Lectures on distributed systems

Clock Synchronization

Paul Krzyzanowski

> *When Charles V retired in weariness from the greatest throne in the world to the solitude of the monastery at Yuste, he occupied his leisure for some weeks trying to regulate two clocks. It proved very difficult. One day, it is recorded, he turned to his assistant and said: "To think that I attempted to force the reason and conscience of thousands of men into one mould, and I cannot make two clocks agree!"*
>
> *Havelock Ellis,*
> *The Task of Social Hygiene, Chapter 9*

Introduction

Clock synchronization deals with understanding the temporal ordering of events produced by concurrent processes. It is useful for synchronizing senders and receivers of messages, determining whether messages are related and their proper ordering, controlling joint activity, and serializing concurrent access to shared objects. Multiple autonomous processes running on different machines need to be able to agree on and be able to make consistent decisions about the ordering of certain events in a system.

Our real-world view of clock synchronization is one of ensuring that multiple processes on multiple machines all see the same time of day. All modern computers have a time-of-day clock and synchronization becomes a matter of keeping these clocks set to an agreed-upon value, usually standard time as defined by UTC, Coordinated Universal Time.

Having synchronized clocks is extremely useful but is not always sufficient. Consider our ability to identify the sequencing and interdependence of events, such as sending or receiving messages or executing a transaction. A timestamp from a time-of-day clock on each such event will identify *when* the event happened has two potential pitfalls.

First, if two events take place at approximately the same time, they may be reported as taking place at the same time since timestamps have a limited precision. Worse, the clocks on different systems may not be precisely synchronized so that an event on one computer may be assigned a later timestamp than that on another even if it took place earlier in time than the other event. That produces the false impression that the first event happened after the event that took place on the other computer. As an example, consider the case where process *A* sends a message with an embedded timestamp of 4:15:00 and machine *B*, on another computer, sends a message with a timestamp of 4:15:05. It is quite possible that process *B*'s message was actually sent prior to that of process *A* if *B*'s clock was over 5 seconds faster. Even if *A*'s and *B*'s clocks were synchronized, it is likely that the clocks run at slightly different speeds, drift apart over time, and eventually report different times.

Second, just by looking at two timestamps, you cannot tell if one event may be the result of another event or if they are completely independent. The messages sent to have identical or even misleading timestamps, as in the above example. If we use algorithms that rely on choosing one timestamp over another, we may not be able to make a consistent decision either by comparing two message timestamps. Worse yet, if we are using a distributed algorithm where each process compares the timestamp in a received message with its own clock, there is no assurance that all systems will yield the same result.

To enable us to get both the time of day as well as an ability to compare events in a meaningful manner, we will use two forms of clocks: *physical* and *logical* clocks.

The concept of a *logical clock* is one where the clock does not have any bearing on the time of day but rather is a number that can be used for comparing sets of events, such as a messages, within a distributed system.

A *physical clock*, on the other hand, reports the time of day. Physical clock synchronization deals with synchronizing time-of-day clocks among groups of machines. In this case, we want to ensure that all machines can report the same time, regardless of how imprecise their clocks may be or what the network latencies are between the machines.

Logical clocks

Let us consider cases that involve assigning sequence numbers ("timestamps") to events upon which all cooperating processes can agree. What matters in these cases is not the time of day at which the event occurred but that all processes can agree on the *order* in which related events occur. Our interest is in getting event sequence numbers that make sense system-wide. These clocks are called *logical clocks*.

If we can do this across all events in the system, we have something called *total ordering*: every event is assigned a unique timestamp (number) and every such timestamp is unique. However, we don't always need total ordering. If processes do not interact then we do not care when their events occur. If we only care about assigning timestamps to potentially related (*causal*) events, then we have something known as *partial ordering*.

Leslie Lamport defined a *happens before* notation to express the relationship between events: $a \rightarrow b$ means that a happens before b. If a represents the timestamp of a message sent and b is the timestamp of that message being received, then $a \rightarrow b$ *must* be true; a message cannot be received before it is sent. This relationship is transitive. If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. If a and b are events that take place in the same process then $a \rightarrow b$ is true if a occurs before b.

With logical time, we would like to assign a time value (sequence number) to each event such that everyone will agree on the final order of events. That is, if *a*→*b* then clock(*a*) < clock(*b*) since the clock (our sequence generator) must never run backwards. If *a* and *b* occur on different processes that *do not* exchange messages (even through third parties) then *a*→*b* is *not* true. These events are said to be *concurrent*: there is no way that *a* could have influenced *b*.

