Network Time with a Consensus on Clock

Handan Kılınç Alper

Web3 Foundation

handan@web3.foundation

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 [23, 13, 4, 5, 12] are secure in the synchronous communication model [8, 22, 24] 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 [28, 33, 30]. In one incident (on November 19, 2012) [6], 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 [36, 21, 31, 20, 37] (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 [8, 22, 24]. 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. [10], 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. [10] 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 [10], 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 [23, 13, 4, 12] 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. [10] 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 [27, 32, 19, 2, 38, 34, 26]. 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 [39] which aims to connect multiple blockchain networks.

1.1 Related Works

UC Clock Models: There are UC models [8, 22, 24] 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 [8, 22, 24] 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 [8, 22, 24] does not need to have this ability.
- Consensus-clock functionality in our model and clock functionalities in [8, 22, 24] 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 [8, 22, 24].

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. [10] 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 [10] 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 [23, 13, 4, 12] is preserved in the synchronous communication model. Ouroboros Genesis [4] is built on top of the clock functionality [24] 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 [28, 33, 30] as we pointed.

Ouroboros Chronos [5] 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 [5, 24] 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 [23]. 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.

Ouroboros Clepsydra [3] has a similar adversarial model as Chronos.

The timeline in the Dfinity consensus protocol [18] 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 [11, 17] 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 [23, 13, 4, 12, 18] 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 [14, 15, 29, 35, 25] have been extensively studied in previous work, as it is a important component in distributed systems. Many protocols [27, 32, 19, 2] exist for consensus clocks with different assumptions. One approach is based on dividing the network into some well-connected clusters [14] that aim to achieve consensus between clusters. There are also fully distributed approaches [38, 34] 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 ITM identifier pid. A party P in UC model is an ITI with the identifier $(\mathsf{sid}, \mathsf{pid})$.
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 [7, 8] for further details about the UC-model.

Definition 2.1. *(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 [9] (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 [13, 4]. 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 functionalities $\mathsf{ListRate}$ which is initially empty. $\mathcal{G}_{\mathsf{refRate}}$ is identified by a session identifier $\mathsf{sid}_{\mathsf{refRate}}$.

- ***(Registration)*** If a functionality $\mathcal{F}$ sends a message $(\mathsf{Register}, \mathsf{sid}, \mathcal{F})$, $\mathcal{G}_{\mathsf{refRate}}$ adds $(\mathcal{F}, \mathsf{sid})$ to $\mathsf{ListRate}$ and sends $(\mathsf{Registered}, \mathsf{sid}, \mathcal{F})$ to $\mathcal{F}$.
- ***(Progress in Time)*** In every tick according to the defined rate, $\mathcal{G}_{\mathsf{refRate}}$ sends $(\mathsf{Tick}, \mathsf{sid}_{\mathsf{refRate}}, P)$ 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 $\mathcal{G}_{\mathsf{refClock}}$ [10] and $\mathcal{G}_{\mathsf{refRate}}$ is that $\mathcal{G}_{\mathsf{refRate}}$ does not have any absolute values related to time (i.e., 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 based on a physical phenomenon (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 [1]"). 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*.

3.1.2 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 a local clock, 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 holds the timer skew which holds the difference between the local timer and correct timer follows the rate of $\mathcal{G}_{\mathsf{refRate}}$ (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 skew is -2 minutes). $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ is accessible by its owner 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. We also let P to reset the timer^* and Σ. The reason of having timer^* and resetting is to define notions drift and skew which cannot be defined without a reference time which is $\mathcal{G}_{\mathsf{refRate}}$ in our case. We define $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ in the GUC model 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 any information related to 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 $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$.

Definition 3.1 (Timer Drift)**.** *The function* $\mathsf{rate}(\mathsf{timer}, t_i, t_j) = t_j + \Sigma[t_j] - t_i - \Sigma[t_i]$ *is a timer rate of* timer *that shows the frequency of ticks between* $\mathsf{timer}^* = t_i$ *and* $\mathsf{timer}^* = t_j$ *where* $t_i \leq t_j$*. We define the timer drift as the difference between timer rate of* timer *and timer rate of* timer^* *between* $\mathsf{timer}^* = t_i$ *and* $\mathsf{timer}^* = t_j$ *i.e.,* $\Delta(\mathsf{timer}, t_i, t_j) = \mathsf{rate}(\mathsf{timer}, t_i, t_j) - (t_j - t_i) = \Sigma[t_j] - \Sigma[t_i]$.

The rate of a timer $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ is not constant because the frequency of the ticks depends on $\mathcal{Z}$. If the local timer drift is negative then it means that the local timer is slower than the real timer and if it is positive, then it means that the local timer is faster than the real timer.

Remark that even if the timer drift is constant for all t_j, t_i where $t_j - t_i = T$, the absolute timer skew increases in time.

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.

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

$\mathcal{Z}$ initiates $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ with a value $t \in \mathbb{Z}$. Then, $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ creates two parameters $\mathsf{timer} = t$ and $\mathsf{timer}^* = t$. Besides, it creates a list $\mathsf{ListTimer}$ to keep registered functionalities and creates an empty array Σ for each sid. $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ is identified with a parameter $\mathsf{sid}_{\mathsf{timer}}$ and its owner $P = (\mathsf{pid}, .)$.

- ***(Registration to $\mathcal{G}_{\mathsf{refRate}}$)*** $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ sends $(\mathsf{Register}, \mathsf{sid}_{\mathsf{timer}}, \mathcal{G}_{\mathsf{timer}}^{\Sigma,P})$ to $\mathcal{G}_{\mathsf{refRate}}$ and receives back $(\mathsf{Registered}, \mathsf{sid}_{\mathsf{timer}}, \mathcal{G}_{\mathsf{timer}}^{\Sigma,P})$. This is done once after the initialization of the functionality.
- ***(Functionality Registration)*** When $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ receives a message $(\mathsf{Register}, \mathsf{sid}, \mathcal{F}, (\mathsf{pid}', \mathsf{sid}'))$ from a functionality $\mathcal{F}$, $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ replies with $(\mathsf{Registered}, \mathsf{sid}, \mathcal{F})$ and adds $\mathcal{F}$ to $\mathsf{ListTimer}$ if $\mathsf{pid}' = \mathsf{pid}$. Otherwise, it replies with $(\mathsf{NotRegistered}, \mathsf{sid}, \mathcal{F})$.
- ***(Increasing timer)*** When $\mathcal{Z}$ sends an $(\mathsf{Increase})$ request, $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ increments timer and lets $\Sigma[\mathsf{timer}^*] = \mathsf{timer} - \mathsf{timer}^*$. Then, it sends the message $(\mathsf{Timer}, \mathsf{sid}, \mathsf{pid}, \mathsf{timer})$ to all $(\mathsf{pid}, \mathsf{sid})$. It also sends the message $(\mathsf{Timer}, \mathsf{sid}', \mathcal{F}, \mathsf{timer})$ to $\mathcal{F}$ where $(\mathcal{F}, \mathsf{sid}') \in \mathsf{ListTimer}$, .
- ***(Increasing timer^*)*** When $\mathcal{G}_{\mathsf{refRate}}$ sends $(\mathsf{Tick}, \mathsf{sid}_{\mathsf{timer}}, \mathcal{G}_{\mathsf{timer}}^{\Sigma,P})$, $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ increments timer^* and lets $\Sigma[\mathsf{timer}^*] = \mathsf{timer} - \mathsf{timer}^*$.
- ***(Resetting)*** When P sends $(\mathsf{Reset}, \mathsf{sid}, P)$, $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ replaces Σ with an empty array and lets $\mathsf{timer}{*} = \mathsf{timer}$.

