Network Time with a Consensus on Clock

Abstract. Decentralized protocols which require synchronous communication usually achieve it with the help of the time that computer clocks show. These clocks are mostly adjusted by *centralized* systems such as Network Time Protocol (NTP) because these adjustments are indispensable to reduce the effects of random drifts on clocks. On the other hand, an attack on these systems (which has happened in the past) can cause corruption of the protocols which rely on the time data that they provide to preserve synchronicity. So, we are facing the dilemma of relying on a *centralized* solution to adjust our timers or risking the security of our *decentralized* protocols. In this paper, we propose a Global Universal Composable (GUC) model for the physical clock synchronization problem in the decentralized systems by modelling the notion of consensus on clocks. Consensus on clocks is agreed upon considering the local clocks of all parties in a protocol which are possibly drifted. In this way, we model the functionality that e.g. NTP provides in a decentralized manner. In the end, we give a simple but useful protocol relying on a blockchain network that realizes our model. Our protocol can be used by the full nodes of a blockchain that need synchronous clocks in the real world 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

In physics, time is defined as 'what a clock measures'. Nowadays, time is usually measurement of the number of vibrations happening on crystal oscillators in a clock e.g., a crystal oscillator vibrate 32768 times per second. So, the frequency of vibrations directly affects a computer's understanding of time. Unfortunately, the frequency of vibrations varies over time with small changes in the environment such as temperature, pressure, humidity thus causing clock drifts. As a result of this, the time knowledge of a clock starts to be inconsistent with the rest of the world. The computer clocks that are connected to the Internet mostly use Network Time Protocol (NTP) to determine the time instead of just relying on crystal oscillators so that the cumulative drift on the computer clock stays close to zero. In this way, not only clocks measure the time correctly but also the protocols such as certificate validation which require synchronized clocks can be executed securely.

Some decentralized protocols such as proof of stake (PoS) based blockchain protocols [28,15,5,6,14] are secure in the synchronous communication model [10,27,29] i.e., round based communication which all parties proceed next synchronously. Parties part of a decentralized protocol usually realize this communication by measuring time with their local clocks in the real world implementations e.g., They start a new round every T ticks of the local clock corrected by

NTP. So, the security of the such *decentralized* protocols reduce to the NTP's security which relies on the security of *centralized* servers. Unfortunately, the safety track record of NTP servers is not clean [33,39,35]. In one incident (on November 19, 2012) [7], two critical NTP servers were set to time twelve years in the past. This caused many important external servers, such as Active Directory authentication servers, to fail for a while. If the same attack happens for example in a PoS blockchain protocol even in a short time, honest parties would stop producing blocks because they think that their round did not come, while malicious full nodes would continue to produce blocks, populating the blockchain entirely with maliciously-produced blocks. Apart from NTP, some clocks are synchronized with the Global Positioning System (GPS). This requires a little more investment in the setup but it is more accurate than NTP and does not have the potential problem of corrupted servers. However, GPS is also vulnerable to spoofing attacks [42,25,36,24,43] (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 an external system to build a correct local clock in decentralized protocols such as blockchains could be a major security vulnerability, as well as going against the ultimate goal of building a fully decentralized system.

In this paper, we model the problem of synchronization of (physical) clocks in a decentralized manner in the Global Universal Composable (GUC) model. In our model, parties have local timers (modelling crystal oscillators) and build local clocks that advances in every certain number of ticks of the timer. The environment can modify the frequency of ticks of timers. In such a model, clearly, the local clocks may drift apart even if they were the same at some point. Therefore, we constructed another functionality called consensus clock which realigns them. The consensus clock functionality receives the local clocks of parties in a protocol as an input. Then, it gives new local clocks to all parties which are close to the consensus clock constructed with respect to the input clocks. Thus, the consensus clock functionality provides a way to decide the current right clock to be aligned in a decentralized network. We have two similar notions timer and clock to be able to represent the difference between two different local clocks with a smaller time unit. Thus, we can define synchronization between local clocks more precisely. We note that our aim *is not* designing a new model for the synchronous communication which has elaborate UC-security models [10,27,29]. We deal with physical clocks which have an ability to measure time while global clock functionalities in the synchronous communication models are logical clocks which proceed based on some events in the protocol independent from actual time.