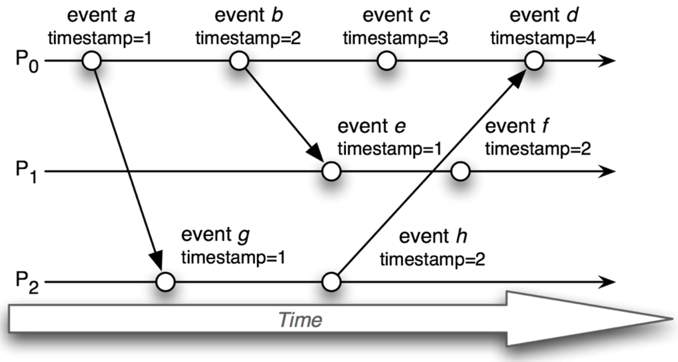

Figure 1. Unsequenced event stamps

Consider the sequence of events depicted in Figure 1 taking place between three processes. Each event is assigned a timestamp by its respective process. Each process simply maintains its own counter that is incremented before each event gets a timestamp.

If we examine the timestamps from a global perspective, we can observe a number of peculiarities. Event *g*, the event representing the receipt of the message sent by event *a*, has the exact same timestamp as event *a* when it clearly had to take place *after* event *a*. Event *e* has an earlier time stamp (1) than the event that sent the message (*b)*, with a timestamp of 2).

Lamport's algorithm remedies the situation by forcing a resequencing of timestamps to ensure that the *happens before* relationship is properly depicted for events related to sending and receiving messages. It works as follows:

> Each process has its own clock, which can be a simple counter that is incremented prior to each event.
>
> The sending of a message is an event and each message carries with it a timestamp obtained from the current value of the clock at that process (a sequence number).
>
> The arrival of a message at a process is also an event and will therefore also receive a timestamp – by the receiving process, of course. The process' clock is incremented prior to timestamping the event, as it would be for any other event. If the process' clock value is *less than or equal to* the timestamp in the received message, the process' clock is adjusted to the (message's timestamp + 1). Otherwise nothing is done. The event is now timestamped. This action ensures that the receipt of the message and all subsequent events at that process will receive higher timestamp values than that of sending the message.

If we apply Lamport's algorithm to the same sequence of messages, we can see that proper message ordering among causally related events is now preserved (Figure 2). Note that between every two events, the clock must tick at least once.

Lamport's algorithm allows us to maintain proper time ordering among causally-related events. In summary, Lamport's algorithm requires a monotonically increasing software counter for a "clock" that has to be incremented at least when events that need to be timestamped take place. These events will have that clock value, called a *Lamport timestamp*, associated with them. For any two events, where $a \rightarrow b$, $L(a) < L(b)$ where $L(x)$ represents the Lamport timestamp for event x.

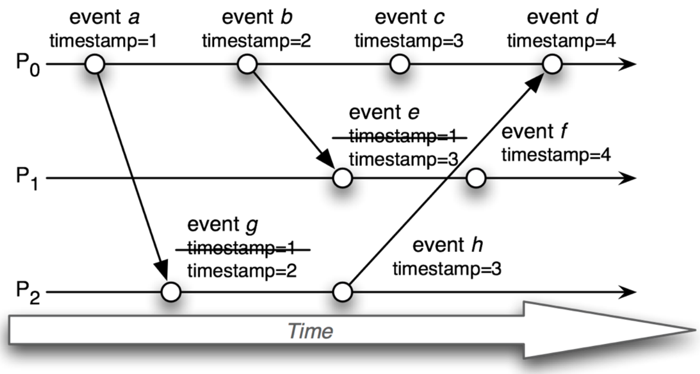

Figure 2. Lamport sequenced event stamps

Lamport timestamps assure us that if there is a causal relationship between two events then the earlier event will have a smaller timestamp than the later event. Causality is achieved by successive events on one process or by the sending and receipt of messages on different processes. As defined by the *happened-before* relationship, causality is transitive. For instance, events *a* and *f* are causally related in Figure 2 (through the sequence *a, b, e, f*).