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

Definition 3.2 (Clock Interval). *Clock interval $T \in \mathbb{N}$ is the minimum time that a clock measures.*

Definition 3.3 (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 3.4 (Clock). *A clock of P is a counter that keeps the clock value based on progression on* timer *of $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$ with respect to clock interval T after an initial match (t^*, c^*) i.e. the clock starts to show c^* when* $\mathsf{timer} = t^*$*. The clock $\boldsymbol{\mathcal{C}}$ of a party P is defined with $[t^*, \mathcal{G}_{\mathsf{timer}}^{\Sigma,P} \rightarrow t, \mathsf{Clock\_Val}(t, T, t^*, c^*)]$.*

Without loss of generality, we assume that the initial clock value is 0 for all clocks in the rest of the paper. In this case, we do not need to add $\mathsf{Clock\_Val}(t, T, t^*, 0)$ as a third element of a clock because it is redundant. As a notation, $\boldsymbol{\mathcal{C}} = [t^*, \mathcal{G}_{\mathsf{timer}}^{\Sigma,P} \rightarrow t]$ represents the whole clock with the access of $\boldsymbol{\mathcal{C}}$'s timer and $\mathcal{C} = [t^*, t]$ represents the momentary clock which represents what P sees in $\boldsymbol{\mathcal{C}}$ instantaneously. Whenever P constructs a clock, he sends sends $(\mathsf{Reset}, \mathsf{sid}, P)$ to $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P}$.

Definition 3.5 (Clock Difference)**.** *Given momentary clocks $\mathcal{C}_i = [t_i^*, t_i]$ and $\mathcal{C}_j = [t_j^*, t_j]$, the time difference of clocks $\boldsymbol{\mathcal{C}_i}$ and $\boldsymbol{\mathcal{C}_j}$ is the difference of between relative time passed from the beginning of the clocks i.e., $\boldsymbol{\mathcal{C}_i} - \boldsymbol{\mathcal{C}_j} = (t_i - t_i^*) - (t_j - t_j^*)$ and $\mathcal{C}_j - \mathcal{C}_i = (t_j - t_j^*) - (t_i - t_i^*)$. We define the absolute time difference of these clocks as $|\boldsymbol{\mathcal{C}_i} - \boldsymbol{\mathcal{C}_j}| = |\boldsymbol{\mathcal{C}_j} - \boldsymbol{\mathcal{C}_i}|$.*

See Figure 3 to visualize the clock difference.

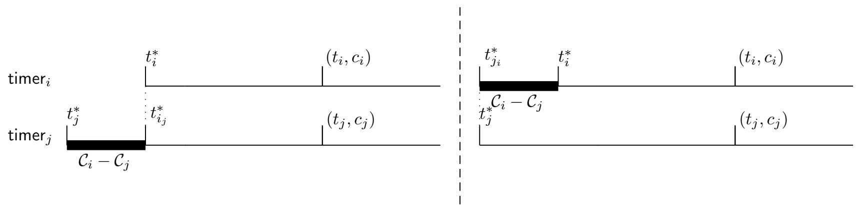

Figure 3: Time difference of clocks: $|\mathcal{C}_i - \mathcal{C}_j|$. It can be computed on the timer of $\mathcal{C}_j$ (left-hand side) or on the timer of $\mathcal{C}_i$ (right-hand side).

Since the clock difference depends on the current timer value of timers, the clock difference may vary over time. For example, assume that when we compute the absolute clock difference timer_i is t_i and timer_j is t_j, the difference is $|(t_i - t_i^*) - (t_j - t_j^*)|$. If we compute the clock difference after X ticks by $\mathcal{G}_{\mathsf{refRate}}$, the difference may not be the same because timer_i does not have to be $t_i + X$ and timer_j does not have to be $t_j + X$ because of the possible drifts introduced by $\mathcal{Z}$ in $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P_i}$ and $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P_j}$.

Assume that the absolute clock difference when the momentary clocks are $\mathcal{C}_i = [t_i^*, t_i]$ and $\mathcal{C}_j = [t_j^*, t_j]$ is D . The absolute clock difference of the future momentary clocks $\mathcal{C}_i = [t_i^*, t_i']$ and $\mathcal{C}_j = [t_j^*, t_j']$ where $t_j' \geq t_j$ and $t_i' \geq t_i$ is equal to $D + |\Delta(\mathsf{timer}_i, t_i, t_i') - \Delta(\mathsf{timer}_j, t_j, t_j')|$.

If a party P_i learns a momentary clock of another party $\mathcal{C}_j = [t_j^*, t_j]$ and wants to construct a clock synchronized with it, he runs the Algorithm 1 below and sends $(\mathsf{Reset}, \mathsf{sid}, P_i)$ to $\mathcal{G}_{\mathsf{timer}}^{\Sigma,P_i}$:

Algorithm 1 SynchronizeWith($\mathcal{C}_j$)

```
1: $t_i \leftarrow \mathcal{G}_{\mathsf{timer}}^{\Sigma,P_i}$ // obtain the current timer value
2: $t_i^* \leftarrow t_i - (t_j - t_j^*)$
3: $\boldsymbol{\mathcal{C}_i} \leftarrow [t_{\mathsf{new}}^*, \mathcal{G}_{\mathsf{timer}}^{\Sigma,P_i} \rightarrow t]$
```

Remark that if $\mathcal{C}_j$ is still $[t_j^*, t_j]$ in the end of the SynchronizeWith algorithm, the difference between clock of $\boldsymbol{\mathcal{C}_j}$ and the new clock $\boldsymbol{\mathcal{C}_i}$ becomes 0 in the end of the SynchronizeWith algorithm.