The closest model for our purposes is the GUC network time model designed by Canetti et al. [12], that models the physical clock adjustment process of a client with corrupted servers. In this model, the goal of a client is to synchronize his local clock close enough to a reference clock incremented by the environment. Canetti et al. [12] designed this model to capture the functionality of NTP. If parties in a decentralized protocol advances a round of the protocol locally according to their local clocks in the network time model [12], the security and

the decentralized manner of the protocol can be violated because of the reasons that we discussed above. Let us summarize them in the network time model with three points:

1. The environment can increment the reference clock too many times between two local clock-synchronization processes with the reference clock so that the local clock of an honest party skips the round. This may cause the exclusion of some honest parties from the protocol.
2. The environment can increment the reference clock too fast so that the protocol collapses since the protocol may not have enough time to execute the specific instructions. For example, this is the case for the protocols [28,15,5,14] where the correctness of rounds depends on the correct execution of previous rounds.
3. How to define the reference clock in the real world is not very clear in decentralized protocols. It cannot be the clock provided by NTP or GPS because it is a centralized solution. It cannot be the clock of one of the parties since it gives too much power to modify the clocks of all other nodes. We cannot have an authority that defines them because we face the same problems that we have in NTP.

These reasons let us consider a new and more appropriate model in the nature of decentralized networks. We note that the issues in 1 and 2 come from the strong adversarial model that Canetti et al. [12] consider in their security model. Therefore, one possible solution could have been to consider a weaker adversarial model in which we do not let the environment modify the reference clock. However, this assumption would not be realistic because the concept of having **one** reference clock implies centralization as pointed in 3. Therefore, instead of a reference clock, we consider a consensus clock notion where the adversarial effect is limited since it is generated with respect to all local clocks. 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 [32,38,23,3,44,40,31]. However, there is currently no security model capturing this except ours.

In more detail, our contributions are as follows:

- We construct a GUC model that captures the notion of consensus on clock and allows parties in a decentralized network to align their clock with it. Our model realizes situations where a party has a local clock constructed based on the ticks of his local timer and wants to synchronize the local clock with the current consensus clock in the protocol. We define a global functionality which sets 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 model is the first security model for the notion of consensus on clock.
- We construct the *Relative Time* protocol on top of blockchain protocols that realizes consensus on clock functionality in our model. The parties construct

local clocks as in our model that count the rounds of the blockchain protocol and behave based on the round information provided by their local clocks. Our protocol ensures that honest clocks preserve closeness to each other by staying close to the consensus clock during the protocol even if local timers drift or network delay exists. We define the consensus clock based on the clock information in the blocks which are in the finalized (consensus) blockchain. Periodically, all parties update their clocks with the time data in these blocks not to be drifted apart. Our protocol can be adapted to all blockchain protocols that fit our abstraction. It does not add any extra communication to the network or any extra weight to the blocks. Thus, we solve the physical clock synchronization problems of PoS-based blockchain protocols in a decentralized manner without putting an extra overhead to the network and the blockchain.

We note that our relative time protocol is currently used in Polkadot [45] which aims to connect multiple blockchain networks.

1.1 Related Works

UC Clock Models: There are UC models [10,27,29] designed to emulate the synchronous communication between parties but they do not provide ways for realizing the functionalities from real-world solutions such as network time protocols or local physical clocks. The clock functionalities in these models are different from physical clocks. They keep the round of a protocol and make sure that all parties are in the same or close rounds which is decent to model the synchronous communication. We note that we do not aim to construct a model for synchronous communication even though there are some name resemblance among clock notions. The difference between clocks in our model and the clock functionality in [10,27,29] are as follows:

- Each local clock represents the round view of a party which does not need to be the same with other parties. Beyond this, we consider local clocks as a mechanism that measures the time while clock functionality [10,27,29] does not need to have this ability.
- Consensus-clock functionality in our model and clock functionalities in [10,27,29] serve completely different purposes. Ours should be imagined as a mechanism which does not let the local clocks drift apart too much. It does not keep the round of the protocol in a systematic way as clock functionalities in [10,27,29].

