Chapter 2

Time in Distributed Systems

> *A man with a watch knows what time it is. A man with two watches is never sure.*

Segal's law

Time and its measurement are very fundamental physical concepts. Ancient civilizations too practiced various ways of representing and measuring time with varying degrees of accuracy. People use the notion of time in many tasks including preparing their daily schedules, coordinating various events, ordering of events, and so on. It is important to note that in all of these applications, we have some inherent notion of how we benefit from knowing the time. Often, we are less concerned with the exact time and are more focused on whether the current time is before or after a particular event. In general, in distributed systems we may not have access to synchronized physical clocks. In such cases, we are more interested in the cause-effect relationships between events. We would like to effect to always appear to happen after the cause. The relationship between a cause and an effect is also known as *causality*, which is of vital significance to us in asynchronous distributed systems (refer to Chapter 1 for a definition).

When we move to computer systems, even one second of physical time is a long time in the processor's time scale. It corresponds to a window in which a large number of actions can be realized. In fact, if we wish to specify actions at the instruction level, we need to specify time in terms of nanoseconds. We shall see that often there is no need to do so and representing the happens-before relationships (causality in some cases) between events is good enough. In the following chapters, we will explore how the concepts of time and

causality play a crucial role in computer systems, and in distributed systems specifically.

Let us quickly see how the notion of "time" is used in computer systems.

- Assigning Timestamps: For many events that take place in a computing system, the operating system assigns a timestamp. Examples include the time at which a file is last accessed or written to.
- Performance and Resource Usage: Timestamps offer a way for the system to monitor the performance of various applications and also track the usage of various resources by processes.
- Timeouts: Some user programs may need to stop after a certain time duration has elapsed. This requires the program to know the time.
- Scheduling and Statistics: The operating system may use time information in process scheduling. Further, the operating system may use time information to keep track of the system time assigned to each process, the overall time each process is running for, tracking the login information of users and maintaining other types of bookkeeping information.
- Event Ordering: Timestamps are a way to arrive at a monotonic ordering of events that happen in the system.

From the above discussion, we note that one of the fundamental activities in a computer is the measurement of time. In the physical world, 1 second is measured as the time it takes the Cesium 133 atom to make exactly 9,192,631,770 transitions. Atomic clocks rely on this phenomenon. However, most computers keep track of time by counting the number of oscillations of a quartz crystal. Specifically, one second on a computer corresponds to 32,768 oscillations of a quartz crystal. Sadly, the oscillation frequency of quartz crystals tends to change with changes in temperature, voltage and even aging. Without frequent calibration, it can vary by 2-5 ppm [124], which can translate to a second lost every 7-11 days. Therefore, unless corrected, the time maintained by a machine (laptop, desktop or server) can differ from the actual clock time, and the errors can also accumulate over time. If this is the case with a single machine, then maintaining globally synchronized time across multiple machines becomes extremely challenging.

Let us first start our study with clock synchronization algorithms that try to maintain a single clock time across a cluster of machines. We need to send frequent messages between them to ensure that no single machine drifts away.

2.1 Clock Synchronization

Having a synchronized clock in a distributed system is extremely beneficial. A clock synchronization algorithm refers to a mechanism that is used to align the clocks of nodes in a distributed system such that every node in the system is sure that the maximum clock drift (across the nodes) is limited by a threshold Δ.

2.1.1 Synchronization of Physical Clocks: Overview

It turns out that maintaining synchronized physical time across computers on a network is unusually difficult. Some of the difficulties arise from the fact that the local time at each node can drift due to varying ambient conditions, network delays of communication links can be unpredictable and possibly unbounded, and nodes themselves or communication links can fail. The lack of access to global knowledge limits the decisions that nodes can take with respect to correcting their clocks. They are limited to using local knowledge. These problems carry over and get amplified in a distributed system. There are additional challenges such as the lack of any centralized control and lack of limits on physical distances between the nodes.

This situation calls for the design of efficient algorithms and protocols that address how nodes in a network can efficiently *synchronize* their clocks. A lot of the foundational work was done by Marzullo [93] in his PhD thesis (published in 1984). there have been a sequence of developments in this direction.

Algorithms for this problem use a range of techniques including peer-to-peer exchange, and master-slave models for synchronizing time across nodes in a network. It is easier to solve such problems when there is a centralized server. The quintessential technique, in both the cases, is for a pair of nodes to estimate the average transmission delay and their clock offset by using a sequence of ping and reply messages. For obvious reasons, algorithms that assume an upper bound on the transmission delay are usually simpler than those that do not make such assumptions.

Other considerations that play a role in the design of these algorithms include the size and extent/spread of the network, the desired accuracy of the synchronization, and the auxiliary support available. There are theoretical limits on the accuracy that can be achieved by *any* algorithm. Specifically, Lundelius and Lynch [89] show that a system with n clocks cannot be synchronized with synchronization error less than $(max - min) \cdot (1 - \frac{1}{n})$, where max and min denote the maximum and minimum transmission delays across

the nodes in the network, respectively. Practical limits on the accuracy depend on multiple factors including the network conditions and the accuracy of the clocks at the individual nodes.

Most of the algorithms for time synchronization employ a range of techniques for fault tolerance. For instance, in a classical ping and reply interaction, if one side does not respond within a reasonable time period, the other side will typically call this round of interaction as unsuccessful and ignores the measurements from such unsuccessful rounds. Given the uncertainties in message transmission and other potential pitfalls such as failed or inaccurate clocks, another common technique is to discard all measurements that show significant deviations.

Recent Developments

There are two notable developments related to time synchronization in distributed systems. First, use cases such as distributed transactions, financial transactions and distributed databases have rekindled interest in synchronizing the time in a distributed system. Such applications require a degree of precision that is of the order of nanoseconds [47]. These use cases led to the design of physical clock synchronization algorithms that used novel techniques from signal processing and estimation theory to achieve better accuracy.

GPS systems, and the like. In this direction, we describe one such protocol, the Google TrueTime protocol that aims to emulate the solution on a network to the distributed system scale.

Finally, we discuss a solution that combines aspects of logical time and physical time, Hybrid Logical Clocks (HLC), to arrive at a simpler way of maintaining time in a distributed system.

