Consensus on Clock in Universally Composable Timing Model

Clock Syncronization Protocol for Full Nodes of a Blockchain

Handan Kılınç Alper

Web3 Foundation

handan@web3.foundation

Abstract

In the blockchain space, there are elaborate proof-of-stake based protocols with an assumption of clock synchronization, i.e. that all of them know the current time of the protocol. However, this assumption is satisfied by relying on the security of centralized systems such as Network Time Protocol (NTP) or Global Positioning System (GPS). An attack on these systems (which has happened in the past) can cause corruption of blockchains that rely on the time data that they provide. To solve this problem in the nature of the decentralized network, we first define a general universally composable (GUC) model that captures the notion of consensus on a clock. A consensus clock is a clock that is agreed upon by honest parties by considering the clocks of all parties. In the end, we give a simple but useful protocol relying on a blockchain network. Our protocol is secure according to our new model. It can be used by full nodes of a blockchain who need to have a notion of common time to preserve the correctness and the security of the blockchain protocol. One advantage of our protocol is that it does not cause any extra communication overhead on the underlying blockchain protocol.

1 Introduction

The first popular decentralized cryptocurrency, Bitcoin, maintains its public distributed ledger with a proof-of-work (PoW) based consensus protocol. PoW has been studied for many decades and its impact in the development of blockchain technologies is undeniable. However, it is also a well-known fact that this protocol consumes vast amounts of energy. Consequently, there are ongoing efforts in the community to replace PoW with more energy-efficient alternatives, that still preserve its decentralized features. *Proof of stake* (PoS) is arguably the most promising replacement among the many recently proposed solutions [16, 5, 18, 1]. It has the same nature as PoW; while in the latter the next block producer is selected from a set of nodes at random proportional to their computational power, in PoS, they are randomly selected proportional to their stake.

One of the most critical issues in PoS is constructing a selection mechanism which cannot be biased by an adversary. Many solutions to the problem of unbiased random selection have been proposed, based on random oracles [35, 14], publicly verifiable secret sharing (PVSS) schemes [27], verifiable random functions (VRF) [15, 13, 21, 22], and threshold cryptography [22]. Using any such selection mechanism, each party must decide if they are eligible to produce the next block within a specific time interval. However, a second critical issue, which has not yet been treated satisfactorily, is how this party detects this time interval correctly. Indeed, in order to preserve security, it is paramount for an honest party to release their produced block at the right time. If they are late, then other producers continue to build on the chain without seeing their block; as a result, the honest party may not have any chance to contribute to the chain growth. If they are early, a similar problem arises.

One may ask why parties of a blockchain cannot simply use their own local computer clock to deduce their time interval. It is a known fact that a party cannot rely solely on the computer clock, as they are controlled by the vibration of crystal oscillators which are not very accurate (up to a few seconds drift in one day). Therefore, these parties usually update their local time via the Network Time Protocol (NTP) which has long been the main clock synchronization protocol on the internet. In the past, NTP servers have been attacked [31, 37, 33]. In one incident (on November 19, 2012) [8], two critical NTP servers were set to a time twelve years in the past. This caused many important external servers, such as Active Directory authentication servers, to fail. If the same attack happens in a proof-of-stake blockchain protocol, honest parties would stop producing blocks because they think that their time did not come, while malicious full nodes would continue to produce blocks, populating the blockchain entirely with maliciously-produced blocks. Another option is to synchronize local time with the Global Posioning System (GPS) clock. Although this requires a little more investment in setup, it is more accurate than NTP and does not have the potential problem of corrupted servers. However, GPS is also vulnerable to spoofing attacks [40, 25, 34, 24, 41] (e.g., delay signals). Even in the absence of an actual attack, mere poor weather conditions can cause inaccurate signals to be received from the GPS satellite. All of these existing solutions show that relying on a centralized system for time synchronization is a major security vulnerability, as well as going against the ultimate goal of building a fully decentralized system.

Sharing timestamps among parties of a blockchain is an appropriate solution, but without an additional mechanism like the one we propose, it is not obvious how to use these timestamps in a system with malicious actors. For example, consider a method that says synchronize the clock based on the timestamp contained in the last block. This is not a secure solution for several reasons: first, not all parties have the same block considered as their last block, and second, it gives power to a malicious block producer to change the clocks of honest parties as he wants. Solutions that give power to the adversary to modify honest parties' clocks of honest parties is certainly not ideal. Beyond this, it is also important to ensure that the clocks of honest parties do not dramatically shift, causing block production to stop (as in the example noted above). Considering these issues, we propose a new synchronization protocol on top of a blockchain protocol that interprets timestamps (clock values) in blocks and arrival times of blocks to obtain a consensus clock between parties.

In designing a solution, the other issue we encountered was proving the security of such synchronization mechanisms. For this, we decided to prove the security of our protocol in a universally composable (UC) model, as we want to make sure that our protocol can be composed on top of blockchain protocols securely. The closest UC model for our purposes is the GUC network time model designed by Canetti et al. [12], which models the clock adjustment of a client with servers which can be corrupted. In their model, clients want to synchronize their clock with the reference clock. However, when we consider the synchronization issue in a blockchain protocol, we see that we do not actually need any notion of a reference clock. In simple terms, the functionality we want to have is to update clocks of honest parties such that they are close enough to each other, and to limit the difference between the updated clocks and the old clocks for the security of the blockchain. For our purposes, we do not need to consider how close the updated clocks are to a reference clock. Therefore, we design a new general universally composable (GUC) consensus clock. In our model, the parties have local timers which let them construct their own clock based on a consensus clock. We note that the notion of consensus on clock is not a new idea. Indeed, it has been studied in other areas outside of cryptography [30, 36, 23, 3, 42, 38, 29]. However, there is not currently any security model capturing this notion except ours. We note that our model is generic and can be used for the security of any synchronization protocol without a reference clock.

Our motivation for this kind of security model without a reference clock comes from blockchain consensus mechanisms. In such distributed and decentralized consensus protocols, accepting one static clock (e.g., reference clock in [12]) as the correct clock goes against the grain of distributed consensus. Imagine a protocol where each party starts with a synchronized clock. As time passes, the synchronization may be broken because of random drift on clocks. At some point, parties may end up with a situation where no relation exists between their clocks. In this case, the question is, in respect to which reference clock should the parties synchronize their clock? We have at least two options for a reference clock:

- An authority defines the reference clock to be synchronized similar to the GUC model Canetti et al. [12].
- Parties agree on the reference clock to be synchronized

We believe most people would agree on the second option being more appropriate if we want to construct completely distributed system without trust in any particular entity. Therefore, we construct a new GUC clock model which avoids any authority deciding with which reference clock parties should synchronize.

1.1 Our Contribution

In more detail, our contribution is as follows:

- We construct a GUC model that captures the notion of consensus on clocks. Our model realizes situations where a party has a local clock constructed based on the ticks of his local timer and wants to update the local clock with a consensus clock. We define a global functionality which defines the rate of timers globally and another functionality of a local timer which does not necessarily follow the global rate to capture the notion of drifted timers in the real world. Our other functionality provides the consensus clock to honest parties. According to our definition, the consensus clock can change as the time passes based on the clocks of parties but this change is limited. To the best of our knowledge, our GUC model is the first model which models the notion of consensus on clock. We note that this model was not designed specifically for blockchain protocols. It can be used by any protocol whose aim is to achieve consensus on clocks in a decentralized manner.
- We construct a generic protocol called *Relative Time* protocol that is realizable in our model. Our protocol ensures that honest clocks are close enough to the consensus clock even if a local timer of a party drifts or network delay exists. Our protocol can be adapted to all blockchain protocols that fit our abstraction. Thus, we solve the synchronization problem in PoS-based blockchain protocols where an eligible block producer creates a block in a certain time interval.

We note that our relative time protocol is currently used in Polkadot [43] which aims to connect multiple blockchain networks.