Total ordering

Note that it is very possible for multiple non-causal (concurrent) events to share identical Lamport timestamps (e.g., *c*, *e,* and *h* in Figure 2). This may cause confusion if multiple processes need to make a decision based on the received timestamps of two concurrent events. The selection of one event over the other may not matter if the events are concurrent but we want all processes to make the same decision. This is difficult if the timestamps are identical. Fortunately, there's an easy remedy.

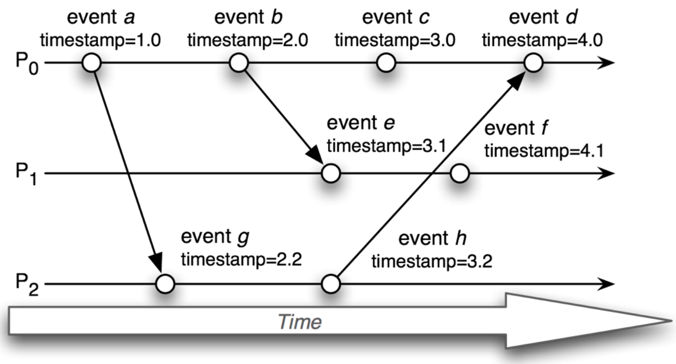

Figure 3. Totally ordered Lamport timestamps

We can create a total order on events by further qualifying them with process ID numbers. We define a global logical timestamp (T_i,i) where T_i represents the local Lamport timestamp and i represents the process ID (in some globally unique way; for example, a concatenation of host address and process ID). We are then able to globally compare these timestamps and conclude that

$$(T_i,i) < (T_j,j)$$

if and only if

$$T_i < T_j$$

or $$T_i = T_j \text{ and } i < j.$$

There is no physical significance to the order since process identifiers can be arbitrary and do not relate to event ordering but the ability to ensure that no two Lamport timestamps are the same globally is helpful in algorithms that need to compare these timestamps. Figure 3 shows an example with a suffix of the process ID added to each timestamp. In real life, depending on the application, one may use a combination of thread ID, process ID, and IP address as a qualifier to the timestamp.

Vector clocks: identifying concurrent events

If two events are causally related and event *e* happened before event *e'* then we know that $L(e) < L(e')$. However, the converse is not necessarily true. With Lamport's algorithm, if $L(e) < L(e')$ we *cannot* conclude that $e \rightarrow e'$. Hence, if we look at Lamport timestamps, we cannot conclude which pairs of events are causally related and which are not. One solution that has been proposed to deal with this problem is the concept of *vector clocks* (proposed by Mattern in 1989 and Fridge in 1991).

A vector clock in a system of N processes is a vector of N integers. Each process maintains its own vector clock (V_i for a process P_i) to timestamp local events. Like Lamport timestamps, vector timestamps (the vector of N integers) are sent with each message. The rules for using vector clocks are:

1. The vector is initialized to 0 at all processes:
   $V_i[j] = 0$ for $i,j = 1, \ldots, N$

2. Before a process P_i timestamps an event, it increments its element of the vector in its local vector:
   $V_i[i] = V_i[i]+1$

3. A message is sent from process P_i with V_i attached to the message.

4. When a process P_j receives a vector timestamp t, it compares the two vectors element by element, setting its local vector clock to the higher of the two values:
   $V_j[i] = max(V_j[i], t[i])$ for $i=1, \ldots, N$

We compare two vector timestamps by defining:

$$V = V' \text{ iff } V[j] = V'[j] \text{ for } i=1, \ldots, N$$

$$V \leq V' \text{ iff } V[j] \leq V'[j] \text{ for } i=1, \ldots, N$$

For any two events *e*, *e'*, if $e \rightarrow e'$ then $V(e) < V(e')$. This is the same as we get from Lamport's algorithm. With vector clocks, we now have the additional knowledge that if $V(e) < V(e')$ then $e \rightarrow e'$. Two events *e*, *e'* are concurrent if *neither* $V(e) \leq V(e')$ nor $V(e') \leq V(e)$.

We can examine the events in Figure 4 with vector clocks and see how events *a* and *e* can be determined to be concurrent by comparing their vector timestamps. If we do an element-by-element comparison, we see that each element in one timestamp is not consistently less than or equal to its corresponding element in the second timestamp. For example, element 1 is greater in *a* than it is in *e* (*1>0*) but element 3 in *a* is less it is in *e* (*0<1*).

