be produced and emitted by alert parties. For synchronized parties, this good beacon set implies that the differences between alert parties' local clocks are still narrow after they enter the next interval and that the shift value they computed is well-bounded. Furthermore, regarding newly joining parties, we also provide an analysis of the joining procedure showing that joining parties starting with no *a-priori* knowledge of the global time, they can listen in and bootstrap their logical clock and become alert parties. The above two aspects imply that a bounded skew is maintained over the whole execution.

Lemma 11. $\textsc{GoodRound}(r-1) \implies \textsc{GoodBeacons}(r)$.

Lemma 13. $\textsc{GoodSkew}(r-1) \implies \textsc{GoodShift}(r)$

Lemma 16. $\textsc{GoodSkew}(r-1) \wedge \textsc{GoodBeacons}(r-1) \implies \textsc{GoodSkew}(r)$.

To sum up, a "safe" start and a (γ, s)-respecting environment guarantee that good properties can be achieved during the whole execution. We work out the related parameters (in Theorem 18) and conclude that Timekeeper solves the clock synchronization problem.

4.5 Typical Executions Maintain a Steady Block Generation Rate

We first show that the adversary cannot create a fork that happened too long ago compared with the current time. We consider a novel partition strategy (different from the approach in [GKL20]) that is better suited to the new target recalculation function.

Lemma 4. $\textsc{GoodRound}(r-1) \implies \textsc{NoStaleChains}(r)$.

Proof. Suppose—towards a contradiction—$\mathcal{C} \in \mathcal{S}_r$ and has not been extended by an honest party for at least $\ell + 2(\Phi + \Delta)$ rounds and r is the least (nominal) round with this property. Let $\mathcal{B}$ be the last honestly-generated block of $\mathcal{C}$ (possibly the genesis) and let w be the (nominal) round it was computed. We consider $S = \{u : w + (\Phi + \Delta) \leq u \leq r - (\Phi + \Delta)\}$ and $U = \{u : w \leq u \leq r\}$ ($|S| \geq \ell$ by assumption). Suppose that the blocks of $\mathcal{C}$ after $\mathcal{B}$ (we denote these blocks by B) span k epochs with corresponding targets $T_1, \ldots, T_k$. For $i \in [k]$ let m_i be the number of blocks with target T_i and set $M = m_1 + \ldots + m_k$ and $d = m_1/T_1 + \ldots + m_k/T_k$. Our plan is to contradict the assumption that $C \in \mathcal{S}_r$ by showing that all chains in $\mathcal{S}_r$ have more difficulty than $\mathcal{C}$. By Chain-Growth Lemma 1, all the honest parties have advanced (in difficulty) during the rounds in U by $Q(S)$. Therefore, to reach a contradiction it suffices to show that $d < Q(S)$.

Consider the following partition on B: we partition B into u sections $B_v, v \in [u]$ and associate each section B_v with the target of its first block T_v. Section B_v starts with either the block after $\mathcal{B}$ (if $v = 1$) or the $\lceil m_i/2 \rceil$-th block in an epoch (if $v > 1$); it ends at either the last block of the chain (if $v = u$) or the $\lfloor m_i/2 \rfloor$-th block such that in epoch $i+1$ the target is less than T_v/τ. Under such construction, the next block after partition B_v is exactly the first block of partition B_{v+1}.

For $u \geq 2$, we claim that for partition B_v, it has the following properties: (1) for all blocks in B_v, their target is at least T_v/τ; (2) the number of blocks in B_v is at least M/2. Property (1) holds because of the strategy of our partition that will stop before it exceeds the lower bound for the targets and thanks to Equation (5) we need to pass at least two boundaries of epochs so the circumstance that no blocks exist in such partition will never happen. To reason why property (2) stands, consider those epochs that are split into two different sections. For an epoch `ep` whose blocks are split into two sections B_v, B_{v+1}, since in epoch $\texttt{ep}+1$ the target is larger than that in `ep` (if not, it does not satisfy the criteria of the partition), there are at least M blocks in epoch `ep`. Otherwise, Equation (5) will raise the target. By the rule of partition, at least M/2 blocks are in each sections. Hence for every partition, either its head or tail has at least M/2 blocks in the same epoch, and this implies the lower bound of the total number of blocks.

For each $v \in \{1, 2, \dots, u\}$, let $j_v \in J$ denote the index of the query during which the first block of the v-th section was computed and set $J_v = \{j : j_v \le j < j_{i+1}\}$ (Definition 7(c) assures $j_i < j_{i+1}$). We have

$$d = \sum_{i=1}^{k} \frac{m_i}{T_i} < \sum_{v=1} (1+\epsilon)|J_v| \le (1+\epsilon)p|J| \le Q(S).$$

The difficulty of the blocks acquired in J_v is at most $A(J_v)$ by property (1) and their number at most $T_v A(J_v)$. Since property (2) above shows that the adversary acquired at least M/2 blocks in J_v, the desired bound follows from Lemma 2(b). The final inequality is Lemma 2(a).

If $u = 1$, let J denote the queries in U starting from the first adversarial query attempting to extend $\mathcal{B}$. Then, $T_1 = T(J)$ and $T_i \ge T(J)/\tau (i > 1)$; thus, $d \le A(J)$. If $A(J) < (1+\epsilon)p|J|$, then $A(J) < Q(S)$ is obtained by Lemma 2(a). Otherwise, $A(J) < (\frac{1}{\epsilon}+1)\tau\alpha(J) = 2(\frac{1}{\epsilon}+1)(\frac{1}{\epsilon}+\frac{\epsilon}{3})\tau\lambda/T(J)$. However, we have

$$\begin{aligned} Q(S) &> (1-\epsilon)[1-(1+\delta)\gamma^2 f]^{\Delta+\Phi} \cdot \frac{p n_u \ell T_1}{\gamma T_1} \\ &> \frac{(1-\epsilon)[1-(1+\delta)\gamma^2 f]^{\Delta+\Phi} f\ell}{2\gamma^3 T(J)} \ge \frac{2(1-\epsilon)(1+3\epsilon)\tau\lambda}{\epsilon^2 T(J)} \ge A(J) \end{aligned}$$

by considering only the first ℓ rounds in S hence $n(S) \ge n_u\ell/\gamma$. ☐

Lemma 5. $\textsc{GoodRound}(r-1) \wedge \textsc{GoodSkew}(r-1) \implies \textsc{CommonPrefix}(r)$.

The proof is presented in Appendix D.

Previous works [GKL17, GKL20] consider the common prefix parameters in terms of the number of blocks parties have to prune, which directly maps to the number of blocks produced in the time period implied by Lemma 5. With an imperfect local clock $\mathcal{F}_{\text{ILClock}}$ as well as the new definition of common prefix (in terms of the number of rounds that we are going to remove), parties have to prune more rounds in order to guarantee it. We establish the new Common Prefix parameter k in Corollary 6, which coincides with the K boundaries of $I_{\text{sync}}(\cdot)$ in Equation (2), and will be helpful in order to argue the good properties of clock skews.

Corollary 6. *For a typical execution in a (γ, s)-respecting environment, if all predicates in Definition 5 hold till $r-1$, for any two alert parties $\mathsf{P}_1, \mathsf{P}_2$ holding chains $\mathcal{C}_1, \mathcal{C}_2$ at round r, it holds that $\mathcal{C}_1^{\lceil k} \preccurlyeq \mathcal{C}_2$, where $k = \ell + 2\Delta + 4\Phi$.*

Proof. Lemma 5 shows that common prefix can be acquired by pruning the blocks produced in the last $\ell + 2\Delta + 2\Phi$ nominal rounds. We consider the timestamp of the first honest block $\mathcal{B}$ produced during such nominal rounds. GoodSkew$(r-1)$ indicates that the timestamp of $\mathcal{B}$ can be up to Φ-rounds behind, if $\mathcal{B}$ is produced by an alert party that is Φ-rounds behind other alert parties. Meanwhile, GoodShift$(r-1)$ implies that up to Φ logical rounds may be skipped at the boundary of an interval. Note that parties can pass at most two intervals within $\ell+2\Delta+2\Phi$ nominal rounds, and setting back local clocks does not hurt common prefix in that it is safe to prune more blocks. By summing them up, we conclude $k = \ell + 2\Delta + 4\Phi$. □

Lemma 7. GoodRound$(r-1)\wedge$GoodChains$(r-1)\wedge$GoodSkew$(r-1)\wedge$GoodShift$(r-1) \implies$ BlockLength(r).

Proof. Suppose—towards a contradiction—that BlockLength(r) is false. Then, there exists a $w \le r$ and a chain $\mathcal{C} \in \mathcal{S}_w$ with an epoch of target T and duration Λ that does not satisfy

$$\frac{1}{2(1+\delta)\gamma^2}\cdot \mathrm{M}f \le \Lambda \le 2(1+\delta)\gamma^2\cdot \mathrm{M}f.$$

We consider the earliest epoch $\mathtt{ep}$ with this property.

For the lower bound, we show that alert parties can alone produce enough blocks. Let u, v denote the first and last nominal round such that all alert parties are in epoch $\mathtt{ep}$. Such u, v exist since GoodSkew$(r-1)$ holds. Define $S = \{i : u+\ell+2\Delta+2\Phi \le i \le v-(\ell+2\Delta+2\Phi)\}$. We consider the length of S. Since one epoch consists of 4 intervals and GoodShift$(r-1)$, GoodSkew$(r-1)$ holds, we get $v-u \ge M-4\Phi-2\Phi = M-6\Phi$ (we prune Φ rounds at the beginning and end to ensure that alert parties stay in $\mathtt{ep}$). Further, by applying Condition (C1) we get $|S| \ge \mathrm{M}-(2\ell+4\Delta+10\Phi) \ge (1-\epsilon)\mathrm{M}$. We have

$$Q(S) > (1-\epsilon)[1+(1+\delta)\gamma^2 f]^{\Delta+\Phi}\cdot\frac{f|S|}{2\gamma^2 T} \ge (1-\epsilon)^3\cdot\frac{Mf}{2\gamma^2 T} \ge \frac{Mf}{2(1+\delta)\gamma^2}\cdot\frac{1}{T}.$$

The first inequality follows that nominal round $u-1$ belongs to target recalculation zone $Z_{\mathtt{ep}}$ and is good (note that $h_{u-1} \ge h_w/\gamma, w \in S$ holds); the second and third inequalities comes from the lower bound on $|S|$ and Condition (C2). By Chain Growth (Lemma 1), this implies that alert parties has produced at least $\mathrm{M}f/2(1+\delta)\gamma^2$ blocks.

For the upper bound, we show that even if the alert parties and the adversary join force, they cannot produce more than $2(1+\delta)\gamma^2\cdot\mathrm{M}f$ blocks with timestamp $\langle\mathtt{ep},\cdot\rangle$. Let u, v denote the first and last nominal round such that at least one alert party is in epoch $\mathtt{ep}$. Define $S = \{i : u \le i \le v\}$ and $S' = \{i : u-(\ell+2\Delta+2\Phi) \le i \le v+\ell+2\Delta+2\Phi\}$, and J the set of queries available to the adversary during the rounds in S' starting with the first query for target T (so that $T(J) = T$). CommonPrefix(r) implies that all adversarial queries that contributed to epoch $\mathtt{ep}$ are all in J. By GoodShift$(r-1)$ and GoodSkew$(r-1)$ we get $|S| \le \mathrm{M}+8\Phi+2\Phi = \mathrm{M}+10\Phi$ (we add Φ rounds at the beginning and end to ensure that at least one alert parties stay in $\mathtt{ep}$). Furthermore, $|S| < |S'| \le \mathrm{M}+2\ell+4\Delta+14\Phi \le (1+\epsilon)\mathrm{M}$ by Condition (C1). Since $u-1$ is a nominal round within target recalculation zone $Z_{\mathtt{ep}}$ and $h_w \le \gamma h_{u-1}$ for all $w \in S'$, it follows that $ph(S) \le p\gamma h_{u-1}|S| \le (1+\delta)\gamma^2 f|S|/T$. Hence,

$$D(S) < (1+\epsilon)ph(S) \le (1+\epsilon)(1+\delta)\gamma^2 f|S|\cdot\frac{1}{T} < (1+\epsilon)^2\cdot(1+\delta)\gamma^2\mathrm{M}f\cdot\frac{1}{T}.$$

Regarding the adversary $\mathcal{A}$, if $\tau TB(J) < \epsilon\mathrm{M}f/4$, the total number of blocks is less than $2(1+\delta)\gamma^2\mathrm{M}f$ and we are done. Otherwise,

$$B(J) < (1+\epsilon)p|J| \le (1+\epsilon)(1-\delta)(1+\delta)\gamma^2 f|S'|\cdot\frac{1}{T} \le (1+\epsilon)^2(1-\delta)(1+\delta)\gamma^2\mathrm{M}f\cdot\frac{1}{T}$$

The second inequality holds because $t_w < (1-\delta)h_w$ for all $w \in S'$, and the last one follows from the upper bound on $|S'|$. Note that $\epsilon < \delta/4$ (by Condition C2), we have $(1+\epsilon)^2(2-\delta) < 2$. I.e., the total number of blocks is less than $2(1+\delta)\gamma^2\mathrm{M}f$. □

Note that the length of S' in the above proof implies that we should require $s \geq \mathrm{M} + 2(\ell + 2\Delta + 7\Phi)$.

Lemma 8. $\textsc{GoodRound}(r-1) \Longrightarrow \textsc{GoodChains}(r)$.

Proof. Recall that the first target recalculation zone Z_1 (which consists of merely the first round) is good in our assumption, it suffices to show that if a recalculation zone Z_{ep} is good, then the next one $Z_{\mathsf{ep}+1}$ is also good. Let Λ denote the number of blocks in epoch ep. we wish to show that, for all $z \in Z_{\mathsf{ep}+1}$, $f/2\gamma \leq ph_zT_{\mathsf{ep}+1} \leq (1+\delta)\gamma f$.

We first conisder the lower bound. Consider a round $w \in Z_{\mathsf{ep}}$ and a round $z \in Z_{\mathsf{ep}+1}$. If $\Lambda \leq \mathrm{M}f/\gamma$, we get $T_{\mathsf{ep}+1} \geq \gamma T_{\mathsf{ep}}$ according to the target recalculation function (see Equation (5)). Hence, $ph_zT_{\mathsf{ep}+1} \geq ph_wT_{\mathsf{ep}+1}/\gamma \geq ph_wT_{\mathsf{ep}} \geq f/2\gamma$ in that w belongs a good recalculation zone.

If not, assume $\Lambda > \mathrm{M}f/\gamma$. Let u, v denote the first and last nominal round such that at least one alert party is in epoch ep and consider $S = \{i : u \leq i \leq v\}, S' = \{i : u - (\ell + 2\Delta + 2\Phi) \leq i \leq v + \ell + 2\Delta + 2\Phi\}$. Let J denote the set of queries available to the adversary in S'. By Lemma 5, all blocks contributed to Λ were computed during honest queries in S or adversarial ones in J. From the discussion in Lemma 7 we learn that $|S| < |S'| \leq \mathrm{M} + 4\Delta + 14\Phi \leq (1+\epsilon)\mathrm{M}$. In addition, for any round $z \in Z_{\mathsf{ep}+1}$, $h(S') \geq h_z|S'|/\gamma$ in that $|S'| < s$; similarly, $h(S) \geq h_z|S|/\gamma$. Thus, recall that $\mathrm{M} = (T_{\mathsf{ep}+1}/T_{\mathsf{ep}})(1/f)\Lambda$, we have

$$B(J) < (1-\delta)(1+\epsilon)ph(S') \leq (1-\delta)(1+\epsilon)^2p\gamma h_u\mathrm{M}$$

and $D(S) < (1+\epsilon)ph(S) \leq (1+\epsilon)^2p\gamma h_u\mathrm{M}$. For any $z \in Z_{\mathsf{ep}+1}$, assume $ph_zT_{\mathsf{ep}+1} < f/2\gamma$, we get he following contradiction.

$$2p\gamma h_zM = 2p\gamma h_z \cdot \frac{T_{\mathsf{ep}+1}}{T_{\mathsf{ep}}} \cdot \Lambda \cdot \frac{1}{f} < \Lambda \cdot \frac{1}{T_{\mathsf{ep}}} \leq D(S) + B(J) < (2-\delta)(1+\epsilon)^2p\gamma h_zM < 2p\gamma h_zM.$$

In order to prove the upper bound ($f \leq (1+\delta)\gamma f$), consider a round $w \in Z_{\mathsf{ep}}$ and a round $z \in Z_{\mathsf{ep}+1}$. If $\Lambda \geq \gamma\mathrm{M}f$, we get $T_{\mathsf{ep}+1} \leq T_{\mathsf{ep}}/\gamma$. Thus, $ph_zT_{\mathsf{ep}+1} \leq p\gamma h_wT_{\mathsf{ep}+1} \leq ph_wT_{\mathsf{ep}} \leq (1+\delta)\gamma f$ in that w belongs a good recalculation zone and we are done.

Otherwise, assume $\Lambda < \gamma\mathrm{M}f$. Let u, v denote the first and last nominal round such that all alert parties are in epoch ep and consider $S = \{i : u + \ell + 2\Delta + 2\Phi \leq i \leq v - (\ell + 2\Delta + 2\Phi)\}$. Following the argument in Lemma 7 we get $|S| \geq (1-\epsilon)\mathrm{M}$. For any $t \in Z_{\mathsf{ep}+1}$, assume $ph_zT_{\mathsf{ep}+1} > (1+\delta)\gamma f$ and recall that $\mathrm{M} = (T_{\mathsf{ep}+1}/T_{\mathsf{ep}})(1/f)\Lambda$, we obtain the following contradiction.

$$\frac{ph_z\mathrm{M}}{(1+\delta)\gamma} = \frac{ph_z}{(1+\delta)\gamma} \cdot \frac{T_{\mathsf{ep}+1}}{T_{\mathsf{ep}}} \cdot \Lambda \cdot \frac{1}{f} > \Lambda \cdot \frac{1}{T_{\mathsf{ep}}} \geq Q(S) \geq (1-\epsilon)(1-(1+\delta)\gamma^2 f)^{\Delta+\Phi} \cdot \frac{ph_z|S|}{\gamma} \geq \frac{ph_z\mathrm{M}}{(1+\delta)\gamma}.$$

The second ineuqality comes from Chain Growth (Lemma 1) (recall that $\mathcal{C} \in \mathcal{S}_r$ and the adversary can discard alert parties for at most $\ell + 2\Delta + 2\Phi$ rounds at the beginning and end of epoch as a result of $\textsc{CommonPrefix}(r)$, Lemma 5); the next one holds in that $\mathrm{M} < s$ and hence $h(S) \geq h_z|S|/\gamma$; the last inequality follows Condition (C2). □

Corollary 9. $\textsc{GoodRound}(r-1) \Longrightarrow \textsc{GoodRound}(r)$.