We believe that precise enough local clocks with the help of the consensus-clock functionality could be used to realize synchronous communication in these models.

The network time model by Canetti et al. [12] is the first model that defines clocks with the ability to measure time. In this model, a party can access immediately to his local clock which can be arbitrarily shifted. The aim of a party is to minimize these shifts by adjusting it with a reference clock that keeps a

counter incremented by the environment. The security of a local clock is defined based upon the closeness of a reference clock. So, the ultimate goal in this model is to obtain local clocks with a time close enough to the reference clock. This type of clock model is useful for the protocols accepting one clock as valid and expect from all other parties to follow this clock. For example, the Public Key Infrastructure (PKI) limits the validity of certificates and expects from all users to consider the validity of a certificate within the same duration. Differently, in our model, the ultimate goal of parties is to have clocks that are close to a consensus clock which is constructed with respect to all existing local clocks. 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 (e.g., blockchain 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 [12] where the reference clock is replaced with the consensus clock. We note that our model is not practical for example for certificate validation in PKI since it requires knowing all clocks of users which is millions in the PKI case to construct the consensus clock.

PoS protocols: The security of some PoS blockchain protocols [28,15,5,14] is preserved in the synchronous communication model. Ouroboros Genesis [5] is built on top of the clock functionality [29] that counts the rounds to preserve synchronous communication. Since in the real world, there is not any such clock functionality, the parties try to realize it with their local clocks that starts a new round when a certain time passed from the last round. Therefore, it is important whether the required synchronous communication is preserved with these local clocks. If they use the local clocks without any adjustment mechanism, the synchronous communication cannot be satisfied because of the unavoidable drifts. If the drifted local clocks are adjusted by NTP, the synchronous communication may not be preserved because of the vulnerabilities of NTP [33,39,35] as we pointed.

Ouroboros Chronos [6] which is an improved version of Ouroboros Genesis adds a mechanism that helps adjusting the physical clocks of parties without relying on any external Internet service such as NTP as our relative time protocol. In this way, loosely synchronous communication in the UC model [6,29] is preserved in Ouroboros Chronos. In order for this, the parties who are selected to produce block in the current round send a synchronization beacon along with the block of the round and others store the arrival time of the beacons. At some point, these beacons are added to a block to agree on which beacons are going to be used for the adjustment. Periodically, parties sort the beacons based on the difference from their own local clocks and arrival times. At the end, they adjust their clocks with the median one. Parties in our relative time protocol also adjust their local clock according to the median clock but in our case it is achieved without using extra messages such as beacons or adding new type of blocks for synchronization. The critical issue in both Ouroboros Chronos and ours is to make sure that all parties find the median with the arrival time of the same messages which are beacons in Chronos and blocks in our protocol. While

Chronos achieves this by deciding them on the blockchain, we use already the existing consensus mechanism of a PoS protocol. In every synchronization period, parties in our protocol obtain the median clock by considering the arrival times of the blocks in the final (consensus) chain after the previous synchronization. We guarantee that the median clock does not dramatically change the local clock of honest parties because we specify the length of one synchronization period sufficiently long so that the final chain includes more honest blocks than malicious blocks (the blocks that are not sent on time). This could be satisfied as long as the chain quality is preserved [28]. The synchronization mechanism in Ouroboros Chronos is tailored for Ouroboros Genesis while our protocol is more generic in this sense which can be constructed on top of coherent PoS protocols with our abstraction. As such, our protocol does not modify the block generation mechanism of the underlying PoS protocol and does not introduce any extra messages in the network. The length of the synchronization period of our protocol depends on the underlying PoS protocol. For example, if our protocol is constructed on top of Ouroboros Genesis [5], it may require a longer synchronization period than Ouroboros Chronos. This is because of the distribution of honest rounds in Ouroboros Genesis [5].

