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 synchronization between senders and receivers of messages, control of joint activity, and the serialization of concurrent access to shared objects. For these kinds of events, we introduce the concept of a *logical clock*, one where the clock need not have any bearing on the time of day but should be useful for generating message sequence numbers. It is also useful for certain applications to have access to an accurate *physical clock*, one that attempts to provide an accurate measure of the current time.

Logical clocks

One aspect of clock synchronization is to provide a mechanism whereby systems can assign sequence numbers ("timestamps") to messages upon which all cooperating processes can agree. What matters in many cases is not the time of day but that all processes can agree on the *order* in which related events occur. In this case, our interest is not on obtaining and maintaining true time, but on getting event sequence numbers that make sense system-wide. These clocks are called **logical clocks**. If processes do not interact then their clocks do not have to be synchronized.

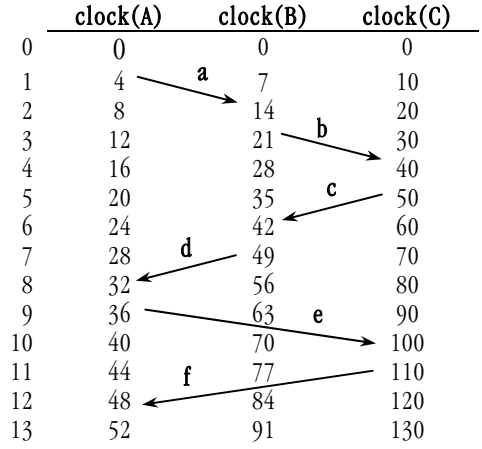

Figure 1. Unsequenced message timestamps

Lamport developed a "happens before" notation to express this: $a \rightarrow b$ means that a happens before b. If a is a message sent and b is a the message being received, then $a \rightarrow b$ *must* be true. A message cannot be received before it is sent[1]. This relationship is transitive. If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. The importance of measuring time is to assign a

msg	*from→ to*	*depart time*	*arrive time*	*OK?*
a	A→B	4	14	yes
b	B→C	21	40	yes
c	C→B	50	42	no
d	B→A	49	32	no
e	A→C	36	100	yes
f	C→A	110	48	no

Figure 2. Message delivery timestamps

[1] It may interest the reader to know that the Psychic Friends Network has filed for bankruptcy.

time value to each event on which everyone will agree on the final order of events. That is, if $a \rightarrow b$ then clock(a) < clock(b) since the clock must never run backwards. If a and b occur on different processes that *do not* exchange messages (even through third parties) then $a \rightarrow b$ is *not* true. These events are said to be **concurrent**.

Consider the sequence of events depicted in Figures 1 and 2 taking place between three machines whose clocks tick at different rates.

In three of the six messages, we get the appearance of moving back in time. Because of this, future messages from those sources appear to have originated earlier than they really have. If we are to sort messages by the timestamps placed upon them when they were sent, the sequence of messages would appear to be {a, b, e, d, c, f} rather than {a, b, c, d, e, f}. **Lamport's algorithm** remedies the situation as follows:

> Each message carries a timestamp of the sending time (according to the sender's clock).
>
> When a message arrives and the receiver's clock is *less* than the timestamp on the received message, the system's clock is forwarded to the message's timestamp + 1. Otherwise nothing is done.

. If we apply this algorithm to the same sequence of messages, we can see that message ordering is now preserved (Figures 3 and 4). Note that between every two events, the clock must tick at least once.

	clock(A)	clock(B)	clock(C)
0	0	0	0
1	4	7	10
2	8	14	20
3	12	21	30
4	16	28	40
5	20	35	50
6	24	42→51	60
7	28	58	70
8	32→59	65	80
9	63	72	90
10	67	79	100
11	71	86	110
12	75→111	93	120
13	115	100	130

Figure 3. Lamport-ordered message delivery

Lamport's algorithm allows us to maintain time ordering among events. We can attach, for example, a process number and host ID to the low order bits of the time stamp (think of it as a fractional extension). Now we have a way to get a total ordering of all events in the system.

msg	*from→ to*	*depart time*	*arrive time*	*Adjust clock*
a	A→B	4	14	-
b	B→C	21	40	-
c	C→B	50	42	51
d	B→A	58	32	59
e	A→C	63	100	-
f	C→A	110	75	111

Figure 4. Lamport clock adjustments

.