2.2 The TEMPO Protocol

One of the early algorithms/protocols to maintain the time in a local network is the TEMPO protocol proposed by Gusella and Zatti [59]. This algorithm adjusts the time of the day of the devices in a local network. Based on this algorithm, Berkeley UNIX (version 4.2 BSD) introduced a system call `adjtime()` that allowed the user to set the time on a system.

2.2.1 The Algorithm

Consider a situation where processes A and B are running on two different nodes in the network. We want to estimate the clock skew between the corresponding machines. Process A sends a message to process B. On receiving such a message, process B sends a reply to process A with a timestamped message. Before sending the message, process B computes the time taken for the message from A to reach B as:

$$D_1 = t_B - t_A \tag{2.1}$$

Notice however that the clocks of A and B may themselves be adrift by Δ_A and Δ_B, respectively. In other words, $\text{timestamp}_A = t_A - e_A$ and $\text{timestamp}_B = t_B - e_B$. Using the above two, and denoting the transmission delay from A to B as T_{AB}, we rewrite Equation 2.1 as:

$$D_1 = t_{B1} - e_{B1} - t_{A1} + e_{A1} = T_1 + \delta - \text{Error}_1 \tag{2.2}$$

In Equation 2.2, the quantity T_1 represents the transmission delay from A to B and δ refers to the offset we are trying to estimate. We also set $\text{Error}_1 = e_{B1} - e_{A1}$.

As B sends a reply to process A, the above calculations can be repeated at process A to obtain the following.

$$D_2 = (t_{A2} - t_{B2}) - (e_{A2} - e_{B2}) = T_2 - \delta - \text{Error}_2 \tag{2.3}$$

Using Equations 2.2, 2.3, process A can also compute the difference in the transmission delay between A to B and B to A as:

$$\Delta' = \frac{D_1 - D_2}{2} = \delta + \frac{T_1 - T_2}{2} - \frac{\text{Error}_1 - \text{Error}_2}{2} \tag{2.4}$$

We now see how we can repeat the above steps to estimate δ. Assume that the random variables corresponding to Error_1 and Error_2 are independent and symmetric. We can repeat the above experiment N times so that the average Δ' can be computed as follows.

$$\begin{aligned} \Delta' &= \frac{\sum_{i=1}^{N} D_{1i} - D_{2i}}{N} \\ &= \delta + \frac{\sum_{i=1}^{N} \frac{T_{1i} - T_{2i}}{2}}{N} - \frac{\sum_{i=1}^{N} \frac{\text{Error}_{1i} - \text{Error}_{2i}}{2}}{N} \end{aligned} \tag{2.5}$$

Notice from Equation 2.5 that the second and the third terms in the right hand side have a mean of 0. Hence, for large N, by appealing to the Strong Law of Large Numbers, the right hand side converges to δ.

This observation can now be used to design the protocol that computers on a local network use to synchronize their clocks. The TEMPO protocol designates one computer in a local network as a *master* and the rest as *slaves*. The master is responsible for initiating and coordinating time synchronization. The master uses the following steps.

1. The master interacts with each of the slave machines, say $S_1, S_2, \cdots$, and obtains estimates of $\Delta'_{S_1}, \Delta'_{S_2}, \cdots$, with respect to each of the slave machines.

2. The master then computes the average of the quantities, $\Delta'_{S_1}, \Delta'_{S_2}, \cdots,$, as the network average error.

3. The master directs slave machine S_i to adjust its clocks by an offset equal to the difference of Δ'_{S_i} and the network average error.

Notice that the above protocol requires some slave machines to set their time to be in the past. This can create situations that are not consistent.

2.2.2 Fault Tolerance

The TEMPO protocol is designed with simple fault tolerance. The failure of a slave to communicate with the master within a specified timeout interval lets the master conclude that the slave machine non-operational. Similarly, if the slave machines do not receive any message from the master within a specified time, they conclude that the master machine is unavailable and start the procedure to elect a new master.

2.3 The Network Time Protocol (NTP)

The Network Time Protocol (NTP) is described in RFC 958 [98]. This protocol indicates how computers on a network should periodically set their time so that all computers on a network have a consistent notion of the physical clock time. To this end, the protocol organizes computers into two sets of entities: Clients and Servers. Clients seek time from a server within the same network and are thus passive in the protocol. Servers are arranged into multiple strata. Servers within a strata are referred to as *peers*.

Stratum 0, the highest level of strata, consists of the most accurate set of time servers, these are also called as *reference clocks*. These use devices such as GPS clocks, atomic clocks, and the like and are highly accurate.

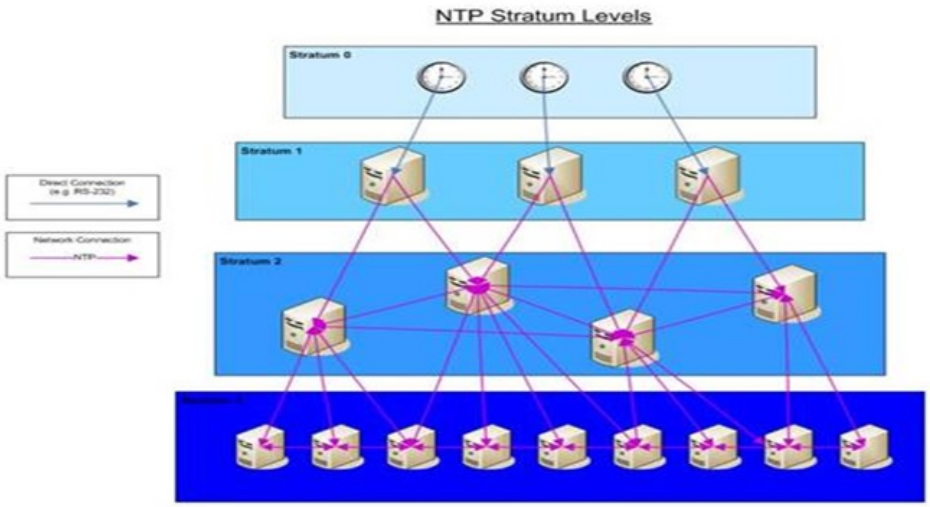

Figure 2.1: A set of servers arranged into various strata in the NTP protocol. Figure borrowed from ntpserver.wordpress.com .