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 consensus blocks and adjust their local clock according to the round inside the last consensus blocks. We note that this is not possible in PoS protocols where the consensus mechanism is probabilistic [28,15,5,14,22] because a party cannot know whether the round in a block is correct since it is not finalized right away as in Algorand.

Synchronization Protocols: Clock synchronization protocols [17,19,34,41,30] have been extensively studied in previous work, as it is a important component in distributed systems. Many protocols [32,38,23,3] 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 [44,40] based on clock skew and network delay estimation. Even though many works exist related to consensus clock, to best of our knowledge, there is 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 $n \geq N$

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

$X \approx Y$ means that two ensembles X and Y are computationally indistinguishable.

2.1 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 1. *(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 Security Model

In this section, we first introduce our new GUC-model for the consensus clock, and then give the UC-model for a partially synchronous network similar to the model in [15,5]. We note that we refer physical clocks that measure the time in our model. They are not logical clocks.

We have multiple ITMs that we call as a party P_i. $\mathcal{Z}$ activates all parties and the adversary and then 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 $\mathcal{A}$.

3.1 The Model for the Consensus of Clocks

We construct a 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 synchronize their clocks without having a global 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 $\mathcal{F}$.
- (Progress in Time) In every tick according to the defined rate, $\mathcal{G}_{\mathsf{refRate}}$ sends (Tick, sid', P) to all $P \in \mathsf{ListRate}$

Fig. 1. The global functionality $\mathcal{G}_{\mathsf{refRate}}$

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 trustless distributed system without a global clock. It does not, because a 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]"). In real world, all physical clocks are designed to follow a global definition of time although they may have difficulties to follow it at some point because of their nature.

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

Local timer ($\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$) We define the global functionality $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ to model the local clock of a computer. We do not directly model it, instead we define a local timer that ticks with a rate which may vary in time (as crystal oscillators of clocks in the real world). We choose the name timer because its absolute values does not matter as a clock. The local clocks measure how much time passed with the help of these timers in our model.

In more details, $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ represents a local timer of a party P. Σ is an array which holds the drift from the real time for each timer value. $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ is accessible by the party P without any delay. $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ stores two types of timer: timer and timer^*. It increments local timer timer whenever it receives a message from $\mathcal{Z}$ and increments the real timer 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 stored in Σ at each timer value. The reason that we define $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ in GUC model is to capture the fact that the real world timers interact with arbitrary protocols.We define it in Figure 2.

We note that $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ does not differ from real-world computer timers because timer^* (the correct measurement of time) or Σ is never shared with its party and the execution of the local timer does not depend on it. Therefore, a computer timer in the real world can be shown as a realization of $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$. It shares Σ with $\mathcal{F}_{\mathsf{C\_Clock}}$ so that $\mathcal{F}_{\mathsf{C\_Clock}}$ can compute clock difference (Definition 6).

Next, we define notions related to local clocks which are constructed with respect to local timers. These clocks can be used to count locally the round of a protocol which progresses depending on time.

Definition 2 (Correspondence of Timer Values). *Assume that P_i and P_j sends $(\mathsf{Get\_Timer}, sid, P_i)$ to $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P_i}$ and $(\mathsf{Get\_Timer}, sid, P_j)$ to $\mathcal{G}_{\mathsf{timer}}^{T,P_j}$ at the same time, respectively and $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P_i}$ responds with $\mathsf{timer}_i = t$ and $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P_j}$ responds with $\mathsf{timer}_j = \hat{t}$. In this moment, the corresponding value of any t_i on timer_i is $\hat{t} + (t_i - t)$ on timer_j and the corresponding value of any $\hat{t}_j$ on timer_j is $t + (\hat{t}_j - \hat{t})$ on timer_j.*

Correspondence of timer values help us to define the difference of time values as below. As it can be seen from the definition, the correspondence of timer values may change in time because of the drifts on timers. Therefore, we define it for the moment when $\mathsf{timer}_i = t$ and $\mathsf{timer}_j = \hat{t}$.