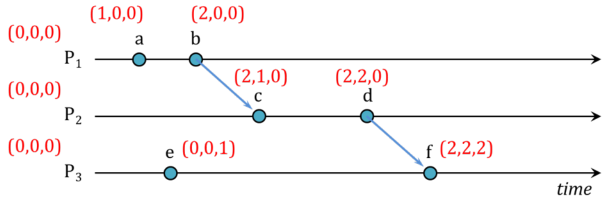

Figure 4. Messages with vector timestamps

The disadvantage with vector clocks is the greater storage and message payload size since an entire vector rather than a single integer must be sent and tracked. There may be cases where the group of processes is either not known to all members or varies over time. In such cases, it does not make sense to send a numeric vector where a process is identified by its position in the vector. In these cases, we can represent the vector by a series of <*process ID, timestamp*> tuples where a process ID, as we discussed earlier, may be a concatenation of a process ID and the host computer's address. To compare these vectors, we find matching process IDs and compare their timestamps. If one vector is missing a process ID, then its value is implicitly 0.

Physical clocks

Most computers today keep track of the passage of time with a battery-backed-up CMOS clock circuit, driven by a quartz resonator. This allows timekeeping to take place even if the machine or the CPU is powered off. When on, an operating system will generally program a timer circuit (typically an Advanced Programmable Interrupt Controller, or APIC, in Intel-based systems) to generate an interrupt periodically. Many Linux systems, for example, 250 interrupts per second by default. The interrupt service procedure simply adds one to a counter in memory to maintain a monotonically increasing value that represents the passage of time. This value is known as the *software clock* or *kernel clock* to differentiate it from the *hardware clock* (also known as the *CMOS clock*).

While the best quartz resonators can achieve an accuracy of one second in 10 years, they are sensitive to changes in temperature and acceleration and their resonating frequency can change as they age. Standard resonators are accurate to 6 parts per million at 31° C, which corresponds to ±½ second per day.

The problem with maintaining a concept of time is when multiple entities expect each other to have the same idea of what the current time is. Two watches hardly ever agree. Computers have the same problem: a quartz crystal on one computer will oscillate at a slightly different frequency than on another computer, causing the clocks to tick at different rates. The phenomenon of clocks ticking at different rates, creating an ever widening gap in perceived time is known as *clock drift*. The difference between two clocks at any point in time is called *clock skew* and is due to both clock drift and the possibility that the clocks may have been set differently on different machines. Figure 5 illustrates this phenomenon with two clocks, *A* and *B*, where clock *B* runs slightly faster than clock *A* by approximately two seconds per hour. This is the clock drift of *B* relative to *A*. At one point in time (five seconds past five o'clock according to *A*'s clock), the difference in time between the two clocks is approximately four seconds. This is the clock skew at that particular time.

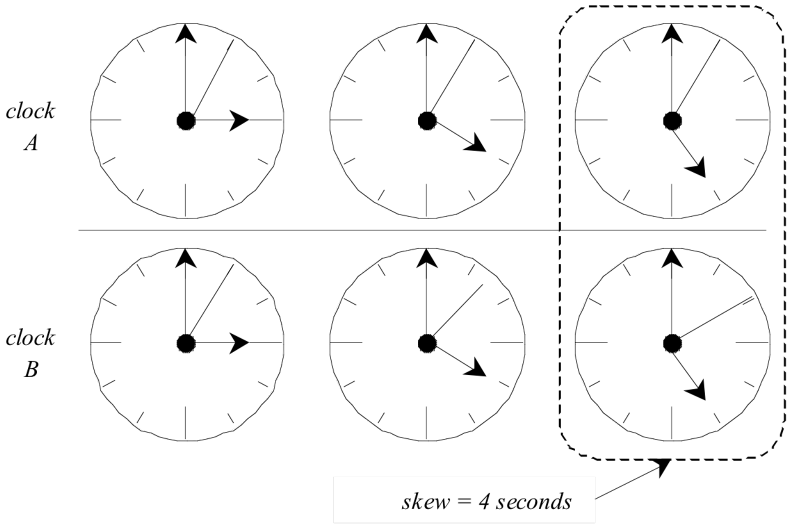

Figure 5. Clock drift and clock skew

Compensating for drift

We can envision clock drift graphically by considering true (UTC) time flowing on the *x*-axis and the corresponding computer's clock reading on the *y*-axis. A perfectly accurate clock will exhibit a slope of one. A faster clock will create a slope greater than unity while a slower clock will create a slope less than unity. Suppose that we have a means of obtaining the true time. One easy, and frequently adopted, solution is to simply update the system time to the true time. This works well for personal computers but may cause problems on servers and other systems that are actively running processes as these processes may see a spontaneous change in time, possibly a jump back in time! To avoid this, one constraint that we will impose on clock synchronization is that it is not a good idea to set the clock back. The illusion of time moving backwards can confuse real-time-based message ordering, users, and software development environments.