Devices in Stratum 1 have a direct connection to one or more devices in stratum 0 in addition to having connections to their peers in the same stratum. Devices in this stratum are also known as primary clocks or primary time servers. They might be few microseconds in offset compared to the time in Stratum 0 clocks. These primary servers can also work as fallback clocks to sync up with each other just in case stratum 0 servers are unavailable.

Devices in Strata 2 and 3 synchronize with devices in the next higher strata and with those in the same strata. Figure 2.1 shows a typical set up of servers into strata.

There are two types of communication between the servers to obtain the current time: one set of communications between the peers, and one from devices in one strata to those in a higher strata. Since servers across strata and also those within a strata may have differences in clocks, and these differences can grow arbitrarily over time, an initial set of synchronization may not be enough. Therefore, peers in the NTP protocol exchange periodic messages to (re)set their local time. The interaction between two peers A and B is characterized by the following messages.

Peer A sends a message to peer B and records the time that the message is sent as T_1. Peer B records the time at which the message is received

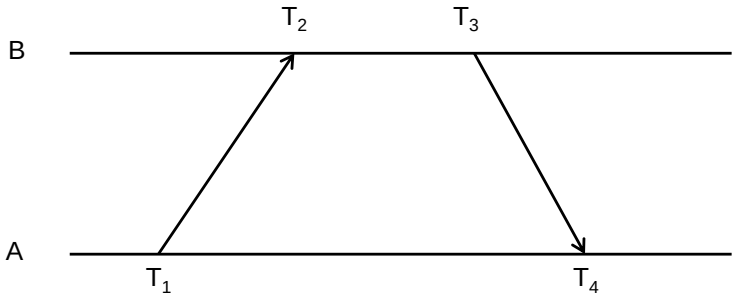

Figure 2.2: A pair of messages exchanged by A and B with send and receive times at A and B recorded at respective machines.

at B as T_2 and sends an acknowledgement to peer A at time T_3. This acknowledgement is received at peer A at time T_4. Each such message and acknowledgement is called as a trial . Figure 2.2 shows these four messages across A and B.

For each trial, the clock offset θ and the round trip delay δ of peer B relative to peer A at time T_4 are approximately given by the following. Measured at peer B, $T_2 = T1 + \delta/2 + \theta$, and $T_3 = T4 - \delta/2 + \theta$. Add the two equations to get $2\theta = a + b$ with $a =$ and $b =$. The round-trip delay $\delta = T_3 - T_1 + T_2 - T_4 = a - b$.

Each peer maintains pairs (O_i, D_i), where O_i is a measure of the offset (θ) and D_i is a measure of the transmission delay of two messages (δ). The eight most recent pairs of (O_i, D_i) are retained. The value of O_i that corresponds to the minimum D_i is chosen The offset corresponding to the minimum delay is chosen as the offset O.

2.3.1 Limitations of NTP

Notice from the above that by having a set of continuously interacting peers, it is possible to have a physical time that is synchronized across computers. The typical errors in synchronization are small. However, in a large scale distributed system, the typical errors can go up to hundreds of milliseconds.

NTP works on the premise that all the systems can be arranged in a hierarchy and the required peering support can be set up. This assumption is difficult to maintain in a fully distributed system. Even if such a hierarchy and peering support is created, the typical physical distances can potentially involve large round trip times resulting in reduced accuracy of the agreed time. In particular, the accuracy using NTP in networked systems is of

the order of few milliseconds. This error can increase to a few hundreds of milliseconds in rare cases. (See the experiment suggested in Question ???). This accuracy compares very unfavorably to running NTP on a LAN which can produce accuracy to the order of less than a millisecond. A global scale distributed system may not be able to function correctly using physical time that has errors of the order of milliseconds or hundreds of milliseconds. These reasons indicate that NTP in its present form is not usable in distributed systems.

Nevertheless, distributed systems too need to measure time for a variety of reasons. Some of these are listed below.

- Distributed Transaction Processing: For instance, measuring time and assigning timestamps is important in transaction processing.
- Distributed Algorithms: Timestamps may also be essential in applications such as mutual exclusion. In some situations, distributed systems use the notion of timeout to abort actions. This requires measuring elapsed time.
- Tracking Dependencies Among Events:
-

Given the above requirements and the difficulty of adopting the NTP approach, what do distributed systems do to maintain time? It turns out that by their nature, distributed systems make progress in spurts and one way to measure time is to understand how various events in a distributed system can be ordered at least in a partial order. This partial order, also known as causality, will then serve as a way of measuring time.

The solutions that distributed systems adopt vary from not using physical time, creating a robust distributed scale NTP-like framework, and using a combination of these two approaches. In the following, we will describe these three approaches in that order.

2.4 Logical Time

In an seminar paper, Lamport [82] showed that a partial ordering of events in a distributed system can provide a mechanism to set the notion of time and its measurement in a distributed system. Such a mechanism allows distributed systems use what is known as logical time instead of physical time.

Logical time is a mapping from events in a distributed system to a time domain. The events in a distributed system correspond to local computation, message send, and message receive. The distributed system in action is thus a system of computers processing events. Events within a machine and those across machines do have causal dependencies of the following kind. An event corresponding to a message send at a machine P has to happen *before* the message is received at machine Q. These causal dependencies induce a partial order on the set of events across machines in a distributed system.

Logical time aims to capture the partial dependencies between events. To make the description formal, let us denote by E_i the set of events happening at processor P_i for $i = 1, 2, \cdots,$. Let $E := \cup E_i$ denote the entire set of events happening at all processors. Let us also number the events that happen at P_i for $i \geq 1$ such that an event e_i^x occurs before e_i^y if $x < y$, for positive integers x and y. In other words, we say that at P_i event e_i^x happens before event e_i^y. Similarly, if P_i and P_j are two different processors, and P_i sends a message to P_j, the corresponding send event must happen before the receive event. We used $\rightarrow_{msg}$ to denote this relation. Finally, we use the binary relation $\rightarrow$ across pairs of events and is defined as follows. Consider any two events e_i^x and e_j^y happening at processors i and j.

$$e_i^x \rightarrow e_j^y \iff \begin{cases} i = j \text{ and } x < y, & \text{or} \\ e_i^x \rightarrow_{msg} e_j^y, & \text{or} \\ \exists e_k^z \in E \text{ such that } e_i^x \rightarrow e_k^z \text{ and } e_k^z \rightarrow e_j^y \end{cases}$$

Notice from the above definition that indeed the relation $\rightarrow$ is a partial order. There could be events that are not related according to $\rightarrow$. We say that for two events e_1 and e_2, if $e_1 \rightarrow e_2$ or vice versa then the two events causally affect each other. Otherwise, the two events are said to *logically concurrent*. We use $e_1 \parallel e_2$ to indicate that the two events are logically concurrent.