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 this "Lamport timestamp" attached to them. For any two events, where $a \rightarrow b$, $L(a) < L(b)$ where $L(x)$ represents the Lamport timestamp for event x.

Consider the sequence of events from three processes depicted in Figure 5. When Lamport timestamps are assigned, we can conclude:

- $a \rightarrow b$, $c \rightarrow d$ because events within a process are sequenced
- $b \rightarrow c$, $d \rightarrow f$ because Lamport imposes a send(m)$\rightarrow$*receive(m) relationship*
- $a \not\rightarrow e$, $e \not\rightarrow a$ because these events are concurrent

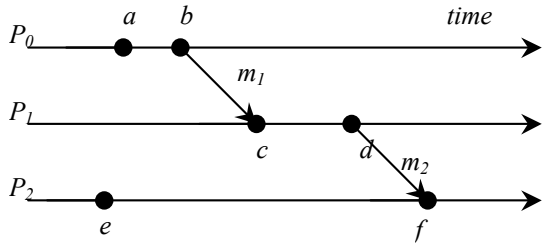

Figure 5. Causally and non-causally related messages

One problem with Lamport's algorithm is that pairs of distinct events (e.g. a, e) can have the same Lamport timestamp. To solve this, we can create a total order on events by taking into account the identities of processes. 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 can globally compare these timestamps and conclude that

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

if and only if

$$T_i < T_j$$

or $T_i = T_j$ 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 can be useful in certain cases (for example, it was used to order entry of processes into a critical section in one mutual exclusion algorithm).

Another problem with Lamport's algorithm is that 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:
   $$V_i[j] = 0 \text{ for } i,j = 1, \ldots, N$$
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:
   $$V_j[i] = max(V_j[i], t[i]) \text{ for } i=1, \ldots, N$$

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 condition that if $V(e) < V(e')$ then $e \rightarrow e'$. Two events e, e' are concurrent if *neither* $V(e) \leq V(e')$ nor $V(e') \leq V(e)$. The disadvantage with vector clocks is the greater storage and message payload size, since an entire vector rather than a single integer must be manipulated. We can examine Figure 5 with vector clocks (Figure 6) and see how events a and

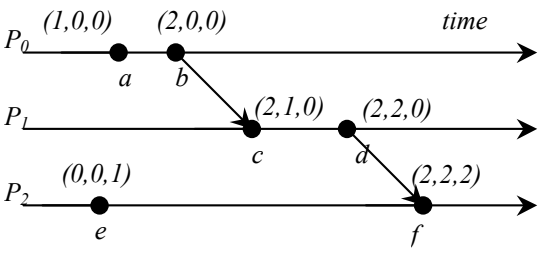

Figure 6. Messages with vector timestamps

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.

Physical clocks

Most computers today keep track of the passage of time with a battery-backed up CMOS clock circuit, driven by a quartz oscillator. This allows the timekeeping to take place even if the machine is powered off. When on, an operating system will generally program a timer circuit to generate an interrupt periodically (common times are 60 or 100 times per second). The interrupt service procedure simply adds one to a counter in memory.

The only problem with maintaining a concept of time is when multiple entities attempt to do it concurrently. 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 a 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 7 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.

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. To complicate matters, one constraint that we'll impose is that it's not a good idea to set the clock back. The illusion of time moving backwards can confuse message ordering and software development environments.