Proof. Consider any $\mathcal{C} \in \mathcal{S}_r$. Let $Z_{\mathtt{ep}}$ be its last target recalculation zone before r. If $r \in Z_{\mathtt{ep}}$, it follows directly by Lemma 8 that it is good. Otherwise, consider a round $w \in Z_{\mathtt{ep}}$ (recall that $f/2\gamma \leq ph_w T_{\mathtt{ep}} \leq (1+\delta)\gamma f$). Since $\textsc{GoodShift}(r-1)$, $\textsc{GoodSkew}(r-1)$ implies $r - w < \mathrm{M} + 8\varPhi + \varPhi = \mathrm{M} + 9\varPhi < s$, we have $h_r/\gamma \leq h_w \leq \gamma h_r$. Combining these two bounds we obtain the desired inequality. □

4.6 Typical Executions Maintain Good Skews

In this section, we show that in a typical execution, alert parties maintain close local clocks, and that the shift that they compute at the end of an interval is well-bounded. Before we consider $\textsc{GoodBeacons}(r)$, we establish the following lemma that bounds the adversary from pre-mining for too long a time.

Lemma 10. *Consider an interval* $\mathtt{itvl}$*, and suppose* r *is the smallest nominal round where all alert parties stay in the beacon mining and inclusion phase. Then the adversary can mine timestamp beacons w.r.t.* $\mathtt{itvl}$ *no earlier than* $r - (2\ell + 4\varDelta + 9\varPhi)$.

Proof. Since we adopt the CRS recorded in the genesis block as the original fresh randomness, by our simultaneously-start assumption, in interval $\mathtt{itvl} = 1$ the adversary can start to mine beacons at most $\mathrm{K} = \ell + 2\varDelta + 4\varPhi$ rounds before the alert parties.

Consider an interval $\mathtt{itvl} > 1$. Recall that in Equation 1, the fresh randomness w.r.t. interval $\mathtt{itvl}$ is computed by hashing the concatenation of all blocks in the previous interval, in order to prove the lemma, it suffices to show that the production time of the last block is no earlier than $r - 2\ell + 4\varDelta + 9\varPhi$. Otherwise, it implies a prediction (cf. Definition 6), which contradicts the fact that execution is typical.

For the sake of a contradiction, assume that the last block $\mathcal{B}$ in interval $\mathtt{itvl} - 1$ is computed at a nominal round $r' < r - 2\ell + 4\varDelta + 9\varPhi$. Let $\mathcal{B}'$ be the first honest block after $\mathcal{B}$. $\mathcal{B}'$ is produced at round at most $r'' = r' + \ell + 2\varDelta + 2\varPhi \leq r - (\ell + 2\varDelta + 7\varPhi)$; otherwise, the chain becomes stale at round r''. We consider the timestamp of $\mathcal{B}'$. Since $\mathcal{B}'$ is at least $\ell + 2\varDelta + 7\varPhi$ (nominal) rounds before all honest parties start to mine beacons (which happens when local clocks of all alert parties pass $\langle \mathtt{itvl}, (\mathtt{itvl}-1)\cdot \mathrm{R} + \ell + 2\varDelta + 4\varPhi \rangle$), $\mathcal{B}'$ cannot report a timestamp $\langle \mathtt{itvl}, \cdot \rangle$. This is because $\textsc{GoodShift}(r-1)$ implies that the backward shift at the end of interval $\mathtt{itvl} - 1$ is at most $2\varPhi$; $\textsc{GoodSkew}(r-1)$ indicates that all alert parties will start to mine beacons at most $\varPhi$ rounds after at least one of them enter $\langle \mathtt{itvl}, (\mathtt{itvl}-1)\cdot \mathrm{R} + \ell + 2\varDelta + 4\varPhi \rangle$ and parties will wait for $\ell + 2\varDelta + 4\varPhi$ round at the beginning of interval $\mathtt{itvl}$. After summing them up we get $\ell + 2\varDelta + 7\varPhi$. Hence, $\mathcal{B}'$ must report a timestamp $\langle \mathtt{itvl} - 1, \cdot \rangle$. This contradicts our assumption that $\mathcal{B}$ is the last block in $\mathtt{itvl} - 1$. □

Next, we prove $\textsc{GoodBeacons}(r)$. This predicate implies that at the boundary of every synchronization interval, alert parties will use the same set of beacons to update their local clock; further, the majority of these beacons are produced and issued by alert parties.

Note that, when the sequence of rounds S and queries J are appropriately selected, random variables $D(S)$ and $A(J)$ directly map to the number of synchronization beacons that protocol participants can produce during the beacon mining and inclusion phase. We are interested in using a typical execution to lower-bound the success of alert parties and upper-bound the success of the adversary (i.e., lower-bound $D(S)$ and upper-bound $A(J)$ in Definition 7).

Lemma 11. $\textsc{GoodRound}(r-1) \implies \textsc{GoodBeacons}(r)$.

Proof. Consider an interval `itvl` and a chain $\mathcal{C}$ held by an alert party at local round $\langle \texttt{itvl}, \texttt{itvl} \cdot \mathrm{R}\rangle$. Let d denote the sum of difficulties of the synchronization beacons recorded in blocks with timestamp round $I_{\text{sync}}(\texttt{itvl})$.

Since $\mathrm{K} = \ell + 2\Delta + 4\Phi$ by Corollary 6, honest parties agree on the beacon mining and inclusion phase in $\mathcal{C}$. Hence, every honest party would have the same view of the beacon set they are going to use. Now we prove that the majority of these beacons are prodcued by alert parties.

Let u denote the first (nominal) round such that all alert parties are mining beacons w.r.t. interval `itvl`. Consider a set of consecutive nominal rounds $S = \{i : u \le i \le v\}$ where $v = u + \mathrm{R} - 2(\ell + 2\Delta + 4\Phi) - \Phi$, where all the honest queries in S are doing 2-for-1 PoW w.r.t. interval `itvl` and hence contribute to the honset beacon set. Let $\mathcal{B}$ be the last block produced by honest parties before round v and denote its production time by w (in terms of the nominal time index). Since $\mathcal{C}$ will become stale if there is no honest block since w for $\ell + 2\Delta + 2\Phi$ rounds, we get that $w < v - (\ell + 2\Delta + 2\Phi)$.

Let $S_1 = \{i : u \le i \le w - (\Delta + \Phi)\}$ and $S_2 = \{i : u - (2\ell + 4\Delta + 9\Phi) \le i \le w + (\ell + 2\Delta + 2\Phi)\}$. S_1 is the time interval that honest success can contribute to the beacon set w.r.t. interval `itvl`; and S_2 is for the adversary. The lower bound of S_1 is derived from the definition of u and the upper bound is because it will take up to $\Delta + \Phi$ rounds for all beacons to be diffused to and accepted by all alert parties. The lower bound of S_2 is acquired due to the unpredictability discussed in Lemma 10. Regarding the upper bound of S_2, it is achieved by considering the first honest block $\mathcal{B}'$ after v, which is produced no later than $w' = w + \ell + 2\Delta + 2\Phi$ (otherwise it violates $\textsc{NoStaleChains}$). The adversary can no longer include beacons to the mining and inclusion phase after w' as it can no longer revert $\mathcal{B}'$, so all the subsecquent beacons produced after w' are invalid w.r.t. the current chain. Note that $|S_1| \ge \mathrm{R} - (3\ell + 7\Delta + 12\Phi) \ge \mathrm{R} - 3(\ell + 2\Delta + 7\Phi)$ and $|S_2 \backslash S_1| = 3\ell + 7\Delta + 12\Phi \le 3(\ell + 2\Delta + 7\Phi)$.

Let J denote the adversarial queries associated with S_2. In order to prove that alert parties can produce at least half of synchronization beacons, it suffices to show that

$$D(S_1) > d/2.$$

We first show that the number of RO queries alert parties can make during S_2 is at most 4ϵ more than those in S_1. We have

$$h(S_2) \le (1+\frac{\gamma|S_2\backslash S_1|}{|S_1|})h(S_1) \le (1+\frac{3\gamma(\ell+2\Delta+7\Phi)}{\mathrm{R}-3(\ell+2\Delta+7\Phi)})h(S_1) \le (1+\frac{3\epsilon}{1-3\epsilon/\gamma})h(S_1) < (1+4\epsilon)h(S_1).$$

The first inequality follows from Fact 1(b); the third one holds since $\ell + 2\Delta + 7\Phi \le \epsilon\mathrm{R}/\gamma$ by Condition (C1); the last inequality is a consequence of Condition (C2) ($\epsilon < 1/12$). Next,

$$\begin{aligned} D(S_1) &\ge (1-\epsilon)ph(S_1) > (1-5\epsilon)ph(S_2) > \frac{1-5\epsilon}{2-\delta}p[h(S_2)+|J|] \\ &> \frac{1-6\epsilon}{2-\delta}[D(S_2)+A(J)] > \frac{1-6\epsilon}{2-\delta}[D(S_1)+A(J)] \ge \frac{1}{2}[D(S_1)+A(J)] = \frac{d}{2}. \end{aligned}$$

The first inequality follows from typical execution (Definition 7(a)); the second one is achieved by substituting $h(S_1)$ with $h(S_2)$; the next inequality follows from the honest majority assumption; and the forth one is by applying Lemma 2(d); the last inequality holds due to Condition (C2) ($\delta \geq 12\epsilon$). $\square$

Given an honest-majority beacon set, we are ready to prove GoodSkew(r) and GoodShift(r). Regarding the proof of GoodSkew(r), note that we will consider both the synchronized parties and the newly joining parties. The proof approach is as follows. We first prove that for parties in the same interval, their local clock can deviate for up to Φ rounds (Lemma 12). Next, we show that the shift computed at the boundary of each interval satisfies our GoodShift(p) redicate (Lemma 13). Then, we argue that the 2Φ-drift holds for parties in different intervals (Lemma 14). For those newly joining parties, we also prove that they can learn a clock value that is Φ-round close to any of the alert parties (Lemma 15). Finally, combining Lemma 12, Lemma 14 and Lemma 15, we obtain the desired clock skews.

Lemma 12. *For a typical execution in a (γ, s)-respecting environment, if all predicates in Definition 5 hold till $r-1$, then at round r, the local time of alert parties in the same interval differs by at most Φ.*

Proof. For nominal rounds that all alert parties stay in interval $\mathtt{itvl} = 1$, their local clock can differ with each other for up to $\Phi_{\text{clock}} < \Phi$ rounds in that no clock synchronization happened and only the adversary can set a Φ_{clock}-bounded drift. Recall that alert parties only adjust their local clock (call SyncProc, see Appendix C.10) when it enters the last round of current interval.

Now we consider a nominal round r such that at least one alert party enters the next interval $\mathtt{itvl}+1$. We show that for those parties that have finished adjusting their clock, the logical time that they report can deviate from each other for up to Φ rounds.

According to Lemma 11, alert parties will agree on the same synchronization beacon set (denoted by $SB_{\mathtt{itvl}}$) at the end of $\mathtt{itvl}$. Fix a beacon $\mathsf{SB} \in SB_{\mathtt{itvl}}$. Consider two alert parties P_1 and P_2 that have enterd $\mathtt{itvl}+1$. We are going to show that the quantity

$$\mu(\mathsf{P}_i, \mathsf{SB}) \triangleq \mathtt{r}_i + \mathsf{Timestamp}(\mathsf{SB}) - \mathsf{P}_i.\mathsf{arrivalTime}(\mathsf{SB})$$

will differ by at most Φ between any two alert parties P_1 and P_2, where $\mathtt{r}_i$ refers to the time that party P_i reports at nominal time r if it does not call SyncProc at the end of $\mathtt{itvl}$. More precisely, $\mu(\mathsf{P}_i, \mathsf{SB})$ is the logical time that P_i will report at nominal round r[11] if it adopts SB and computes the corresponding shift shift_i to update its clock (at the end of $\mathtt{itvl}$).

The arrival time of SB bookkeeped by P_i can be represented by

$$\mathsf{P}_i.\mathsf{arrivalTime}(\mathsf{SB}) = \mathtt{r}_i - (r - r_{\mathsf{SB}}) + \delta_{\mathsf{P}_i,\mathsf{SB}} + \varphi_{\mathsf{P}_i,\mathsf{SB}}.$$

r_{SB} is the nominal round that SB is emitted to the network if SB is honest, and is the first nominal round such that at least one honest party receives SB if SB is adversarial. $\delta_{\mathsf{P}_i,\mathsf{SB}} \in [\Delta]$ is the time elapsed (counted by rounds in terms of the nominal time) for SB to be delivered to P_i. $\varphi_{\mathsf{P}_i,\mathsf{SB}} \in [-\Phi_{\text{clock}}, \Phi_{\text{clock}}]$ is the drift of P_i when SB was sent (note that for two alert parties $\mathsf{P}_1, \mathsf{P}_2$ we have $|\varphi_{\mathsf{P}_1,\mathsf{SB}} - \varphi_{\mathsf{P}_2,\mathsf{SB}}| \leq \Phi_{\text{clock}}$ by the clock drift assumption). By substituting we get

$$\mu(\mathsf{P}_i, \mathsf{SB}) = \mathtt{r}_i + \mathsf{shift}_i = \mathsf{Timestamp}(\mathsf{SB}) + r - r_{\mathsf{SB}} - \delta_{\mathsf{P}_i,\mathsf{SB}} - \varphi_{\mathsf{P}_i,\mathsf{SB}}. \tag{7}$$

[11] We note that while an alert party may pass several logical rounds in a nominal round, it does not hurt our argument here as long as the adversarial drift is bounded by an absolute value.

Note that for different parties at nominal round r, $\mathsf{Timestamp}(\mathtt{SB}) + r - r_{\mathtt{SB}}$ is a constant. Hence $|\mu(\mathsf{P}_1, \mathtt{SB}) - \mu(\mathsf{P}_2, \mathtt{SB})| \leq \Phi$. We highlight this holds for adversarially generated $\mathtt{SB}$ in that after $\mathtt{SB}$ is delivered to at least one honest party (denoted by $r_{\mathtt{SB}}$), it will be delivered to all honest parties within Δ rounds.

Consider two alert parties $\mathsf{P}_1, \mathsf{P}_2 \in \mathcal{P}_{\mathsf{alert}}[r]$ with timestamp $\langle \mathtt{itvl}+1, \cdot \rangle$ (i.e., they have finished synchronization). Tuples $(\mu(\mathsf{P}_1, \mathtt{SB}))_{\mathtt{SB} \in \mathcal{S}_{\mathtt{itvl}}}$ and $(\mu(\mathsf{P}_2, \mathtt{SB}))_{\mathtt{SB} \in \mathcal{S}_{\mathtt{itvl}}}$ are all the potential times that they will report. These two tuples are of the same size, and $\forall \mathtt{SB} \in \mathcal{S}_{\mathtt{itvl}}, |\mu(\mathsf{P}_1, \mathtt{SB}) - \mu(\mathsf{P}_2, \mathtt{SB})| \leq \Phi$. Due to Fact 2 we get

$$\left| \mathsf{med}\Big((\mu(\mathsf{P}_1, \mathtt{SB}))_{\mathtt{SB} \in \mathcal{S}_{\mathtt{itvl}}} \Big) - \mathsf{med}\Big((\mu(\mathsf{P}_2, \mathtt{SB}))_{\mathtt{SB} \in \mathcal{S}_{\mathtt{itvl}}} \Big) \right| \leq \Phi.$$

Recalling Equation (3), $\mathsf{med}(\mu(\mathsf{P}_i, \mathtt{SB}))_{\mathtt{SB} \in \mathcal{S}_{\mathtt{itvl}}} = r_i + \mathsf{med}\{\mathsf{Timestamp}(\mathtt{SB}) - \mathsf{P}_i.\mathsf{arrivalTime}(\mathtt{SB}) \,|\, \mathtt{SB} \in \mathcal{S}_{\mathtt{itvl}}\} = r_i + \mathsf{shift}^{\mathsf{P}_i}_{\mathtt{itvl}}$ is the logical time that P_i reports after the synchronization. Therefore alert clocks that have already entered $\mathtt{itvl} + 1$ will deviate from each other for up to Φ rounds.

Finally, in order to conclude the proof, it suffices to show that during nominal rounds in interval $\mathtt{itvl} > 1$ that no adjustment happens, the difference of alert parties' local clocks will not deviate for more than Φ rounds. This always holds in that the difference of honest parties' drifts in the execution of $\mathsf{Timekeeper}$ is an absolute term (recall Φ_{clock} and the condition check in CLOCK-FORWARD as well as CLOCK-BACKWARD in $\mathcal{F}_{\text{ILCLOCK}}$), thus adversarial manipulation cannot set $\varphi_{\mathsf{P}_i, \mathtt{SB}}$ to a value that violates the upper bound of clock skews. □

Lemma 13. GOODSKEW$(r-1) \Longrightarrow$ GOODSHIFT(r)

Proof. For a party P that calls $\mathsf{SyncProc}$ at nominal round r, we prove that $-2\Phi < \mathsf{shift} < \Phi$.