We now build on the relation $\rightarrow$ between pairs of events to define the notion of logical time. Just like physical time, logical time also assigns a timestamp to each event. These timestamps ensure that if an event e_1 that causally affects another event e_2, then the timestamp of e_1 is smaller than that of e_2. One can infer the causality between events by comparing the (logical) timestamps.

A *logical clock* $\mathcal{LC}$ is a function that maps an event e in a distributed system to an element in the time domain T, denoted as $\mathcal{LC}(e)$ and called the timestamp of e, and is defined as a mapping $\mathcal{LC} : E \rightarrow T$ such that the following property is satisfied. For two events e_i and e_j, $e_i \rightarrow e_j \Rightarrow \mathcal{LC}(e_i) <$

$\mathcal{LC}(e_j)$. This monotonicity property is called as the clock consistency property of the logical clock system.

Based on the above idea, it is possible to define several systems of logical time. These systems have the following commonalities. To use the time system, each processor in the distributed system has two data structures. A *local logical clock* denoted lc_i at processor P_i helps P_i measure its own progress. A *logical global clock* denoted gc_i at processor P_i is a way of measuring the view of the logical global time of process P_i. The second of these is required so that P_i can assign consistent timestamps to events that occur at P_i. The logical time system also provides for rules to update the data structures so as to ensure the time consistency condition. In the following, we study the logical scalar time system proposed by Lamport [82].

2.4.1 Logical Scalar Time

In this model, a combined integer c_i represents the data structures lc_i and gc_i at process P_I. The time domain is the set of non-negative integers. There is a common increment d by which the time is incremented as necessary. Process P_i, before executing an event e increments its local time c_i to $c_i + d$. Process P_i also includes its current time on every message that it sends. This piggybacking of the current time, denoted c_{msg}, on each message helps the recipient P_j to update its time as $c_j = \max\{c_j, c_{\text{msg}}\}$. Subsequently, P_j increments its local time c_j by d before delivering the message. These steps ensure that the timestamp of the corresponding message receive event is larger than the timestamp of the message send event. Figure 2.3 shows an example.

It can be noticed that the scalar time as described ensures the consistency property. Extending the notion of consistency, a logical time is said to be strongly consistent if it holds that for any two events e_i and e_j, $e_i \rightarrow e_j \iff \mathcal{LC}(e_i) < \mathcal{LC}(e_j)$. Logical scalar time is however not strongly consistent. Check Figure 2.3 and the events with timestamp 5 and 4 at processes P_1 and P_2, respectively. While $\mathcal{LC}(e_i) < \mathcal{LC}(e_j)$, it does not hold that $e_i \rightarrow e_j$.

Logical scalar time has other properties. If we set the increment $d = 1$ at all times, then an event e with a timestamp of t depends on exactly $t-1$ events in the past. In addition, the scalar time can be used to induce an ordering on events as follows. Consider the tuple (t, i) associated with every event e so that t is the scalar timestamp of e and i is the id of the processor where the event e happens. For two tuples (t_1, i_1) and (t_2, i_2), we define $(t_1, i_1) < (t_2, i_2)$ iff $t_1 < t_2$ or if $t_1 = t_2$ and $i_1 < i_2$. Under this ordering, it holds that for any two events e_1 and e_2, $e_1 < e_2 \rightarrow$, either $e_1 \rightarrow e_2$ or

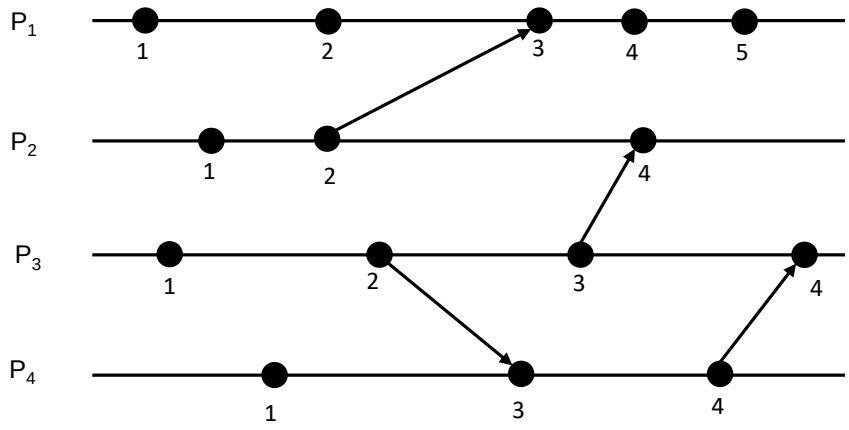

Figure 2.3: A set of four processors using scalar time.

$e_1 \parallel e_2$.

The notion of logical scalar time has found applications in several fundamental problems in distributed computing. For instance, we will see in Chapter 4 that scalar time is useful in designing an algorithm for ensuring mutual exclusion in the distributed setting.

To address the limitation of scalar time that it does not satisfy strong consistency, other logical time systems include the vector time[1] [40, 86, 94] and its extension to matrix time [127]. We briefly review the notion of vector time now.