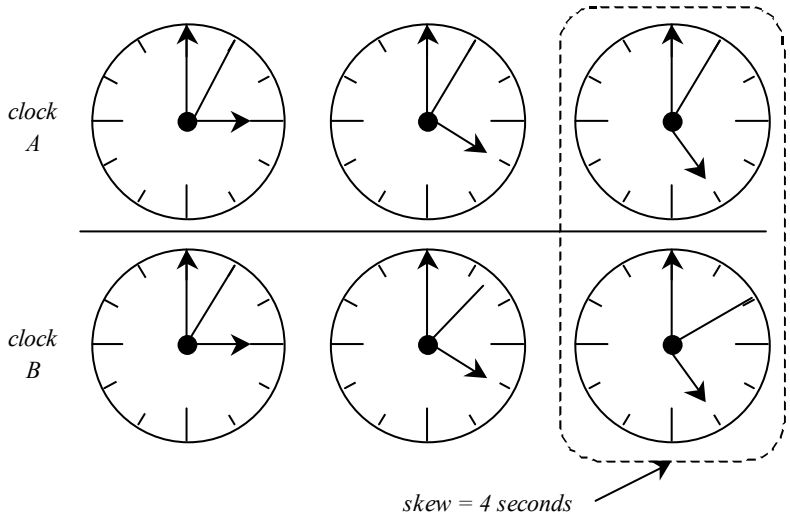

Figure 7. Clock drift and clock skew

If a clock is fast, it simply has to be made to run slower until it synchronizes. If a clock is slow, the same method can be applied and the clock can be made to run faster until it synchronizes. 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. This adjustment changes the slope of the system time and is known as a *linear compensating function* (Figure 8). After the synchronization period is reached, one can choose either 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 UNIX System V man page for *adjtime*.

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 get the time first so that we would know what to adjust.

One possibility is to attach a GPS (Global Positioning System) receiver to each computer. A GPS receiver will provide time within ± 1 msec. of UTC time and can be had for as little as US $100. Alternatively, if the machine is in the U.S., one can attach a WWV radio receiver to obtain time broadcasts from Boulder, CO or Washington, DC giving accuracies of ± 3–10 msec., 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 msec. of UTC time. Unfortunately, for reasons of economy, convenience, and reception, these are not practical solutions for *every* machine. Most machines will set their time by asking another machine for the time (preferably one with one of the aforementioned time sources). A machine that provides this information is called a time server.

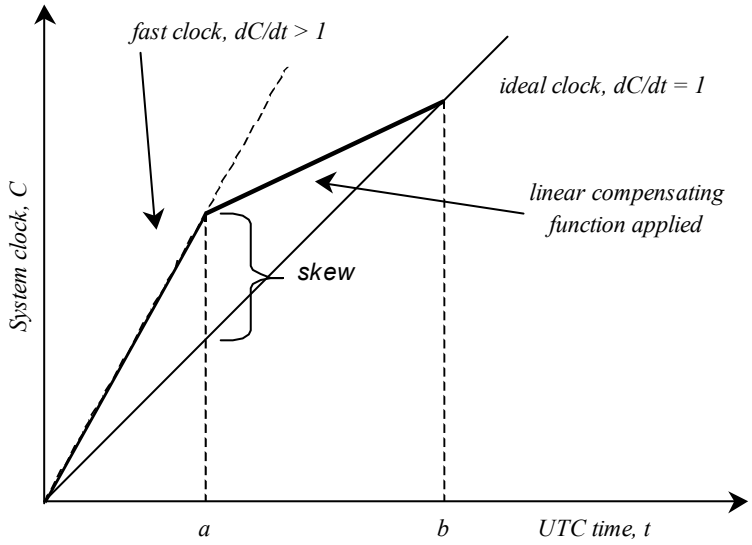

Figure 8. Compensating for drift with a linear compensating function.

Cristian's algorithm

The simplest algorithm for setting the time would be to simply issue a remote procedure call to a time server and obtain the time. That does not account for the network and processing delay. We can attempt to 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 (we have no reason to believe otherwise). 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:

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

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:

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

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.

Cristian's algorithm suffers from the problem that afflicts all single-server algorithms: the server might fail and clock synchronization will be unavailable. It is also subject to malicious interference.