1.2 Related Works

UC Timing Model: There are UC models [10, 26] designed to emulate the synchronous communication between parties but they do not provide ways for realizing the functionalities from a real world solutions such as network time. The closest timing model to ours is called the network time model by Canetti et al. [12]. Their model has a reference clock that keeps a counter incremented by the adversary. The network time model captures the physical clocks where a party can access immediately but that may be arbitrarily shifted and which can be updated by asking other parties' clock. The security of the clock is defined based upon the closeness of the reference clock. So, the ultimate goal in this model is to obtain local clocks with a time close enough to the reference clock. Differently, in our model, the ultimate goal of parties is to have clocks that are close to each other. It is not important how close the consensus clock is to a reference clock (e.g. a clock defined by NTP, GPS). Our model is useful for protocols which do not need to rely on any real world notion of the correct time for security and completeness. In a nutshell, our model is a version of the network time model where the reference clock is defined based on the consensus on a clock.

PoS protocols: In Ouroboros [27] and Ouroboros Praos [15] protocols, there is a common timeline that is divided into *slots*, where each slot may or may not have a block producer. Their security is based on the assumption that all parties know when each slot starts and ends. The Ouroboros Genesis [6] protocol is similar to Ouroboros Praos, and it explicitly mentions that violation of this assumption breaks the security of the chain selection rule. The timeline in the Dfinity consensus protocol [22] progresses based on certain events, but it is not clear how everyone can agree on the time at which any event occurs in a partially synchronous network. On the other hand, the Algorand protocol [13, 21] executes Byzantine agreement on each block, hence parties can trust all blocks and adjust their local clock according to the round inside the blocks. We note that this is not possible in Ouroboros [27], Ouroboros Praos [15] or Ouroboros Genesis [6] because a party cannot know whether a block is produced by an honest party, or whether it is sent during the correct time slot. Snow White [14, 35] is the only protocol where this timing issue is considered in their analysis, and the authors propose adding up the maximum time difference between parties into the network delay. However, they do not propose any protocol to obtain clocks with this maximum difference.

Recently, a new protocol called Ouroboros Chronos [7] was proposed to solve the timing issues in Ouroboros Genesis [6] that we mention. Their adversarial model is different than ours since they assume that there exists a core set of parties (termed "alert parties") who are honest and synchronized. The synchronization algorithm that they propose helps new joining parties to synchronize themselves with the alert parties. The existence of such a core set of parties is based on the assumption that their clocks follow almost the same rate. However, we know that modern computer clocks do not have such an accurate rate. Our model and theirs differ on this point, as we assume that the clocks of parties can arbitrarily shift considering the nature of real world computer clocks based on crystal oscillators. Therefore, synchronization between parties cannot be maintained for a long time in our model unless there is a mechanism, such as the one we propose, to preserve it. In addition to having a different adversarial model, our protocol is generic and can thus be applied any blockchain protocol compatible with our abstraction (e.g. Ouroboros Praos, Genesis [15, 6]). Ouroboros Clepsydra [4] has the similar adversarial model as Chronos.

Synchronization Protocols: Independently from blockchains, synchronization protocols [17, 19, 32, 39, 28] have been deeply studied in previous work, as it is a important component in distributed systems. Many protocols [30, 36, 23, 3] exist for consensus clocks with different assumptions. One approach is based on dividing the network into some well-connected clusters [17] that aim to achieve consensus between clusters. There are also fully distributed approaches [42, 38] based on clock skew and network delay estimation. Even though many works exist related to consensus clock, to best of our knowledge, there exists no formal cryptographic security model as we defined for these types of functionality . Differently, our protocol works on top of a secure blockchain protocol with a consensus mechanism which lets parties also reach consensus on clocks. Compared to the previous protocols, our proposed protocol has the advantage of building the synchronization protocol on top of an existing consensus mechanism.

2 Preliminaries

Notations: We use $\mathcal{D}$ to define a distribution. $x \leftarrow \mathcal{D}$ shows that x is selected with respect to the distribution $\mathcal{D}$.

Two ensembles $X = \{X_1, X_2, ..., X_n\}$ and $Y = \{Y_1, Y_2, ..., Y_n\}$ are computationally indistinguishable if for all probabilistic polynomial time (PPT) algorithms D and for all $c > 0$, there exists an integer N such that for all $n \geq N$

$$|\mathsf{Pr}[\mathsf{D}(X_i) = 1] - \mathsf{Pr}[\mathsf{D}(Y_i) = 1]| < \frac{1}{n^c}.$$

$\approx$ means that two ensembles are computationally indistinguishable (i.e. $X \approx Y$).

2.1 Blockchain

A protocol that defines the construction of a blockchain is called a blockchain protocol. Garay et al. [20] define the properties defined below in order to obtain a secure blockchain protocol. A blockchain protocol progresses in rounds $C_1, C_2, ...$ which can also executed in loose and asynchronous way. The formal definition of the concept of C_1, C_2 is in Section 3.1. If a blockchain protocol satisfies the properties below, it is called secure.

Definition 2.1 (Common Prefix (CP) Property [20]). *The CP property with parameters $k \in \mathbb{N}$ ensures that any blockchains B_1, B_2 owned by two honest parties at the onset of rounds $C_1 < C_2$ satisfy that $B_1^{\lceil k}$ is the prefix of B_2 where $B_1^{\lceil k}$ is B_1 without last k blocks.*

In other words, the CP property ensures that blocks which are k blocks before the last block of an honest party's blockchain cannot be changed. We call all unchangeable blocks *finalized* blocks.

Definition 2.2 (Chain Growth (CG) Property [20]). *The CG property with parameters $\tau \in (0, 1]$ and $s_{cg} \in \mathbb{N}$ ensures that if the length of a blockchain owned by an honest party at the onset of a round C_u is ℓ_u and the length of the same blockchain at round C_v where $C_v \leq C_u - s_{cg}$ is ℓ_v, then the $\ell_u - \ell_v \geq \tau s_{cg}$.*

In other words, the CG property guarantees if a chain is owned by an honest party at a round, then this chain has grown τs_{cg} blocks in every s_{cg} rounds.

Definition 2.3 (Chain Quality (CQ) Property [20]). *The CQ property with parameters $\mu \in (0, 1]$ and $k \in \mathbb{N}$ ensures that the ratio of honest blocks in any k length portion of a blockchain owned by an honest party is at least μ.*

The CQ property ensures the existence of sufficient honest blocks on any blockchain owned by an honest party.

We also define a new property which is necessary for our protocol.

Definition 2.4 (Chain Density (CD) Property)**.** *The CD property with parameters $s_{cd} \in \mathbb{N}$ ensures that if a blockchain owned by an honest party at the onset of a round C_u is B then any portion of B spanning s_{cd} prior rounds with n blocks contains number of n_h honest blocks where $\frac{n_h}{n} > \frac{1}{2}$.*

The CD property ensures that a blockchain adopted by an honest party contains more honest blocks than malicious ones in a sufficiently long span of that blockchain. A blockchain which has been constructed based on the longest chain rule implies the CD property if it satisfies CG and CQ [27].

2.2 Universally Composable (UC) Model:

The UC model consists of an ideal functionality that defines the execution of a protocol in an ideal world where there is a trusted entity. The real-world execution of a protocol (without a trusted entity) is called UC-secure if running the protocol with the ideal functionality $\mathcal{F}$ is indistinguishable by any external environment $\mathcal{Z}$ from the protocol running in the real-world.