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

$\mathcal{Z}$ initiates $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ with a value $t \in \mathbb{Z}$ and empty array Σ. Then, $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ creates two parameters $\mathsf{timer} = t$ and $\mathsf{timer}^* = t$ and lets $\Sigma[t] = 0$.

- (Registration to $\mathcal{G}_{\mathsf{refRate}}$) $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ sends $(\mathsf{Register}, sid, \mathcal{G}_{\mathsf{timer}}^{\Sigma,P})$ to $\mathcal{G}_{\mathsf{refRate}}$ and receives back $(\mathsf{Registered}, sid, \mathcal{G}_{\mathsf{timer}}^{\Sigma,P})$. This is done once after the initialization of the functionality.
- (Registration) If $\mathcal{F}_{\mathsf{C\_Clock}}$ (See Figure 4) sends a message $(\mathsf{Register}, sid, \mathcal{F}_{\mathsf{C\_Clock}})$, $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ replies with $(\mathsf{Registered}, sid, \mathcal{F}_{\mathsf{C\_Clock}})$.
- (Increasing timer) When $\mathcal{Z}$ sends $(\mathsf{Increase}, sid, P)$, $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ increments timer and lets $\Sigma[\mathsf{timer}] = \mathsf{timer}^* - \mathsf{timer}$. Then, it sends the message $(\mathsf{Timer}, sid, \mathsf{timer}, P)$ to P and the message $(\mathsf{Timer}, sid, \mathsf{timer}, \Sigma[\mathsf{timer}], \mathcal{F}_{\mathsf{C\_Clock}})$ if $\mathcal{F}_{\mathsf{C\_Clock}}$ registered.
- (Increasing timer^*) When $\mathcal{G}_{\mathsf{refRate}}$ sends $(\mathsf{Tick}, sid, \mathcal{G}_{\mathsf{timer}}^{\Sigma,P})$, $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ increments timer^* and lets $\Sigma[\mathsf{timer}] = \mathsf{timer}^* - \mathsf{timer}$. P and the message $(\mathsf{Timer}, sid, \mathsf{timer}, \Sigma[\mathsf{timer}], \mathcal{F}_{\mathsf{C\_Clock}})$ if $\mathcal{F}_{\mathsf{C\_Clock}}$ registered.
- If P sends $(\mathsf{Get\_Timer}, sid, P)$, $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ sends the message $(\mathsf{Timer}, sid, \mathsf{timer}, P)$ to P.

Fig. 2. The global functionality $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$

Definition 3 (Clock Interval). *Clock interval $T \in \mathbb{N}$ is the minimum time that a clock measures.*

Definition 4 (Clock Value). *Given that the initial clock value $c^* \in \mathbb{N}$ is mapped with $\mathsf{timer} = t^*$, we define the clock value for all $\mathsf{timer} = t \geq t^*$ of a $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ as:*

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

where T is the clock interval.

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 value (i.e., starting from t^*, every T progress in timer, increase the clock value).

Definition 5 (Clock). *A clock of P is a counter that keeps the clock value based on progression on timer of $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ after an initial match (t^*, c^*). The clock $\mathcal{C}$ of a party P when $\mathsf{timer} = t_{curr}$ is defined with $[(t^*, c^*), (t_{curr}, c_{curr}), \mathcal{G}_{\mathsf{timer}}^{\Sigma,P}]$ where $c_{curr} = \mathsf{Clock\_Val}(t_{curr}, T, t^*, c^*)$[1].*

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

Given

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

Definition 6 (Clock Difference). *Consider two clocks* $\mathcal{C}_i = [(c^*, t^*), (c, t), \mathcal{G}_{\mathsf{timer}}^{\Sigma, P_i}]$ *and* $\mathcal{C}_j = [(\hat{c}^*, \hat{t}^*), (\hat{c}, \hat{t}), \mathcal{G}_{\mathsf{timer}}^{\Sigma, P_i}]$ *given by a party* P_i *and* P_j*. The timer of* $\mathcal{C}$ *is* timer_i *and the timer of* $\mathcal{C}_j$ *is* timer_j*.*