If a clock is fast, it simply has to be made to run slower until it synchronizes. If a clock is slow, the clock can be made to run faster until it synchronizes. In theory, the operating system can do this by changing the rate at which it requests interrupts. For example, suppose the system requests an interrupt every 17 milliseconds (pseudo-milliseconds, really – the computer's idea of what a millisecond is) and the clock runs a bit too slowly. The system can request interrupts at a faster rate, say every 16 or 15 milliseconds, until the clock catches up. However, this is not always a practical approach since we may not have enough precision in the timer. It is easier to avoid mucking around with the hardware and just redefine the rate at which system time is advanced with each interrupt. Hence, whenever the operating system will read the software clock, it will apply an adjustment to the counter to compensate for drift.

This adjustment changes the slope of the system time and is known as a *linear compensation function* (Figure 6). After the synchronization period is reached, one can choose to resynchronize periodically and/or keep track of these adjustments and apply them continually to get a better running clock. This is analogous to noticing that your watch loses a minute every two months and making a mental note to adjust the clock by that amount every two months (except the system does it continually). For an example of clock adjustment, see the Linux man page for *adjtime*.

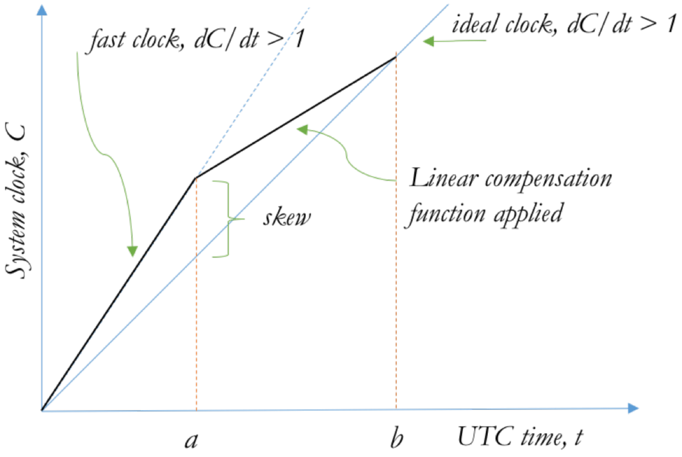

Figure 6. Compensating for drift with a linear compensation function

Setting the time on physical clocks

With physical clocks, our interest is not in advancing them just to ensure proper message ordering, but to have the system clock keep good time. We looked at methods for adjusting the clock to compensate for skew and drift, but it is essential that we can find the precise time first so that we would know how we need to adjust our clock.

One possibility is to attach a GPS (Global Positioning System) receiver to each computer. A GPS receiver will provide time from within ±100 ns[1] to ±1 µs of UTC time. USB-connected ones can be had for under US $30. Some devices, such as phones and tablets, have a GPS receiver built into them (the chip cost is under $5). Unfortunately, they often do not work well indoors. If the machine is in the U.S., one can attach a WWV radio receiver to obtain time broadcasts from the National Institute of Standards and Technology at Boulder, Colorado or Washington, DC, with accuracies of ±3–10 ms, depending on the distance from the source. Another option is to obtain a GOES (Geostationary Operational Environment Satellites) receiver, which will provide time within ± 0.1 ms of UTC time. For reasons of economy, convenience, and reception, these are not practical solutions for *every* machine. Most systems will set their time by asking another computer for the time, preferably one with one of the aforementioned time sources connected to it. A computer that provides this information is called a *time server*.

[1] Just a reminder: 1ns is 1 nanosecond, or one billionth of a second. 1 µs is 1 microsecond, or one millionth of a second. 1 ms is 1 millisecond, or 1 thousandth of a second.