A protocol π is defined with distributed interactive Turing machines (ITM). Each ITM has an inbox collecting messages from other ITMs, adversary $\mathcal{A}$ or environment $\mathcal{Z}$. Whenever an ITM is activated by $\mathcal{Z}$, the ITM instance (ITI) is created. We identify ITI's with an identifier consisting of a session identifier *sid* and the party identifier P.
π in the Real World: $\mathcal{Z}$ initiates all or some ITM's of π and the adversary $\mathcal{A}$ to execute an instance of π with the input $z \in \{0,1\}^*$ and the security parameter κ. The output of a protocol execution in the real world is denoted by $\mathsf{EXEC}(\kappa, z)_{\pi,\mathcal{A},\mathcal{Z}} \in \{0,1\}$. Let $\mathsf{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$ denote the ensemble $\{\mathsf{EXEC}(\kappa, z)_{\pi,\mathcal{A},\mathcal{Z}}\}_{z\in\{0,1\}^*}$.
π in the Ideal World: The ideal world consists of an incorruptible ITM $\mathcal{F}$ which executes π in an ideal way. The adversary $\mathcal{S}$ (called simulator) in the ideal world has ITMs which forward all messages provided by $\mathcal{Z}$ to $\mathcal{F}$. These ITMs can be considered corrupted parties and are represented as $\mathcal{F}$. The output of π in the ideal world is denoted by $\mathsf{EXEC}(\kappa, z)_{\mathcal{F},\mathcal{S},\mathcal{Z}} \in \{0,1\}$. Let $\mathsf{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ denote the ensemble $\{\mathsf{EXEC}(\kappa, z)_{\mathcal{F},\mathcal{S},\mathcal{Z}}\}_{z\in\{0,1\}^*}$.

$\mathcal{Z}$ outputs whatever the protocol in the real world or ideal world outputs. We refer to [9, 10] for further details about the UC-model.

Definition 2.5. *(UC-security of π) Let π be the real-world protocol and $\mathcal{F}$ be the ideal-world functionality of π. We say that π UC-realizes $\mathcal{F}$ (π is UC-secure) if for all PPT adversaries $\mathcal{A}$ in the real world, there exists a PPT simulator $\mathcal{S}$ such that for any environment $\mathcal{Z}$,*

$$\mathsf{EXEC}_{\pi,\mathcal{A},\mathcal{Z}} \approx \mathsf{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$$

π in the Hybrid World: In the hybrid world, the parties in the real world interact with some ideal functionalities. We say that a protocol π in hybrid world UC-realizes $\mathcal{F}$ when π consists of some ideal functionalities.

Generalized UC model [11] (GUC) formalizes the global setup in a UC-model. In GUC model, $\mathcal{Z}$ can interact with arbitrary protocols and ideal functionalities $\mathcal{F}$ can interact with GUC functionalities $\mathcal{G}$.

3 UC-Model for Relative Time

In this section, we describe our new GUC-model for consensus clocks and a partially synchronous network for blockchain protocols.

In our model we have multiple ITMs that we call as a party P_i. $\mathcal{Z}$ activates all parties and the adversary and creates ITIs. $\mathcal{A}$ is also activated whenever the ideal functionalities are invoked. $\mathcal{A}$ can corrupt parties $P_1, P_2, ..., P_n$. When $\mathcal{Z}$ permits a corruption of a party P_i, it sends the message $(\mathsf{Corrupt}_i, P_i)$. If a party is corrupted, then whenever it is activated, its current state is shared with $\mathcal{A}$.

We first introduce our new GUC-model for the concept of consensus clocks, and then give the UC-model for a partially synchronous network similar to the model in [15, 6].

3.1 GUC-Model of Relative Time

A similar model to ours, "GUC network time model", is introduced by Canetti et al. [12]. This models the clock adjustment of a client connected to servers which can be corrupted. In their model, clients attempt to adjust their clock to be as close as possible to the reference clock which can be controlled by the adversary. In our model, we do not necessarily have a reference clock that progresses depending on inputs of $\mathcal{Z}$. Instead, the parties come to a consensus on a clock which can be considered as reference clock in the GUC network time model [12].

In a nutshell, we construct a GUC- model where parties have access to their local timers (e.g., computer clocks in real life) that are constructed to tick according to a certain global metric time but it may not follow the metric time because of adversarial interruptions. Parties construct their clocks according to their local timers. In such an environment, we model how parties achieve consensus on a clock. Our model consists of the following functionalities:

Functionality $\mathcal{G}_{\mathsf{refRate}}$

$\mathcal{G}_{\mathsf{refRate}}$ has a list of registered parties $\mathsf{ListRate}$ which is initially empty.

- (Registration) If a functionality $\mathcal{F}$ sends a message $(\mathsf{Register}, sid, \mathcal{F})$, $\mathcal{G}_{\mathsf{refRate}}$ adds $\mathcal{F}$ to $\mathsf{ListRate}$ and sends $(\mathsf{Registered}, sid, \mathcal{F})$ to P.
- (Progress in Time) In every tick according to the defined rate, $\mathcal{G}_{\mathsf{refRate}}$ sends (Tick, sid', P) to all $P \in \mathsf{ListRate}$

Figure 1: The global functionality $\mathcal{G}_{\mathsf{refRate}}$

3.1.1 Reference Rate ($\mathcal{G}_{\mathsf{refRate}}$)

This functionality defines a metric time for timers. More precisely, it can be considered as a global timer that ticks with respect to the metric time (e.g., it ticks every second.). The entities who want to be notified in every tick register with $\mathcal{G}_{\mathsf{refRate}}$. Whenever $\mathcal{G}_{\mathsf{refRate}}$ ticks, it informs them. The difference between $\mathcal{G}_{\mathsf{refClock}}$ [12] and $\mathcal{G}_{\mathsf{refRate}}$ is that $\mathcal{G}_{\mathsf{refRate}}$ does not have any absolute values related to time (e.g., it does not keep how many ticks it had). We note that ITI's *cannot* contact with $\mathcal{G}_{\mathsf{refRate}}$. The details are in Figure 1.

One may question whether the concept of a reference rate contradicts the idea of a trustless distributed system. It does not, because metric system for time is universally defined and unchangeable (e.g., "One second is the time that elapses during 9.192631770×10^9 cycles of the radiation produced by the transition between two levels of the cesium 133 atom [2]").

We note that we do not aim to construct protocols that precisely follow this global rate. This functionality is intended to be useful in the notion of relative time.

3.1.2 Local timer ($\mathcal{F}_{\mathsf{timer}}^{\Sigma,P}$)

The functionality $\mathcal{F}_{\mathsf{timer}}^{\Sigma,P}$ represents a local timer of a party P. $\Sigma \in \mathbb{Z}$ represents how much the local timer has drifted from the correct relative time (e.g. 1 hour passed according to $\mathcal{G}_{\mathsf{refRate}}$ since the initialization of P, but the local timer indicates that 58 minutes have passed. In this case, the total drift is -2 minutes). $\mathcal{F}_{\mathsf{timer}}^{\Sigma,P}$ is accessible by the party P without any delay. $\mathcal{F}_{\mathsf{timer}}^{\Sigma,P}$ stores two types of timer: timer and timer^*. It increments timer whenever it receives a message from $\mathcal{Z}$ and increments timer^* whenever it receives a signal from $\mathcal{G}_{\mathsf{refRate}}$. However, it never shares timer^* (the real timer) with the party. The reason of having timer^* is to allow us to calculate the amount of total drift (Σ) at any given time. We define it in Figure 2.

Functionality $\mathcal{F}_{\mathsf{timer}}^{\Sigma,P}$

$\mathcal{Z}$ initiates $\mathcal{F}_{\mathsf{timer}}^{\Sigma,P}$ with a value $t \in \mathbb{Z}$. Then, $\mathcal{F}_{\mathsf{timer}}^{\Sigma,P}$ creates two parameters $\mathsf{timer} = t$ and $\mathsf{timer}^* = t$.