2.4.2 Logical Vector Time

Logical scalar time uses only one variable to represent both the local logical time at a processor and also the logical global time. Vector time solves this issue by representing time as a vector of n dimensions where n is the number of processors in the distributed system. Let $V_i[1..n]$ denote the vector at process P_i. Each element of the vector is a non-negative integer. The ith component of the vector V_i is the local logical time at processor P_i. In addition, the jth component of V_i represents the latest knowledge of P_i about the local time at process P_j. In particular, if $V_i[j] = x$, then process

[1]It has long been not clear as to who is the first one to propose vector time. See the blog by Kuper `https://decomposition.al/blog/2023/04/08/who-invented-vector-clocks/`

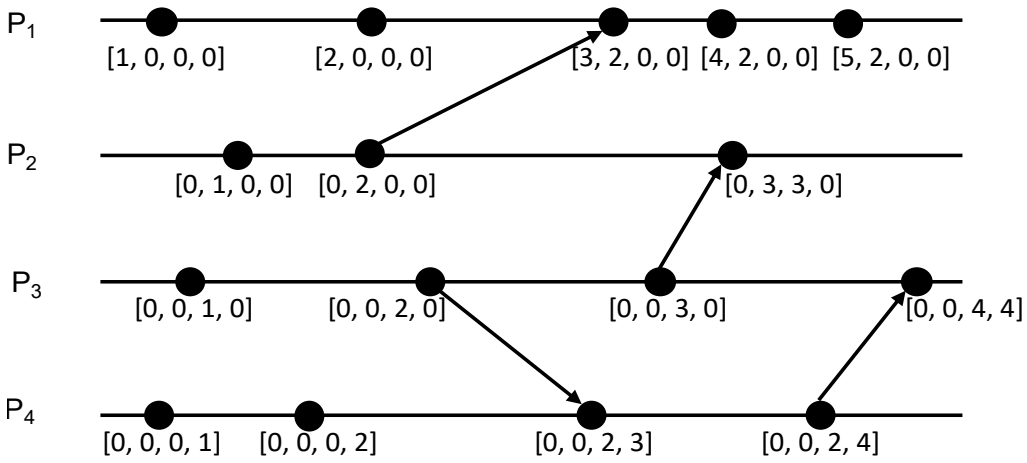

Figure 2.4: A set of four processors using vector logical time.

P_i knows that local time at process P_j has progressed till x. The entire vector V_i constitutes the view of the global logical time at P_i and is used by P_i to timestamp events. The timestamp of an event is now a vector of n dimensions.

When a process P_i executes an event e, it increments $V_i[i]$ by d and assigns e the timestamp of V_i. When a process P_i sends a message to a process P_j, the current time V_i is sent along with the message as V_{msg}. On receipt if the message, process P_j updates V_j as follows. For each $1 \leq k \leq n$, $V_j[k] := \max\{V_j[k], V_{msg}[k]$. This is followed by incrementing $V_j[j]$ by d before delivering the message. This ensures that the event corresponding to the receipt of the message has a larger timestamp than that of the corresponding send event. Figure 2.4 shows an example of using vector time.

To compare two vector time stamps, we use the following rule. Let v and w be two vectors of n dimensions each. We say that $v = w$ if for every $1 \leq i \leq n$, $v[i] = w[i]$. We say that $v \leq w$ if for every $1 \leq i \leq n$, $v[i] \leq w[i]$. We say that $v < w$ if $v \leq w$ and there exists an index i $(1 \leq i \leq n)$ such that $v[i] < w[i]$. Finally, we say that $v \parallel w$ if neither $v < w$ nor $w < v$. As an example, the vector $[1, 3, 4]$ is less than the vector $[1, 5, 6]$. The vectors $[2, 5, 3]$ and $[3, 4, 4]$ are concurrent.

From the definition of vector time, it is apparent that the ith component of the vector clock at process P_i, $V_i[i]$, denotes the number of events that have occurred at P_i until that instant. If an event e has timestamp v, then $v[j]$ denotes the number of events executed by process P_j that causally precede e. Further, $(\sum_{j=1}^{n} V[j]) - 1$ is the total number of events that causally precede

e in the distributed computation.

It can be noted that vector time is strongly consistent. For any two events e and f, it is possible to determine if the events are causally related.

Optimized Implementation of Vector Time

In both of these extensions, the data structures used for representing time increase with the increase in the number of systems. This can pose a limitation trouble especially since the current time is sent on each message. There are some techniques to reduce the overhead of the time information carried on each message. One such technique, due to Singhal and Kshemkalyani [115] is to use two additional information at every process in the form of two arrays. The LastUpdate array corresponds to when each process i last updated its information about the time at a process j, and the LastSent array corresponds to the time when process i last sent its time information to process j. With the help of these two arrays, when process i needs to piggyback a message with its current vector clock to process j, it includes only entries from its vector clock, $vt_i[k]$ such that $\text{LastSent}_i[j] < \text{LastUpdate}_i[k]$. However, these techniques are not generic and work only in a heuristic manner. We refer the reader to [115] for more details on these optimizations.

There are other mechanisms that optimize the amount of time information to be sent as piggyback on messages. Notable ones include the direct dependency work of Fowler and Zwaenepoel [44] and that of Jard and Jourdan [69]. In the direct dependency technique of Fowler and Zwaenepoel [44], processes maintain information pertaining to direct dependence of events on other processes. The actual vector time of an event can be computed offline by using a recursive search on the recorded direct dependencies to account for transitive dependencies. In particular, if an event e_j at a process P_j occurs before the event e_i at process P_i, it can be inferred that events e_0 to e_{j-1} at process P_j also happen before event e_i. In this case, it suffices to record at P_i the latest of the events at P_j the happened before the event e_i. In a similar fashion, P_j would record the latest event that e_j would directly depend on. This recursive trail of dependencies can be used to obtain the vector clock of e_i when needed. One important step in implementing this approach is to make sure that every process updates the dependency information after receiving a message and before it sends out any messages.

More details and the recursive algorithm needed to identify the transitive dependencies are in [44]. Since this technique involves additional computation, its applicability is limited to specific settings such as causal breakpoints and asynchronous checkpoint recovery.

Beyond vector time, Wu and Bernstein [127] shows use cases for matrix time. In matrix time, each processor stores time as a two dimensional matrix. At processor P_i, the (i, j)th entry in the matrix time represents the progress at Processor j that P_i is aware of. At processor P_i, the entry at (k, ℓ), for $k \neq i$, refers to the progress that processor P_i knows about what processor P_k knows of the progress at P_ℓ. While being useful in some specific applications, matrix time has the big disadvantages including the size of the timestamp to be sent along every message.

2.4.3 Limitations of Logical Time

The solution proposed by Lamport [82] to circumvent using physical time in a distributed system and instead use logical time is elegant. However, logical time has its own shortcoming that impact its practicality. If one insists on strong consistency, then scalar clocks do not suffice and one needs to use vector clocks or matrix clocks. These come with a huge overhead on message sizes and the overhead grows with increasing size of the distributed system. Recall from Section 2.4.2 that the timestamp information in a system using vector clocks grows linear in the number of nodes in the system. With current emphasis on large-scale systems, such an overhead is not practical.