Let $\mathtt{SB}$ denote the median beacon P adopted, and shift the corresponding shift value. Since GOODBEACONS$(r-1)$ holds, we will have two alert parties P_1 and P_2 who produce synchronization beacons $\mathtt{SB}_1$ and $\mathtt{SB}_2$, respectively. We denote the shifts w.r.t. $\mathtt{SB}_1, \mathtt{SB}_2$ by $\mathsf{shift}_1, \mathsf{shift}_2$ (based on P's local arrival timetable) and learn that $\mathsf{shift}_1 \leq \mathsf{shift} \leq \mathsf{shift}_2$. This implies $\mu(\mathsf{P}, \mathtt{SB}_1) \leq \mu(\mathsf{P}, \mathtt{SB}) \leq \mu(\mathsf{P}, \mathtt{SB}_2)$ (recall the definition of $\mu(\mathsf{P}, \mathtt{SB})$ in Lemma 12). Further, we extract these quantities by Equation (7) in the following way. For $\mu(\mathsf{P}, \mathtt{SB})$, we consider it with respect to the shift value, and for the rest we substitute them in terms of the timestamp recorded in $\mathtt{SB}_i$ (i.e., the last expression in Equation (7)). We have

$$\mathsf{Timestamp}(\mathtt{SB}_1) - r_{\mathtt{SB}_1} - \delta_{\mathsf{P}, \mathtt{SB}_1} - \varphi_{\mathsf{P}, \mathtt{SB}_1} \leq \mathtt{r}_i - r + \mathsf{shift} \leq \mathsf{Timestamp}(\mathtt{SB}_2) - r_{\mathtt{SB}_2} - \delta_{\mathsf{P}, \mathtt{SB}_2} - \varphi_{\mathsf{P}, \mathtt{SB}_2}.$$

By letting $\delta_{\mathsf{P}, \mathtt{SB}_1} = \Delta$, $\varphi_{\mathsf{P}, \mathtt{SB}_1} = \Phi_{\text{clock}}$ and $\delta_{\mathsf{P}, \mathtt{SB}_2} = \varphi_{\mathsf{P}, \mathtt{SB}_2} = 0$, we get

$$\mathsf{Timestamp}(\mathtt{SB}_1) - r_{\mathtt{SB}_1} - \Phi \leq \mathtt{r}_i - r + \mathsf{shift} \leq \mathsf{Timestamp}(\mathtt{SB}_2) - r_{\mathtt{SB}_2}. \tag{8}$$

Note that no clock adjustment on P happened during nominal time $[\min\{r_{\mathtt{SB}_1}, r_{\mathtt{SB}_2}\}, r)$. Moreover, if the adversary does not set a drift and the beacon is delivered to all alert parties immediately after it was mined, we will have $\mathtt{r}_i - r = \mathsf{Timestamp}(\mathtt{SB}_i) - r_{\mathtt{SB}_i}$. Since the adversarial drift is upper bounded by the absolute term Φ_{clock}, and the network is Δ-bounded delay, we learn that the difference of $\mathsf{Timestamp}(\mathtt{SB}_i) - r_{\mathtt{SB}_i}$ and $\mathtt{r}_i - r$ is bounded by $\Phi_{\text{clock}} + \Delta = \Phi$. Therefore, we get

$$\left| (\mathsf{Timestamp}(\mathtt{SB}_i) - r_{\mathtt{SB}_i}) - (\mathtt{r}_i - r) \right| \leq \Phi. \tag{9}$$

After combining Equations (8) and (9), we get the desired range for shift, namely, $-2\Phi < \mathsf{shift} < \Phi$. □

Lemma 14. *For a typical execution in a (γ, s)-respecting environment, if all predicates in Definition 5 hold till $r-1$, then at round r, the local time of alert parties in different interval differs by at most 2Φ.*

Proof. Consider two alert parties $\mathsf{P}_1, \mathsf{P}_2$ that P_1 finishes running SyncProc at nominal time r and P_2 does not. Note that before the synchronization, $\mathtt{r}_1 - \Phi \leq \mathtt{r}_2 < \mathtt{r}_1$ holds becasue they are alert and hence skew predicate satisfies (and, since P_1 runs SyncProc first, its local time is larger than P_2). Let $\mathtt{r}'_1 = \mathtt{r}_1 + \mathsf{shift}_1$ denote the logical time that P_1 owns after synchronization, we have

$$|\mathtt{r}_2 - \mathtt{r}'_1| \leq 2\Phi$$

after combining $\mathtt{r}_1 - \Phi \leq \mathtt{r}_2 < \mathtt{r}_1$ and $-2\Phi < \mathsf{shift}_1 < \Phi$ from Lemma 13. □

Lemma 15. *For a typical execution in a (γ, s)-respecting environment, if all predicates in Definition 5 hold till $r-1$, then for a newly joining party who concludes the join procedure at round r, the following properties hold.*

(a) Let $\mathcal{C}_{\mathsf{join}}$ denote the chain held by $\mathsf{P}_{\mathsf{join}}$ at nominal round $r' \leq r$ and $\mathsf{P}_{\mathsf{join}}$ is in Phase C or D; let $\mathcal{C}_{\mathsf{alert}}$ denote a chain held by $\mathsf{P}_{\mathsf{alert}}$ at the same nominal time. Then $\mathcal{C}_{\mathsf{alert}}^{\lceil K} \preceq \mathcal{C}_{\mathsf{join}}$.

(b) The index value i^* set in line 34 of its joining procedure JoinProc satisfies $i^* \geq 1$.

(c) When the joining party becomes alert, it will report a logical clock that is at most Φ-rounds apart from any other alert party that is in the same interval.

Proof. Regarding claim(a), notice that the heaviest chain selection rule used in Selectchain (Appendix C.8) does not involve the local time `localTime` of the party executing it. $\mathsf{P}_{\mathsf{join}}$ and $\mathsf{P}_{\mathsf{alert}}$ would make the same decision except that $\mathsf{P}_{\mathsf{join}}$ may consider and adopt those chains in the `futureChains` set of $\mathsf{P}_{\mathsf{alert}}$. Therefore, it suffices to show that no chain with future timestamps and more accumulated difficulties than $\mathcal{C}_{\mathsf{alert}}$ can have a fork from $\mathcal{C}_{\mathsf{alert}}$ that is too earlier with respect to r'.

Assume at nominal round r', $\mathsf{P}_{\mathsf{join}}$ is in Phase C or D holding a chain $\mathcal{C}_{\mathsf{join}}$ such that $\mathrm{diff}(\mathcal{C}_{\mathsf{join}}) > \mathrm{diff}(\mathcal{C}_{\mathsf{alert}})$ and $\mathcal{C}_{\mathsf{alert}}^{\lceil K} \not\preceq \mathcal{C}_{\mathsf{join}}$. Consider a *virtual execution* for party $\mathsf{P}_{\mathsf{alert}}$ (cf. [BGK+18]). This is an artificial random experiment that consists of the execution of the protocol with an additional "virtual" party $\mathsf{P}_{\mathsf{virt}}$ that participates from the beginning, and is always alert. $\mathsf{P}_{\mathsf{virt}}$ does not query the random oracle thus is passive. Starting from the point of execution where $\mathsf{P}_{\mathsf{join}}$ joins the system, $\mathsf{P}_{\mathsf{virt}}$ advances exactly like $\mathsf{P}_{\mathsf{join}}$ and receives the same messages in the same round and order as $\mathsf{P}_{\mathsf{join}}$. When $\mathsf{P}_{\mathsf{join}}$ adopts a chain $\mathcal{C}_{\mathsf{join}}$ that $\mathsf{P}_{\mathsf{virt}}$ does not adopt at nominal round r', since it contains more difficulties, we have $\mathcal{C}_{\mathsf{join}} \in \mathcal{S}_{r'}$. By Lemma 5 we know that the creation time of $\mathrm{head}(\mathcal{C}_{\mathsf{join}} \cap \mathcal{C}_{\mathsf{virt}})$ is larger than $r' - (\ell + 2\Delta + 2\Phi)$. With the similar argument in Corollary 6 (note that a chain received by $\mathsf{P}_{\mathsf{virt}}$ will be considered by $\mathsf{P}_{\mathsf{alert}}$ within $(\Delta + \Phi)$ nominal rounds) we get $\mathcal{C}_{\mathsf{alert}}^{\lceil K} \preceq \mathcal{C}_{\mathsf{join}}$.

Moving to claim(b), in order to prove $i^* \geq 1$, we show that $\mathsf{P}_{\mathsf{join}}$ has observed at least one full synchronization interval that started at least t_{pre} rounds after the beginning of Phase C. Let $r_{\mathsf{start}}^{(j)}$ denote the nominal time such that at least one alert party passes local round $\langle j, (j-1)\cdot \mathrm{R} + K\rangle$ (in other words, enters the beacon mining and inclusion phase w.r.t. interval j). Since $\textsc{GoodSkew}(r-1)$ and $\textsc{GoodShift}(r-1)$ holds, we have

$$r_{\mathsf{start}}^{(j+1)} - r_{\mathsf{start}}^{(j)} \leq \mathrm{R} + 3\Phi.$$

Let i^* denote the minimal i such that $r^{(i)}_{\mathsf{start}} \geq t_{\text{join}} + t_{\text{off}} + t_{\text{pre}}$ where t_{join} is the norminal time such that $\mathsf{P_{join}}$ start to run $\mathsf{JoinProc}$. In other words, $r^{(i^*)}_{\mathsf{start}}$ occurs at least t_{pre} rounds after the beginning of $\mathsf{P_{join}}$'s Phase C. We have

$$r^{(i^*)}_{\mathsf{start}} \leq t_{\text{join}} + t_{\text{off}} + t_{\text{pre}} + \mathrm{R} + 3\Phi \leq t_{\text{join}} + t_{\text{off}} + t_{\text{gather}} - \mathrm{R}.$$

The first inequality comes from the upper bound on the distance of $r^{(j)}_{\mathsf{start}}$ and $r^{(j+1)}_{\mathsf{start}}$; the next ineuqality follows from the values of t_{gather} and t_{pre}.

Since the length of a beacon mining and inclusion phase is $\mathrm{R} - 2\mathrm{K}$, $r^{(i^*)}_{\mathsf{start}}$ is at least R nominal rounds before the end of Phase C. On the other hand, it takes at most $\Delta + \Phi$ rounds to diffuse the beacons. $\mathsf{P_{join}}$ will record the arrival time of all beacons in interval i^*. Note that the length of t_{pre} and the upper bound on duration of pre-mining stage in Lemma 10 guarantees that the adversary cannot produce valid beacons w.r.t. interval i^* before the begining of Phase C.

Finally, for claim(c), we show that every time $\mathsf{P_{join}}$ will adopt the same beacon set as $\mathsf{P_{alert}}$ to update its local clock, and then they will update to a local time that maintains a good skew. Recall that in claim(b) we show that $r^{(i^*)}_{\mathsf{start}}$ is at least R nominal rounds before the end of Phase C, at the end of Phase C the blocks in beacon mining and inclusion phase w.r.t. interval i^* went into the settled blockchain. This also concludes that $\mathsf{P_{join}}$ will use the same beacon set as $\mathsf{P_{alert}}$. Regarding $i \geq i^*$ they can be proved in the same way.

The rest of the proof follows that in Lemma 12. Note that in Equation (7), the new local time after adjustment is irrelevant to its previous local time, hence after the computation, the distance of $\mathsf{P_{alert}}$ and $\mathsf{P_{join}}$'s local clocks will differ for at most Φ rounds. □

Lemma 16. $\textsc{GoodSkew}(r-1) \wedge \textsc{GoodBeacons}(r-1) \Longrightarrow \textsc{GoodSkew}(r)$.

The proof directly follows from Lemma 12, Lemma 14 and Lemma 15.

Theorem 17. *For a typical execution in a $(\gamma, M + 2(\ell + 2\Delta + 7\Phi))$-respecting environment, if Condition C1 and Condition C2 are satisfied, then all predicates in Definition 5 hold.*

Finally, we are able to prove our promised goal:

Theorem 18. *Consider an execution of* $\mathsf{Timekeeper}$ *in a $(\gamma, M+2(\ell+2\Delta+7\Phi))$-respecting environment. If Conditions* (C1) *and* (C2) *are satisfied, then the protocol achieves clock synchronization (Definition 1) with parameter values*

$$\mathsf{Skew} = 2\Phi, \quad \mathsf{shiftLB} = 3\Phi/\mathrm{R}, \quad \mathsf{shiftUB} = 2\Phi/\mathrm{R},$$

except with probability negligibly small in κ and λ.

Proof. The proof of bounded skews and the value of Skew directly follow from Lemma 16. Regarding linear envelope, consider a nominal round $r > \mathrm{R}$ and an alert party $\mathsf{P} \in \mathcal{P}_{\mathsf{alert}}[r]$. Consider a *virtual execution* for party $\mathsf{P_{alert}}$ (cf. [BGK+18]) with an additional party $\mathsf{P_{virt}}$ who keeps alert from the beginning of the execution. Intuitively, the largest local time of $\mathsf{P_{virt}}$ is achieved when $\mathsf{P_{virt}}$ keeps skipping Φ rounds at the beginning of each interval; the lowest local time will be reported when $\mathsf{P_{virt}}$ always set backward his local time for 2Φ rounds instead. In other words, when the local clock of $\mathsf{P}_{\mathsf{virt}}$ passes one interval, the nominal time elapsed is bounded by $\mathrm{R} - \varPhi$ and $\mathrm{R} + 2\varPhi$. Now, consider the local time $\mathtt{r}$ that $\mathsf{P}_{\mathsf{alert}}$ will report at nominal round r. We have

$$r - 2\varPhi \cdot \left\lceil \frac{r - \mathrm{R}}{\mathrm{R} + 2\varPhi} \right\rceil \leq \mathtt{r} \leq r + \varPhi \cdot \left\lceil \frac{r - \mathrm{R}}{\mathrm{R} - \varPhi} \right\rceil.$$

Note that there is no adjustment at the beginning of first interval. By dropping the ceil and rearranging we get

$$\frac{1}{1 + 3\varPhi/\mathrm{R}} \cdot r \leq \mathtt{r} \leq \frac{1}{1 - 2\varPhi/\mathrm{R}} \cdot r.$$

We then conclude an alert party's local clock stays in the (U, L)-linear envelope with parameters $\mathsf{shiftLB} = 3\varPhi/\mathrm{R}$ and $\mathsf{shiftUB} = 2\varPhi/\mathrm{R}$. □

References

[ADD+19] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous byzantine agreement with expected O(1) rounds, expected $o(n^2)$ communication, and optimal resilience. In Ian Goldberg and Tyler Moore, editors, *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*, volume 11598 of *Lecture Notes in Computer Science*, pages 320–334. Springer, 2019.

[Bah13] Lear Bahack. Theoretical bitcoin attacks with less than half of the computational power (draft). Cryptology ePrint Archive, Report 2013/868, 2013. `https://ia.cr/2013/868`.

[BGK+18] 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*, CCS '18, pages 913–930, New York, NY, USA, 2018. Association for Computing Machinery.

[BGK+21] Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Dynamic ad hoc clock synchronization. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021*, pages 399–428, Cham, 2021. Springer International Publishing.

[BKT+19] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 585–602, New York, NY, USA, 2019. Association for Computing Machinery.

[BMTZ17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 324–356, Cham, 2017. Springer International Publishing.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145. IEEE Computer Society, 2001.

[DHS86] Danny Dolev, Joseph Y. Halpern, and H. Raymond Strong. On the possibility and impossibility of achieving clock synchronization. *J. Comput. Syst. Sci.*, 32(2):230–250, 1986.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, Apr 1988.

[DP09] Devdatt P. Dubhashi and Alessandro Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms.* Cambridge University Press, 2009.

[GK20] Juan A. Garay and Aggelos Kiayias. Sok: A consensus taxonomy in the blockchain era. In Stanislaw Jarecki, editor, *Topics in Cryptology - CT-RSA 2020 - The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings*, volume 12006 of *Lecture Notes in Computer Science*, pages 284–318. Springer, 2020.

[GKL15] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015*, pages 281–310, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[GKL17] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 291–323, Cham, 2017. Springer International Publishing.

[GKL20] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. Full analysis of nakamoto consensus in bounded-delay networks. Cryptology ePrint Archive, Report 2020/277, 2020. `https://ia.cr/2020/277`.

[GKO$^+$20] Juan Garay, Aggelos Kiayias, Rafail M. Ostrovsky, Giorgos Panagiotakos, and Vassilis Zikas. Resource-restricted cryptography: Revisiting mpc bounds in the proof-of-work era. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 129–158, Cham, 2020. Springer International Publishing.

[HSSD84] Joseph Y. Halpern, Barbara Simons, Ray Strong, and Danny Dolev. Fault-tolerant clock synchronization. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, pages 89–102, New York, NY, USA, 1984. Association for Computing Machinery.

[KMTZ13] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In *Proceedings of the 10th Theory of Cryptography Conference on Theory of Cryptography*, TCC'13, pages 477–498, Berlin, Heidelberg, 2013. Springer-Verlag.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[LL22] Christoph Lenzen and Julian Loss. Optimal clock synchronization with signatures. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, PODC'22, pages 440–449, New York, NY, USA, 2022. Association for Computing Machinery.

[LMS84] Leslie Lamport and P. M. Melliar-Smith. Byzantine clock synchronization. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, pages 68–74, New York, NY, USA, 1984. Association for Computing Machinery.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. `http://bitcoin.org/bitcoin.pdf`.

[PS17a] Rafael Pass and Elaine Shi. Fruitchains: A fair blockchain. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 315–324, New York, NY, USA, 2017. Association for Computing Machinery.

[PS17b] Rafael Pass and Elaine Shi. Rethinking large-scale consensus. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 115–129. IEEE Computer Society, 2017.

[PSS17] Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*, volume 10211 of *Lecture Notes in Computer Science*, pages 643–673, 2017.

[ST87] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *J. ACM*, 34(3):626–645, 1987.

[WL88] Jennifer Lundelius Welch and Nancy Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, 1988.

A Mathematical Facts

Fact 1 captures some facts (Especially, the relation between the total number of hash queries in S and the number of queries in a specific round) within a (γ, s)-respecting environment.

Fact 1. *Let U be a set of at most s consecutive rounds in a (γ, s)-respecting environment and $S \subseteq U$.*

(a) For any $h \in \{h_r : r \in U\}$, $\frac{h}{\gamma} < \frac{h(S)}{|S|} \leq \gamma h$.

(b) $h(U) \leq (1 + \frac{\gamma|U \setminus S|}{|S|} h(S))$.

(c) $|S| \sum_{r \in S} (ph_r)^2 \leq \gamma(ph(S))^2$.