3.1.3 Synchronization with a Consensus on Clock ($\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$)

$\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ helps parties to synchronize their clocks with a clock close to a consensus on clock. The functionality defines the consensus with an algorithm ConsensusClock which receives momentary clocks according to Definition 3.4 as an input and outputs either $\perp$ if consensus does not exist or a clock as a consensus to be synchronized with. The momentary clocks in the algorithm ConsensusClock can be imagined as a vote of what the clock should be when the algorithm is run. We note that ConsensusClock can be defined based on the needs of a real-world protocol. e.g., the mostly agreed clock or minimum clock can be defined as a consensus on clock or the way that we define for our protocol in Section 4.

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

$\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ is parameterized with T and $\Theta = (\Theta_c, \Theta_p)$ given by $\mathcal{Z}$ and identified by a session id $\mathsf{sid}_{\mathsf{C\_Clock}}$. It maintains a set of parties (ITI's) $P_i = (\mathsf{pid}_i, \mathsf{sid})$ in $\mathcal{P}$. $\mathcal{P}$ is initially empty but it is populated when a party registers with its identifier pid_i and its session identifier sid. $\mathcal{P}_h \subset \mathcal{P}$ is the set of honest parties.
$\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ first sends $(\mathsf{Register}, \mathsf{sid}_{\mathsf{C\_Clock}}, \mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}})$ to $\mathcal{G}_{\mathsf{refRate}}$ and receives back $(\mathsf{Registered}, \mathsf{sid}_{\mathsf{C\_Clock}}, P)$. Thus, it constructs its own timer $\mathsf{timer}_{\mathsf{C\_Clock}}$.
When an honest (P_i, sid) registers, $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ sends $(\mathsf{Register}, \mathsf{sid}_{\mathsf{C\_Clock}}, \mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}})$ to $\mathcal{G}^{\Sigma,P_i}_{\mathsf{timer}}$ and receives back $(\mathsf{Registered}, \mathsf{sid}_{\mathsf{C\_Clock}}, P)$.

- ***(Initial Consensus)*** Each party $P_j \in \mathcal{P}_h$ sends the message $(\mathsf{InitialTimer}, \mathsf{sid}, \mathsf{pid}_j, t^*_j)$ where t^*_j is the initial timer value of the clock of P_j. Then, $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ obtains $\mathcal{C}_j = [t^*_j, t_j]$ where t_j is the current timer value of $\mathcal{G}^{\Sigma,P_j}_{\mathsf{timer}}$ for all $(\mathsf{pid}_j, \mathsf{sid}) \in \mathcal{P}_h$ and runs the algorithm ConsensusClock $(\{\mathcal{C}_j\}_{(.,\mathsf{sid})\in\mathcal{P}_h})$.
  - If ConsensusClock outputs $\perp$, then $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ sends $(\mathsf{Abort}, \mathsf{sid}, \mathsf{pid}_j\ \perp)$ to all $P_j \in \mathcal{P}$ and aborts the protocol.
  - If it outputs $\bar{\boldsymbol{\mathcal{C}}}_{\boldsymbol{h}}$, it sends $(\mathsf{Clock}, \mathsf{sid}, \mathcal{A}, \mathcal{C}_j = [t^*_j, t_j])$ to $\mathcal{A}$ for all $P_j \in \mathcal{P}_h$ whenever $\mathcal{G}^{\Sigma,P_j}_{\mathsf{timer}}$ outputs a new timer value t_j.
- ***(New Clocks)*** $\mathcal{A}$ replies with $(\mathsf{NewClock}, \mathsf{sid}, \mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}, \{\tilde{\mathcal{C}}_j\}_{(.,\mathsf{sid})\in\mathcal{P}})$. If there exists $\tilde{\mathcal{C}}_j = \mathsf{null}$, $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ aborts and sends $(\mathsf{Abort}, \mathsf{sid}, \mathsf{pid}_j, \perp)$ to all $P_j \in \mathcal{P}$. Otherwise, it continues with the next phase.
- ***(New Consensus)*** $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ runs ConsensusClock$(\{\mathcal{C}_j\}_{(.,\mathsf{sid})\in\mathcal{P}})$.
  - If it does not output $\perp$ and $|\bar{\boldsymbol{\mathcal{C}}} - \bar{\boldsymbol{\mathcal{C}}}_{\boldsymbol{h}}| \leq \Theta_c$ and $|\bar{\boldsymbol{\mathcal{C}}} - \tilde{\boldsymbol{\mathcal{C}}}_{\boldsymbol{j}}| \leq \Theta_p$ for all $P_i \in \mathcal{P}_h$, it sends $(\mathsf{SynchronizeWith}, \mathsf{sid}, \mathsf{pid}_j, \mathcal{C}_j)$ $\tilde{\mathcal{C}}_j$ to each party $P_j \in \mathcal{P}_h$.
  - Otherwise, it sends $(\mathsf{SynchronizeWith}, \mathsf{sid}, \mathsf{pid}_j, \bar{\mathcal{C}}_h)$ to each party $P_j \in \mathcal{P}_h$ and $(\mathsf{SynchronizeWith}, \mathsf{sid}, \mathsf{pid}_i, \bar{\mathcal{C}}_i)$ to all $P_i \in \mathcal{P} \setminus \mathcal{P}_h$.

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

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

(**Timer Registration:**) $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ registers itself to the local timer functionalities $\mathcal{G}^{\Sigma,P}_{\mathsf{timer}}$ of honest parties to keep their clocks and $\mathcal{G}_{\mathsf{refRate}}$ to construct its own timer $\mathsf{timer}_{\mathsf{C\_Clock}}$.

(Initial Consensus): $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ receives all initial timer values of each honest parties to construct their clocks and then checks whether the initial consensus between honest parties exist. If it does not exist from the beginning, $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ aborts the protocol.

In more detail, each honest party $P_i \in \mathcal{P}_h$ having a clock sends the initial timer value t^*_i of his clock to $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$. If P_i does not have any clock, he sends null as an indicator that he does not have a clock. After receiving all of inputs from honest parties, $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ constructs clocks of each P_i as $\boldsymbol{\mathcal{C}_i} = [t^*_i, \mathcal{G}^{\Sigma,P_i}_{\mathsf{timer}} \to t_i]$ according to the Definition 3.4 if $t^*_i \neq \mathsf{null}$. Otherwise, it lets the clock of P_i be $\boldsymbol{\mathcal{C}_i} = \mathsf{null}$. Remark that $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ is capable of constructing local clocks of each honest party P because it has a right to access each $\mathcal{G}^{\Sigma,P}_{\mathsf{timer}}$.