Cristian's algorithm

The simplest method for setting the time would be simply to issue a remote procedure call to a time server and obtain the time. The result, however, does not account for the network and processing delay. Cristian's algorithm improves this result by accounting for the latency of getting the timestamp. We can compensate for this by measuring the time (in local system time) at which the request is sent (T_0) and the time at which the response is received (T_1). Our best guess at the network delay in each direction is to assume that the delays to and from are symmetric. The estimated overhead due to the network delay is then (T_1- T_0)/2. The new time can be set to the time returned by the server plus the time that elapsed since the server generated the timestamp:

$$T_{new} = T_{server} + \frac{T_1 - T_0}{2}$$

The second part of Cristian's algorithm is to place bounds on the accuracy of the result. Suppose that we know the smallest time interval that it could take for a message to be sent between a client and server (either direction). Let's call this time T_{min}. This is the time when the network and CPUs are completely unloaded. Knowing this value allows us to place bounds on the accuracy of the result obtained from the server. If we sent a request to the server at time T_0, then the *earliest* time stamp that the server could generate the timestamp is $T_0 + T_{min}$. The *latest* time that the server could generate the timestamp is $T_1 - T_{min}$, where we assume it took only the minimum time, T_{min}, to get the response. The range of these times is: $T_1 - T_0 - 2T_{min}$, so the accuracy of the result is:

$$\pm\left|\frac{T_1 - T_0}{2} - T_{\min}\right|$$

Errors are cumulative. If process *A* synchronizes from a server *B* and gets an accuracy of ±5 ms but server *B* in turn got its time from server *C* with an accuracy of ±7 ms, the net accuracy at machine *A* is ±(5+7), or ±12 ms.

Several time requests may be issued consecutively in the hope that one of the requests may be delivered faster than the others (e.g., it may be submitted during a time window when network activity is minimal). This can achieve improved accuracy since we can select the interaction with the smallest round-trip time and hence the lowest error.

Cristian's algorithm suffers from the problem that afflicts all single-server algorithms: the server might fail and clock synchronization will be unavailable. It may also be subject to malicious interference: a forged response may make the client set an incorrect time[2].

Berkeley algorithm

The Berkeley algorithm, developed by Gusella and Zatti in 1989, does not assume that any machine has an accurate time source with which to synchronize. Instead, it opts for obtaining an average time from the participating computers and synchronizing all machines to that average.

[2] Read up on the use of timestamps in guarding against *replay attacks* to see why a malicious party may want to set a computer to the wrong time.

The computers involved in the synchronization each run a time dæmon[3] process that is responsible for implementing the protocol. One of these computers is elected (or designated) to be the master. The others are slaves. The server polls each computer periodically, asking it for the time. The time at each system may be estimated by using Cristian's method to account for network delays, if desired. When all the results are in, the master computes the *average* time (including its own time in the calculation). The hope is that the average cancels out the individual clock's tendencies to run fast or slow.

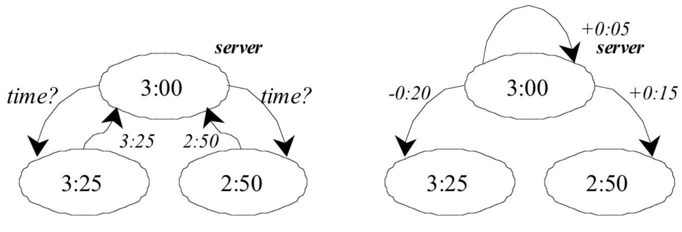

Figure 7. Berkeley synchronization algorithm

Instead of sending the updated time back to the slaves, which would introduce further uncertainty due to network delays, it sends each computer the offset by which its clock needs adjustment. The operation of this algorithm is illustrated in Figure 7. Three machines have times of 3:00, 3:25, and 2:50. The machine with the time of 3:00 is the server (master). It sends out a synchronization query to the other machines in the group. Each of these machines sends a timestamp as a response to the query. The server now averages the three timestamps: the two it received and its own, computing (3:00+3:25+2:50)/3 = 3:05. Now it sends an offset to each machine so that the machine's time will be synchronized to the average once the offset is applied. The computer with a time of 3:25 gets sent an offset of -0:20 and the computer with a time of 2:50 gets an offset of +0:15. The server needs to adjust its own time by +0:05.

The algorithm also has provisions to ignore readings from clocks whose skew is too great. The master may compute a *fault-tolerant average* – averaging values from machines whose clocks have not drifted by more than a certain amount. If the master fails, any other slave could be elected to take over.