Definition 8. *[DP09, Definition 5.3] A sequence of random variables $(X_0, X_1, \ldots)$ is a martingale with respect to the sequence $(Y_0, Y_1, \ldots)$, if, for all $n \geq 0$, X_n is determined by $Y_0, \ldots, Y_n$ and $\mathbb{E}[X_{n+1} \mid Y_0, \ldots, Y_n] = X_n$.*

Theorem 19. *Let $(X_0, X_1, \ldots)$ be a martingale with respect to the sequence $(Y_0, Y_1, \ldots)$. Suppose an event G implies*

$$X_k - X_{k-1} \leq b \text{ (for all } k\text{) and } V = \sum_k \mathbf{Var}[X_k - X_{k-1} \mid Y_1, \ldots, Y_{k-1}] \leq v.$$

Then, for non-negative n and t,

$$\mathbf{Pr}[X_n \geq X_0 + t \wedge G] \leq \exp\Big\{-\frac{t^2}{2v + 2bt/3}\Big\}.$$

Fact 2. *Let $(a_i)_{i=1}^n$ and $(b_i)_{i=1}^n$ be two sequences of n integers each, with the property that $\forall i \in [n], |a_i - b_i| \leq \varPhi$. Then we have $|\mathsf{med}((a_i)_{i=1}^n) - \mathsf{med}((a_i)_{i=1}^n)| \leq \varPhi$.*

B Model (Cont'd)

B.1 Imperfect Local Clock Functionality

Functionality $\mathcal{F}_{\text{ILClock}}^{\varPhi_{\text{clock}}}$

The functionality manages the set P of registered identities (i.e., parties $\mathsf{P} = (\text{pid}, \text{sid})$) and the set $\mathcal{F}$ of functionalities (together with their session identifier). It also manages an integer variable $\texttt{time}$. Initially, $\mathcal{P} := \emptyset, \mathcal{F} := \emptyset$ and $\texttt{time} = 1$.

For each identity $\mathsf{P} := (\text{pid}, \text{sid}) \in \mathcal{P}$ it manages variable $d_{\mathsf{P}} \in \{\mathsf{ongoing}, \mathsf{done}\}$, an integer b_{P} and an integer $\texttt{drift}_{\mathsf{P}}$. Initially, $d_{\mathsf{P}} = \mathsf{ongoing}, b_{\mathsf{P}} = 1$ and $\texttt{drift}_{\mathsf{P}} = 0$. For each pair $(\mathcal{F}, \text{sid}) \in F$ it manages variable $d_{(\mathcal{F},\text{sid})} \in \{\mathsf{ongoing}, \mathsf{done}\}$ (initially set to $\mathsf{ongoing}$).

Synchronization:

- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C)$ from some party $\mathsf{P} \in \mathcal{P}$ set $d_{\mathsf{P}} := \mathsf{done}$ and $b_{\mathsf{P}} := b_{\mathsf{P}} - 1$; execute *Round-Update* and forward $(\text{CLOCK-UPDATE}, \text{sid}_C, \mathsf{P})$ to $\mathcal{A}$.
- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C)$ from some functionality $\mathcal{F}$ in a session sid such that $(\mathcal{F}, \text{sid}) \in F$ set $d_{(\mathcal{F},\text{sid})} := 1$, execute ROUND-UPDATE and return $(\text{CLOCK-UPDATE}, \text{sid}_C, \mathcal{F})$ to this instance of $\mathcal{F}$.
- Upon receiving $(\text{CLOCK-FORWARD}, \text{sid}_C, \mathsf{P})$ from $\mathcal{A}$ where $\mathsf{P} \in \mathcal{P}$, if $\texttt{drift}_{\mathsf{P}} - \min\{\texttt{drift}_{\mathsf{P}}\} \geq \varPhi_{\text{clock}}$ or $\texttt{drift}_{\mathsf{P}} \geq \varPhi_{\text{clock}}$ or $d_{\mathsf{P}} = \mathsf{ongoing}$, ignore the message. Otherwise, update $\texttt{drift}_{\mathsf{P}} := \texttt{drift}_{\mathsf{P}} + 1$, $d_{\mathsf{P}} = \mathsf{ongoing}$ and $b_{\mathsf{P}} := b_{\mathsf{P}} + 1$; return $(\text{CLOCK-FORWARD-OK}, \text{sid}_C, \mathsf{P})$ to $\mathcal{A}$.
- Upon receiving $(\text{CLOCK-BACKWARD}, \text{sid}_C, \mathsf{P})$ from $\mathcal{A}$ where $\mathsf{P} \in \mathcal{P}$, if $\max\{\texttt{drift}_{\mathsf{P}}\} - \texttt{drift}_{\mathsf{P}} \geq \varPhi_{\text{clock}}$ or $\texttt{drift}_{\mathsf{P}} \leq -\varPhi_{\text{clock}}$ or $d_{\mathsf{P}} = \mathsf{ongoing}$, ignore the message. Otherwise, update $\texttt{drift}_{\mathsf{P}} := \texttt{drift}_{\mathsf{P}} - 1$ and $b_{\mathsf{P}} := b_{\mathsf{P}} - 1$; return $(\text{CLOCK-BACKWARD-OK}, \text{sid}_C, \mathsf{P})$ to $\mathcal{A}$.
- Upon receiving $(\text{CLOCK-TICK}, \text{sid}_C)$ from any participant P—including the environment on behalf of a party—or the adversary on behalf of a corrupted party P (resp. from any ideal—shared or local—functionality $\mathcal{F}$), execute procedure *Round-Update*, return $(\text{CLOCK-TICK}, \text{sid}_C, d_{\mathsf{P}})$ (resp. $(\text{CLOCK-TICK}, \text{sid}_C, d_{(\mathcal{F},\text{sid})})$) to the requestor (where sid is the sid of the calling instance).
- Upon receiving $(\text{CLOCK-READ}, \text{sid}_C)$ from the environment, the adversary or the wapper functionalities of random oracle $(\mathcal{W}(\mathcal{F}_{\text{RO}}))$, return $(\text{CLOCK-READ}, \text{sid}_C, \texttt{time})$ to the requestor.

Procedure Round-Update: For each session sid do: If $d_{(\mathcal{F},\text{sid})} = \mathsf{done}$ for all $\mathcal{F} \in F$ and $d_{\mathsf{P}} = \mathsf{done} \wedge b_{\mathsf{P}} \leq 0$ for all honest parties $\mathsf{P} = (\cdot, \text{sid}) \in \mathcal{P}$, then update $\texttt{time} := \texttt{time} + 1$, $d_{(\mathcal{F},\text{sid})} := \mathsf{ongoing}$ and $b_{\mathsf{P}} := b_{\mathsf{P}} + 1$ for all parties $\mathsf{P} = (\cdot, \text{sid}) \in \mathcal{P}$. Afterwards, for all parties $\mathsf{P} = (\cdot, \text{sid}) \in \mathcal{P}$ with $b_{\mathsf{P}} > 0$, update $d_{\mathsf{P}} := \mathsf{ongoing}$.

Functionality $\mathcal{F}_{\text{ILClock}}$ maintains an internal variable $\texttt{time}$ (the nominal time) to record how many times the sub-procedure *Round-Update* resets the round status for suitable honest parties and other functionalities. This variable will never be revealed to neither honest parties nor registered functionalities, which captures the fact that clocks are local, and it is only accessible by $\mathcal{F}_{\text{ILClock}}$, the environment $\mathcal{Z}$, the adversary $\mathcal{A}$ and the (wrapped) random oracle functionality$\mathcal{W}(\mathcal{F}_{\text{RO}})$, after they send a CLOCK-READ command. Notably, $\mathcal{W}(\mathcal{F}_{\text{RO}})$ has to know the exact nominal time in order to coordinate with the environment so as to apply restrictions on adversarial queries and model dynamic participation. (More details on the wrapped RO functionality and the notion of a respecting environment below.)

$\mathcal{F}_{\text{ILClock}}$ also keeps track of the set of honest parties $\mathcal{P}$ and the registered functionalities $\mathcal{F}$. For all honest parties and functionalities, $\mathcal{F}_{\text{ILClock}}$ associates each of them with a bit variable $d \in \{\mathsf{ongoing}, \mathsf{done}\}$ which tracks if the corresponding party/functionality has claimed finishing its local round, and returns a response to command CLOCK-TICK to indicate whether the requester should update its local clock. Importantly, every honest party P is associated with a (possibly negative) integer b_{P} which indicates the "tick budget" allocated for P during the current nominal round. If no adversarial clock drift is set, at the end of a nominal round b_{P} is reset to 1 by *Round-Update* and consumed by the party when P sends CLOCK-UPDATE in the next nominal round. This captures the fact that, without adversarial manipulation, local clocks proceed with exactly the same speed. Additionally, a (possibly negative) integer $\texttt{drift}$ is associated with P to record the total drift that is applied to its local clock.

An honest party P sends the CLOCK-UPDATE command to $\mathcal{F}_{\text{ILClock}}$ to indicate conclusion of its current local round. The main purpose of this command is to set the variable d associated with P to done and consume one tick from its budget (i.e., $b_{\mathsf{P}} \leftarrow b_{\mathsf{P}} - 1$). When party P is activated, ti sends $(\text{CLOCK-TICK}, \text{sid})$ to $\mathcal{F}_{\text{ILClock}}$ to check if its corresponding indicator d has been reset to $\mathsf{ongoing}$. If this happens, P updates its local clock and enters the next local round; if not, this implies that P need to wait for other participants.

Regarding the *Round-Update* procedure, $\mathcal{F}_{\text{ILClock}}$ will update the nominal time only when (1) all honest parties and registered functionalities have claimed finishing the current round (i.e., the corresponding indicator $d = \mathsf{done}$), and (2) all the tick budgets allocated to honest parties have been consumed (i.e., $b_{\mathsf{P}} \leq 0$ for every party). When the party/functionality statuses pass the above checks, *Round-Update* moves the nominal time forward ($\texttt{time} \leftarrow \texttt{time} + 1$). Then, *Round-Update* issues a new tick budget for all honest parties (i.e., all budget variables b are increased by 1); afterwards, for all registered functionalities and honest parties whose b_{P} is positive, their d indicators are reset to $\mathsf{ongoing}$.

Next, we elaborate on the "imperfect" aspect of the clock and on the adversarial manipulation of clock drifts. Specifically, in addition to *Round-Update*, we also allow the adversary to set some drifts to parties' local clocks, which will accelerate or stall their local clock; such values are globally bounded by Φ_{clock}. This assumption allows local clocks to proceed at "roughly" the same speed.

Further, the adversary $\mathcal{A}$ can adaptively manipulate the drift of honest parties' clocks by sending CLOCK-FORWARD and CLOCK-BACKWARD[12] after they conclude the current round. If $\mathcal{A}$ issues CLOCK-FORWARD for party P, it will enter a new local round before $\mathcal{F}_{\text{ILClock}}$ updates the nominal time, and this can be repeated as long as P's drift is not Φ_{clock} rounds larger than other honest

[12]As such, our clock functionality is a more natural model of the real world compared to [BGK+21]'s, as it allows $\mathcal{A}$ to manipulate the clock in both directions, backward, and forward; in [BGK+21], only forward manipulation is allowed. Nonetheless, this does not result in a more powerful adversary.

parties. On the other hand, if $\mathcal{A}$ issues CLOCK-BACKWARD, it will set P's budget to a negative value, thus preventing $\mathcal{F}_{\text{ILClock}}$ from updating d_{P} at the end of the nominal round. I.e., P will still be in the same logical round during these two nominal rounds. Again, this process can be repeated by $\mathcal{A}$ as long as the drift on P is not Φ_{clock} rounds smaller than others. As a consequence, the targeted party's local clock may remain static for several nominal rounds.

B.2 Global Random Oracle Functionality and its Wrapper

By convention, we model parties' calls to the hash function used to generated proofs of work as assuming access to a random oracle; this is captured by the functionality $\mathcal{F}_{\text{RO}}$. $\mathcal{F}_{\text{RO}}$ internally maintains an updatable table H; it is parameterized with security parameter κ as H's output length. Upon receiving a query $(\text{EVAL}, \text{sid}, x)$, if $H(x) = \perp$ (i.e., no pair of the form $(x, \cdot)$ is in H), a value y is chosen uniformly at random from $\{0,1\}^\kappa$ and returned to the party ($\mathcal{F}_{\text{RO}}$ also updates $H(x) = y$). If $H(x) \neq \perp$ (i.e., x has been queried before), the corresponding y is returned.

Functionality $\mathcal{F}_{\text{RO}}$

The functionality is parametrized by the security parameter κ. It maintains a dynamically updatable function table H where $H[x] = \perp$ denotes the fact that no pair of the form $(x, \cdot)$ is in H. Initially, $H = \emptyset$.

- Upon receiving $(\text{EVAL}, \text{sid}, x)$ from some party $\mathsf{P} \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of a corrupted P), do the following:
  1. If $H[x] = \perp$ sample a value y uniformly at random from $\{0,1\}^\kappa$ and set $H[x] \leftarrow y$.
  2. Return $(\text{EVAL}, \text{sid}, x, H[x])$ to the requestor.

Notice that with regards to bounding access to real-world resources, functionality $\mathcal{F}_{\text{RO}}$ as defined fails to limit the adversary on making a certain number of queries per round. Hence, we adopt a functionality wrapper [BMTZ17, GKO$^+$20] $\mathcal{W}(\mathcal{F}_{\text{RO}})$ that wraps the corresponding resource to capture such restrictions. We highlight that our wrapper $\mathcal{W}(\mathcal{F}_{\text{RO}})$ improves on previous wrappers in two aspects, in order to provide a more natural model of the real world:

1. We capture the pre-mining stage by letting the adversary query the RO with no restrictions before the CRS is released. More specifically, $\mathcal{W}(\mathcal{F}_{\text{RO}})$ is initialized with internal time counter $\texttt{time} = \perp$, and the adversary can make as many queries as he wants (albeit polynomially bounded) and the wrapper simply forwards all these queries to $\mathcal{F}_{\text{RO}}$ as long as $\texttt{time} = \perp$. This pre-mining stage ends when the time counter $\texttt{time}$ is set to 1, which happens immediately after $\mathcal{W}(\mathcal{F}_{\text{RO}})$ receives $(\text{RETRIEVED}, \text{sid})$ from the CRS functionality $\mathcal{F}_{\text{CRS}}^{\mathcal{D}}$.

   This pre-mining stage captures the fact that a hash function is usually available before the protocol execution begins. Nonetheless, thanks to the CRS with sufficiently high entropy, all adversarial queries made during the pre-mining stage can benefit the adversary in the later execution only with negligible probability.

2. The wrapper limits adversarial access per nominal round by bounding the total number of queries that $\mathcal{A}$ can make. After the CRS is released, $\mathcal{W}(\mathcal{F}_{\text{RO}})$ keeps track of the corrupted party set. Upon receiving queries from corrupted parties, $\mathcal{W}(\mathcal{F}_{\text{RO}})$ first checks if $\mathcal{F}_{\text{ILClock}}$ advances the nominal time ($\mathcal{W}(\mathcal{F}_{\text{RO}})$ can directly access $\mathcal{F}_{\text{ILClock}}$'s internal time counter). If $\mathcal{F}_{\text{ILClock}}$ enters the next round, $\mathcal{W}(\mathcal{F}_{\text{RO}})$ updates its internal clock counter as well and resets the adversarial RO counter $t_{\mathcal{A}}$ to 0. As long as $t_{\mathcal{A}} < t_{\texttt{time}}$, where $t_{\texttt{time}}$ is a predetermined value provided by the environment $\mathcal{Z}$ for nominal round $\texttt{time}$, $\mathcal{W}(\mathcal{F}_{\text{RO}})$ forwards the queries to $\mathcal{F}_{\text{RO}}$. When $t_{\mathcal{A}} \geq t_{\texttt{time}}$, which implies that the adversary has consumed all its allowed

budget, the wrapper stops interaction with the adversary.

This "migration" from the q-bounded adversarial model ([GKL15] and follow-ups) to bounding the total number of queries that $\mathcal{A}$ can make allows us to dispose of the "flat" computational model considered in previous works. We now define the computational power in terms of the number of RO queries per round, which makes it possible to further refine the notion of a "respecting environment" that is suited for imperfect local clocks.

Wrapper Functionality $\mathcal{W}(\mathcal{F}_{\text{RO}})$

The wrapper functionality is parameterized by a set of parties $\mathcal{P}$, and an upper bound t which restricts the $\mathcal{F}$-evaluations of all corrupted party per round. (To keep track of rounds the functionality registers with the global clock $\mathcal{F}_{\text{ILClock}}$.) The functionality manages the variable $\texttt{time}$ (positive integer or $\perp$) and the current set of corrupted miners $\mathcal{P}$. It also manages variable $t_{\mathcal{A}}$. Initially, $\texttt{time} = \perp$.

General:

- The wrapper stops the interaction with the adversary as soon as the adversary tries to exceed its budget of $t_{\texttt{time}}$ queries per nominal round.

Relaying inputs to the random oracle:

- Upon receiving $(\text{EVAL}, \text{sid}, x)$ from $\mathcal{A}$ on behalf of a corrupted party $P \in \mathcal{P}$, if $\texttt{time} = \perp$, forward the request to $\mathcal{F}_{\text{RO}}$ and return to $\mathcal{A}$ whatever $\mathcal{F}_{\text{RO}}$ returns. Otherwise, first execute *Round Reset*. Then, set $t_{\mathcal{A}} := t_{\mathcal{A}} + 1$ and only if $t_{\mathcal{A}} \le t_{\texttt{time}}$ forward the request to $\mathcal{F}_{\text{RO}}$ and return to $\mathcal{A}$ whatever $\mathcal{F}_{\text{RO}}$ returns.
- Upon receiving (RETRIEVED) from $\mathcal{F}^{\mathcal{D}}_{\text{CRS}}$, set $\texttt{time} = 1$.
- Any other request from any participant or the adversary is simply relayed to the underlying functionality without any further action and the output is given to the destination specified by the hybrid functionality.

Corruption Handling:

- Upon receiving $(\text{CORRUPT}, \text{sid}, \mathsf{P})$ from the adversary, set $\mathcal{P} := \mathcal{P} \cup \mathsf{P}$.

Procedure Round-Reset:

Send $(\text{CLOCK-READ}, \text{sid}_C)$ to $\mathcal{F}_{\text{ILClock}}$ and receive $(\text{CLOCK-READ}, \text{sid}_C, \texttt{time}')$ from $\mathcal{F}_{\text{ILClock}}$. If $|\texttt{time} - \texttt{time}'| > 0$ (i.e., a new round started), then set $t_{\mathcal{A}} := 0$ for the adversary and set $\texttt{time} := \texttt{time}'$.

B.3 Common Reference String

Functionality $\mathcal{F}^{\mathcal{D}}_{\text{CRS}}$

When activated for the first time on input $(\text{RETRIEVE}, \text{sid})$, choose a value $d \leftarrow \mathcal{D}$, and send $(\text{RETRIEVE}, d)$ back to the activated party; also send $(\text{RETRIEVED}, \text{sid})$ to $\mathcal{W}(\mathcal{F}_{\text{RO}})$.
In each other activation return the value d to the activated party.