- (Registration) $\mathcal{F}_{\mathsf{timer}}^{\Sigma,P}$ sends $(\mathsf{Register}, sid, \mathcal{F}_{\mathsf{timer}}^{\Sigma,P})$ and receives back $(\mathsf{Registered}, sid, \mathcal{F}_{\mathsf{timer}}^{\Sigma,P})$. This is done once after the initialization of the functionality.
- (Increasing timer) When $\mathcal{Z}$ sends $(\mathsf{Increase}, sid, P)$, $\mathcal{F}_{\mathsf{timer}}^{\Sigma,P}$ increments timer and lets $\Sigma = \mathsf{timer} - \mathsf{timer}^*$.
- (Increasing timer^*) When $\mathcal{G}_{\mathsf{refRate}}$ sends $(\mathsf{Tick}, sid, \mathcal{F}_{\mathsf{timer}}^{\Sigma,P})$, $\mathcal{F}_{\mathsf{timer}}^{\Sigma,P}$ increments timer^* and lets $\Sigma = \mathsf{timer} - \mathsf{timer}^*$.
- If P sends $(\mathsf{Get\_Timer}, sid, P)$, $\mathcal{F}_{\mathsf{timer}}^{\Sigma,P}$ sends the message $(\mathsf{Timer}, sid, \mathsf{timer}, P)$.

Figure 2: The global functionality $\mathcal{F}_{\mathsf{timer}}^{\Sigma,P}$

We note that $\mathcal{F}_{\mathsf{timer}}^{\Sigma,P}$ does not differ from real-world computer timers because timer^* (the real clock) is never shared with the parties and the execution of the local timer does not depend on it. Therefore, a computer timer in the real world can be proven as a realization of $\mathcal{F}_{\mathsf{timer}}^{\Sigma,P}$.

Definition 3.1 (Correspondence of Timer Values)**.** *Assume that P_i and P_j sends $(\mathsf{Get\_Timer}, sid, P_i)$ to $\mathcal{F}_{\mathsf{timer}}^{T,P_i}$ and $(\mathsf{Get\_Timer}, sid, P_j)$ to $\mathcal{F}_{\mathsf{timer}}^{T,P_j}$ at the same time, respectively and $\mathcal{F}_{\mathsf{timer}}^{T,P_i}$ responds with $\mathsf{timer}_i = t_i$ and $\mathcal{F}_{\mathsf{timer}}^{T,P_j}$ responds with $\mathsf{timer}_j = t_j$. We call t_i is the corresponding timer value of t_j in timer_j in that moment (the moment that parties asked) and similarly we call that t_j is the corresponding timer value of t_i in timer_j in that moment.*

We denote by an algorithm $\mathsf{map}(\mathsf{timer}_i, \mathsf{timer}_j, t_i)$ the corresponding value of t_i in timer_j.

Before defining our consensus clock functionality, we first give some definitions to define clocks.

Definition 3.2 (Clock Value). *A clock value is some natural number. Given that the initial clock value which is 0 is matched with* $\mathsf{timer} = t^*$, *we define the clock value when* $\mathsf{timer} = t \geq t^*$ *for all* timer *of* $\mathcal{F}_{\mathsf{timer}}^{\Sigma,P}$ *of any* P *as:*

$$\mathsf{Clock}(t, T, t^*) = \lfloor \frac{t - t_{init}}{T} \rfloor. \tag{1}$$

where $T \in \mathbb{N}$ *is the measure of time interval between two consequent clock values.*

t^* serves as a local reference point to obtain a clock (defined below) that lets a clock determine its clock value at a given timer (i.e., after t^*, for every T increment in timer, increase the clock value).

Definition 3.3 (Clock). *A clock of* P *is a counter that keeps the clock value based on progression on* timer *of* $\mathcal{F}_{\mathsf{timer}}^{\Sigma,P}$ *after an initial assignment at* t^*. *The clock* $\mathcal{C}$ *of a party when* $\mathsf{timer} = t_{curr}$ *is defined with* $[t^*, (t_{curr}, c_{curr})]$ *where* $c_{curr} = \mathsf{Clock}(t_{curr}, T, t^*)$[1].

$\mathcal{C}$ represents the clock at t_{curr} while c represents the clock value, which is any output from Equation (1).

Definition 3.4 (Time Difference of Clocks). *Given that two clocks* $\mathcal{C}_i = [t_i^*, (c_i, t_j)]$ *and* $\mathcal{C}_j = [t_j^*, (c_j, t_j)]$ *at the same time that increments according to* timer_i *and* timer_j, *respectively, we define the time difference of clock* $\mathcal{C}_i$ *and* $\mathcal{C}_j$ $(\mathcal{C}_i - \mathcal{C}_j)$ *as the elapsed time between the first time* $\mathcal{C}_j$*'s clock value is changed to* c *and the first time* $\mathcal{C}_j$*'s clock value is changed to* c *where* $c \in \mathbb{N}$.

More formally, $\mathcal{C}_i - \mathcal{C}_j = |t_j^* - t_{i_j}^*| = |t_i^* - t_{j_i}^*|$ *where* $t_{i_j}^* = |t_j - (t_i - t_i^*|)$ *is the corresponding timer value of* t_i^* *in* timer_j *and* $t_{j_i}^* = |t_i - (t_j - t_j^*|)$ *is the corresponding timer value of* t_j^* *in* timer_i.

The time difference of two clocks can be visualized as in Figure 3.1.2.

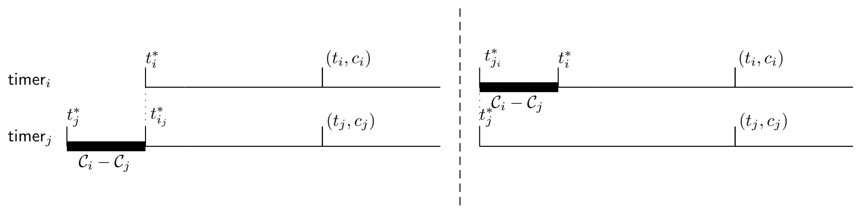

Figure 3: Time difference of clocks: $|\mathcal{C}_i - \mathcal{C}_j|$. It can be computed on the timer of $\mathcal{C}_j$ (left-hand side) or on the timer of $\mathcal{C}_i$ (right-hand side).

We note that the time difference between clocks does not have to be constant, because the timers can drift backward or forward. Therefore, we define the difference between clocks instantaneously.

3.1.3 Consensus Clock Provider ($\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$)

$\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$ has a similar role with $\mathcal{G}_{\mathsf{Clock}}$ in the network time model [12]. Differently, in our functionality, the new clock is determined based on a consensus algorithm `Clock_Consensus` which outputs the current agreed clock or $\perp$ given a set of clocks provided by parties.

[1]It is redundant to have c_{curr} in a clock $[t^*, (t_{curr}, c_{curr})]$ but we have it there for the sake of clarity.

Functionality $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$

$\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$ is parameterized with T and $\Theta = (\Theta_c, \Theta_p)$ given by $\mathcal{Z}$. $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$ first sends $(\mathsf{Register}, sid, \mathcal{F}_{\mathsf{Clock}}^{T,\Theta})$ and receives back $(\mathsf{Registered}, sid, P)$. Thus, it constructs its own timer.

- **(Initial Consensus)** Each party $P_i \in \mathcal{P}_h$ sends the message $(\mathsf{Get\_Clock}, sid, P_j, \mathcal{C}_j)$ where $\mathcal{C}_j$ is the clock of the party P_j. $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$ runs the algorithm $\mathtt{Clock\_Consensus}(\{\mathcal{C}_j\}_{P_j \in \mathcal{P}_h})$. If $\mathtt{Clock\_Consensus}$ outputs $\perp$, then $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$ sends $(\mathsf{Current\_Clock}, sid, \perp)$ to each P_j and aborts the protocol. If it outputs $\bar{\mathcal{C}}_h$, it sends $\{(\mathsf{Get\_Clock}, sid, P_j, \mathcal{C}_j)\}_{P_j \in \mathcal{P}_h}$ to $\mathcal{A}$.
- **(New Clocks)** $\mathcal{A}$ replies with $(\mathsf{Init\_Clock}, sid, \mathcal{F}_{\mathsf{Clock}}^{T,\Theta}, \tilde{C}_1, \tilde{C}_2, ..., \tilde{C}_n)$ for all $P_i \in \mathcal{P}_h$. If there exists $\tilde{C}_i = \mathsf{null}$, $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$ aborts.
- **(New Consensus)** $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$ runs the algorithm $\mathtt{Clock\_Consensus}(\{\mathcal{C}_j\}_{P_j \in \mathcal{P}_h \cup \mathcal{P}_c})$. If it does not output $\perp$ and $|\bar{\mathcal{C}} - \bar{\mathcal{C}}_h| \leq \Theta_c$ and $|\bar{\mathcal{C}} - \tilde{\mathcal{C}}_i| \leq \Theta_p$ for all $P_i \in \mathcal{P}_h$, it sends the $\tilde{\mathcal{C}}_i$ to each party $P_i \in \mathcal{P}_h$. Otherwise, it sends $\bar{\mathcal{C}}_h$ to each party $P_i \in \mathcal{P}_h$.