In addition, even as logical time can be used to provide for a total order of the events there is no guarantee that two events e_1 and e_2 with the timestamp of e_1 smaller than that of e_2 are such that e_1 indeed occurs physically before e_2. Further, the total order induced is not unique. Such inference may be needed in distributed systems especially in cases where one wishes to enforce certain notions of consistency. Examples include mutually exclusive access to resources which is possible to solve using the total ordering induced by logical clocks but the solution does not ensure that the access is granted in physical time order.

Logical clocks also assume that all interactions of the participants in the distributed system happen within the system and there are no other communication channels. With the rapid rise of cyber-physical systems and Internet-of-Things, there is now more scope for systems to talk via multiple channels and today's systems are highly integrated in their operational environment but yet loosely coupled.

From the preceding discussion, we note that the existing solutions fall short of the practical requirements. Using physical time comes with limitations as mentioned by Lamport [82]. In fact, the solution from [82] makes two assumptions. Firstly, the clock C_i at any process i can only be set forward and never be set back. Secondly, the network of processes is to be

strongly connected. Thirdly, it is also assumed that there exist two constants δ and p such that each link carries a message whose delay is unpredictable yet bounded by δ every p seconds. Finally, the offset between the clocks at any two processes in the system is bounded by $d(2\kappa\delta + p)$ where κ is a constant that bounds the drift in the clock at any process over time, and d is the diameter of the network.

2.5 NTP at Distributed Scale

Corbett et al. [29] showcases a few details of how to engineer and run a protocol similar to NTP in a modern planet-scale distributed system. A brief summary of TrueTime follows.

Consider a setting where data centers are spread across the planet. Each data center has one or more TimeMaster servers, also known as time servers. The TrueTime solution has two kinds of TimeMaster servers.

- GPS Time Master: These TimeMaster servers contain GPS receiver nodes and can interface with GPS signals and receive time information via satellites. Figure 2.5 shows an example TimeMaster set up.
- Armageddon Master: These TimeMaster servers contain atomic clocks that are highly accurate. Time information from atomic clocks acts as a supplement to the time information from the GPS TimeMasters in case the satellite connections suffer any unexpected outage.

The engineering choice of using GPS based time information and atomic clocks is to account for good redundancy. Usually, it is observed that the factors that affect the failure of these two time systems are independent. For instance, satellite communication on which GPS relies on may be impacted by interference from radio waves. These radio waves do not impact the functioning of atomic clocks.

TimeMasters periodically exchange and compare their time with other TimeMasters. This exchange is along the lines of the NTP protocol. Each TimeMaster also checks the local clock for integrity. In case of these checks fail, the TimeMaster voluntarily stops being a TimeServer pending resolution of the failure.

We now outline how clients use TrueTime. A client pings a set of TimeMaster servers of both kinds, the GPS TimeMasters and the Armageddon TimeMasters, across the network. As in NTP, from the replies received from the TimeMasters, clients use statistical techniques to drop the outlier responses and know the best candidate time from the set of responses.

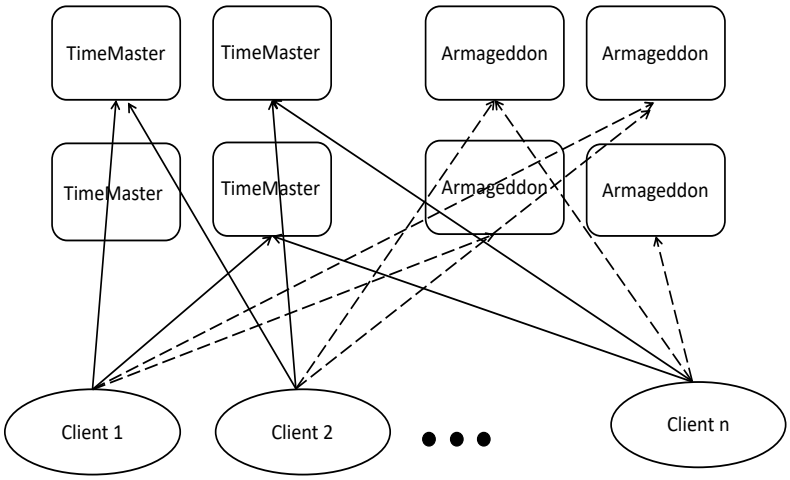

Figure 2.5: An illustration of the Google TrueTime time architecture. Solid lines represent clients interacting with TrueTime Master and dashed lines incited client interacting with Armegeddon master.

While these actions are going on, clients advertise a time interval that has increasing degree of uncertainty. This uncertainty is in proportion to the local clock drift.

Applications using TrueTime follow the TrueTime API. In this API, the timestamp of an event is an interval that accounts for the limited uncertainty. Each timestamp has the datatype TTInterval of form [earliest, latest] which correspond to the beginning and ending times of the interval. The API has three functions: the function TT.now() which returns the current timestamp as the above interval, the function TT.after(t) which returns True if time t has certainly elapsed, and the function TT.before(t) which returns True if time t is certainly yet to occur.

TrueTime guarantees that for an invocation of $tt = TT.now()$, $tt.earliest \leq t_{abs}(e) \leq tt.latest$, where is the absolute time when event happened. The interval given by *earliest* and *latest* are guaranteed to be within a bounded uncertainty. In practice, Google states that this interval lasts between 1 ms to 7 ms.

To summarize, notice that TrueTime brings two novel contributions. One is to represent time as an interval instead of as an absolute scalar quantity. The second novel aspect of TrueTime is to utilize GPS and atomic clock based time references. The latter allows TrueTime to use algorithms similar to that of NTP to synchronize time and provide strong guarantees

on the interval of uncertainty.

TrueTime facilitated many distributed applications that rely on the availability of time. A good example is the Google spanner distributed database system that uses TrueTime to induce a strict concurrency control for transactions at a global scale. More details of this is provided in Chapter 9.

2.5.1 Other Related Systems

The design of TrueTime by Corbett et al. [29] led to the design of similar systems. Amazon launched its version of time service labeled as timesync using GPS based and atomic clocks. This service also uses many ideas similar to that of TrueTime and NTP. The TimeSync service is now used in similar Amazon products such as Amazon DynamoDB [116].