B.4 Diffusion Functionality

There are three types of messages that are exchanged in the network: blockchain messages (e.g., when a party mines a new block); regular messages (usually transactions), which are diffused to the network when received by the environment; and synchronization beacons—a special message that parties will use to perform the local time adjustment. For simplicity, we follow the convention that each type of messages is diffused by its own network. We denote the network functionality disseminating blockchain information by $\mathcal{F}^{\mathsf{bc}}_{\text{Diffuse}}$; the one diffusing transactions by $\mathcal{F}^{\mathsf{tx}}_{\text{Diffuse}}$; and the one circulating synchronization beacons by $\mathcal{F}^{\mathsf{sync}}_{\text{Diffuse}}$, respectively. Parties will communicate with $\mathcal{F}^{\mathsf{bc}}_{\text{Diffuse}}$ and $\mathcal{F}^{\mathsf{tx}}_{\text{Diffuse}}$ to receive new chains and transactions in FetchInformation, and they will receive synchronization beacons by sending requests to $\mathcal{F}^{\mathsf{sync}}_{\text{Diffuse}}$ in ProcessBeacons. Refer to Appendices C.5 and C.7 for further details.

Functionality $\mathcal{F}^{\Delta}_{\text{Diffuse}}$

The functionality is parameterized with a set possible senders and receivers $\mathcal{P}$. Any newly registered (resp. deregistered) party is added to (resp. deleted from) $\mathcal{P}$.

- **Honest sender diffusion**. Upon receiving $(\text{DIFFUSE}, \text{sid}, m)$ from some $\mathsf{P} \in \mathcal{P}$, where $\mathcal{P} = \{U_1, \ldots, U_n\}$ denotes the current party set, choose n new unique message-IDs $\text{mid}_1, \ldots, \text{mid}_n$ of the form $\text{mid}_i = (\text{mid}_n, i)$, initialize $2n$ new variables $D_{\text{mid}_1} := D^{MAX}_{\text{mid}_1} \ldots := D_{\text{mid}_n} := D^{MAX}_{\text{mid}_n} := 1$, a per message delay $\Delta_{\text{mid}_i} = \Delta$ for $i = 1, \ldots, n$ and set $\mathbf{M} := \mathbf{M} \parallel (m, \text{mid}_1, D_{\text{mid}_1}, U_1) \parallel \ldots \parallel (m, \text{mid}_n, D_{\text{mid}_n}, U_n)$, and send $(\text{DIFFUSE}, \text{sid}, m, \mathsf{P}, (U_1, \text{mid}_1), \ldots, (U_n, \text{mid}_n))$ to the adversary.
- **Adversarial sender diffusion**. Upon receiving $(\text{DIFFUSE}, \text{sid}, m)$ from some $\mathsf{P} \in \mathcal{P}$ (where $= \{U_1, \ldots, U_n\}$ denotes the current party set), do execute it the same way as an honest-sender diffusion, with the only difference that $\Delta_{\text{mid}_i} = \infty$.
- **Honest party fetching.** Upon receiving $(\text{FETCH}, \text{sid})$ from $\mathsf{P} \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of P if P is corrupted):
  1. For all tuples $(m, \text{mid}, D_{\text{mid}}, \mathsf{P}) \in \mathbf{M}$, set $D_{\text{mid}} := D_{\text{mid}} - 1$.
  2. Let $\mathbf{M}^{\mathsf{P}}_0$ denote the subvector $\mathbf{M}$ including all tuples of the form $(m, \text{mid}, D_{\text{mid}}, \mathsf{P})$ with $D_{\text{mid}} = 0$ (in the same order as they appear in $\mathbf{M}$). Then, delete all entries in $\mathbf{M}^{\mathsf{P}}_0$ from $\mathbf{M}$ and in case some $(m, \text{mid}, D_{\text{mid}}, \mathsf{P})$ is in $\mathbf{M}^{\mathsf{P}}_0$, where P is honest, set $\Delta_{\text{mid}'} = \Delta$ for any $(m, \text{mid}', D_{\text{mid}'}, \mathsf{P}')$ in $\mathbf{M}$ and replace this record by $(m, \text{mid}', \min\{D_{\text{mid}'}, \Delta\}, \mathsf{P}')$. Finally, send $\mathbf{M}^{\mathsf{P}}_0$ to P.
- **Adding adversarial delays**. Upon receiving $(\text{DELAYS}, \text{sid}, (T_{\text{mid}_{i_1}}, \text{mid}_{i_1}), \ldots, (T_{\text{mid}_{i_\ell}}, \text{mid}_{i_\ell}))$ from the adversary do the following for each pair $(T_{\text{mid}_{i_j}}, \text{mid}_{i_j})$: if $D^{MAX}_{\text{mid}_{i_j}} + T_{\text{mid}_{i_j}} \leq \delta_{\text{mid}_{i_j}}$ and mid_{i_j} is a message-ID registered in the current $\mathbf{M}$, set $D_{\text{mid}_{i_j}} := D_{\text{mid}_{i_j}} + T_{\text{mid}_{i_j}}$ and set $D^{MAX}_{\text{mid}_{i_j}} := D^{MAX}_{\text{mid}_{i_j}} + T_{\text{mid}_{i_j}}$; otherwise, ignore this pair.
- **Adversarially reordering messages**. Upon receiving $(\mathsf{swap}, \text{sid}, \text{mid}, \text{mid}')$ from the adversary, if mid and mid' are message-IDs registered in the current $\mathbf{M}$, then swap the triples $(m, \text{mid}, D_{\text{mid}}, \cdot)$ and $(m, \text{mid}', D_{\text{mid}'}, \cdot)$ in $\mathbf{M}$. Return $(\text{SWAP}, \text{sid})$ to the adversary.

C The Timekeeper Protocol

In this section we give the full specification of the Timekeeper protocol. The description is presented in UC-like notation.

C.1 The Main Protocol Instance

We first introduce the main Timekeeper protocol instance that dispatches to the relevant subprocesses.

Protocol $\mathsf{Timekeeper}_k(\mathsf{P}, \mathrm{sid}; \mathcal{G}_{\textsc{Ledger}}, \mathcal{F}_{\textsc{ILClock}}, \mathcal{F}_{\mathrm{RO}}, \mathcal{F}^{\Delta}_{\mathrm{Diffuse}})$

Global Variables:

- Read-only: R, M, t_{off}, t_{gather}, t_{pre}
- Read-write: `localTime`, `ep`, `r`, $\mathcal{C}_{\mathsf{loc}}$, $T^{\mathsf{ep}}_{\mathsf{P}}$, `isInit`, t_{work}, `buffer`, `futureChains`, `isSync`, `fetchCompleted`, `lastTimeAlert`, $\mathsf{arrivalTime}_{SB}(\cdot)$.

Registration / Deregistration:

- Upon receiving input $(\textsc{register}, \mathcal{R})$, where $\mathcal{R} \in \{\mathcal{G}_{\textsc{Ledger}}, \mathcal{F}_{\textsc{ILClock}}, \mathcal{F}_{\mathrm{RO}}\}$ execute protocol $\mathsf{Registration\text{-}Timekeeper}(\mathrm{P}, \mathrm{sid}, \mathtt{Reg}, \mathcal{R})$.
- Upon receiving input $(\textsc{de-register}, \mathcal{R})$, where $\mathcal{R} \in \{\mathcal{G}_{\textsc{Ledger}}, \mathcal{F}_{\textsc{ILClock}}, \mathcal{F}_{\mathrm{RO}}\}$ execute protocol $\mathsf{Deregistration\text{-}Timekeeper}(\mathrm{P}, \mathrm{sid}, \mathtt{Reg}, \mathcal{R})$.
- Upon receiving input $(\textsc{is-registered}, \mathrm{sid})$ return $(\textsc{register}, \mathrm{sid}, 1)$ if the local registry `Reg` indicates that this party has successfully completed a registration with $\mathcal{R} = \mathcal{G}_{\textsc{Ledger}}$ (and did not de-register since then). Otherwise, return $(\textsc{register}, \mathrm{sid}, 0)$.

Interacting with the Ledger:

Upon receiving a ledger-specific input $I \in \{(\textsc{submit}, \ldots), (\textsc{read}, \ldots), (\textsc{maintain-ledger}, \ldots)\}$ verify first that all resources are available. **If** not all resources are available, **then** ignore the input; **else** (i.e., the party is operational and time-aware) execute one of the following steps depending on the input I:

- **If** $I = (\textsc{submit}, \mathrm{sid}, \mathtt{tx})$ **then** set $\mathtt{buffer} \leftarrow \mathtt{buffer} \,\|\, \mathtt{tx}$, and send $(\textsc{diffuse}, \mathrm{sid}, \mathtt{tx})$ to $\mathcal{F}^{\Delta}_{\mathrm{Diffuse}}$.
- **If** $I = (\textsc{maintain-ledger}, \mathrm{sid}, \mathrm{minerID})$ **then** invoke protocol $\mathsf{LedgerMaintenance}(\mathcal{C}_{\mathsf{loc}}, \mathsf{P}, \mathrm{sid}, \mathrm{R})$; **if** $\mathsf{LedgerMaintenance}$ halts **then** halt the protocol execution (all future input is ignored).
- **If** $I = (\textsc{read}, \mathrm{sid})$ **then** invoke protocol $\mathsf{ReadState}(\mathcal{C}_{\mathsf{loc}}, \mathsf{P}, \mathrm{sid}, \mathrm{R})$.
- **If** $I = (\textsc{export-time}, \mathrm{sid})$ **then** do the following: if `isSync` or `isInit` is false, then return $(\textsc{export-time}, \mathrm{sid}, \bot)$ to the caller. Otherwise, call $\mathsf{UpdateTime}(\mathsf{P}, R)$ and return $(\textsc{export-time}, \mathrm{sid}, \mathtt{localTime})$ to the caller.

Handling calls to the shared setup:

- Upon receiving $(\textsc{clock-tick}, \mathrm{sid}_C)$, forward it to $\mathcal{F}_{\textsc{ILClock}}$ and output $\mathcal{F}_{\textsc{ILClock}}$'s response.
- Upon receiving $(\textsc{clock-update}, \mathrm{sid}_C)$, record that a clock-update was received in the current round. If the party is registered to all its setups, then do nothing further. Otherwise, do the following operations *before concluding this round*:
  1. If this instance is currently time-aware but otherwise stalled or offline, then call $\mathsf{UpdateTime}(\mathsf{P}, R)$ to update `localTime`. If the party has passed a synchronization interval, then set $\mathtt{isSync} \leftarrow \mathsf{false}$.
  2. If this instance is only stalled but $\mathtt{isSync} = \mathsf{true}$, then additionally execute $\mathsf{FetchInformation}(\mathsf{P}, \mathrm{sid})$, extract all new synchronization beacons from the fetched chains and record their arrival times and set $\mathtt{fetchCompleted} \leftarrow \mathsf{true}$. Also, any unfinished interruptible execution of this round is marked as completed.
  3. Forward $(\textsc{clock-update}, \mathrm{sid}_C)$ to $\mathcal{F}_{\textsc{ILClock}}$ to finally conclude the round.
- Upon receiving $(\textsc{eval}, \mathrm{sid}_{RO}, x)$ forward the query to $\mathcal{F}_{\mathrm{RO}}$ and output $\mathcal{F}_{\mathrm{RO}}$'s response.

C.2 Registration and De-registration

In order to perform basic operations, a party P needs to register to all resources. Note that P is aware whether he is not synchronized not and will set the bit variable `isSync` correspondingly.

```
Protocol Registration − Timekeeper($\mathsf{P}$, sid, reg, $\mathcal{G}$)

1: if $\mathcal{G} \in \{\mathcal{F}_{\text{ILClock}}, \mathcal{F}_{\text{RO}}\}$ then
2:     send (REGISTER, sid) to $\mathcal{G}$, set registration status to registered with $\mathcal{G}$, and output
   the value received by $\mathcal{G}$.
3: end if
4: if $\mathcal{G} = \mathcal{G}_{\text{LEDGER}}$ then
5:     if the party is not registered with $\mathcal{F}_{\text{ILClock}}$ or $\mathcal{F}_{\text{RO}}$ or already registered with all
   setups then
6:         ignore this input.
7:     else
8:         Send (CLOCK-TICK, $\text{sid}_C$) to $\mathcal{F}_{\text{ILClock}}$ and receive (CLOCK-TICK, $\text{sid}_C$, tick).
9:         Send (REGISTER, sid) to $\mathcal{F}^{\Delta}_{\text{Diffuse}}$.
10:        Set localTime := $\langle 1, 1\rangle$ and isSync ← false.
11:        If this is the first registration invocation for this ITI, then set isInit ← false.
12:        Output (REGISTER, sid, $\mathsf{P}$) once completing the registration with all the above
   resources $\mathcal{F}$.
13:    end if
14: end if
```

The deregistration process is an analogous aciton. Note that parties will record the last alert time which might be used to synchronize if it is only stalled.

```
Protocol Deregistration − Timekeeper($\mathsf{P}$, sid, reg, $\mathcal{G}$)

1: If the party is alert, set lastTimeAlert ← localTime.
2: if $\mathcal{G} \in \{\mathcal{F}_{\text{ILClock}}, \mathcal{F}_{\text{RO}}\}$ then
3:     if $\mathcal{G} = \mathcal{F}_{\text{ILClock}}$ then
4:         Set isSync ← false
5:     end if
6:     Send (DE-REGISTER, sid) to $\mathcal{G}$ and set registration status as de-registered with $\mathcal{G}$.
7:     Output the value received by $\mathcal{G}$.
8: end if
9: if $\mathcal{G} = \mathcal{G}_{\text{LEDGER}}$ then
10:    Set isSync ← false
11:    Send (DE-REGISTER, sid) to $\mathcal{F}^{\Delta}_{\text{Diffuse}}$, set its registration status as de-registered with
   $\mathcal{F}^{\Delta}_{\text{Diffuse}}$ and output (DE-REGISTER, sid, $\mathsf{P}$).
12: end if
```

C.3 Ledger Maintenance

We group all the steps in the main ledger operation in LedgerMaintenance. Note that what a party might execute depends on its status.

Protocol $\mathsf{LedgerMaintenance}(\mathcal{C}_{\mathsf{loc}}, \mathsf{P}, \mathrm{sid}, R)$

The following steps are executed in an $(\textsc{maintain-ledger}, \mathrm{sid}, \mathrm{minerID})$-interruptible manner:

```
1:  if $\texttt{isInit} = \mathsf{false}$ then
2:      Send $(\textsc{Retrieve}, \mathrm{sid})$ to $\mathcal{F}^{\mathcal{D}}_{\mathrm{CRS}}$ and receive $(\textsc{Retrieved}, d)$.
3:  end if
4:  // From here the variables $\texttt{localTime}$, $\mathcal{C}_{\mathsf{loc}}$, $\texttt{isSync}$, can be used to read from as they
    are guaranteed to be initialized.
5:  if $\texttt{isSync}$ and stalled before (and now up and running) then
6:      Call $\mathsf{SimulateClockAdjustments}(\mathsf{P}, \mathrm{sid}, \mathrm{R})$
7:  end if
8:  if not $\texttt{isSync}$ then
9:      Call $\mathsf{JoinProc}(\mathsf{P}, \mathrm{sid}, \mathrm{R}, t_{\mathrm{off}}, t_{\mathrm{gather}})$
10: end if
11: // normal operation when alert
12: Call $\mathsf{FetchInformation}(\mathsf{P}, \mathrm{sid})$ and denote the output by $(\mathcal{C}_1, \ldots, \mathcal{C}_N), (\texttt{tx}_1, \ldots, \texttt{tx}_k)$
13: Set $\texttt{buffer} \leftarrow \texttt{buffer} \,\|\, (\texttt{tx}_1, \ldots, \texttt{tx}_k)$ and define $\texttt{futureChains} \leftarrow \texttt{futureChains} \,\|$
    $(\mathcal{C}_1, \ldots, \mathcal{C}_N)$
14: Call $\mathsf{UpdateTime}(\mathsf{P}, \mathrm{R})$
15: // Ensures the processing of new beacons arrived in chains only.
16: Extract beacons $B \leftarrow \{\mathsf{SB}_1, \ldots, \mathsf{SB}_n\}$ contained in $\mathcal{C}_1, \ldots, \mathcal{C}_N$ and not yet contained in
    $\texttt{syncBuffer}$.
17: Call $\mathsf{ProcessBeacons}(\mathsf{P}, \mathrm{sid}, B)$
18: Let $\mathcal{N}_0$ be the subsequence of $\texttt{futureChains}$ s.t. $\mathcal{C} \in \mathcal{N}_0 :\Leftrightarrow \forall \mathcal{B} \in \mathcal{C} : \mathsf{Timestamp}(\mathcal{B}) \leq$
    $\texttt{localTime}$
19: Remove each $\mathcal{C} \in \mathcal{N}_0$ from $\texttt{futureChains}$.
20: $\texttt{fetchCompleted} \leftarrow \mathsf{true}$
21: Call $\mathsf{SelectChain}(\mathsf{P}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}}, \mathcal{N}_0, \mathrm{R})$ to update $\mathcal{C}_{\mathsf{loc}}$
22: if $t_{\mathrm{work}} < \texttt{localTime}$ then
23:     Call $\mathsf{UpdateMiningTarget}(\mathsf{P}, \mathrm{M})$ to udpate $T^{\mathcal{C}}_{\mathsf{ep}}$
24:     Call $\mathsf{UpdateFreshRandomness}(\mathsf{P}, \mathrm{R})$ to update $\eta_{\texttt{itvl}}$
25:     Call $\mathsf{MiningProcedure}(\mathsf{P}, \mathsf{ep}, \texttt{r}, \texttt{buffer}, \mathcal{C}_{\mathsf{loc}})$
26:     Set $t_{\mathrm{work}} \leftarrow \texttt{localTime}$
27:     if $\texttt{r} = \texttt{itvl} \cdot \mathrm{R}$ then
28:         Call $\mathsf{SyncProc}(\mathsf{P}, \mathrm{sid}, R)$
29:     end if
30: end if
31: Call $\mathsf{FinishRound}(\mathsf{P})$ // Mark normal round actions as finished.
```

C.4 Validity Checks of Chains and Beacons

Chain verification. A core procedure is to distinguish valid from invalid chains. The procedure is depicted below. It adopts the verification rule of the main blockchain structure similar to that in Bitcoin (cf. [GKL17, GKL20]), and extends it with beacon verifications.