Finally, $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ runs the algorithm ConsensusClock by inputting honest momentary clocks to check whether the initial consensus exists. If the algorithm outputs $\perp$, $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ sends an abort message to the adversary $\mathcal{A}$ and honest parties. So, the protocol ends. If the algorithm outputs consensus on a clock $\tilde{\boldsymbol{\mathcal{C}}}_{\boldsymbol{h}} = [\tilde{t}^*_h, \mathsf{timer}_{\mathsf{C\_Clock}} \to \tilde{t}_h]$, $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ gives access of the honest clocks to the adversary $\mathcal{A}$. In order to do this, $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ sends momentary clock $\mathcal{C}_i$ of each honest party P_i to $\mathcal{A}$ in every update of $\mathcal{C}_i$ (i.e., whenever timer_i is incremented in $\mathcal{G}^{\Sigma,P_i}_{\mathsf{timer}}$) without any delay. In meantime, $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ waits for a response from $\mathcal{A}$.

(New Clocks:) $\mathcal{A}$ gives new momentary clocks $\bar{\mathcal{C}}_i$ for each $P_i \in \mathcal{P}_h$ and momentary clocks on behalf of corrupted parties to $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ as a response. If there exists a null clock among new clocks for honest parties then $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ aborts.

(New Consensus): At this point, $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ checks if a new consensus is possible with the new momentary clocks. If the new consensus exists and it is close enough to the initial one, then all honest parties receive the corresponding new momentary clocks provided by $\mathcal{A}$ to be synchronized with.

In more detail, $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ first runs the algorithm ConsensusClock by inputting new momentary clocks. If ConsensusClock outputs $\perp$, $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ sends the initial consensus $\bar{\mathcal{C}}_h$ to the honest parties. If it outputs a clock $\bar{\mathcal{C}}$ as consensus on clock, $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ continues as follows:

- If $|\bar{\boldsymbol{\mathcal{C}}} - \bar{\boldsymbol{\mathcal{C}}}_h| \leq \Theta_c$ and $|\bar{\boldsymbol{\mathcal{C}}} - \tilde{\boldsymbol{\mathcal{C}}}_i| \leq \Theta_p$ 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 $P_i \in \mathcal{P}_h$.

Whenever an honest party receives a momentary clock $\mathcal{C}$ from $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$, they output SynchronizeWith($\mathcal{C}$). Otherwise, he outputs $\perp$. We note that the new consensus phase happens without any delay after the new clocks phase. More details of $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ are in Figure 4.

In a nutshell, this functionality aims to provide clocks to honest parties which do not drift apart much from the initial consensus that they have and which is close enough to the new consensus on 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 $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$. For example, (a very extreme example) we do not want to end up with a new 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. If a protocol does not need such a bound, θ_c should be considered as ∞.

We note that the more stronger version of $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ can be defined by removing the condition of having an initial consensus.

3.2 UC- Partially Synchronous Network Model

Our network model $\mathcal{F}_{\mathsf{DDiffuse}}$ is similar to the network model of Ouroboros Praos [13, 23, 16]. 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 $\mathcal{G}_{\mathsf{refRate}}$.

$\boldsymbol{\mathcal{F}^{\delta}_{\mathsf{DDiffuse}}}$**:** The message handling functionality $\mathsf{Diffuse}$ was first introduced by Garay et al. [16] in a synchronous network where message delivery is executed in a certain amount of known time. Then, David et al. [13] define a new functionality "delayed diffuse" ($\mathsf{DDiffuse}$)

that realizes a partially synchronous network where a message arrives to others eventually, but parties do not know how long it takes. $\mathsf{DDiffuse}$ is parameterized with the network delay parameter Δ and ensures that all messages are received at most Δ-rounds later. However, it does not use any clock that measures length of a round. $\mathcal{F}^{\delta}_{\mathsf{DDiffuse}}$ is a version of $\mathsf{DDiffuse}$ which can access of $\mathcal{G}_{\mathsf{refRate}}$ to count the round based on relative time. It simply sends a given message from a party P_i to all other parties within a bound δ-ticks. $\mathcal{F}^{\delta}_{\mathsf{DDiffuse}}$ first registers to $\mathcal{G}_{\mathsf{refRate}}$ and creates a local timer $\mathsf{timer} = 0$. Whenever $\mathcal{G}_{\mathsf{refRate}}$ sends a message with Tick, 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 at latest δ-ticks later.

3.3 The Result from Combination of Models

In this section, we give a nice outcome of $\mathcal{G}^{\Sigma,P}_{\mathsf{timer}}$ and $\mathcal{F}_{\mathsf{DDiffuse}}$ which let the parties in these models to preserve the distance between their clocks according to Definition 3.4.

Consider m parties $P_1, ..., P_m$ initiated with parameters n and $a \leq n$ and with clocks $\boldsymbol{\mathcal{C}_k} = [t^*_k, \mathcal{G}^{P_k,\Sigma}_{\mathsf{timer}} \to t_k]$ given by $\mathcal{Z}$. Whenever $\mathcal{Z}$ requests to a random party P_k to send a message, P_k sends the message B'_i including current clock value $r_i = \mathsf{Clock\_Val}(t_{curr}, T, t^*_k, 0)$ with $\mathcal{F}_{\mathsf{DDiffuse}}$. When a party P_ℓ receives a message B_i, it stores $u_{i\to\ell} = t'_{i\to\ell} - r_i T$ to a list list_ℓ where $t'_{i\to\ell}$ is the arrival time of B_i according to its timer timer_ℓ. After receiving n messages, each party P_k constructs a new clock $\boldsymbol{\mathcal{C}_k} = [\mathtt{sort}(\mathsf{list}_k)[a], \mathcal{G}^{P_k,\Sigma}_{\mathsf{timer}}]$ where $\mathtt{sort}(\mathsf{list}_\ell)[a]$ is the a^{th} element of sorted list_ℓ.

Lemma 3.1. *In the end of a session as described above, assuming that $\mathcal{Z}$ requests to send message exactly n-times, the time difference is at most D between two requests according to $\mathcal{G}_{\mathsf{refRate}}$, $\Delta_{\max} = \max\{\Delta(\mathsf{timer}_i, t, t+T) : \mathcal{G}^{P_i,\Sigma}_{\mathsf{timer}} \to t\}$, and timer drift from the start of the session and end of the session is between $[-\Delta_{\mathsf{sid}}, \Delta_{\mathsf{sid}}]$, the difference between new clocks of parties can be at most $3\delta + 3\lceil \frac{D}{T} \rceil \Delta_{\max} + 2\Delta_{\mathsf{sid}}$ where $\Delta_{\max}$ is the maximum drift on a timer on two different ticks.*

Remark that each $u_{i\to\ell}$ represents the timer value when clock value is 0 in the momentary clock of sender according to the view of the receiver party P_ℓ. So, each list_ℓ is the list of timer values of clock value 0's on timer timer_ℓ with respect to each momentary clock when the message each sent according to the view of P_ℓ.

Proof. If $\mathcal{Z}$ requests exactly n-times to send a message during the session, it implies that all parties receive the messages before ending the session. So, each list_k of a P_k has the timer values of the same messages.