We let the start time of c *be* $t^s = t^* + (c - c^*)T$ *on* timer_i *and be* $\mathbf{t^s} = t^s + \Sigma_i[t_s]$ *on* timer_j^**. Similarly, we let the start time of* $\hat{c}$ *be* $\hat{t}^s = \hat{t}^* + (\hat{c} - \hat{c}^*)T$ *on* timer_j *and be* $\hat{\mathbf{t}}^{\mathbf{s}} = \hat{t}^s + \Sigma_j[\hat{t}^s]$ *on* timer_j^**. The corresponding timer value of* t^s *on* timer_j *is* t_j^s *and the corresponding timer value of* $\hat{t}^s$ *on* timer_i *is* $\hat{t}_i^s$ *(See Definition 2).*

Without loss of generality, we assume that $c \leq \hat{c}$*. We define the clock difference* $\mathcal{C}_i - \mathcal{C}_j | t_j^s + \Sigma_j[t_j^s] - (\hat{\mathbf{t}}^{\mathbf{s}} - (\hat{c} - c)T)| = |\hat{t}_i^s + \Sigma_i[\hat{t}_i^s] - (\mathbf{t^s} + (\hat{c} - c)T)|$.

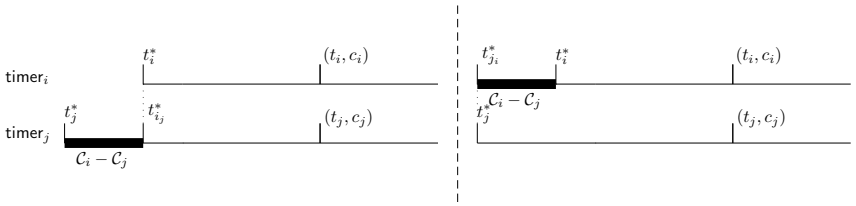

Fig. 3. Time difference of clocks: $|\mathcal{C}_i - \mathcal{C}_j|$. It can be computed on the timer of $\mathcal{C}_j$ (left-hand side) or on the timer of $\mathcal{C}_i$ (right-hand side).

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.

Consensus Clock Provider ($\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$) $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ helps parties to synchronize their clocks with respect to consensus on a clock. In a nutshell, $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ takes all clocks and construct a new clock with the algorithm `Clock_Consensus` as a reference clock to be synchronized.

$\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ is defined with the parameter T which is the clock interval and Θ. $\Theta = (\Theta_c, \Theta_p)$ is the desynchronization parameter. We call $\mathcal{P}_h$ is the set of honest parties and $\mathcal{P}_c$ is the set of corrupted parties. In more detail, $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ works as follows:

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

[2] `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.

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

$\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ is parameterized with T and $\Theta = (\Theta_c, \Theta_p)$ given by $\mathcal{Z}$. $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ first sends $(\mathsf{Register}, sid, \mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta})$ 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. $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ runs the algorithm $\texttt{Clock\_Consensus}(\{\mathcal{C}_j\}_{P_j \in \mathcal{P}_h})$. If `Clock_Consensus` outputs $\perp$, then $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ 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 $(\mathsf{Init\_Clock}, sid, \mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}, \tilde{C}_1, \tilde{C}_2, ..., \tilde{C}_n)$ for all $P_i \in \mathcal{P}_h$. If there exists $\tilde{C}_i = \mathsf{null}$, $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ aborts.
- **(New Consensus)** $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ runs the algorithm $\texttt{Clock\_Consensus}(\{\mathcal{C}_j\}_{P_j \in \mathcal{P}_h \cup \mathcal{P}_c})$. 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 $P_i \in \mathcal{P}_h$.