Network Time Protocol (NTP)

The Network Time Protocol [1991, 1992, 2010] is an Internet standard (version 4, RFC 5905-5908) whose goals are to:

- Enable clients across the Internet to be accurately synchronized to UTC (universal coordinated time) despite message delays. Statistical techniques are used for filtering data and gauging the quality of the results.

- Provide a reliable service that can survive lengthy losses of connectivity. This means having redundant paths and redundant servers.

- Enable clients to synchronize frequently and offset the effects of clock drift.

- Provide protection against interference; authenticate that the data is from a trusted source.

[3] A dæmon is a program that runs in the background.

NTP servers are arranged into *strata*. The first stratum contains the primary servers, which are computers that are connected directly to an accurate time source (the time source hardware itself is considered to be at stratum 0). The second stratum contains the secondary servers. These machines that synchronized from the primary stratum machines. The third stratum contains tertiary servers that synchronized from the secondaries, and so on. Together, all these servers form the *synchronization subnet* (Figure 8).

A computer will often try to synchronize with several servers, using the best of all the results to set its time. The *best* result is a function of a number of qualities, including: round-trip delay, consistency of the delay, round-trip error, server's stratum, the accuracy of the server's clock, the last time the server's clock was synchronized, and the estimated drift on the server.

Because a system may synchronize with multiple servers, its stratum is dynamic: it is based on the server used for the latest synchronization. If you synchronized from a secondary NTP server then you are in the third stratum. If, next time, you used a primary NTP server to synchronize, you are now in the second stratum.

Computers synchronize in one of the following modes:

- *Symmetric active/passive mode*: This is intended for synchronization among a set of low-stratum NTP time servers to enable the servers to operate as mutual backups for each other. If one of the systems in the group loses its access to time servers or ceases to run, other servers in the group can take over. In this mode of operation, a host announces its willingness to synchronize and be synchronized by the peer. This mode offers the highest accuracy and is intended for use by master servers. A pair of servers exchanges messages with each other containing timing information. Timing data are retained to improve accuracy in synchronization over time.

- *Procedure call mode*: This is the most common mode of operation and is similar to Cristian's algorithm. A client announces its willingness to by synchronized by the server, but not to synchronize the server.

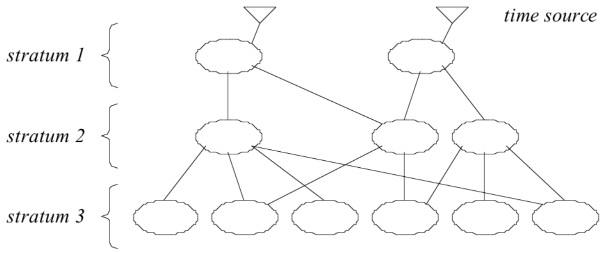

Figure 8. NTP synchronization subnet

- *Broadcast/multicast mode*: This is intended for high speed LANs, particularly those with a large number of clients. In this mode, the server sends a broadcast message containing a time stamp at regular intervals. Multicast mode is similar but uses IP multicast, which directs the messages to a group of systems that registered their interest in receiving these messages. The broadcast/multicast mode offers relatively low accuracy since there is no way to account for network delays, but is acceptable for many applications, particularly when servers and clients are on the same high-speed local area network.

All messages are delivered unreliably via UDP[4]. In both the procedure call mode and symmetric mode, messages are exchanged in pairs. Each message has the following timestamps:

T_{i-3}: local time when previous NTP message was sent.

T_{i-2}: local time when previous NTP message was received.

T_{i-1}: local time when current NTP message was sent.

The server notes its local time, T_i. For each pair, NTP calculates the offset (estimate of the actual offset between two clocks) and delay (total transit time for two messages). In the end, a process determines three products:

1. *Clock offset*: this is the amount that the local clock needs to be adjusted to have it correspond to a reference clock.

2. *Roundtrip delay*: this provides the client with the capability to launch a message to arrive at the reference clock at a particular time; it gives us a measure of the transit time of the message to a particular time server.

3. *Dispersion*: this is the "quality of estimate" based on the accuracy of the server's clock and the consistency of the network transit times. It represents the maximum error of the local clock relative to the reference clock.