Now, imagine a non-existing list $\mathsf{reallist}_k$ of a party P_k which includes $u_{i\to k} = t_{i\to k} - r_i T$ of each message B_i which is sent on timer value $t_{i\to k}$ according to real timer (not drifted) timer^*_k of P_k. Remark that the respective value of it in list_u is $u'_{i\to k} = t'_{i\to k} - r_i T$ where $t'_{i\to k}$ is the arrival time of B_i to P_k according to timer_k. If a message B_i arrives to P_k without any delay, then $t'_{i\to k} = t_{i\to k} + \Sigma[t_{i\to k}]$. If the message B_i arrives to P_k with a maximum delay δ, $t'_{i\to k} = t_{i\to k} + \delta + \Sigma[t_{i\to k} + \delta]$. Therefore, for all P_k, $u_{i\to k} + \Sigma[t_{i\to k}] \leq u_i \leq u_{i\to k} + \delta + \Sigma[t_{i\to k} + \delta]$.

Let's define a new notation $\preceq$ to order messages. We call $B'_i \preceq B'_j$ if corresponding timer value of B_i in a list is less than or equal to the corresponding timer value of B_j in the same list. For example, $B_i \preceq B_j$ in list_k if $u'_{i\to k} \leq u'_{j\to k}$ in list_k. Remark that the order of messages are the same in $\mathsf{reallist}_k$ of each P_k because $(t_{i\to k} - t_{j\to k}) = (t_{i\to\ell} - t_{j\to\ell})$ so $(u_{i\to k} - u_{j\to k}) = (u_{i\to\ell} - u_{j\to\ell})$ for all P_k and P_ℓ. However, we cannot say that the order of the messages are the same for list_u and list_v because of the arrival time difference of these blocks to P_u and P_v i.e. $(t'_{i\to k} - t'_{j\to k}) \neq (t'_{i\to\ell} - t'_{j\to\ell})$. Now, we analyze under which condition(s) the order of two messages are not the same.

Consider two messages B_i and B_j where $B_j \preceq B_i$ in list_k and $B_i \preceq B_j$ in $\mathsf{reallist}_k$. This implies that $u'_{j\to k} \leq u'_{i\to k}$ and $u_{i\to k} \leq u_{j\to k}$. Since $u_{j\to k} + \Sigma[t_{j\to k}] \leq u'_{j\to k} \leq u_{j\to k} + \delta + \Sigma[t_{j\to k} + \delta]$, $u_{i\to k} + \Sigma[t_{i\to k}] \leq u'_{i\to k} \leq u_{i\to k} + \delta + \Sigma[t_{i\to k} + \delta]$, we can see that $u_{j\to k} - u_{i\to k} \leq \delta + \Sigma[t_{i\to k} + \delta] - \Sigma[t_{j\to i}] = \delta + \Delta(\mathsf{timer}_k, t_{i\to k}, t_{i\to k} + \delta) \leq \delta + \lceil \frac{T}{\delta} \rceil \Delta_{\max}$. This shows us that if the order of two messages B_i and B_j in two list list_k and list_ℓ are different then

$$|u_{j\to k} - u_{i\to k}| = |u_{j\to\ell} - u_{i\to\ell}| \leq \delta + \lceil \frac{\delta}{T} \rceil \Delta_{\max} \tag{2}$$

Now consider that $B_i \preceq B_j$ in list_k and list_ℓ. However, $\mathtt{sort}(\mathsf{list}_k)[a] = u'_{i\to k}$ and $\mathtt{sort}(\mathsf{list}_\ell)[a] = u'_{j\to k}$ where $i \neq j$. This means that there exists $x > 0$ such that $\mathtt{sort}(\mathsf{list}_k)[a + x] = u'_{j\to k}$ and there exits $y > 0$ such that $\mathtt{sort}(\mathsf{list}_\ell)[a - y] = u'_{i\to\ell}$. This means that there are some$y - 1$ elements between $u'_{w\to k} \in \mathtt{sort}(\mathsf{list}_k)(a - y : a)$. In order to increase the order of B_i to a in list_ℓ, there should be y-elements which are less than $u'_{i\to\ell}$. It means that there exists B_w such that $B_i \preceq B_j \preceq B_w$ in list_k but $B_w \preceq B_i \preceq B_j$ in list_ℓ. From the inequality (2)), we can say that distance of $u_{w\to k}$ (or $u_{w\to\ell}$) to both $u_{i\to k}$ (or $u_{i\to\ell}$) and $u_{j\to k}$ (or $u_{j\to\ell}$) is at least $\delta + \lceil \frac{\delta}{T} \rceil \Delta_{\max}$. Therefore, if the orders of two messages are different, it means that

$$|u_{j\to k} - u_{i\to k}| \leq 2\delta + 2\lceil \frac{\delta}{T} \rceil \Delta_{\max} \tag{3}$$

Finally, consider P_k runs $\mathtt{sort}(\mathsf{list}_k)[a] = u'_{i\to k}$ which corresponds to a message B_i and P_ℓ runs $\mathtt{sort}(\mathsf{list}_\ell)[a] = u'_{j\to\ell}$ which corresponds to block B_j. P_k constructs a clock $\boldsymbol{\mathcal{C}_k} = [u'_{i\to k}, \mathcal{G}^{\Sigma, P_k}_{\mathsf{timer}}]$ as soon as receiving n messages when $\mathsf{timer}^*_k = t_k$ and P_ℓ constructs a clock $\boldsymbol{\mathcal{C}_\ell} = [u'_{j\to\ell}, \mathcal{G}^{\Sigma, P_\ell}_{\mathsf{timer}}]$ as soon as receiving n messages when $\mathsf{timer}^*_k = t_\ell$. We analyze the absolute difference of the momentary clocks of parties just after all parties construct their clock as if no clock drift exists between the first clock construction and the last clock construction. Given that the momentary clock of P_k is $\mathcal{C}_k = [u_{i\to k}, t_k]$ and the momentary clock of P_ℓ is $\mathcal{C}_\ell = [u'_{j\to\ell}, t_\ell]$, the absolute difference of them is $|(t_k - u'_{i\to k}) - (t_\ell - u'_{j\to\ell})|$ (See Definition 3.4). Assume that when B_j arrives to party P_ℓ, the real timer of P_i (timer^*_i) is $t_{j\to\ell^k}$. Therefore, $u_{j\to\ell^k} = t_{j\to\ell^k} - r_j T$. We know that the real time difference $t_\ell - u'_{j\to\ell} - \Delta(\mathsf{timer}_j, t'_{j\to\ell}, t_\ell)$ is equal to the real time difference $t_k - u'_{j\to\ell^k} - \Delta(\mathsf{timer}_k, t'_{j\to\ell^k}, t_k)$. So,

$$\begin{aligned}
|(t_k - u'_{i\to k}) - (t_\ell - u'_{j\to\ell})| &= |t_k - u'_{i\to k} - (t_k - u'_{j\to\ell^k} - \Delta_k + \Delta_\ell)| \\
&= |u'_{j\to\ell^k} - u'_{i\to k} + 2\Delta_{\mathsf{sid}} \text{ (Since } \max(\Delta_k) = \Delta_{\mathsf{sid}}, \min(\Delta_k) = -\Delta_{\mathsf{sid}})| \\
&\leq |u_{j\to k} + \delta + \Sigma[t_{j\to k}] - u_{i\to k} - \Sigma[t_{i\to k}] + 2\Delta_{\mathsf{sid}}| \\
&\leq |u_{j\to k} - u_{i\to k} + \delta + \lceil \frac{D}{T} \rceil \Delta_{\max} + 2\Delta_{\mathsf{sid}}| \qquad (4)
\end{aligned}$$