Figure 4: The functionality $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$

$\mathcal{G}_{\mathsf{Clock}}^{T,\Theta}$ is defined with the parameter T and Θ. T as in Equation (1) is the amount of time that defines the duration between one increment in the clock. $\Theta = (\Theta_c, \Theta_p)$ is the desynchronization parameter. $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$ provides agreed clock to a requester party based on a consensus.

We call $\mathcal{P}_h$ is the set of honest parties and $\mathcal{P}_c$ is the set of corrupted parties.

(Initial Consensus): At first, $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$ finds the initial consensus between honest parties. If it does not exist from the beginning, $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$ aborts the protocol. In more detail, $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$ first collects clocks from honest parties $P_i \in \mathcal{P}_h$. We note that some clocks can be null meaning that the honest party does not have any clock. $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$ runs the algorithm $\mathtt{Clock\_Consensus}$[2] with the honest clocks and obtains either $\perp$ or a consensus clock of honest parties $\bar{\mathcal{C}}_h$. If it is $\perp$, $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$ sends an abort message to the adversary $\mathcal{A}$ and the honest parties and the protocol ends. Otherwise, $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$ gives the honest clocks to the adversary $\mathcal{A}$.

(New Clocks:) After receiving clocks of honest parties, $\mathcal{A}$ gives new clocks for honest parties to $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$. If there exists a null clock among new clocks then $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$ aborts.

(New Consensus): At this point, $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$ checks if a new consensus is possible with the new clocks. If the new consensus clock exists and it is close enough to the initial one, then all parties continue with their new clocks provided by $\mathcal{A}$. In more detail, $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$ runs the algorithm $\mathtt{Clock\_Consensus}$ with the new honest clocks provided by the adversary. If $\mathtt{Clock\_Consensus}$ outputs $\perp$, $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$ sends $\bar{\mathcal{C}}_h$ (the initial consensus) to the honest parties. If it outputs a consensus clock $\bar{\mathcal{C}}$, $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$ continues as follows: If $|\bar{\mathcal{C}} - \bar{\mathcal{C}}_h| \leq \Theta_c$ and $|\bar{\mathcal{C}} - \tilde{\mathcal{C}}_i| \leq \Theta_p$ for all $P_i \in \mathcal{P}_h$, it sends the $\tilde{\mathcal{C}}_i$ to each party $P_i \in \mathcal{P}_h$. Otherwise, it sends $\bar{\mathcal{C}}_h$ to each party $P_i \in \mathcal{P}_h$. More details are in Figure 4.

In a nutshell, this functionality aims to provide clocks to honest parties which do not drift apart from the initial consensus that they have and which is close enough to their own clock.

[2] $\mathtt{Clock\_Consensus}$ can be defined based on the needs of the real-world protocol. e.g., the mostly agreed or minimum clock can be the consensus clock or the way that we define for our protocol in Section 4.

Having the difference limit between the initial consensus clock $\bar{\mathcal{C}_h}$ and the new consensus clock $\bar{\mathcal{C}}$ is useful not to slow down the protocols relying on the output of $\mathcal{F}^{T,\Theta}_{\mathsf{Clock}}$. For example, (a very extreme example) we do not want to end up with a new consensus clock that says that we are in the year 2001 when we were in 2019 according to the previous consensus. In this case, parties may wait 18 years to execute an action that is supposed to be done in 2019.

We may have a stronger version of this functionality where honest parties do not have the initial consensus clock and still obtain a consensus clock. Our protocol in the next section runs on top of a blockchain protocol which assumes initial parties have a consensus on clock. Therefore, we do not consider the stronger version.

3.2 UC- Partially Synchronous Blockchain Network Model

Our network model $\mathcal{F}_{\mathsf{DDiffuse}}$ is similar to the network model of Ouroboros Praos [15, 27, 20]. Differently, it accesses to $\mathcal{G}_{\mathsf{refRate}}$ in order to have the notion of relative time. Now, we define the functionality $\mathcal{F}_{\mathsf{DDiffuse}}$ which models a partially synchronous network with the time delay δ. Here, δ represents number of δ-increment message by $\mathcal{G}_{\mathsf{refRate}}$.
$\mathcal{F}^{\delta}_{\mathsf{DDiffuse}}$**:** The message handling functionality Diffuse for a blockchain network was first introduced by Garay et al. [20] in a synchronous network where message delivery is executed in a certain amount of known time. Then, David et al. [15] define a new functionality "delayed diffuse" (DDiffuse) for a blockchain network that realizes a partially synchronous network where a message arrives to others eventually, but parties do not know how long it takes. DDiffuse is parameterized with the network delay parameter Δ and ensures that all messages are received at most Δ-slots later. Here, a slot is the duration to produce one block. However, it does not use any clock to determine how many slots have passed since a message was sent. $\mathcal{F}^{\delta}_{\mathsf{DDiffuse}}$ is a version of DDiffuse which can access of $\mathcal{G}_{\mathsf{refRate}}$. It simply sends a given message from a party P_i to all other parties within a bound δ. $\mathcal{F}^{\delta}_{\mathsf{DDiffuse}}$ first registers to $\mathcal{G}_{\mathsf{refRate}}$ and creates a local timer $\mathsf{timer} = 0$. Whenever $\mathcal{G}_{\mathsf{refRate}}$ sends a message with Increase, it increments timer.

Each honest party P_i can access its inbox anytime. $\mathcal{A}$ can read all messages sent by the parties and decide their delivery order before they arrive to inboxes of honest parties. For any message coming from an honest party, $\mathcal{A}$ can label it as $\mathsf{delayed}_i$. When $\mathcal{F}_{\mathsf{DDiffuse}}$ receives $\mathsf{delayed}_i$ for a message to P_i, it marks it with the current local timer value t_i. A delayed message is not moved to the inbox of P_i until $\mathcal{A}$ lets $\mathcal{F}^{\delta}_{\mathsf{DDiffuse}}$ move it or the timer reaches $t_i + \delta$. In the end, all parties always receive any message sent by a party.

4 Realization of Consensus Clock

In this section, we describe our relative time protocol for blockchain protocols that realizes the functionality $\mathcal{F}^{T,\Theta}_{\mathsf{Clock}}$. Before describing the protocol, we first define the algorithm `Clock_Consensus` (Algorithm 4.1) that we designed for the relative time protocol. We note that this is our way of defining consensus on clocks while every protocol can have a different `Clock_Consensus` algorithm. After defining the algorithm, we describe our relative time protocol that lets parties agree on a clock according to `Clock_Consensus`.

4.1 Clock Consensus

The `Clock_Consensus` algorithm receives clocks of multiple parties as input, and outputs one of them as a consensus clock. We say that two clocks are *synchronous* if they output the same clock at any time between a consecutive κ ticks by $\mathcal{G}_{\mathsf{refRate}}$. For example, assume that κ is 5 seconds. A clock that starts to output the clock value c in the first second, and another clock that starts to output c in the fourth second, are still considered synchronized, even though they do not output the same clock value in the sixth second (i.e., one outputs $c+1$ and the other outputs c in the sixth second). We say that clocks are synchronized if all pairs of these clocks are synchronized.