Facebook in 2020 also moved to a distributed scale NTP engineering effort called `Chrony`. Details of this can be seen in [38].

2.5.2 Limitations of TrueTime and Similar Systems

Truetime and other related systems come under a class of highly engineered solutions that require the presence of special purpose hardware on time servers in the form of GPS receivers and precision atomic clocks. The atomic clocks used by Google are accurate to the order of a difference of 1 second in 10 million years. Such high level of precision instrumentation cannot be feasible for many settings. Secondly, it is not the hardware and instrumentation alone that supports systems such as TrueTime. The communication latency in the interactions between clients and TrueTime Master servers and the Armageddon Masters can play a role too. In particular, the TrueTime system relies on having a dedicated, fault-tolerant, high-speed communication fabric that ensures that the window of uncertainty in the TrueTime system is as small as possible.

In addition, TrueTime returns a time interval as a timestamp. When the interval is small enough, this suffices to order events and compare the timestamps of events to know which event precedes another event. In the unlikely situation that such an ordering is not possible, TrueTime recommends that applications wait out the period of uncertainty. These introduce unexpected delays in application event processing and also reduced the concurrency achievable in the system.

2.6 Hybrid Logical Clocks – Combining Physical and Logical Times

Kulkarni et al. [80] combine ideas from logical time and physical time to arrive at a simpler way of maintaining time in a distributed system. They call this as Hybrid Logical Clocks (HLC). In the following, we summarize the basic ideas from [80].

The goal of HLC is to support causality across events similar to what is provided by logical clocks while maintaining the logical clock value to be always close to the physical clock. The requirements of HLC can be captured formally as follows.

Let pt be the physical time at any node in the distributed system. We use $pt.e$ to denoted the physical timestamp at the node where the of event e happens. Given a distributed system, assign a timestamp ℓ for every e, denoted $\ell.e$, such that:

- For any two events e and f, $e \rightarrow f \Rightarrow \ell.e < \ell.f$.
- The timestamp $\ell.e$ can be stored in space of $O(1)$ integers
- The timestamp $\ell.e$ is represented in bounded space, and
- $\ell.e$ is close to $pt.e$, in symbols, $|\ell.e - pt.e|$ is bounded.

The above requirements can be understood in the following manner. The first requirement captures the notion of causality in distributed systems. The second requirement ensures that updates to $\ell.e$ is possible in $O(1)$ operations and hence reduces the overhead of maintaining the logical time. The third requirement indicates that the logical time has a bounded space requirement and does not depend on parameters that can grow unbounded such as the number of nodes in the system. The last requirement captures the constraint that $\ell.e$ is close to $pt.e$ which enables HLC to be used in place of physical time. This decoupling from physical time allows applications the replace using physical time with hybrid logical clock.

HLC maintains its logical clock at every node j as two variables $\ell.j$ and $c.j$. The tuple $\langle \ell.e, c.e \rangle$ is the logical timestamp assigned to event e. The first variable $\ell.j$ is used to maintain the maximum of pt information that j learnt so far. The second variable $c.j$ is used to capture causality updates when the ℓ values of two events are equal. This separation allows site j to *reset* $c.j$ when the information about maximum pt that j is aware of equals or goes beyond $\ell.j$. Notice from the description that for two events e and f

at a node j, it is possible that $\ell.e = \ell.f$. Thus, unlike logical time, ℓ need not be incremented for every event. This feature ensures that one of the following holds: (1) a node receives a message with a larger ℓ resulting in an update to ℓ and a reset of c, or (2) if the node does not get any messages from other nodes, then its ℓ stays the same, and its pt will increase naturally which results in an update to ℓ and reset c.

The algorithm HLC uses is shown below. Initially, at all nodes j we set $\ell.j = 0$ and $c.j = 0$. When executing a local event e, including a send event at node j, node j updates ℓ and c as follows.

- Set $\ell'.j = \ell.j$
- Set $\ell.j = \max\{\ell.j, pt.j\}$
- If $\ell.j == \ell'.j$, then $c.j = c.j + 1$ Else $c.j = 0$.
- Set the timetstamp of e as $ts.e := \langle \ell.j, c.j \rangle$.

When executing a receive event of message m at node j, it does the following.

- Set $\ell'.j = \ell.j$
- Set $\ell.j = \max\{\ell.j, \ell.m, pt.j\}$
- If $(\ell.j == \ell'.j == \ell.m)$, then $c.j = \max\{c.j, c.m\} + 1$ Else if $\ell.j == \ell'.j$, then $c.j = c.j + 1$ Else if $(\ell.j = \ell.m)$ then $c.j := c.m + 1$ Else $c.j = 0$
- Set the timetstamp of e as $ts.e := \langle \ell.j, c.j \rangle$.

From the above description, we notice that when a new send event f is created at node h, $\ell.j$ is set to $\max(\ell.e, pt.j)$, where e is the previous event at j. Such an update to ℓ ensures that $\ell.j \geq pt.j$ at all times. However, looking at events e and f at node j, it is possible that $\ell.e = \ell.f$. The increment to the second part of the timestamp tuple helps ensure that $\langle \ell.e, c.e \rangle < \langle \ell.f, c.f \rangle$ when compared lexicographically. If $\ell.e$ differs from $\ell.f$ then $c.j$ is reset, and this allows us to guarantee that c values remain bounded. Figure 2.6 shows an example of using hybrid logical clocks.

Similarly, to process a receive event at node j, $\ell.j$ is updated to to $\max\{\ell.e, \ell.m, pt.j\}$. Depending on whether $\ell.j$ equals both of $\ell.e, \ell.m$, or neither of them, $c.j$ is set accordingly.

The following theorems prove that the HLC construction satisfies the four conditions needed.