Hence, using inequalities (2) and (3), we can conclude that

$$|\mathcal{C}_k - \mathcal{C}_\ell| \leq \begin{cases} \delta + \lceil \frac{D}{T} \rceil \Delta_{\max} + 2\Delta_{\mathsf{sid}} & \text{if } i = j \\ 2\delta + 2\lceil \frac{D}{T} \rceil \Delta_{\max} + 2\Delta_{\mathsf{sid}} & \text{if } i \neq j, B_j \preceq B_i \text{ in } \mathsf{list}_k \text{ and } B_i \preceq B_j \text{ in } \mathsf{list}_\ell \\ 3\delta + 3\lceil \frac{D}{T} \rceil \Delta_{\max} + 2\Delta_{\mathsf{sid}} & \text{if } i \neq j, B_j \preceq B_i \text{ in } \mathsf{list}_k \text{ and } \mathsf{list}_\ell \end{cases}$$

If $j = i$, then $|(t_k - u'_{i \shortmid k}) - (t_\ell - u'_{j \shortmid \ell})| \leq \delta + \lceil \frac{D}{T} \rceil + 2\Delta_{\max}$. If $j \neq i$, it means that the order of B_i and B_j

□

This lemma shows us that if we can construct a protocol where parties agree on the messages with timestamps and construct a new clock with respect to the order of messages ($\preceq$) as described in Lemma 3.1, then the difference of the clocks is bounded even if there is no synchronization at the beginning. Therefore, we construct our protocol in the next session on top of a blockchain protocol which make parties agree on messages without any extra communication overhead.

4 Realization of Consensus on 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. [16] 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 [8, 22, 24]. We note that these rounds do not have to advance based on the measurement of time as our clocks.

Definition 4.1 (Common Prefix (CP) Property [16]). *The CP property with parameters $k \in \mathbb{N}$ ensures that any blockchains $\boldsymbol{B_1}, \boldsymbol{B_2}$ owned by two honest parties at the onset of rounds $r_1 < r_2$ satisfy that $\boldsymbol{B_1}$ without the last k blocks is the prefix of $\boldsymbol{B_2}$.*

In other words, the CP property ensures that blocks which are k blocks before the last block of an honest party's blockchain will be always in the blockchain that is owned by an honest party. We call all these stable 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 4.2 (Chain Density (CD) Property). *The CD property with parameters $s_{cd} \in \mathbb{N}, \mu \in (0, 1]$ ensures that any portion $\boldsymbol{B}[r_u : r_v]$ of a final blockchain $\boldsymbol{B}$ spanning between rounds r_u and $r_v = r_u + s_{cd}$ contains at least $|\boldsymbol{B}[r_u : r_v]|\mu$ honest blocks where $|\boldsymbol{B}[r_u : r_v]|$ is the number of blocks of $\boldsymbol{B}[r_u : r_v]$.*

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 ConsensusClock in $\mathcal{F}^{T,\Theta}_{\mathsf{C\_Clock}}$ that defines consensus clock in our protocol. ConsensusClock receives momentary clocks of multiple parties as an input, and outputs one of them as a consensus on clock. In our setup, we say that two clocks are *synchronous* if their absolute difference is Θ_p according to Definition 3.5. For example, assume that Θ_p 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 momentary clocks $\mathcal{C}_j = [t^*_j, t_j]$, ConsensusClock first finds the pairwise difference of all honest clocks $\boldsymbol{\mathcal{C}_j}$ as defined in Definition 3.5. If there exists two honest clock with the difference greater than κ (no synchronization exist), ConsensusClock outputs $\perp$ meaning that no consensus exists. Otherwise, it selects the median clock as consensus clock as described in Algorithm 2.

Algorithm 2 ConsensusClock($\{\mathcal{C}_j\}$) where $\mathcal{C}_j = [t^*_j, t_j]$

```
1: $\mathsf{relative\text{-}lst} \leftarrow \emptyset$
2: for all $\mathcal{C}_u, \mathcal{C}_v$ where $\mathcal{C}_u \neq null, \mathcal{C}_v \neq null$ and $P_u, P_v \in \mathcal{P}_h$ do
3:     if $|\boldsymbol{\mathcal{C}_u} - \boldsymbol{\mathcal{C}_v}| > \Theta_p$ then
4:         return $\perp$
5: for all $\mathcal{C}_j \neq null$ and $P_j \in \mathcal{P}_h$ do
6:     $\mathsf{relative}_j = (t_j - t^*_j)$ // the relative time since the start of $\boldsymbol{\mathcal{C}_j}$
7:     add $\mathsf{relative}_j$ to $\mathsf{relative\text{-}lst}$
8: $\mathsf{relative}_i \leftarrow$ Median($\mathsf{relative\text{-}lst}$) // Median sorts the input list and returns the median
9: return $\mathcal{C}_i$
```

4.3 Relative Time Protocol

The relative time protocol realizes $\mathcal{F}^{T,\theta}_{\mathsf{C\_Clock}}$ (See Figure 4) meaning that it let's parties to obtain a clock which is close enough to the consensus on clock. We build our protocol on top of a blockchain protocol where parties produce blocks when it is their turn. In more detail, we consider a blockchain protocol consisting of rounds where the length of each round is defined with the T ticks by $\mathcal{G}_{\mathsf{refRate}}$: After the genesis block is released, the round 0 starts. At the same time, each party constructs their local clock $\boldsymbol{\mathcal{C}} = [t^*, \mathcal{G}^{\Sigma,P}_{\mathsf{timer}} \to t]$ to count the rounds locally where t^* is its timer value when he receives the genesis block. Thus, in every T tick of their timer according to $\mathcal{G}^{\Sigma,P}_{\mathsf{timer}}$ (See Figure 2), they move to the next round. The blockchain protocol has a selection mechanism which tells parties at which round they are supposed to produce a block and send it with $\mathcal{F}^{\delta}_{\mathsf{DDiffuse}}$. Selected parties produce blocks only when their round starts according to their timer. Whenever they produce a block they also add their local round number 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 most of the PoS blockchain protocols [23, 13, 4, 18, 12] fit this abstraction. There are two missing mechanism in such a blockchain protocol to preserve the security and correctness.