Given clocks $\mathcal{C}_j = [t_j^*, (c_j, t_j)]$, `Clock_Consensus` finds the corresponding time values $\hat{t}_j^c$ (Definition 3.1) of $t_j^c = t_j^* + cT$ on $\mathsf{timer}_{\mathcal{F}_{\mathsf{Clock}}}$ of $\mathcal{F}_{\mathsf{Clock}}^{T,\Theta}$ and checks whether the differences of $\hat{t}_j^c$'s that are retrieved from clocks of parties are less than κ to check if clocks are synchronized. If clocks are not synchronized, then the algorithm outputs $\perp$ meaning that no consensus exists. Otherwise, the clock of a party that corresponds to the median of all $\hat{t}_j^c$'s is selected as a consensus clock. We give the algorithm of `Clock_Consensus` in Algorithm 1.

Algorithm 1 `Clock_Consensus`$(\{\mathcal{C}_j\})$ where $\mathcal{C}_j = [t_j^*, (c_j, t_j)]$

```
1:  $\mathsf{start\_lst} = \emptyset$
2:  **pick** $c \in \mathbb{N}$ **such that** for all $j$ such that $c > c_j$
3:  **for all** $\mathcal{C}_j \neq null$ **do**
4:      $t_j^c \leftarrow t_j^* + cT$
5:      $\hat{t}_j^c \leftarrow \mathsf{map}(\mathsf{timer}_j, \mathsf{timer}_{\mathcal{F}_{\mathsf{Clock}}}, t_j^c)$
6:      **add** $\hat{t}_j^c$ **to** $\mathsf{start\_lst}$
7:  **if** $|\mathsf{start\_lst}| > 0$ **then**
8:      **for all** $\hat{t}_u^c, \hat{t}_v^c \in \mathsf{start\_lst}$ **do**
9:          **if** $|\hat{t}_v^c - \hat{t}_u^c| > \kappa$ **then**
10:             **return** $\perp$
11:     $\hat{t}_i^c \leftarrow$ Median(start_lst) // Median(start_lst) sorts the list and returns the median
12:     **return** $\mathcal{C}_i$
13: **else**
14:     **return** $\perp$
```

4.2 Relative Time Protocol

The relative time protocol realizes $\mathcal{F}_{\mathsf{Clock}}^{T,\theta}$ (See Figure 4) meaning that it let's parties to obtain a clock which is close enough to the consensus clock. We build it on top of a blockchain protocol where parties produce blocks when it is their turn. In more detail, the blockchain protocol is defined as follows: After the genesis block is released, each party starts their local timer. In every T tick of their timer according to $\mathcal{F}_{\mathsf{timer}}^{\Sigma,P}$ (See Figure 2), they increment their clock which is initially 0 when they receive the genesis block. The blockchain protocol has a selection mechanism which tells parties at which clock value they are supposed to produce a block. Selected parties produce blocks only when their clock reaches the right clock value. Whenever they produce a block they also add their current clock value to the block as a timestamp. The environment $\mathcal{Z}$ activates parties just before the genesis block. It then can stop these parties or add new parties. We believe that Ouroboros [27], Ouroboros Praos [15], Ouroboros Genesis [6], Dfinity [22] and Snow White [14] can all easily fit into this abstraction.

According to our abstraction, all initial active parties when the genesis block is released have clocks that differ by at most δ (network delay bound). However, after a while, the cumulative drift in their local clock (Σ in $\mathcal{F}_{\mathsf{timer}}^{\Sigma,P}$) may change and so the difference between clocks can increase. In addition, new parties which are activated after the genesis block need to another way to initiate their clock. Therefore, all parties in such a blockchain protocol need to run the relative time protocol to obtain a clock which is close enough to the consensus clock. Our protocol does not provide perfect synchronization, but it does preserve a maximum difference between the consensus clock and the clock that the algorithm offers to the parties during the execution of the blockchain protocol. The relative time protocol works as follows:

We divide the protocol into epochs. In each epoch, all active parties run the relative time protocol and update their clocks according to output of the protocol in the beginning of the next epoch. The first epoch starts just after the genesis block is released. The other epochs start when the clock value of the last finalized block is c_e which is the smallest clock value such that $c_e - c_{e-1} \geq s_{cd}$ where c_{e-1} is the clock value of the last finalized block in epoch $e-1$. Here, s_{cd} is the parameter of the chain density (CD) property (Definition 2.4). If the previous epoch is the first epoch then $c_{e-1} = 0$. We define the last finalized block as follows: Retrieve the best blockchain according to the chain selection rule of the blockchain protocol, then trim the last k blocks of the best chain, and the last block of the trimmed best chain is the last finalized block. Here, k is defined according to the common prefix property (Definition 2.1).

The party P constantly stores the arrival time of *valid* blocks according to the underlying blockchain protocol. Whenever it receives a new valid block B_i', it sends $(\mathsf{Get\_Timer}, sid, P)$ to $\mathcal{F}_{\mathsf{timer}}^{\Sigma,P}$ and obtains the arrival time t_i of B_i' according to its local timer. Let us denote the clock value of B_i' by c_i'. We note that the clock values in these blocks do not have to be in a certain order because desynchronized or malicious parties may not send their blocks on time. At the end of the epoch, P retrieves the arrival times of **valid and finalized blocks** which have a clock value c_x where $c_{e-1} < c_x \leq C_e$. Let us assume that there are n such blocks that belong to the current epoch. Then, P selects a clock value $c > c_e$ Then, P runs the median algorithm (Algorithm 2) which finds some candidate start times of c using the arrival time of blocks and then picks the median of them.

Algorithm 2 Median$(c, \{t_i, c_i'\}_{i=1}^{n})$

```
1: lst ← ∅
2: for $i = 0$ to $n$ do
3:     $a_i \leftarrow c - c_i'$
4:     store $(t_i + a_i T)$ to lst // start time of $c$ according to the party that produced $B_i'$
5: $lst$ ← sort(lst)
6: return median(lst)
```

Assume that t is the output of the median algorithm. Then, P considers t as a start time of the clock value c (i.e., a local reference clock (c, t)) and adjust its clock so that it shows c when its timer is t.

The security of our protocol is based on the security of the CP and CD properties. The CP property guarantees that all honest parties accept the same blocks as finalized blocks. Therefore, all honest parties run the median algorithm using the arrival time of the same blocks. Thus, since the network delay is at most δ, the difference between the median outputs of each honest party is also at most δ. Therefore, after each epoch, the difference between the clocks are at most δ as in the right after the genesis block is released. The difference between a new clock and an old clock of an honest party is limited thanks to the CD property that the blockchain protocol provides. The reason for this is that CD property guarantees that more than half of the blocks used in the median algorithm belong to honest parties. Thanks to a nice property of the median operation, the output of the median should be between the minimum and maximum honest of clocks. The formal proof is as follows:

Theorem 4.1. *Assuming that the blockchain protocol preserves the common prefix property with the parameter k, and the chain density property with the parameter s_{cd} as long as the maximum difference between honest clocks is $2\delta + 2|\Sigma|$ where $|\Sigma|$ is the maximum cumulative drift between epochs, $\Theta_c = 2\delta + |\Sigma|$ and $\Theta_p = \delta$, the relative time protocol in $\mathcal{F}^{\delta}_{\mathsf{DDiffuse}}$ and $\mathcal{F}^{T,\Sigma}_{\mathsf{timer}}$-hybrid model realizes $\mathcal{F}^{T,\Theta}_{\mathsf{Clock}}$ except with the probability $p_{cp}+p_{cd}$ which are the probability of breaking CP and CD properties, respectively.*