Fig. 4. The functionality $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$

adversary $\mathcal{A}$ and the honest parties and the protocol ends. Otherwise, $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ 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 $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$. If there exists a null clock among new clocks then $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ aborts.

(New Consensus): At this point, $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ 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, $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ runs the algorithm `Clock_Consensus` with the new honest clocks provided by the adversary. If `Clock_Consensus` outputs $\perp$, $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ sends $\bar{\mathcal{C}}_h$ (the initial consensus) to the honest parties. If it outputs a consensus clock $\bar{\mathcal{C}}$, $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ 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. 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 $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$. 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. However, if a protocol do not need such bound, θ_c should be considered as ∞.

We note that the requirement of initial consensus can be removed by defining the algorithm `Clock_Consensus` accordingly. We have it in our definition to be more general.

3.2 UC- Partially Synchronous Blockchain Network Model

Our network model $\mathcal{F}_{\mathsf{DDiffuse}}$ is similar to the network model of Ouroboros Praos [15,28,20]. 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 $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$. Before describing the protocol, we give some preliminary definitions related to security of blockchain.

4.1 Blockchain

A protocol that defines the construction of a blockchain structure is called a blockchain protocol. Garay et al. [20] define some properties given below to obtain a secure blockchain protocol. In these definitions, the blockchain protocol

follows the logical clock $r_1, r_2, ...$ in the synchronous communication [10,27,29]. We note that these rounds do not have to advance based on the measurement of time as our clocks.

Definition 7 (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 $r_1 < r_2$ satisfy that B_1 without the last k blocks is the prefix of B_2.*

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 and the blockchain including the finalized blocks final blockchain.

We modify the chain quality property (CQ) by Garay et al. and give chain density property. CQ property ensures the existence of sufficient honest blocks on any blockchain owned by an honest party.

Definition 8 (Chain Density (CD) Property). *The CD property with parameters $s_{cd} \in \mathbb{N}, \mu \in (0, 1]$ ensures that any portion $B[s_u : s_v]$ of a final blockchain B spanning between rounds s_u and $s_v = s_u + s_{cd}$ contains $\mathsf{length}(B[s_u : s_v])\mu$ honest blocks.*

The CD property ensures a minimum ratio of honest blocks in the final sub-blockchain. In our protocol, we need CD property with $\mu > 0.5$ which implies that the sufficiently long span of the final blockchain contains more honest blocks than malicious ones.

4.2 Consensus Clock

We first describe the algorithm `Clock_Consensus` in $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ that defines consensus clock in our protocol. `Clock_Consensus` receives clocks of multiple parties as input, and outputs one of them as a consensus clock. In our setup, we say that two clocks are *synchronous* if their difference is κ according to Definition 6. 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 $\mathcal{C}_j = [(t_j^*, c_j^*), (c_j, t_j), \mathcal{G}_{\mathsf{timer}}^{\Sigma,P_j}]$, `Clock_Consensus` first finds the pairwise difference of all clocks as defined in Definition 6. We note that $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ has enough source to compute the difference. It can access the drift information of each timer value by asking their local timer functionalities and know the corresponding timer values between timers because all t_i's and t_j's given as a current timer value in $\mathcal{C}_i$'s and $\mathcal{C}_j$'s are their corresponding timer value. If there exists two clock with the difference greater than κ (no synchronization exist), `Clock_Consensus` outputs $\perp$ meaning that no consensus exists. Otherwise, it selects the median clock as consensus clock as described in Algorithm 1.

Algorithm 1 `Clock_Consensus`$(\{\mathcal{C}_j\})$ where $\mathcal{C}_j = [(c_j^* t_j^*), (c_j, t_j), \mathcal{G}_{\mathsf{timer}}^{\Sigma, P_j}]$