1. The mechanism that lets newly joining parties after the genesis block construct a clock which is close to other clocks of existing parties.
2. Even if the first mechanism exist, we need another mechanism that do not let the clocks drift apart. If we do not have this mechanism, the difference between clocks cannot be bounded because of unbounded drifts on timers.

Our relative time protocol provides these mechanisms. In more detail, it works as follows: We divide the blockchain 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 the last finalized block (See Definition 4.1) is r_e. We define r_e as the smallest clock value such that $r_e - r_{e-1} \geq s_{cd}$ where r_{e-1} is the timestamp of the last finalized block in epoch $e-1$. Here, s_{cd} is the parameter of the chain density (CD) property (Definition 4.2). 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 in each epoch. 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 timestamp of B_i by r_i. We note that rounds 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 timestamp r_x where $r_{e-1} < r_x \leq r_e$. Let us assume that there are n such blocks that belong to the current epoch. T Then, P runs the median algorithm (Algorithm 3) which finds prospective initial timer value of each clock of blocks according to his timer and then picks the median of them.

Algorithm 3 MEDIAN($\{t'_i, r_i\}_{i=1}^{n}$)

```
1: list ← ∅
2: for $i = 0$ to $n$ do
3:     store $(t'_i - r_iT)$ to list // prospective initial timer value of clock of the party that produced $B_i$
4: return median(list)
```

Assume that t is the output of the median algorithm. Then, P considers t as a new initial timer value of his clock and updates his clock with $\mathcal{C} = [t, \mathcal{G}_{\mathsf{timer}}^{\Sigma,P} \to t_x]$.

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, we can use the result of Lemma 3.1. Therefore, after each epoch, the difference between the clocks are bounded. 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:

In the below theorem, the time difference is at most D between sending two different block according to $\mathcal{G}_{\mathsf{refRate}}$, $\Delta_{\max} = \max\{\Delta(\mathsf{timer}_i, t, t+T) : \mathcal{G}_{\mathsf{timer}}^{P_i,\Sigma} \to t\}$, and timer drift from the start of the session and end of the session is between $[-\Delta_{\mathsf{sid}}, \Delta_{\mathsf{sid}}]$, $\Theta_c = 2\delta + |\Sigma|$. We let $M = 3\delta + 3\lceil\frac{D}{T}\rceil\Delta_{\max} + 2\Delta_{\mathsf{sid}}$.

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 $s_{cd}, \mu > 0.5$ as long as the maximum difference between honest clocks is $M + 2\Delta_{\mathsf{sid}}$ and $\Theta_p = M, \Theta_q = M + \delta + \Delta_{\max}\lceil\frac{\delta}{T}\rceil + \Delta_{\mathsf{sid}}$, the relative time protocol in $\mathcal{F}_{\mathsf{DDiffuse}}^{\delta}$ and $\mathcal{F}_{\mathsf{timer}}^{T,\Sigma}$-hybrid model realizes $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ 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 $\mathcal{F}_{\mathsf{DDiffuse}}^{\delta}$. Therefore, it first registers $\mathcal{G}_{\mathsf{refRate}}$ to emulate it. The simulation is straightforward. As soon as $\mathcal{S}$ receives momentary clocks $\mathcal{C}_i = [t_i^*, t_i]$ or null clocks of each honest party P_i from $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$, $\mathcal{S}$ simulates honest parties in the underlying blockchain protocol during one epoch decided by $\mathcal{Z}$. Given that the first round of the epoch is r, $\mathcal{S}$ adjust the clocks of honest parties by preserving their difference according to round r. For this, $\mathcal{S}$ chooses one of the momentary clock $\mathcal{C}_j = [t_j^*, t_j]$ and finds the start timer value of round r according to $\boldsymbol{\mathcal{C}_j}$ which is $t_j^* + rT$. Then, it obtains $D_r = (t_j^* + rT) - t_j$ which is the distance of momentary clock of P_i to the start of the round r. In the end, it adjusts all honest clocks $\boldsymbol{\mathcal{C}_i}$ which are not null by constructing clocks $\boldsymbol{\mathcal{C}_i^r} = [t_i^*, D_r + \mathcal{G}_{\mathsf{timer}}^{P_i,\Sigma} \to t_i]$ and simulates the honest parties according to these clocks in the blockchain protocol. Remark that the adjusted clocks preserve the difference of honest clocks. $\mathcal{S}$ needs to adjust them to simulate the honest parties in the block production mechanism of the underlying blockchain protocol in the current epoch. $\mathcal{S}$ registers the honest parties who has a clock to contribute the block production mechanism in the current epoch and learns the rounds of each registered honest party that they are supposed to produce a block. $\mathcal{S}$ simulates the block production for each honest party P_j who does not have a null clock as in the protocol with respect to adjusted clock $\boldsymbol{\mathcal{C}_j^r}$. For this, it checks the momentary clock updates $[t_i^*, t]$ given by $\mathcal{F}_{\mathsf{C\_Clock}}^{T,\Theta}$ for each $P_j \in \mathcal{P}_h$ and updates $\mathcal{C}_i^r = [t_i^*, D_r + t]$. accordingly. It produces a block when $\mathsf{Clock\_Val}(D_r + t, T, t_i^*, 0)$ is equal the round of P_j. After producing the block, $\mathcal{S}$ sends the block of P_j to $\mathcal{A}$ (since $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{DDiffuse}}^{\delta}$ 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. 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 t_j which is the current timer value in $\mathcal{C}_j^r = [t_j^*, t_j]$ when a block arrives from any other party. Remark that the difference of arrival time of a block is at most δ according to $\mathcal{G}_{\mathsf{refRate}}$. Therefore, this difference can be at most $\delta + |\Sigma|$ according to a timer of an honest party. In the end of the epoch, $\mathcal{S}$ runs the Median algorithm (Algorithm 3) and updates the clocks of honest parties accordingly. $\mathcal{S}$ sends the updated clocks of honest parties and adversarial clocks given by $\mathcal{Z}$. Finally, $\mathcal{S}$ outputs the clocks of honest parties. We note that since our CONSENSUSCLOCK algorithm does not use the adversarial clock when computing the consensus on a clock, the outcome of parties does not change based on the adversarial clocks.