By performing several NTP exchanges with several servers, a process can determine which server to favor. The preferred ones are those with a lower stratum and the lowest total filter dispersion. A higher stratum (less accurate) time source may be chosen if the communication to the more accurate servers is less predictable.

The *Simple Network Time Protocol*, SNTP (RFC 2030), is a subset of the Network Time Protocol that allows operation in a stateless remote procedure call mode or multicast mode. It is not a new protocol but just a subset of NTP. It is intended for environments when the full NTP implementation is not needed or is not justified. The intention is that SNTP be used at the ends of the synchronization subnet (high strata) rather than for synchronizing time servers.

SNTP can operate in either a unicast, multicast, or anycast modes:

- In unicast mode, a client sends a request to a designated server.

- In multicast mode, a server periodically sends a broadcast or multicast message and expects no requests from clients. This matches NTP's multicast mode and suffers from the same problem: there is no ability to estimate the delay.

[4] Using a reliable protocol, such as TCP, will introduce significant asymmetric latencies whenever packet loss occurs.

- In anycast mode, a client sends a request to a local broadcast or multicast address and takes the first response received by responding servers. From then on, the protocol proceeds as in unicast mode.

NTP and SNTP messages are both sent via UDP. The message structure contains:

Leap indicator	warns of impending leap second (last minute has either 59, 60, or 61 seconds)
Version number	
Mode	symmetric active, symmetric passive, client, server, broadcast
Stratum	stratum
Poll interval	maximum interval between successive messages (power of 2)
Precision	8-bit signed integer indicating the precision of the local clock, seconds to nearest power of two
Root delay	32-bit number indicating total roundtrip delay to primary reference source (16 bit seconds, and 16 bits of decimal seconds)
Root dispersion	32-bit number indicating the nominal error relative to the primary reference source
Reference identifier	identify the reference source – four character ASCII string. Possible sources are: local uncalibrated clock, atomic clock, NIST dial-up modem service, USNO modem service, PTB (Germany) dial-up modem service, Allouis (France) radio, Boulder (CO, USA) radio, LORAN-C radionavigation system, Global Positioning System (GPS), Geostationary Orbit Environment Satellite(GOES), & cetera.
Reference timestamp	time at which local clock was last set or corrected
Originate timestamp	time at which request departed the client for the server
Receive timestamp	time at which the request arrived at the server
Transmit timestamp	time at which the reply departed the server
Key identifier	used if the NTP authentication scheme is implemented
Message digest	used if the NTP authentication scheme is implemented

In unicast mode, the roundtrip delay and local offset are calculated as follows:

1. The client sets the transmit timestamp in the request to the time of day according to the client clock. (T_1).

2. The server copies this field to the originate timestamp in the reply and sets the receive timestamp and transmit timestamps to the time of day according to the server clock (T_2, T_3).

3. When the server reply is received, the client determines a destination timestamp as the time of arrival according to its clock (T_4).

Timestamp name	*ID*	*when generated*
originate timestamp	T_1	time request sent by client
receive timestamp	T_2	time request received by server
transmit timestamp	T_3	time reply sent by server
destination timestamp	T_4	time reply received by client

The roundtrip delay *d* is defined as:

$$d = (T_4 - T_1) - (T_2 - T_3)$$

Note that the delay estimates the time spent sending and receiving data over the network, and subtracts out the processing delay at the server. The local clock offset *t* is defined as:

$$t = ((T_2 - T_1) + (T_3 - T_4)) / 2$$

The client, after computing this offset, adds this amount to its clock.

References

Time, Clocks, and the Ordering of Events in a Distributed System, Leslie Lamport, Communications of the ACM, July 1978, Volume 21, Number 7, pp. 558-565.

The Network Time Protocol (NTP) Distribution, The Network Time Foundation, http://doc.ntp.org

Distributed Systems: Concepts and Design, G. Coulouris, J. Dollimore, T. Kindberg, ©1996 Addison Wesley Longman, Ltd.

Distributed Operating Systems, Andrew Tanenbaum, © 1995 Prentice Hall.

Modern Operating Systems, Andrew Tanenbaum, ©1992 Prentice Hall.

RFC1305: Network Time Protocol version 3. This can be found in many locations. One place is http://www.faqs.org/rfcs/rfc1305.html

RFC 2030: Simple Network Time Protocol version 4. This can be found in many places. One place is http://www.faqs.org/rfcs/rfc2030.html