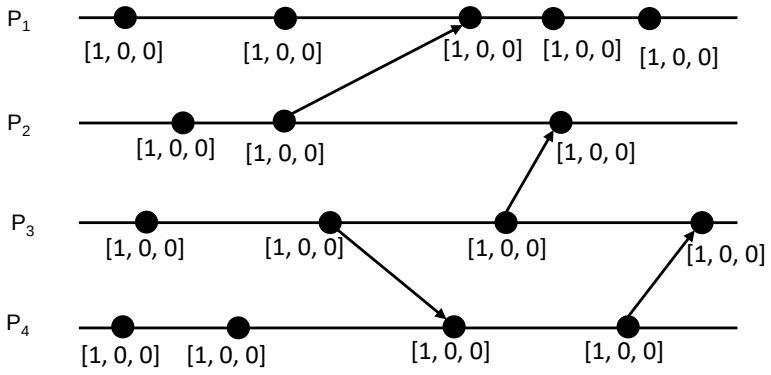

Figure 2.6: An illustration of Hybrid Logical Clocks.

Theorem 2.6.1

For any two events e and f, if $e \rightarrow f$ then $ts(e) < ts(f)$.

Proof: Notice that comparing timestamps in HLC is done via the lexicographic ordering of the tuple $\langle \ell.e, c.e \rangle$. (In particular, we say that $\langle a, b \rangle < \langle c, d \rangle$ if and only if either $a < c$ or $a = c$ and $b < d$.)

Based on how the timestamps of events are assigned, the theorem holds easily. □

We move to see how HLC meets condition 4 via the following sequence of theorem.

Theorem 2.6.2

For any event e, $\ell.e \geq pt.e$.

Theorem 2.6.3

Let e be any event. Then, $\ell.e$ denotes the maximum clock value that e is aware of. In particular, $\ell.e > pt.e \Rightarrow (\exists f : e \rightarrow f \wedge pt.f = \ell.e)$.

Proof: The proof can be done by induction on events. □

Let ϵ denote the physical clock synchronization uncertainty. Notice that ϵ depends on physical factors including the protocol such as NTP used to synchronize physical time. We now use Theorem 2.6 and the notion of ϵ to show that $\ell.e - pt.e|$ is bounded for any event e.

Theorem 2.6.4

For any event e, $|\ell.e - pt.e| \leq \epsilon$.

Proof:

Due to the nature of ϵ, we cannot have two events e and f such that $e \rightarrow f$ and $pt.e > pt.f + \epsilon$. Now, appealing to Theorem 2.6, we get the desired result. □

We now show that HLC satisfies condition 3 using once again the causality among events.

Theorem 2.6.5

Let e be an event with $c.e$ be $k > 0$. Then, there exists a sequence of k events $f_1, f_2, \cdots, f_k$ such that (i) for $1 \leq i < k$, $f_i \rightarrow f_{i+1}$, and (ii) for $1 \leq i \leq k$, $\ell.f_i = \ell.e$, and (iii) $f_k \rightarrow e$.

Proof: The proof of the theorem follows via induction. □

Theorem 2.6 immediately gives the following observation which limits the nature of events that lead to a high value for c.

Observation 2.6.1

For any event e, $c.e \leq |\{f | f \rightarrow e \wedge \ell.f = \ell.e\}|$.

Finally, we get the following bound on c.

Theorem 2.6.6

Let N denote the number of nodes in the distributed system. For any event e, $c.e \leq N \cdot (\epsilon + 1)$.

Proof: We appeal to Observation 2.6 and obtain that $c.e$ has a value bounded by the number of events f that precede e and have the same ℓ value. Note that f can occur at any node in the system. Further, from the construction of HLC, it holds that $\ell.f \geq pt.f$ for any event f. Finally, due to the clock synchronization constraint and the fact that $e \rightarrow f$, we have that $\ell.f \leq pt.e + \epsilon$.

From the above, we note that the events f that satisfy $f \rightarrow e$ and $\ell.f = \ell.e$ are those that occur at some node when the physical time at that node is in the interval $[\ell.e, \ell.e + \epsilon]$. Since the physical clock is incremented by at least one for every event, there are at most $1 + \epsilon$ such events at any

node. With N nodes in the system, the bound on $c.e \leq N \cdot (\epsilon + 1)$ follows. □

In summary, what HLC offers is a novel way to decouple physical time and logical time yet obtain guarantees on the window of uncertainty. Moreover, HLC does not rely on any expensive or special purpose mechanisms to synchronize physical time, nor does it rely on any particular physical time synchronization protocol. HLC uses the physical time but does not update the physical time at any node. The update to physical time is left to the physical time synchronization protocol. However, updates to the logical timestamp follow information obtained through timestamps on messages received, which are a function of the physical time at the sender.

Finally, HLC is not the silver bullet to solve time the problem of time synchronization in distributed systems. HLC still includes a window of uncertainty and applications may need to wait out this window of uncertainty, just as in using TrueTime [29].

2.7 Leap Seconds and Smearing

Despite all the careful theoretical and engineering efforts, there is still one issue that most practical systems face with. Notice that time is also a measure of how the Earth revolved around the sun. Changes in this speed occasionally or as a continuing pattern can cause changes in the accuracy of the time systems we use. It is observed that the speed with which the Earth revolves around the Sun varies slightly due to several factors. This results in a difference of roughly one second over a couple of years between physical clocks and the time as measured by the rotation of the Earth around the Sun. To adjust for this difference, the international body General Conference on Weights and Measurements (CGPM) [1] adopted the practice of adding or removing one second from the physical clocks. The last such addition to the local time at Hyderabad happened in June 2017. This extra second added or removed from physical clocks is called as *leap second*.

Systems such as TrueTime also adopt the model of adding a leap second as needed. One additional technique that is used is to spread the change across multiple instances. This practice goes by the name of leap second smearing.

Questions

Question 2.1. Set up a small experiment to simulate NTP in a LAN

environment. You can scale the simulation to systems running across larger physical distances by introducing a delay in proportion to the physical distance plus some noise.

Question 2.2. In logical scalar time and logical vector time, is the increment d needed to be common across processes in the distributed system? Justify your answer.

Question 2.3. In the context of the Hybrid Logical Clocks, if the time for message transmission is long enough so that the physical clock of every node is incremented by at least d, where d is a given parameter, redo the bound on $c.f$ for any event f. From Page 6 of the HLC paper.

TO MOVE TO EARLIER PART:

Modern distributed systems is also about creating large-scale highly reliable, fault-free, and available systems using inherently unreliable and limited in scale with additional challenges including degrees of asynchrony and uncertainty.

All problems in computer science can be solved by another level of indirection. –David Wheeler

no changes.