Recall that $I_{\mathrm{sync}}(\texttt{itvl})$ defined in Equation (2) is a useful function which extracts the valid timestamp set in beacon mining and inclusion phase w.r.t. interval $\texttt{itvl}$. In addition, $\mathsf{isvalidate}$

is a prediate that checks whether transactions in the blockchain achieves a valid ledger state (for details, see [BMTZ17]).

Protocol $\mathsf{IsValidChain}(\mathsf{P}, \mathrm{sid}, \mathcal{C}, \mathrm{R})$

```
1:  if $\mathcal{C}$ contains empty intervals or starts with a block with hash reference other than CRS,
    or $\mathsf{isvalidstate}(\overrightarrow{\mathtt{st}}) = 0$ then
2:      return false
3:  end if
4:  if $\mathtt{isSync}$ and $(\exists \mathcal{B} \in \mathcal{C} : \mathsf{Timestamp}(\mathcal{B}) > \mathtt{localTime})$ then
5:      return false
6:  end if
7:  for each interval $\mathtt{itvl}'$ do
8:      // Derive target and randomness for interval $\mathtt{itvl}'$ as indicated by $\mathcal{C}$
9:      $\mathsf{ep}' \leftarrow \mathsf{TargetRecalcEpoch}(\langle \mathtt{itvl}', \cdot \rangle)$
10:     Set $T^{\mathcal{C}}_{\mathsf{ep}'}$ to be the target for epoch $\mathtt{ep}'$ in $\mathcal{C}$.
11:     Set $\eta_{\mathtt{itvl}'} \leftarrow G(\eta_{\mathtt{itvl}'-1} \,\|\, \mathtt{itvl}' \,\|\, v)$ where $v$ is the concatenation of all block hash in
    interval $\mathtt{itvl}' - 1$, and $\eta_1 \triangleq$ CRS.
12:     for each block $\mathcal{B}$ in $\mathcal{C}$ from interval $\mathtt{itvl}'$ do // check 2-for-1 pow.
13:         Parse $\mathcal{B}$ as $\langle ctr, h, \mathtt{st}, \langle \mathtt{itvl}', \mathtt{r}' \rangle, txLabel \rangle$.
14:         // Check hash
15:         Set $\mathsf{badhash} \leftarrow (h \neq H(\mathcal{B}^{-1}))$, where $\mathcal{B}^{-1}$ is the last block in $\mathcal{C}$ before $\mathcal{B}$.
16:         // Check nonce
17:         Set $\mathsf{badnonce} \leftarrow (H(ctr, h, \mathtt{st}, \langle \mathtt{itvl}', \mathtt{r}' \rangle, txLabel) < T^{\mathcal{C}}_{\mathsf{ep}'}) \wedge (ctr < 2^{32})$
18:         // Check beacons
19:         if $\exists \mathtt{SB} \in \mathcal{B}$ and $\mathtt{r}' \notin I_{\text{sync}}(\mathtt{itvl}')$ then
20:             Set $\mathsf{badBeacon} \leftarrow \mathsf{true}$
21:         else if $\exists \mathtt{SB} \in \mathcal{B} : \mathsf{Timestamp}(\mathtt{SB}) > \mathsf{Timestamp}(\mathcal{B})$ then
22:             Set $\mathsf{badBeacon} \leftarrow \mathsf{true}$
23:         else
24:             Parse $\mathtt{SB}$ as $\langle \langle \mathtt{itvl}', \mathtt{r}' \rangle, \mathsf{P}', \eta_{\mathtt{SB}}, ctr, blockLabel \rangle$
25:             if $\mathcal{C}$ contains more than one beacon with $(\mathtt{r}', \mathsf{P}', \cdot)$ or $\eta_{\mathtt{SB}} \neq \eta_{\mathtt{itvl}'}$ or $\mathtt{r}' \notin$
    $I_{\text{sync}}(\mathtt{itvl}')$ then
26:                 Set $\mathsf{badBeacon} \leftarrow \mathsf{true}$
27:             end if
28:             $u \leftarrow H(ctr, \mathtt{r}', blockLabel, \langle \mathtt{itvl}', \mathtt{r}' \rangle, \mathsf{P}', \eta_{\mathtt{itvl}'})$
29:             Set $\mathsf{badBeacon} \leftarrow ([u]^{\mathsf{R}} < T^{\mathcal{C}}_{\mathsf{ep}'}) \wedge (ctr' < 2^{32})$
30:         end if
31:         if $(\mathsf{badhash} \vee \mathsf{badnonce} \vee \mathsf{badBeacon})$ then
32:             return false
33:         end if
34:     end for
35: end for
```

The beacon validity predicate. Beacons validity is related to chain validity as one need the corresponding target as well as fresh randomness to check beacon validity. The details are found below.

Protocol $\mathsf{ValidSB}(\mathsf{P}, \mathrm{sid}, \mathtt{SB}, \mathcal{C}, \mathrm{R})$

```
// Precondition: Chain $\mathcal{C}$ is valid. Returns true if the beacon is a valid beacon w.r.t. $\mathcal{C}$,
// undecided if no judgement is possible, and false if the beacon is invalid w.r.t. $\mathcal{C}$.
1: Parse $\mathtt{SB}$ as $\langle\langle \mathtt{itvl}', \mathtt{r}'\rangle, \mathsf{P}', \eta_{\mathtt{SB}}, ctr, blockLabel\rangle$
2: if $\mathcal{C}$ contains no block in interval $\mathtt{itvl}'$ then
3:     return undecided // no judgement possible for this beacon
4: end if
5: // Derive target and randomness for interval $\mathtt{itvl}'$ as indicated by $\mathcal{C}$
6: $\mathsf{ep}' \leftarrow \mathsf{TargetRecalcEpoch}(\langle \mathtt{itvl}', \mathtt{r}'\rangle)$
7: Set $T^{\mathcal{C}}_{\mathsf{ep}'}$ to be the target for epoch $\mathsf{ep}'$ in $\mathcal{C}$.
8: Set $\eta_{\mathtt{itvl}'} \leftarrow G(\eta_{\mathtt{itvl}'-1} \,\|\, \mathtt{itvl}' \,\|\, v)$ where $v$ is the concatenation of all block hash in
   interval $\mathtt{itvl}' - 1$, and $\eta_1 \triangleq \mathrm{CRS}$.
9: // Check nonce value and freshness
10: $u \leftarrow H(ctr, \mathtt{r}', blockLabel, \langle \mathtt{itvl}', \mathtt{r}'\rangle, \mathsf{P}', \eta_{\mathtt{itvl}'})$
11: if $([u]^{\mathsf{R}} < T^{\mathcal{C}}_{\mathsf{ep}}) \wedge (ctr' < 2^{32}) \wedge (\eta_{\mathtt{itvl}'} = \eta_{\mathtt{SB}})$ then
12:     return true
13: end if
14: return false
```

C.5 Fetch information

Parties fetch information from two diffusion network—$\mathcal{F}^{\mathsf{bc}}_{\mathrm{Diffuse}}$ and $\mathcal{F}^{\mathsf{tx}}_{\mathrm{Diffuse}}$—to learn new chains and transactions.

Protocol $\mathsf{FetchInformation}(\mathsf{P}, \mathrm{sid})$

```
1: if fetchCompleted then
2:     Set fetchCount $\leftarrow 0$
3: else
4:     Set fetchCount $\leftarrow 1$
5: end if
6: // Fetch on $\mathcal{F}^{\mathsf{bc}}_{\mathrm{Diffuse}}$
7: Send fetchCount fetch-queries $(\textsc{fetch}, \mathrm{sid})$ to $\mathcal{F}^{\mathsf{bc}}_{\mathrm{Diffuse}}$; denote the $i$-th response from
   $\mathcal{F}^{\mathsf{bc}}_{\mathrm{Diffuse}}$ by $(\textsc{fetch}, \mathrm{sid}, b_i)$.
8: Extract chains $\mathcal{C}_1, \ldots, \mathcal{C}_k$ from $b_1 \ldots b_{\mathsf{fetchCount}}$.
9: // Fetch on $\mathcal{F}^{\mathsf{tx}}_{\mathrm{Diffuse}}$
10: Send fetchCount fetch-queries $(\textsc{fetch}, \mathrm{sid})$ to $\mathcal{F}^{\mathsf{tx}}_{\mathrm{Diffuse}}$; denote the $i$-th response from
    $\mathcal{F}^{\mathsf{tx}}_{\mathrm{Diffuse}}$ by $(\textsc{fetch}, \mathrm{sid}, b_i)$.
11: Extract received transactions $\mathtt{tx}_1, \ldots, \mathtt{tx}_k$ from $b_1 \ldots b_{\mathsf{fetchCount}}$.
12: if not isSync or P is stalled then
13:     buffer $\leftarrow$ buffer $\|\, (\mathtt{tx}_1, \ldots, \mathtt{tx}_n)$
14:     futureChains $\leftarrow$ futureChains $\cup \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$
15: end if
OUTPUT: The protocol outputs $(\mathcal{C}_1, \ldots, \mathcal{C}_k)$ and $(\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$ to its caller (but not to $\mathcal{Z}$).
```

C.6 Update Mining Target, Fresh Randomness and Local Time

Following Equation (5) and the rule regarding dampening filter τ, the targets for each epoch ep are computed as following. Recall that $\mathsf{EpochBlockCount}$ defined in Equation (4) outputs the number of blocks in the corresponding epoch, and T_0 is a pre-determined target value (hardcoded in the protocol) that guarantees a "safe" start in our assumption.

Protocol $\mathsf{UpdateMiningTarget}(\mathsf{P}, \mathrm{M})$

```
1: $\mathsf{ep} \leftarrow \mathsf{TargetRecalcEpoch}(\mathtt{localTime})$
2: **if** $\mathsf{ep} = 1$ **then**
3:     $T_{\mathsf{ep}} \leftarrow T_0$
4: **else**
5:     $\Lambda = |\mathsf{EpochBlocks}(\mathcal{C}_{\mathsf{loc}}, \mathsf{ep} - 1)|$
6:     $\Lambda = \min\{\max\{\Lambda, \Lambda_{\mathsf{epoch}}/\tau\}, \Lambda_{\mathsf{epoch}} \cdot \tau\}$
7:     $T_{\mathsf{ep}} \leftarrow (\Lambda_{\mathsf{epoch}}/\Lambda) \cdot T_{\mathsf{ep}-\mathrm{M/R}}$
8: **end if**
OUTPUT: The protocol outputs $T_{\mathsf{ep}}$ to its caller (but not to $\mathcal{Z}$).
```

Parties will extract the fresh randomness in every synchronization interval from the block hash in previous epoch. Note that $G(\cdot)$ is another hash function different from $H(\cdot)$ therefore it does not interact with the random oracle $\mathcal{F}_{\mathrm{RO}}$ in the mining procedure.

Protocol $\mathsf{UpdateFreshRandomness}(\mathsf{P}, \mathrm{R})$

```
1: Set $\eta_{\mathtt{itvl}} \leftarrow G(\eta_{\mathtt{itvl}-1} \| \mathtt{itvl} \| v)$ where $v$ is the concatenation of all block hash in interval
   $\mathtt{itvl} - 1$, and $\eta_1 \triangleq \mathrm{CRS}$.
OUTPUT: The protocol outputs $\eta_{\mathsf{ep}}$ to its caller (but not to $\mathcal{Z}$).
```

Parties will send CLOCK-TICK to $\mathcal{F}_{\mathrm{ILClock}}$ to check if it receives a tick $= 0$, which indicates the beginning of a new (local) round. Note that alert parties will never change the interval index `itvl` here when adding 1 to `localTime` (regarding the rules of addtion, see Section 3.1); they will only adjust `itvl` in $\mathsf{SyncProc}$ (code in Appendix C.10). Meanwhile, for those stalled parties, their local time will increase as if no adjuetment happens.

Protocol $\mathsf{UpdateTime}(\mathsf{P}, \mathrm{R})$

```
// Precondition: Only executed if time-aware.
1: Send $(\textsc{clock-tick}, \mathrm{sid}_C)$ to $\mathcal{F}_{\mathrm{ILClock}}$ and receive $(\textsc{clock-tick}, \mathrm{sid}_C, \mathrm{tick})$
2: **if** $\mathrm{tick} = 0$ **then**
3:     $\mathtt{localTime} \leftarrow \mathtt{localTime} + 1$
4:     $\mathtt{fetchCompleted} \leftarrow \mathsf{false}$
5: **end if**
6: $\mathsf{ep} \leftarrow \mathsf{TargetRecalcEpoch}(\mathtt{localTime})$
OUTPUT: The protocol outputs localTime, ep to its caller (but not to $\mathcal{Z}$).
```

C.7 Process Beacons and Arrival Times

The following procedure processes imcoming beacons, bookkeeps their arrival times and filters out invalid as well as duplicate beacons. The predicate to verify becaons is presented in Appendix C.4. Regarding the duplicate beacons recording the same timestamp and miner indentity, only one with the earliest arrival time will be preserved.

Protocol $\mathsf{ProcessBeacons}(\mathsf{P}, \mathrm{sid}, B)$

```
1: if not $\mathtt{fetchCompleted}$ then
2:     Send $(\textsc{fetch}, \mathrm{sid})$ to $\mathcal{F}^{\mathsf{sync}}_{\mathrm{Diffuse}}$. denote the $i$-th response from $\mathcal{F}^{\mathsf{sync}}_{\mathrm{Diffuse}}$ by $(\textsc{fetch}, \mathrm{sid}, b)$
3:     Extract all received beacons $(\mathtt{SB}_1, \ldots, \mathtt{SB}_k)$ contained in $b \cup B$.
4:     for each $\mathtt{SB}_i$ with $\mathsf{arrivalTime}_{SB}(\mathtt{SB}) = \perp$ do
5:         $\mathtt{syncBuffer} \leftarrow \mathtt{syncBuffer} \cup \{\mathtt{SB}\}$
6:         Let $\mathtt{itvl}'$ be the interval index $\mathsf{Timestamp}(\mathtt{SB})$ belongs to
7:         if $\mathtt{isSync} \wedge (\mathtt{itvl} \geq \mathtt{itvl}')$ then
8:             Set $\mathsf{arrivalTime}_{SB}(\mathtt{SB}_i) \leftarrow (\mathtt{localTime}, \mathsf{final})$ // The measurement is final.
9:         else// Will be adjusted upon next time shift.
10:            $\mathsf{arrivalTime}_{SB}(\mathtt{SB}_i) \leftarrow (\mathtt{localTime}, \mathsf{temp})$
11:        end if
12:    end for
13:    // Buffer cleaning. Keep one representative arrival time.
14:    if $\mathtt{isSync}$ then
15:        Remove from $\mathtt{syncBuffer}$ all beacons s.t. $\mathsf{ValidSB}(\mathsf{P}, \mathrm{sid}, \mathtt{SB}, \mathcal{C}_{\mathsf{loc}}, \mathrm{R})$ returns $\mathsf{false}$
16:        $\mathtt{syncBuffer}_{\mathrm{valid}} \leftarrow \{\mathtt{SB}' \in \mathtt{syncBuffer} \mid \mathsf{ValidSB}(\mathsf{P}, \mathrm{sid}, \mathtt{SB}', \mathcal{C}_{\mathsf{loc}}, \mathrm{R}) = \mathsf{true}$
17:        Let $L = (\mathtt{SB}_1, \ldots, \mathtt{SB}_n)$ be a canonical ordering of $\mathtt{syncBuffer}_{\mathrm{valid}}$
18:        for each $\mathtt{SB} = \langle\langle \mathtt{itvl}', \mathtt{r}'\rangle, \mathsf{P}', \eta_{\mathtt{SB}}, \mathit{ctr}, \mathit{blockLabel}\rangle \in L$ do
19:            $Q_{\mathtt{SB}} \leftarrow \{\mathtt{SB}' = \langle\langle \mathtt{itvl}'', \mathtt{r}''\rangle, \mathsf{P}'', \cdot, \cdot, \cdot\rangle \in L \mid \mathsf{P}' = \mathsf{P}'' \wedge \langle \mathtt{itvl}', \mathtt{r}'\rangle = \langle \mathtt{itvl}'', \mathtt{r}''\rangle\}$
20:            $\min_{\mathtt{SB}} \leftarrow \min\{\mathsf{arrivalTime}(\mathtt{SB}') \mid \mathtt{SB}' \in Q_{\mathtt{SB}}\}$
21:            $\mathtt{SB}' \leftarrow \min\{\mathtt{SB}'' \in Q_{\mathtt{SB}} \mid \mathsf{arrivalTime}(\mathtt{SB}'') = \min_{\mathtt{SB}}\}$ // Min w.r.t. ordering in $L$
22:            Remove from $\mathtt{syncBuffer}$ all beacons $\langle\langle \mathtt{itvl}', \mathtt{r}'\rangle, \mathsf{P}', \cdot, \cdot, \cdot\rangle$ except $\mathtt{SB}'$
23:        end for
24:    end if
25: end if
OUTPUT: ok to its caller (but not to $\mathcal{Z}$).
```

C.8 Chain Selection

Parties drop invalid chains, and then select the a chain by *heaviest difficulty* chain selection rule (cf. $\mathsf{maxvalid}$ in [GKL17]).

More specifically, $\max(\mathcal{C}_1, \mathcal{C}_2)$ will return the *most difficult* of the two. In case $\mathrm{diff}(\mathcal{C}_1) = \mathrm{diff}(\mathcal{C}_2)$, $\max(\cdot, \cdot)$ always return the first operand to reflect the fact that parties adopt the first chain they obtain from the network.

Protocol $\mathsf{SelectChain}(\mathsf{P}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}}, \mathcal{N} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}, \mathrm{R})$

```
1: Set $\mathcal{C}_{\max} \leftarrow \mathcal{C}_{\mathsf{loc}}$
2: for $i = 1$ to $n$ do
3:     if $\mathsf{IsValidChain}(\mathsf{P}, \mathrm{sid}, \mathcal{C}_i, \mathrm{R})$ returns $\mathtt{true}$ then
4:         $\mathcal{C}_{\max} \leftarrow \max(\mathcal{C}_{\max}, \mathcal{C}_i)$
5:     end if
6: end for
7: Replace $\mathcal{C}_{\mathsf{loc}}$ by $\mathcal{C}_{\max}$.
OUTPUT: The protocol outputs $\mathcal{C}_{\max}$ to its caller (but not to $\mathcal{Z}$).
```

C.9 Mining Procedure

Once a party P has prepared all information and updated its state, it can run the core mining procedure formally given below. When $\mathtt{localTime}$ reports a timestamp that satisfies $\mathtt{r} \in I_{\mathrm{sync}}(\mathtt{itvl})$ (i.e., P stays in the beacon mining and inclusion phase), P will include the fresh beacons and check if he succeeds in the beacon mining procedure.