Proof. In order to prove the theorem, we construct a simulator $\mathcal{S}$ where $\mathcal{S}$ emulates $\mathcal{F}^{\delta}_{\mathsf{DDiffuse}}$ and $\mathcal{F}^{T,\Sigma}_{\mathsf{timer}}$. The simulation is straightforward. $\mathcal{S}$ simulates honest parties in the underlying blockchain protocol as well, based on the clocks given by $\mathcal{F}^{T,\Theta}_{\mathsf{Clock}}$. For this, $\mathcal{S}$ selects an epoch randomly which is less than the current epoch of the blockchain protocol and rewinds the adversary to the beginning of the epoch. Given that the clock value of the beginning of the epoch is c, $\mathcal{S}$ rewinds or forwards the clocks of honest parties to the value c of the blockchain protocol i.e., given $\mathcal{C}_i = [t^*_i, (c_i, t_i)]$, set $\mathcal{C}_i = [t^*_i, (c, t^*_i + cT)]$. Then, $\mathcal{S}$ sends $(\mathsf{Register}, sid, \mathcal{S})$ to $\mathcal{G}_{\mathsf{refRate}}$ to emulate $\mathcal{F}^{\Sigma,P_i}_{\mathsf{timer}}$ by setting $\mathsf{timer}_i = t^*_i + cT$ and behave the same as $\mathcal{F}^{\Sigma,P_i}_{\mathsf{timer}}$. After setting up the time and clocks, $\mathcal{S}$ starts to simulate each honest party P_i in the real protocol according to these clocks and $\mathcal{F}^{\Sigma,P_i}_{\mathsf{timer}}$. $\mathcal{S}$ produces a block on behalf of P_j if P_j is eligible to produce a block when the clock value is c according to underlying blockchain protocol. If P_j is eligible, $\mathcal{S}$ sends the block of P_j to $\mathcal{A}$ (since $\mathcal{S}$ emulates $\mathcal{F}^{\delta}_{\mathsf{DDiffuse}}$ too). If $\mathcal{A}$ moves the block to the inbox of other honest parties, $\mathcal{S}$ stores the time that the block moved to the inbox of honest parties as the arrival time of this block. If the block is delayed by $\mathcal{A}$, $\mathcal{S}$ waits until $\mathcal{A}$ permits the block to move it. If the permission is not received after δ consecutive ticks by $\mathcal{G}_{\mathsf{refRate}}$, $\mathcal{S}$ moves the block to the inbox of honest parties. In either case, it stores timer of $\mathcal{F}^{\Sigma,P_i}_{\mathsf{timer}}$ when a block arrives from any other party. Recall that $\mathcal{S}$ knows the duration of δ because it receives the exact rate from $\mathcal{G}_{\mathsf{refRate}}$ while simulating the local clocks. During the simulation, $\mathcal{S}$ learns the clocks of corrupted parties in the epoch since it simulates $\mathcal{F}^{P,\Sigma}_{\mathsf{timer}}$ for a corrupted party as well. At the end of the epoch, $\mathcal{S}$ runs the Median algorithm (Algorithm 2) and updates the clocks of honest parties accordingly. $\mathcal{S}$ sends the clocks of honest parties. Finally, $\mathcal{S}$ outputs the clocks of honest parties.

The output of an honest party in the real world and the honest party in the ideal world are not the same if

1. there is no consensus according to `Clock_Consensus` or
2. the difference between the initial consensus and the new consensus clock is more than Θ_c or
3. the difference between the final consensus clock and the new clock of an honest party is more than Θ_p.
4. at least one of the new clocks of honest parties is null.

Now, we analyze the probability of having such bad events in our simulation in any epoch.

(1. Case and 3. Case): According to our `Clock_Consensus` (Algorithm 1), the consensus on clocks exists if and only if the difference between honest clocks is at most $\Theta_p = \delta$. Therefore, if we show that a consensus on clocks exists after the update then we also show that the difference between clocks of honest parties and the new consensus clock is at most δ. We can then show that a consensus on clocks after the update exists given that **CP property is not broken during an epoch** except with probability p_{cp}. All honest parties run the median algorithm with the arrival time of the same blocks thanks to the CP property. $\mathcal{F}^{\delta}_{\mathsf{DDiffuse}}$ guarantees that a block arrives at all honest parties within δ-ticks. Therefore, the time difference in arrival time of any block differs at most δ between honest parties, as well. This implies that the time difference between the median of all honest parties' lst in Algorithm 2 can be at most δ. Thus, for all $P_i, P_j \in \mathcal{P}_h$, $|\mathcal{C}_j - \mathcal{C}_i| \leq \delta$ and so $|\bar{\mathcal{C}} - \mathcal{C}_i| \leq \delta = \Theta_p$. Now, we need to show that **the CP property is satisfied during all epochs** with induction. We know that at the beginning of the first epoch, the maximum difference between clocks of honest parties is δ because of our assumption after release of the genesis block. During the first epoch, the difference between the honest parties can be at most $2|\Sigma| + \delta$ because of clock drifts. Therefore, the CP property is preserved during the first epoch. Assume that the CP property is satisfied during the epoch x. Then, we show that the CP property is satisfied during the epoch $x + 1$. We know that if CP property is satisfied then the difference between clocks of honest parties is at most δ after running the median algorithm in the the end of the epoch x. So, honest parties start the epoch $x + 1$ with a clock which has difference δ at most. For the same reasons as of the first epoch, the CP property is satisfied during the epoch $x + 1$ as well.

(2. Case) We know that the clocks of honest parties before simulation starts have consensus since $\mathcal{F}^{T,\Theta}_{\mathsf{Clock}}$ gave it to them. Therefore, the simulation starts with the honest clock has difference at most δ. We know that the total drift of timers of honest parties during the simulation is at most $|\Sigma|$. Therefore, the clock difference of honest clocks can be at most $\delta + 2|\Sigma| \leq \Theta$ during the simulation. It is $2|\Sigma|$ because the drift can be forward or backward in the timeline. Therefore, the CD property is satisfied during an epoch. It means that majority of the blocks (at least $\lfloor \frac{n}{2} \rfloor + 1$ finalized blocks in the epoch) used in the median algorithm are honest ones except with the probability p_{cd}.

We now show the difference between the new consensus clock $\bar{\mathcal{C}}$ and the consensus clock $\bar{\mathcal{C}}_h$ just before the simulation starts is at most Θ_c assuming that $\lfloor \frac{n}{2} \rfloor + 1$ of the finalized blocks during the simulation were sent by honest parties. Let us assume that for an honest party P_u, the median algorithm outputted $\tilde{t} = t + a_i T$ where t is the arrival time of the block with clock value c according to P_u's timer. For the sake of clarity, all timer values are corresponding timer values on timer_u. If the block with the clock value c is sent by an honest party P_v, it is sent at t' which is the start of c according to $\mathcal{C}_v$. Because of $\mathcal{F}^{\delta}_{\mathsf{DDiffuse}}$, this block may be delayed before received by P_u. Therefore, $t' \leq t \leq t' + \delta$. The difference between the clock of P_u after updating its clock and the clock of P_v before updating its clock is $0 \leq \tilde{\mathcal{C}}_u - \mathcal{C}_v = t - t' \leq \delta$. Since $\bar{\mathcal{C}}$ is one of new clocks of honest parties $\tilde{\mathcal{C}}_i$'s and $\tilde{\mathcal{C}}_u - \mathcal{C}_v = t - t' \leq \delta$ for all $P_u \in \mathcal{P}_h$, $0 \leq \bar{\mathcal{C}} - \mathcal{C}_v \leq \delta$. We know that the difference between $\bar{\mathcal{C}}_h$ and the clock $\mathcal{C}_v$ is at the beginning of the epoch is at most δ and could be at most $\delta + |\Sigma|$ at the end. So, $0 \leq \mathcal{C}_v - \bar{\mathcal{C}}_h \leq \delta + \Sigma$. We know that $\bar{\mathcal{C}} - \mathcal{C}_h \leq 2\delta + |\Sigma|$ so $\bar{\mathcal{C}} - \mathcal{C}_h \leq 2\delta + \Sigma \leq \Theta_c$.