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 ConsensusClock 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 ConsensusClock (Algorithm 2), the consensus on clocks exists if and only if the difference between honest clocks is at most Θ_p. Therefore, if we show that a consensus on clock exists after the update then we also show that the difference between clock of honest parties and the consensus is at most Θ_p. Since $\mathcal{F}_{\mathsf{timer}}^{T,\Theta}$ gives the initial clock of parties, it means that the consensus exists with the initial clock. If the consensus exists, the initial clocks of honest parties are synchronized. It means that initial clock difference of honest parties is at most Θ_p. During the simulation of an epoch the difference between initial clocks becomes at most $\Theta_p + 2\Delta_{\mathsf{sid}}$. Since as long as the difference between clock of parties is $\Theta_p+2\Delta_{\mathsf{sid}}$, **CP property is not broken during an epoch** except with probability p_{cp}. Thanks to the CP property, all honest parties run the median algorithm with the arrival time of the same blocks. It means that list in Algorithm 3 includes the prospective initial timer value of clocks of the same blocks. Therefore, the difference between the new clocks is at most M thanks to Lemma 3.1 i.e., for all $P_i, P_j \in \mathcal{P}_h$, $|\boldsymbol{\mathcal{C}_j} - \boldsymbol{\mathcal{C}_i}| \leq M \leq \Theta_p$. Therefore, the consensus exists in $\mathcal{F}_{\mathsf{timer}}^{T,\Theta}$ with the new clocks.

(2. Case) Since the clock difference of honest parties is at most $M + 2\Delta_{\mathsf{sid}}$ during the simulation, 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 $\bar{\boldsymbol{\mathcal{C}}}$ and the consensus clock $\bar{\boldsymbol{\mathcal{C}}}_{\boldsymbol{h}}$ 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. The difference between clocks before starting the simulation is at most M. So, the difference of initial clocks with the initial consensus clock $\bar{\boldsymbol{\mathcal{C}}}_{\boldsymbol{h}}$ is at most M. Remember that the values in list_k is the initial timer value of the clock of a block according to the view of P_k. Assume that the corresponding timer values of initial timer values of initial clock is in $[t_1^*, t_m^*]$ on a timer of a party where m is the number of honest clocks. In the end of the epoch, list_k of P_k includes number of n candidate initial timer of honest clocks which are in $[t_1^* - \Delta_{\mathsf{sid}}, t_m^* + \delta + \Delta_{\max}\lceil \frac{\delta}{T} \rceil + \Delta_{\mathsf{sid}}]$. The addition $\delta + \Delta_{\max}\lceil \frac{\delta}{T} \rceil + \Delta_{\mathsf{sid}}$ to t_m^* comes from the fact that the latest initial timer value drifts at most Δ_{sid} and the blocks sent with respect to its timer arrives to a party P_k at latest $\delta + \Delta_{\max}\lceil \frac{\delta}{T} \rceil$ ticks later according to timer of P_k. Similarly the earliest initial timer value drifts backward at most Δ_{sid} so it increases the difference between the first and latest initial timer value Δ_{sid} more. Since there are at least $\lfloor \frac{n}{2} \rfloor + 1$ timer values corresponding to honest clocks, the median algorithm outputs one of this. The initial clock value of $\bar{\boldsymbol{\mathcal{C}}}_{\boldsymbol{h}}$ is in $[t_1^*, t_m^*]$ and initial clock value of a new clock is in $[t_1^* - \Delta_{\mathsf{sid}}, t_m^* + \delta + \Delta_{\max}\lceil \frac{\delta}{T} \rceil + \Delta_{\mathsf{sid}}]$. Therefore, the difference of a new clock and $\bar{\boldsymbol{\mathcal{C}}}_{\boldsymbol{h}}$ is at most $M + \delta + \Delta_{\max}\lceil \frac{\delta}{T} \rceil + \Delta_{\mathsf{sid}} = \Theta_c$. Since the new consensus clock is one the new clocks $|\bar{\boldsymbol{\mathcal{C}}} - \bar{\boldsymbol{\mathcal{C}}}_{\boldsymbol{h}}| \leq \Theta_c$.

(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 3 is never empty, so new clocks are never null.

□

5 Conclusion

In this paper, we constructed a GUC model that models the real world clocks and lets clocks preserve synchronization among them without a reference clock. Our model is the first GUC model that captures the notion of consensus on clock. It can be used in decentralized protocols that requires synchronization between physical clocks for the security and completeness.

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. The difference between clocks of parties can be at $3\delta + 3\lceil \frac{D}{T} \rceil \Delta_{\max} + 2\Delta_{\mathsf{sid}}$ where δ is the maximum network delay and Δ values are clock drifts between different intervals. The difference between old clocks and new clocks can be at most $\Theta_c = 4\delta + 4\Delta_{\max}\lceil \frac{\delta}{T} \rceil + 3\Delta_{\mathsf{sid}}$. It means that the protocol can stop at most Θ_c ticks.

We collected the clock drift data of 32 full nodes of Polkadot blockchain network [39] to analyze clock drifts in the real world clocks of full nodes. NTP collects these data and stores them in the `/var/lib/ntp/ntp.drift file`. The data in the drift file of each full node measuring parts per million (ppm). We converted this data to drifts in one day in terms of second as shown in Figure 5. The graph in Figure 5 shows this data during one day. As it can be seen, the clock drift does not change dramatically during a day because it depends on the frequency error of the timer rate which is bounded. The most drifted clock drifts around 2 second per day according to Figure 5. If one epoch takes one hour then we can say that the clock drift in one epoch which Δ_{sid} is 0.083 second which is very small comparing to network delay δ. Therefore, we can say that the difference between clocks in our protocol is directly related with the network delay since the drifts in a short time is negligible.

References

[1] Second (s or sec).

[2] 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.

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

[4] 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.

[5] 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.

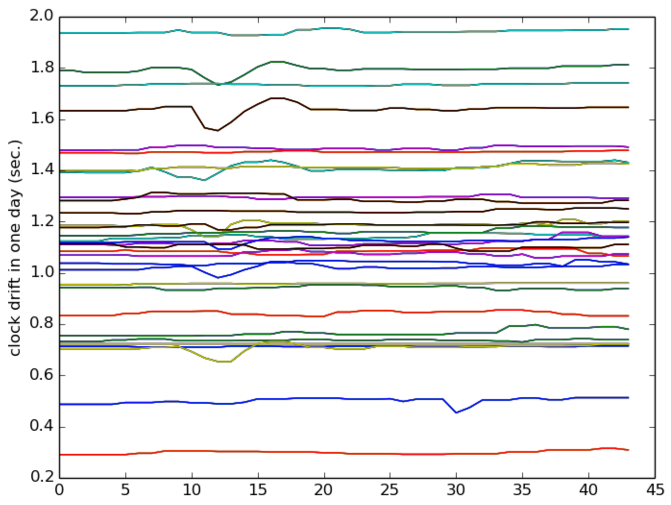

Figure 5: Clock drifts of 32 full node in one day in terms of second

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

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

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

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

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

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

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

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

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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