Note that, in the "prepare block content" part, we follow [BMTZ17] and adopt several predicates such as $\mathsf{blockify}_{\mathsf{OC}}$ and $\mathsf{validTX}_{\mathsf{OC}}$ to generate the merkle root ($\mathtt{st}$) of the block content. $\mathsf{blockify}_{\mathsf{OC}}$ is used to compute the root value, and $\mathsf{validTX}_{\mathsf{OC}}$ can justify whether an incoming transaction is valid w.r.t. the current blockchain state. This part is not relevant to the topic of this paper; for further discussion, we refer to [BMTZ17].

Protocol $\mathsf{MiningProcedure}(\mathsf{P}, \mathrm{sid}, \mathtt{itvl}, \mathtt{r}, \mathtt{buffer}, \mathcal{C}_{\mathsf{loc}})$

The following steps are executed in an (MAINTAIN-LEDGER, sid, minerID)-interruptible manner:

```
1: // Check if switch to a new chain.
2: if $h \neq H(\mathsf{head}(\mathcal{C}_{\mathsf{loc}}))$ then
3:     $h \leftarrow H(\mathsf{head}(\mathcal{C}_{\mathsf{loc}}))$
4: end if
5: // Prepare block content.
6: Set $\mathtt{buffer}' \leftarrow \mathtt{buffer}, \mathbf{N} \leftarrow \mathtt{tx}_{\mathsf{P}}^{\mathrm{base-tx}}$ , and $\mathtt{st} \leftarrow \mathsf{blockify}_{\mathsf{OC}}(\mathbf{N})$.
7: repeat
8:     Parse $\mathtt{buffer}'$ as sequence $(\mathtt{tx}_1, \ldots, \mathtt{tx}_n)$
9:     for $i = 1$ to $n$ do
10:        if $\mathsf{validTX}_{\mathsf{OC}}(\mathtt{tx}_i, \overrightarrow{\mathtt{st}} \,\|\, \mathtt{st}) = 1$ then
11:            $\mathbf{N} \leftarrow \mathbf{N} \,\|\, \mathtt{tx}_i$
12:            Remove $\mathtt{tx}'$ from $\mathtt{buffer}'$
13:            Set $\mathtt{st} \leftarrow \mathsf{blockify}_{\mathsf{OC}}(\mathbf{N})$
14:        end if
15:    end for
16: until $\mathbf{N}$ does not increase any more
17: // Check if a beacon should be included.
18: if $I_{\mathrm{sync}}(\mathtt{localTime}) = \mathsf{true}$ then
19:     $B \leftarrow \{\mathtt{SB}' \in \mathtt{syncBuffer} \mid \mathsf{ValidSB}(\mathsf{P}, \mathrm{sid}, \mathtt{SB}', \mathcal{C}_{\mathsf{loc}}, \mathrm{R}) = \mathsf{true}\}$
```

```
20:     Remove from $B$ all beacons $\texttt{SB} = \langle\langle \texttt{itvl}', \texttt{r}'\rangle, \mathsf{P}', \cdot, \cdot, \cdot\rangle$ that satisfy $(\mathsf{Timestamp}(\texttt{SB}) > \texttt{localTime}) \vee (I_{\text{sync}}(\mathsf{Timestamp}(\texttt{SB})) = \mathsf{true}) \vee \mathcal{C}_{\mathsf{loc}}$ contains a beacon $\langle\langle \texttt{itvl}', \texttt{r}'\rangle, \mathsf{P}', \cdot, \cdot, \cdot\rangle$.
21:     $\mathbf{N} \leftarrow \mathbf{N} \,\|\, B$
22:     Set $\texttt{st} \leftarrow \mathsf{blockify}_{\mathsf{OC}}(\mathbf{N})$
23: end if
24: // prepare 2-for-1 PoW.
25: $\mathit{blockLabel} \leftarrow h \,\|\, \texttt{st}, \mathit{txLabel} \leftarrow \mathsf{P}, \|\eta_{\texttt{itvl}}$
26: $u \leftarrow H(\mathit{ctr}, h, \texttt{st}, \texttt{localTime}, \mathsf{P}, \eta_{\texttt{itvl}})$
27: // Check if block mining succeed.
28: if $(u < T^{\mathcal{C}}_{\mathsf{ep}}) \wedge (\mathit{ctr} < 2^{32})$ then
29:     Set $\mathcal{B} \leftarrow \langle h, \texttt{st}, \texttt{localTime}, \mathit{ctr}, \mathit{txLabel}\rangle$ and update $\mathcal{C}_{\mathsf{loc}} \leftarrow \mathcal{C}_{\mathsf{loc}} \,\|\, \mathcal{B}$
30:     Send $(\textsc{diffuse}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}})$ to $\mathcal{F}^{\mathsf{bc}}_{\text{Diffuse}}$ and proceed from here upon next activation of this procedure. // Diffuse the extended chain and wait.
31: end if
32: // Check if a PoW timestamp transaction mining succeed.
33: if $([u]^{\mathsf{R}} < T^{\mathcal{C}}_{\mathsf{ep}}) \wedge (\mathit{ctr} < 2^{32}) \wedge (I_{\text{sync}}(\texttt{localTime}) = \mathsf{true})$ then
34:     $\texttt{SB} \leftarrow \langle \texttt{localTime}, \mathsf{P}, \mathit{ctr}, \eta_{\texttt{itvl}}, \mathit{blockLabel}\rangle$
35:     Send $(\textsc{diffuse}, \text{sid}, \texttt{SB})$ to $\mathcal{F}^{\mathsf{sync}}_{\text{Diffuse}}$ and set anchor at end of procedure to resume on next maintenance activation.
36: else
37:     Give up activation and set anchor at end of procedure to resume on next maintenance activation.
38: end if
39: $\mathit{ctr} \leftarrow \mathit{ctr} + 1$
```

C.10 Synchronization Procedure

The synchronization procedure is called when party's local clock enters a clock synchronization interval boundary (i.e., $\texttt{localTime} = \langle \texttt{itvl}, \texttt{itvl} \cdot \mathrm{R}\rangle$). Note that thanks to the timestamp scheme in $\mathsf{Timekeeper}$, parties will only pass the interval boundary for once.

Protocol $\mathsf{SyncProc}(\mathsf{P}, \mathrm{sid}, \mathrm{R})$

```
1: // Only called when: $\mathsf{P}$ is alert, $\texttt{localTime} = \langle \texttt{itvl}, \texttt{itvl} \cdot \mathrm{R}\rangle$ and $\texttt{itvl} > 0$
2: $B \leftarrow \{\mathcal{B} \mid (\mathcal{B} \in \mathcal{C}_{\mathsf{loc}}) \wedge (\mathsf{Timestamp}(\mathcal{B}) = \langle \texttt{itvl}, \cdot\rangle) \wedge (I_{\text{sync}}(\mathsf{Timestamp}(\mathcal{B})) = \mathsf{true})\}$
3: $SB \leftarrow \{\texttt{SB} \mid (\texttt{SB} \in \mathcal{B} \in B) \wedge (\mathsf{Timestamp}(\texttt{SB}) = \langle \texttt{itvl}, \cdot\rangle) \wedge (I_{\text{sync}}(\mathsf{Timestamp}(\texttt{SB})) = \mathsf{true})\}$
4: for each $\texttt{SB} = \langle\langle \texttt{itvl}', \texttt{r}'\rangle, \mathsf{P}', \mathit{ctr}, \eta_{\texttt{SB}}, \mathit{blockLabel}\rangle \in SB$ do
5:     // Find representative beacon and compute recommendation.
6:     Find unique $\texttt{SB}' = \langle\langle \texttt{itvl}', \texttt{r}'\rangle, \mathsf{P}', \cdot, \cdot, \cdot\rangle \in \texttt{syncBuffer}$. If inexistent, set $\texttt{SB}' \leftarrow \perp$.
7:     if $\texttt{SB}' \neq \perp$ then
8:         Set $\mathsf{arrivalTime}_{SB}(\texttt{SB}) \leftarrow \mathsf{arrivalTime}_{SB}(\texttt{SB}')$
9:         $\mathsf{recom}(\texttt{SB}) \leftarrow \mathsf{Timestamp}(\texttt{SB}) - \mathsf{arrivalTime}(\texttt{SB})$
10:    else
11:        $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\texttt{SB}\}$ // Negligible probability event in execution.
12:    end if
13: end for
```

```
14: $\mathsf{shift}_i \leftarrow \mathsf{med}\{\mathsf{recom}(\mathtt{SB}) \mid \mathtt{SB} \in SB\}$
15: for each $\mathtt{SB}$ with $\mathsf{arrivalTime}_{SB}(\mathtt{SB}) = (a, \mathsf{temp})$ do
16:     $\mathsf{arrivalTime}_{SB}(\mathtt{SB}) \leftarrow (a + \mathsf{shift}_i, \mathsf{final})$
17: end for
18: if $\mathsf{shift}_i = 0$ then // No adjustment, simply enter next epoch
19:     $\mathtt{itvl} \leftarrow \mathtt{itvl} + 1$
20: end if
21: if $\mathsf{shift}_i > 0$ then // Move fast forward
22:     $\mathsf{newTime} \leftarrow \langle \mathtt{itvl} + 1, \mathtt{r} + \mathsf{shift}_i \rangle, \mathtt{localTime} \leftarrow \langle \mathtt{itvl} + 1, \mathtt{r} \rangle, \mathcal{M}_{\text{chains}} \leftarrow \emptyset$
23:     while $\mathtt{localTime} < \mathsf{newTime}$ do
24:         $\mathtt{localTime} \leftarrow \mathtt{localTime} + 1$ // increment round counter in localtime by 1
25:         Let $\mathcal{N}_0$ be the subsequence of $\mathtt{futureChains}$ s.t. $\mathcal{C} \in N_0 :\Leftrightarrow \forall \mathcal{B} \in \mathcal{C} : \mathsf{Timestamp}(\mathcal{B}) < \mathtt{localTime}$
26:         Remove each $\mathcal{C} \in \mathcal{N}_0$ from $\mathtt{futureChains}$
27:         Call $\mathsf{SelectChain}(\mathsf{P}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}}, \mathcal{N}_0, \mathrm{R})$ to update $\mathcal{C}_{\mathsf{loc}}$
28:     end while
29:     Send $(\textsc{diffuse}, \mathrm{sid}, \mathcal{M}_{\text{chains}})$ to $\mathcal{F}^{\mathsf{bc}}_{\text{Diffuse}}$ and proceed from here upon next activation of this procedure.
30: end if
31: if $\mathsf{shift}_i < 0$ then // Set clock back
32:     Set $\mathtt{localTime} \leftarrow (\mathtt{itvl} + 1, \mathtt{r} + \mathsf{shift}_i)$
33: end if
OUTPUT: The protocol outputs ok to its caller (but not to $\mathcal{Z}$).
```

C.11 Reading the Ledger State

In order to read the ledger state, the party P first processes all relevant information and then extracts the state (the settled ledger).

Protocol $\mathsf{ReadState}(\mathsf{P}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}}, \mathrm{R})$

```
1: if $\mathtt{isInit} = \mathsf{false} \lor \mathtt{isSync} = \mathsf{false}$ then
2:     Output the empty state $(\textsc{read}, \mathrm{sid}, \varepsilon)$ (to Z).
3: else
4:     Call $\mathsf{FetchInformation}(\mathsf{P}, \mathrm{sid})$ and denote the output by $(\mathcal{C}_1, \ldots, \mathcal{C}_N), (\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$
5:     Set $\mathtt{buffer} \leftarrow \mathtt{buffer} \,\|\, (\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$ and define $\mathcal{N} \leftarrow \{\mathcal{C}_1, \ldots, \mathcal{C}_N\}$
6:     Call $\mathsf{UpdateTime}(\mathsf{P}, \mathrm{R})$
7:     Call $\mathsf{ProcessBeacons}(\mathsf{P}, \mathrm{sid})$
8:     Let $\mathcal{N}_0 := \{\mathcal{C} \in \mathcal{N} \cup \mathtt{futureChains} \mid \forall \mathcal{B} \in \mathcal{C} : \mathsf{Timestamp}(\mathcal{B}) \leq \mathtt{localTime}\}$
9:     Let $\mathcal{N}_1 := \{\mathcal{C} \in \mathcal{N} \mid \exists \mathcal{B} \in \mathcal{C} : \mathsf{Timestamp}(\mathcal{B}) > \mathtt{localTime}\}$
10:    $\mathtt{futureChains} \leftarrow (\mathtt{futureChains} \setminus \mathcal{N}_0) \cup \mathcal{N}_1$
11:    $\mathtt{fetchCompleted} \leftarrow \mathsf{true}$
12:    Call $\mathsf{SelectChain}(\mathsf{P}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}}, \mathcal{N}_0, \mathrm{R})$ to update $\mathcal{C}_{\mathsf{loc}}$
13:    Extract the state $\mathtt{st}$ from the current local chain $\mathcal{C}_{\mathsf{loc}}$
14:    Output $(\textsc{read}, \mathrm{sid}, \mathtt{st}^{\lceil k})$ (to $\mathcal{Z}$) // $\mathtt{st}^{\lceil k}$ denotes the prefix of $\mathtt{st}$ by pruning blocks with timestamps reporting the last $k$ rounds
```

```
15: end if
```

C.12 Simulate Clock Adjustments

If parties are merely de-registered from the random oracle $\mathcal{F}_{\mathrm{RO}}$ (namely, stalled for a limited time), they can bootstrap easily to the reliable state and time. Note that if parties are stalled, their `localTime` is still updated in UpdateTime and by computing the distance of round numbers in `lastTimeAlert` and `localTime` we get the exact number of rounds that have elapsed during the stall period.

Protocol SimulateClockAdjustment($\mathsf{P}, \mathrm{sid}, \mathrm{R}$)

```
1: $\mathsf{simulatedTime} \leftarrow \mathtt{lastTimeAlert}$
2: Set $\mathtt{r}'$ as round index in $\mathsf{simulatedTime}$
3: for $\mathtt{r} - \mathtt{r}'$ iterations do
4:     Let $\mathcal{N}_0$ be the subsequence of futureChains s.t. $\mathcal{C} \in \mathcal{N}_0 :\Leftrightarrow \forall \mathcal{B} \in \mathcal{C} : \mathsf{Timestamp}(\mathcal{B}) \leq \mathsf{simulatedTime}$
5:     Remove each $\mathcal{C} \in \mathcal{N}_0$ from futureChains.
6:     Emulate $\mathsf{SelectChain}(\mathsf{P}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}}, \mathcal{N}_0, \mathrm{R})$ with simulated time $\mathsf{simulatedTime}$ (instead of localTime) to update $\mathcal{C}_{\mathsf{loc}}$
7:     if $\mathsf{simulatedTime} = \langle \mathtt{itvl}, \mathtt{itvl} \cdot \mathrm{R} \rangle$ for interval itvl then
8:         Emulate $\mathsf{SyncProc}(\mathsf{P}, \mathrm{sid}, \mathrm{R})$ on simulated time $\mathsf{simulatedTime}$ (instead of localTime)
9:     end if
10: end for
11: Set $\mathtt{localTime} \leftarrow \mathsf{simulatedTime}$
12: Set $t_{\mathrm{work}} \leftarrow \mathtt{localTime} - 1$
OUTPUT: The protocol outputs ok to its caller (but not to $\mathcal{Z}$).
```

C.13 Round Finish Procedure

Once a party P has done its actions in a round, P claims finishing current round by calling FinishRound and sending CLOCK-UPDATE to $\mathcal{F}_{\mathrm{ILClock}}$. For details of the way to interact with $\mathcal{F}_{\mathrm{ILClock}}$ in a UC treatment, see [BGK+21].

Protocol FinishRound(P)

```
1: while A (CLOCK-UPDATE, $\mathcal{Z}$) has not been received during the current round do
2:     Give up activation (set the anchor here)
3: end while
4: Send (CLOCK-UPDATE, $\mathrm{sid}_C$) to $\mathcal{F}_{\mathrm{ILClock}}$. // Party will lose its activation here.
```

C.14 The Joining Procedure

Another main procedure of Timekeeper is JoinProc, where newly joint parties synchronize with other alert parties by passively listening to the protocol execution, building blockchain and processing beacons to derive a local time that is close to all alert parties. The default value of the parameters in each phase are summarized in Table 2.