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. The machines involved each run a time daemon process that is responsible for implementing the protocol. One of the machines is elected (or designated) to be the master. The others are slaves. The server polls each machine periodically, asking it for the time. The time at each machine may be estimated by using Cristian's method to account for network delays. When all the results are in, the master computes the average time (including its own). The hope is that the average cancels out the individual clock's tendencies to run fast or slow. Instead of sending the updated time back to the slaves, which would introduce further uncertainty due to network delays, it sends each machine the offset by which its clock needs adjustment. The operation of this algorithm is illustrated in Figure 9. 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

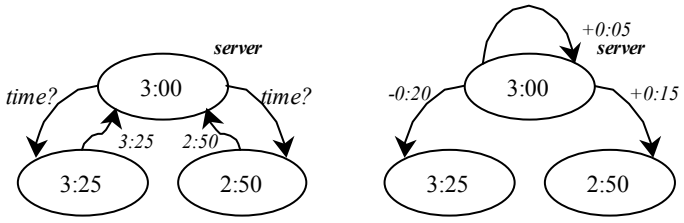

Figure 9. Berkeley synchronization algorithm

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 machine with a time of 3:25 gets sent an offset of -0:20 and the machine with a time of 2:50 gets an offset of +0:15. The server has 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 machine fails, any other slave could be elected to take over.

Network Time Protocol (NTP)

The Network Time Protocol [1991, 1992] is an Internet standard (version 3, RFC 1305) 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.

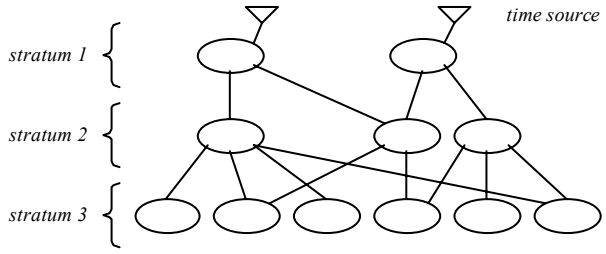

Figure 10. NTP synchronization subnet

The NTP servers are arranged into strata: the first stratum contains the primary servers, which are machines that are connected directly to an accurate time source. The second stratum contains the secondary servers. These machines are synchronized from the primary stratum machines. The third stratum contains tertiary servers that are synchronized from the secondaries, and so on. Together, all these servers form the **synchronization subnet** (Figure 10).

Machines synchronize in one of the following modes:

- symmetric active mode: a host sends periodic messages regardless of the reachability state or stratum of its peer
- symmetric passive: this mode is created when a system receives a message from a peer operating in symmetric active mode and persists as long as the peer is reachable and operating at a stratum less than or equal to the host. This is a mode where the 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: similar to Cristian's algorithm; a client announces its willingness to by synchronized by the server, but not to synchronize the server.

- multicast mode: intended for high speed LANs; relatively low accuracy but fine for many applications.
-

All messages are delivered unreliably via UDP. 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 mesge to a particular time server.
3. Dispersion: this is the "quality of estimate" (also known as *filter dispersion*) 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 an adaptation of the Network Time Protocol that allows operation in a stateless remote procedure call mode or multicast mode. 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
- 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[2].

NTP and SNTP messages are both sent via UDP (there is no point in having time reports delayed by possible TCP retransmissions). The message structure contains:

Leap indicator	warn 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,

[2] This is a somewhat different definition of *anycast* than that used in IPv6.

	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 (64 bits)	time at which local clock was last set or corrected
Originate timestamp (64 bits)	time at which request departed the client for the server
Receive timestamp (64 bits)	time at which the request arrived at the server
Transmit timestamp (64 bits)	time at which the reply departed the server
Key identifier (32 bits)	used if the NTP authentication scheme is implemented
Message digest (128 bits)	used if the NTP authentication scheme is implemented

In unicast mode, the roundtrip delay and local offset are calculated as follows (from RFC2030):

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

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.globecom.net/(eng,nocl)/ietf/rfc/rfc1305.shtml. Another is http://www.garlic.com/~lynn/rfcidx4.htm#1305.