We can now show that the same inequality holds even if the median $\tilde{t}$ is computed from the clock value of an adversarial block. In this case, there exists a $t_x, t_y \in \mathsf{lst}$ (lst in Algorithm 2) where t_x and t_y are generated from clock values of honest blocks such that $t_x \leq \tilde{t} \leq t_y$ because at least $\lfloor \frac{n}{2} \rfloor + 1$ of the collected blocks were sent by honest parties. Since $\bar{\mathcal{C}} - \mathcal{C}_h \leq 2\delta + \Sigma \leq \Theta_c$

holds for all clocks between t_x and t_y, it should hold for adversarial $\tilde{t}$.

(4. Case): Since CD and CP property is preserved between epochs as shown in case 1, 2 and 3, there are finalized blocks between epochs so lst in Algorithm 2 is never empty, so new clocks are never null

□

5 Conclusion

In this paper, we proposed a generic synchronization protocol that works on top of a blockchain protocol. Our synchronization protocol takes advantage of a regular messaging process (e.g., blocks are sent regularly) to preserve consensus between honest parties' clocks. We also designed the first formal security model to capture the notion of consensus on clocks. Our security model is not specific to blockchain protocols. It can be used to show the existence of a consensus clock in arbitrary protocol. We proved that our protocol is secure in our new GUC security model.

References

[1] Proof of authority. .`https://github.com/paritytech/parity/wiki/Proof-of-Authority-Chains`.

[2] Second (s or sec).

[3] H. Aissaoua, M. Aliouat, A. Bounceur, and R. Euler. A distributed consensus-based clock synchronization protocol for wireless sensor networks. *Wireless Personal Communications*, 95(4):4579–4600, 2017.

[4] H. K. Alper. Ouroboros clepsydra: Ouroboros praos in the universally composable relative time model.

[5] G. Ateniese, I. Bonacina, A. Faonio, and N. Galesi. Proofs of space: When space is of the essence. In *International Conference on Security and Cryptography for Networks*, pages 538–557. Springer, 2014.

[6] C. Badertscher, P. Gaži, A. Kiayias, A. Russell, and V. Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 913–930. ACM, 2018.

[7] C. Badertscher, P. Gazi, A. Kiayias, A. Russell, and V. Zikas. Ouroboros chronos: Permissionless clock synchronization via proof-of-stake. Cryptology ePrint Archive, Report 2019/838, 2019. `https://eprint.iacr.org/2019/838`.

[8] L. Bicknell. NTP issues today. outages mailing list. `https://mailman.nanog.org/pipermail/nanog/2012-November/053449.html`.

[9] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. `https://eprint.iacr.org/2000/067`.

[10] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 2001 IEEE International Conference on Cluster Computing*, pages 136–145. IEEE, 2001.

[11] R. Canetti, Y. Dodis, R. Pass, and S. Walfish. Universally composable security with global setup. In *Theory of Cryptography Conference*, pages 61–85. Springer, 2007.

[12] R. Canetti, K. Hogan, A. Malhotra, and M. Varia. A universally composable treatment of network time. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 360–375. IEEE, 2017.

[13] J. Chen and S. Micali. Algorand. *arXiv preprint arXiv:1607.01341*, 2016.

[14] P. Daian, R. Pass, and E. Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proofs of stake. *Cryptology ePrint Archive*, 2017.

[15] B. David, P. Gaži, A. Kiayias, and A. Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 66–98. Springer, 2018.

[16] S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak. Proofs of space. In *Annual Cryptology Conference*, pages 585–605. Springer, 2015.

[17] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. *ACM SIGOPS Operating Systems Review*, 36(SI):147–163, 2002.

[18] B. Fisch. Tight proofs of space and replication. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 324–348. Springer, 2019.

[19] S. Ganeriwal, R. Kumar, and M. B. Srivastava. Timing-sync protocol for sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 138–149. ACM, 2003.

[20] J. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 281–310. Springer, 2015.

[21] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017.

[22] T. Hanke, M. Movahedi, and D. Williams. Dfinity technology overview series, consensus system. *arXiv preprint arXiv:1805.04548*, 2018.

[23] J. He, P. Cheng, L. Shi, J. Chen, and Y. Sun. Time synchronization in wsns: A maximum-value-based consensus approach. *IEEE Transactions on Automatic Control*, 59(3):660–675, 2013.

[24] T. E. Humphreys, B. M. Ledvina, M. L. Psiaki, B. W. O'Hanlon, and P. M. Kintner. Assessing the spoofing threat: Development of a portable gps civilian spoofer. In *Radionavigation laboratory conference proceedings*, 2008.

[25] R. G. Johnston and J. Warner. Think GPS cargo tracking= high security? think again. *Proceedings of Transport Security World*, 2003.

[26] J. Katz, U. Maurer, B. Tackmann, and V. Zikas. Universally composable synchronous computation. In *Theory of Cryptography Conference*, pages 477–498. Springer, 2013.

[27] A. Kiayias, A. Russell, B. David, and R. Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*, pages 357–388. Springer, 2017.

[28] C. Lenzen, P. Sommer, and R. Wattenhofer. Pulsesync: An efficient and scalable clock synchronization protocol. *IEEE/ACM Transactions on Networking (TON)*, 23(3):717–727, 2015.

[29] B. Luo, L. Cheng, and Y.-C. Wu. Fully distributed clock synchronization in wireless sensor networks under exponential delays. *Signal Processing*, 125:261–273, 2016.

[30] M. K. Maggs, S. G. O'keefe, and D. V. Thiel. Consensus clock synchronization for wireless sensor networks. *IEEE sensors Journal*, 12(6):2269–2277, 2012.

[31] A. Malhotra, I. E. Cohen, E. Brakke, and S. Goldberg. Attacking the network time protocol. In *NDSS*, 2016.

[32] M. Maróti, B. Kusy, G. Simon, and Á. Lédeczi. The flooding time synchronization protocol. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 39–49. ACM, 2004.

[33] D. L. Mills. Computer network time synchronization. In *Report Dagstuhl Seminar on Time Services Schloß Dagstuhl, March*, volume 11, page 332. Springer, 1997.

[34] P. Papadimitratos and A. Jovanovic. Gnss-based positioning: Attacks and countermeasures. In *MILCOM 2008-2008 IEEE Military Communications Conference*, pages 1–7. IEEE, 2008.

[35] R. Pass and E. Shi. The sleepy model of consensus. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 380–409. Springer, 2017.

[36] L. Schenato and F. Fiorentin. Average timesynch: A consensus-based protocol for clock synchronization in wireless sensor networks. *Automatica*, 47(9):1878–1886, 2011.

[37] J. Selvi. Breaking ssl using time synchronisation attacks. In *DEF CON Hacking Conference*, 2015.

[38] R. Solis, V. S. Borkar, and P. Kumar. A new distributed time synchronization protocol for multihop wireless networks. In *Proceedings of the 45th IEEE Conference on Decision and Control*, pages 2734–2739. IEEE, 2006.

[39] P. Sommer and R. Wattenhofer. Gradient clock synchronization in wireless sensor networks. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, pages 37–48. IEEE Computer Society, 2009.

[40] N. O. Tippenhauer, C. Pöpper, K. B. Rasmussen, and S. Capkun. On the requirements for successful GPS spoofing attacks. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 75–86. ACM, 2011.

[41] J. Warner and R. Johnston. A simple demonstration that the global positioning system (gps) is vulnerable to spoofing, j. of secur. *Adm*, (1-9), 2002.

[42] G. Werner-Allen, G. Tewari, A. Patel, M. Welsh, and R. Nagpal. Firefly-inspired sensor network synchronicity with realistic radio effects. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 142–153. ACM, 2005.

[43] G. Wood. Polkadot: Vision for a heterogeneous multi-chain framework. *White Paper*, 2016.