Protocol $\mathsf{JoinProc}(\mathsf{P}, \mathrm{sid}, R, t_{\mathrm{off}}, t_{\mathrm{gather}})$

```
1: // Phase A, state-reset
2: Call $\mathsf{UpdateTime}(\mathsf{P}, \mathrm{R})$ // Align with newest round.
3: if $\texttt{localTime} > \langle 1,1\rangle$ then
4:     Set $\texttt{localTime} \leftarrow \langle 1,1\rangle$
5:     $\texttt{fetchCompleted} \leftarrow \mathsf{false}, \texttt{futureChains} \leftarrow \emptyset, \texttt{buffer} \leftarrow \emptyset$
6:     Set beacon arrival timetable as empty array
7: end if
8: // Phase B, chain-convergence
9: while $\texttt{localTime} < \langle 1,1\rangle + t_{\mathrm{off}}$ do
10:     if $\texttt{fetchCompleted} = \mathsf{false}$ then
11:         Call $\mathsf{FetchInformation}(\mathsf{P}, \mathrm{sid})$ and denote fetched chains by $\mathcal{N} := (\mathcal{C}_1, \ldots, \mathcal{C}_N)$
12:         Call $\mathsf{SelectChain}(\mathsf{P}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}}, \mathcal{N}, \mathrm{R})$ to update $\mathcal{C}_{\mathsf{loc}}$
13:         $\texttt{fetchCompleted} \leftarrow \mathsf{true}$
14:         $\mathsf{FinishRound}(\mathsf{P})$
15:     end if
16:     Call $\mathsf{UpdateTime}(P, R)$ to update $\texttt{localTime}$
17: end while
18: // Phase C, beacon-gathering
19: while $\texttt{localTime} \leq \langle 1,1\rangle + t_{\mathrm{off}} + t_{\mathrm{gather}}$ do
20:     if $\texttt{fetchCompleted} = \mathsf{false}$ then
21:         Call $\mathsf{FetchInformation}(\mathsf{P}, \mathrm{sid})$ and denote the output by $(\mathcal{C}_1, \ldots, \mathcal{C}_N), (\texttt{tx}_1, \ldots, \texttt{tx}_k)$
22:         Set $\texttt{buffer} \leftarrow \texttt{buffer} \parallel (\texttt{tx}_1, \ldots, \texttt{tx}_k)$
23:         Set $\texttt{futureChains} \leftarrow \texttt{futureChains} \parallel (\mathcal{C}_1, \ldots, \mathcal{C}_N)$
24:         Call $\mathsf{ProcessBeacons}$ to collect new beacons in this round. // All arrival times are
    temporary
25:         Call $\mathsf{SelectChain}(\mathsf{P}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}}, \texttt{futureChains}, R)$ to update $\mathcal{C}_{\mathsf{loc}}$
26:         $\texttt{fetchCompleted} \leftarrow \mathsf{true}$
27:         $\mathsf{FinishRound}(\mathsf{P})$
28:     end if
29:     Call $\mathsf{UpdateTime}(P, R)$ to update $\texttt{localTime}$
30: end while
31: // Phase D, shift-computation
32: $\texttt{syncBuffer}_{\mathrm{valid}} \leftarrow \{\texttt{SB}' \in \texttt{syncBuffer} \mid \mathsf{ValidSB}(\mathsf{P}, \mathrm{sid}, \texttt{SB}', \mathcal{C}_{\mathsf{loc}}, \mathrm{R}) = \mathsf{true}\}$
33: Initialize $i := 0$.
34: Set $i$ to be the minimum positive integer such that $\forall \texttt{SB} \in \mathcal{B} \in B : \texttt{SB} \in \texttt{syncBuffer}_{\mathrm{valid}} \wedge$
    $\mathsf{arrivalTime}(\texttt{SB}) \geq \langle 1,1\rangle + t_{\mathrm{off}} + t_{\mathrm{pre}}$ where $B \leftarrow \{\mathcal{B} \mid (\mathcal{B} \in \mathcal{C}_{\mathsf{loc}}) \wedge (\mathsf{Timestamp}(\mathcal{B}) =$
    $\langle i, \cdot\rangle) \wedge (I_{\mathrm{sync}}(\mathsf{Timestamp}(\mathcal{B})) = \mathsf{true})\}$.   // if no interval exists, $i$ is unchanged.
35: if $i \geq 1$ then
36:     for at most $(t_{\mathrm{gather}}$ div R$)$ iterations do
37:         $B \leftarrow \{\mathcal{B} \mid (\mathcal{B} \in \mathcal{C}_{\mathsf{loc}}) \wedge (\mathsf{Timestamp}(\mathcal{B}) = \langle i, \cdot\rangle) \wedge (I_{\mathrm{sync}}(\mathsf{Timestamp}(\mathcal{B})) = \mathsf{true})\}$
38:         $SB \leftarrow \{\texttt{SB} | \texttt{SB} \in \mathcal{B} \in B \wedge (\mathsf{Timestamp}(\texttt{SB}) = \langle i, \cdot\rangle) \wedge (I_{\mathrm{sync}}(\mathsf{Timestamp}(\texttt{SB})) = \mathsf{true})\}$
39:         for each $\texttt{SB} = \langle\langle \texttt{itvl}', \texttt{r}'\rangle, \mathsf{P}', ctr, \eta_{\texttt{SB}}, blockLabel\rangle \in SB$ do
```

```
40:             $Q_{\mathtt{SB}} \leftarrow \{\mathtt{SB}' = \langle\langle \mathtt{itvl}'', \mathtt{r}''\rangle, \mathsf{P}'', \cdot, \cdot, \cdot\rangle \in \mathtt{syncBuffer}_{\mathrm{valid}} \,|\, \mathsf{P}' = \mathsf{P}'' \wedge \langle \mathtt{itvl}', \mathtt{r}'\rangle = \langle \mathtt{itvl}'', \mathtt{r}''\rangle\}$
41:             if $Q_{\mathtt{SB}} \neq \emptyset$ then
42:                 $\min_{\mathtt{SB}} \leftarrow \min\{\mathsf{arrivalTime}(\mathtt{SB}) \mid \mathtt{SB} \in Q_{\mathtt{SB}}\}$
43:                 $\mathsf{arrivalTime}_{SB}(\mathtt{SB}) \leftarrow (\min_{\mathtt{SB}}, \mathsf{final})$
44:                 $\mathsf{recom}(\mathtt{SB}) \leftarrow \mathsf{Timestamp}(\mathtt{SB}) - \mathsf{arrivalTime}(\mathtt{SB})$
45:             else
46:                 $\mathcal{S} \leftarrow \mathcal{S} \backslash \{\mathtt{SB}\}$ // Negligible probability event in execution
47:             end if
48:         end for
49:         $\mathsf{shift}_i \leftarrow \mathsf{med}\{\mathsf{recom}(\mathtt{SB}) \mid \mathtt{SB} \in SB\}$
50:         for each SB with $\mathsf{arrivalTime}_{SB}(\mathtt{SB}) = (a, \mathsf{temp})$ do
51:             $\mathsf{arrivalTime}_{SB}(\mathtt{SB}) \leftarrow (a + \mathsf{shift}_i, \mathsf{temp})$
52:         end for
53:         if $\mathtt{r} + \mathsf{shift}_i \leq (i+1)\mathrm{R}$ then
54:             Set $\mathtt{localTime} \leftarrow \langle i+1, \mathtt{r} + \mathsf{shift}_i\rangle$
55:             Break
56:         else
57:             Set $\mathtt{localTime} \leftarrow \langle i+1, \mathtt{r}+\mathsf{shift}_i\rangle$ // Temporarily invalid; will be adjust later.
58:             set $i \leftarrow i + 1$ // continue iteration.
59:         end if
60:     end for
61:     $\mathtt{isSync} \leftarrow \mathsf{true}$; run $\mathsf{SelectChain}(\mathsf{P}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}}, \mathtt{futureChains}, \mathrm{R})$ to update $\mathcal{C}_{\mathsf{loc}}$; $t_{\mathrm{work}} \leftarrow \langle \mathtt{itvl}, \mathtt{r} - 1\rangle$
62:     for each beacon $\mathtt{SB} \in \mathtt{syncBuffer}_{\mathrm{valid}}$ with $\mathsf{Timestamp}(\mathtt{SB}) = \langle \mathtt{itvl}, \cdot\rangle$ do
63:         Parse $\mathsf{arrivalTime}_{SB}(\mathtt{SB})$ as $(a, \mathsf{temp})$. Define $\mathsf{arrivalTime}_{SB}(\mathtt{SB}) \leftarrow (a, \mathsf{final})$
64:     end for
65: end if
```

D Proofs Omitted from the Main Body

Proof of Lemma 1. If two blocks are obtained at nominal rounds which are at distance at least $\Delta + \Phi$, then we are certain that the later block increased the accumulated difficulty. To be precise, assume $S^* \subseteq S$ is such that, for all $i, j \in S^*$, $|i - j| \geq \Delta + \Phi$ and $Y_i > 0$. We argue that, by round v, every honest party has a chain of difficulty at least

$$d + \sum_{r \in S^*} Y_r \geq d + Q(S).$$

Observe first that every honest party will receive the chain of difficulty d by round $u + \Delta + \Phi$ and so the first block obtained in S^* extends a chain of weight at least d. Next, note that if a block obtained in S^* is the head of a chain of weight at least d', then the next block in S^* extends a chain of weight at least d'. ☐

Proof of Lemma 2. For the proof of Lemma 2(a)(b)(c), we refer to [GKL20]. Regarding Lemma 2(d), note that we are under the same conditon as that in Lemma 2(c), we have

$$A(J') < A(J) < (1-\delta+3\epsilon)Q(S) \leq (1-\delta+3\epsilon)(1+\epsilon)ph(S) < (1-\delta+3\epsilon)(1+\epsilon)(1-\delta)p|J'| < (1+\epsilon)p|J'|.$$

The second inequality follows Lemma 2(c); the third one comes from Definition 7(a); the last inequality is a consequence of Condition (C2).

By combining the upper bound on $A(J')$ with $D(S) < (1+\epsilon)ph(S)$ (which comes from the definitions) we get the desired inequality. $\square$

Proof of Theorem 3. Definition 7 extends the definition of typical executions in [GKL20] by adding lower bound on $D(S)$ in item (a). Hence regarding the proof of the rest part in Definition 7, we refer to [GKL20]. In this proof we only consider the probablity of violating the lower bound on $D(S)$ and the eventual error probability.

Fix a set of $R \geq \ell$ consecutive rounds $S = s_1, s_2 \ldots, s_R$ and let $h_j, j \in [R]$ denote the number of honest queries make during round s_j. We work on per qeury that alert parties make during S. Let J denote the queries in S, $\nu = |J| = h(S)$, and Z_i the difficulty of any block obtained from query i. Consider the sequence of random variables

$$X_0 = 0; X_k = \sum_{i\in[k]} Z_i - \sum_{i\in[k]} \mathbb{E}[Z_i|\mathcal{E}_{i-1}], k \in [\nu].$$

This is a martingale sequence with respect to sequence $(\mathcal{E}_0, \mathcal{E}_1, \ldots, \mathcal{E}_\nu)$ in that

$$\mathbb{E}[X_k|\mathcal{E}_{k-1}] = \mathbb{E}[Z_k - \mathbb{E}[Z_k|\mathcal{E}_{k-1}]|\mathcal{E}_{k-1}] + \mathbb{E}[X_{k-1}|\mathcal{E}_{k-1}] = X_{k-1}.$$

The last equation follows the linearity of conditional expectation and the facth that X_{k-1} is a deterministic function with respect to $\mathcal{E}_{k-1}$.

Regarding the details relevant to Theorem 19, we pick t as

$$\epsilon \sum_{i\in[\nu]} \mathbb{E}[Z_i|\mathcal{E}_{i-1} = E_{i-1}] \leq \epsilon \sum_{j\in[R]} ph_j = \epsilon ph(S) \overset{\text{def}}{=} t$$

and consider an execution satisfying G_t. Fix $i \in [\nu]$, let j be the round s_j that query i belongs to. Let

$$Z_i - \mathbb{E}[Z_i|\mathcal{E}_{i-1} = E_{i-1}] \leq \frac{1}{T_j^{\min}} = \frac{ph_j}{ph_j T_j^{\min}} \leq \frac{\gamma ph(S)}{ph_j T_j^{\min} R} \leq \frac{\gamma ph(S)}{fR/(2\gamma^2)} = \frac{2\gamma^3 t}{\epsilon f R} \overset{\text{def}}{=} b$$

and we see that the event G implies $X_k - X_{k-1} \leq b$. To get the bound on V, note that

$$\mathbf{Var}(X_k - X_{k-1}|\mathcal{E}_{k-1}) = \mathbb{E}[(Z_i - \mathbb{E}[Z_i|\mathcal{E}_{i-1}])^2|\mathcal{E}_{k-1}] = \mathbb{E}[Z_i^2|\mathcal{E}_{k-1}] - (\mathbb{E}[Z_k|\mathcal{E}_{k-1}])^2 \leq \mathbb{E}[Z_i^2|\mathcal{E}_{k-1}].$$

Hence, based on the independence of random variables as well as Fact 1(c), we pick v as

$$\begin{aligned}\sum_{k\in[\nu]} \mathbb{E}[Z_i^2|\mathcal{E}_{k-1} = E_{k-1}] &\leq \sum_{j\in[R]}\sum_{i\in[h_j]} \frac{1}{T_i^2} \cdot pT_i = \sum_{j\in[R]} \frac{ph_j}{T_j^{\min}} = \sum_{j\in[R]} \frac{(ph_j)^2}{ph_j T_j^{\min}} \\ &\leq \frac{2\gamma^2}{f} \sum_{j\in[R]} (ph_j)^2 \leq \frac{2\gamma^3}{fR} \cdot (ph(S))^2 \leq \frac{2\gamma^3 t^2}{\epsilon^2 f R} \overset{\text{def}}{=} v.\end{aligned}$$

In view of these bounds (note that $bt = \epsilon v$), by Theorem 19 and Condition (C1), we have

$$\mathbf{Pr}[-X_\nu \geq t \wedge G_t] \leq \exp\Big\{-\frac{t^2}{2v(1+\frac{\epsilon}{3})}\Big\} \leq \exp\Big\{-\frac{\epsilon^2 f R}{4\gamma^3(1+\frac{\epsilon}{3})}\Big\} \leq e^{-\lambda}. \tag{10}$$

Combining Equation (10) and those error probablities in [GKL20], we get asymptotically the same result. I.e., the probablity that "$\mathcal{E}$ is not typical" is bounded by $O(L^2)(e^{-\lambda} + 2^{-\kappa})$. $\square$

Proof of Lemma 5. Suppose head$(\mathcal{C} \cap \mathcal{C}')$ was created in round v and let $u \leq v$ be the greatest round in which an honest party computed a block on $\mathcal{C} \cap \mathcal{C}'$. Let $U = \{i : u < i \leq r\}$, $S = \{i : u + \Delta + \Phi \leq i \leq r - (\Delta + \Phi)\}$, and let J denote the adversarial queries that correspond to the rounds in U. Consider the following claim.

Below we say that $d \in \mathbb{R}$ is *contained* in a block $\mathcal{B}$ (and write $d \in \mathcal{B}$), when $\mathcal{B}$ extends a chain $\mathcal{C}$ and $\operatorname{diff}(\mathcal{C}) < d \leq \operatorname{diff}(\mathcal{C}\mathcal{B})$.

Claim. *If $r - v \geq \ell + 2(\Delta + \Phi)$, then $2Q(S) \leq D(U) + A(J)$.*

Proof. Associate with each $r \in S$ such that $Q_r > 0$ an arbitrary honest block that is computed at round r for difficulty Q_r. Let B be the set of these blocks and note that their difficulties sum to $Q(S)$. We argue the existence of a set of blocks B' computed in U such that $B \cap B' = \emptyset$ and $\{d \in \mathcal{B} : \mathcal{B} \in B\} \subseteq \{d \in \mathcal{B} : \mathcal{B} \in B'\}$. This suffices, because each block in B' contributes either to $D(U) - Q(S)$ or to $A(J)$ and so $Q(S) \leq D(U) - Q(S) + A(J)$.

Consider, then, a block $\mathcal{B} \in B$ extending a chain $\mathcal{C}^*$ and let $d = \operatorname{diff}(\mathcal{C}^*\mathcal{B})$. If $d \leq \operatorname{diff}(\mathcal{C} \cap \mathcal{C}')$ (note that $u < v$ in this case and head$(\mathcal{C} \cap \mathcal{C}')$ is adversarial), let $\mathcal{B}'$ be the block of $\mathcal{C} \cap \mathcal{C}'$ containing d. Such a block clearly exists and and was computed after u. Furthermore, $\mathcal{B}' \notin B$, since $\mathcal{B}'$ was computed by the adversary. If $d > \operatorname{diff}(\mathcal{C} \cap \mathcal{C}')$, note that there is a unique $\mathcal{B} \in B$ such that $d \in \mathcal{B}$ (recall the argument in Chain Growth Lemma 1). Since $\mathcal{B}$ cannot simultaneously be on $\mathcal{C}$ and $\mathcal{C}'$, there is a $\mathcal{B}' \notin B$ either on $\mathcal{C}$ or on $\mathcal{C}'$ that contains d. □

Since $S \geq \ell$ and Lemma 4 implies that neither $\mathcal{C}$ nor $\mathcal{C}'$ are stale, from Lemma 2(a)(c) we get that $D(U) < (1+5\epsilon)Q(S)$ and $A(J) < (1-\delta+3\epsilon)Q(S)$, which implies $D(U)+A(J) < 2Q(S)$ when $\delta \geq 8\epsilon$ (by Condition (C2)). This contradicts the claim above. Hence, $r - v < \ell + 2(\Delta + \Phi)$. □

E Glossary

E.1 Main Parameters of Timekeeper

E.2 Main Variables of Timekeeper Participants

Parameter	**Description**
h_r	The number of honest RO queries in (nominal) round r.
t_r	The number of RO quries by $\mathcal{A}$ in (nominal) round r.
δ	Advantage of honest parties ($t_r < (1-\delta)h_r$ for all r).
f	The probability at least one honest RO query out of n_0 computes a block for target T_0.
R	The length of a clock synchronization interval in number of rounds.
M	The length of a target recalculation epoch in number of rounds.
K	The length of convergence phase in a clock synchronization interval in number of rounds.
Δ	Network delay in rounds.
Φ_{clock}	The upper bound of the drift that $\mathcal{A}$ can set.
Φ	The upper bound of the difference between honest parties' local clocks. We require that $\Phi = \Phi_{\text{clock}} + \Delta$.
κ	Security parameter; length of the hash function output.
(γ, s)	Respecting environment parameter.
ϵ	Quality of concentration of random variables.
λ	Related to the properties of the protocol.

Table 3: Main Parameters of Timekeeper.

Variable	**Description**
`localTime`	The party P's current timestamp in the form of $\langle \texttt{itvl}, \texttt{r} \rangle$.
`itvl`	The interval that `r` belongs to.
`ep`	The epoch that `r` belongs to.
$\mathcal{C}_{\mathsf{loc}}$	The local chain the party adopts based on which it does mining and exports the ledger state.
`buffer`	a buffer of transactions.
`futureChains`	A buffer to store chains with blocks whose timestamps belong to the future (logical) rounds.
$\mathsf{arrivalTime}_{SB}(\cdot)$	A map that assigns to each synchronization beacon a pair (a, b), where a is the arrival time and b is an indication of whether a is final or temp.
$\mathsf{arrivalTime}(\cdot)$	Shorthand for the first (arrival time) element of the pair $\mathsf{arrivalTime}_{SB}(\cdot)$.
`isSync`	A bit variable to store the synchronization status.
`fetchCompleted`	A variable to store whether the round messages have been fetched.
`lastTimeAlert`	The timestamp which stores the last alert time. Used for re-joining if the party was only stalled.

Table 4: Main state variables in Timekeeper.