1: $\mathsf{start\_lst} = \emptyset$
2: $t \leftarrow$ value of $\mathsf{timer}_{\mathcal{F}_{\mathsf{timer}}^{T,\Theta}}$ when clocks are given
3: **add** all clocks list
4: **for all** $\mathcal{C}_u, \mathcal{C}_v \in \mathsf{list}$ **where** $\mathcal{C}_u \neq null$ and $\mathcal{C}_v \neq null$ **do**
5: **if** $\mathcal{C}_u - \mathcal{C}_v > \kappa$ **then**
6: **return** $\perp$
7: **pick** $c \in \mathbb{N}$ **such that** for all j such that $c > c_j$
8: **for all** $\mathcal{C}_j \neq null$ **do**
9: $t_j^c \leftarrow t_j^* + cT$ // start time of clock value on timer_j
10: $\hat{t}_j^c \leftarrow t + (t_j - t_j^c)$ // correspondence value of t_j^c on $\mathsf{timer}_{\mathcal{F}_{\mathsf{timer}}^{T,\Theta}}$ (See Definition 2)
11: **add** $\hat{t}_j^c$ **to** start_lst
12: $\hat{t}_i^c \leftarrow$ `Median`(start_lst) // `Median`(start_lst) sorts the list and returns the median
13: **return** $\mathcal{C}_i$

4.3 Relative Time Protocol

The relative time protocol realizes $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\theta}$ (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 [28], Ouroboros Praos [15], Ouroboros Genesis [5], 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 of physical clocks, 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. c_e 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 8). If the previous epoch is the first epoch then $c_{e-1} = 0$.

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: list ← ∅
2: for i = 0 to n do
3:     a_i ← c − c'_i
4:     store (t_i + a_i T) to list // start time of c according to the party that produced B'_i
5: lst ← sort(list)
6: return median(list)
```

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 1. *Assuming that the blockchain protocol preserves the common prefix property with the parameter k, and the chain density property with the parameter $s_{cd}, \mu > 0.5$ 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 $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ 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 $\mathcal{F}^{\delta}_{\mathsf{DDiffuse}}$. The simulation is straightforward. $\mathcal{S}$ simulates honest parties in the underlying blockchain protocol as well, and their local clocks based on the clock increments by $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$. 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' list 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 $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ 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 $t_x, t_y \in \mathsf{list}$ (list 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 list 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. 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.
5. 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.
6. 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`.
7. L. Bicknell. NTP issues today. outages mailing list. `https://mailman.nanog.org/pipermail/nanog/2012-November/053449.html`.
8. D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. Verifiable delay functions. In *Annual international cryptology conference*, pages 757–788. Springer, 2018.

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. Y. T. Kalai, Y. Lindell, and M. Prabhakaran. Concurrent composition of secure protocols in the timing model. *Journal of Cryptology*, 20(4):431–492, 2007.
27. J. Katz, U. Maurer, B. Tackmann, and V. Zikas. Universally composable synchronous computation. In *Theory of Cryptography Conference*, pages 477–498. Springer, 2013.

28. 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.
29. A. Kiayias, H.-S. Zhou, and V. Zikas. Fair and robust multi-party computation using a global transaction ledger. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 705–734. Springer, 2016.
30. 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.
31. 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.
32. 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.
33. A. Malhotra, I. E. Cohen, E. Brakke, and S. Goldberg. Attacking the network time protocol. In *NDSS*, 2016.
34. 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.
35. D. L. Mills. Computer network time synchronization. In *Report Dagstuhl Seminar on Time Services Schloß Dagstuhl, March*, volume 11, page 332. Springer, 1997.
36. P. Papadimitratos and A. Jovanovic. Gnss-based positioning: Attacks and countermeasures. In *MILCOM 2008-2008 IEEE Military Communications Conference*, pages 1–7. IEEE, 2008.
37. 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.
38. L. Schenato and F. Fiorentin. Average timesynch: A consensus-based protocol for clock synchronization in wireless sensor networks. *Automatica*, 47(9):1878–1886, 2011.
39. J. Selvi. Breaking ssl using time synchronisation attacks. In *DEF CON Hacking Conference*, 2015.
40. 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.
41. 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.
42. 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.
43. 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.
44. 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.
45. G. Wood. Polkadot: Vision for a heterogeneous multi-chain framework. *White Paper*, 2016.