Clock Synchronization

Amruth Dattatreya
Computer Science and Engineering
University of Texas at Arlington
dattatre@cse.uta.edu

March 26th 2002

Abstract:

The concept of time is fundamental to our way of thinking. Similarly the distributed system uses clocks to synchronize between different processes. Accurate and synchronized clocks are extremely needed and useful to co-ordinate activities between co-operating processors and therefore very important in a distributed environment. Although computers usually have a hardware-based clock, most of them are imprecise and have a substantial drift. Furthermore, these clocks are prone to faults and malicious resetting. Hence in order to behave as a single, unified computing resource, distributed systems need a fault-tolerant, clock synchronization service. A clock synchronization service ensures that spatially dispersed processors in a distributed system share a common notion of time. This clock synchronization service must deal with communication delay uncertainties, clock imprecision and drift, as well as link and processor faults. This paper examines the different ways the clocks are synchronized and the algorithms implemented. Many of the algorithms proposed for Clock Synchronization take either software or a hardware approach.

1.Introduction:

Digital computers have become essential to critical real-time applications such as aerospace systems, life support system, nuclear power plants and computer integrated manufacturing systems. All these applications demand for maximum reliability and high performance from computer controllers. The performance to a large extent depends on the quality of synchronization between the processors. Because of such stringent requirements Clock Synchronization becomes one of the fundamental problems of distributed computing [12]. The discrepancy between clock readings is called the ***tightness*** of synchronization.

The basic difficulty in clock synchronization is that timing information tends to deteriorate over the temporal and spatial axes. A fault free hardware clock, even if initially synchronized with a standard time reference, tends to drift away from the standard over a period of time. As a result, an interval of time measured with such a clock tends to be in error. However, the rate at which the hardware clock deviates from the standard is bounded by a constant, which is known as ***maximum drift rate*** of the clock. A direct consequence of this phenomenon is that clocks in a distributed system gradually deviate from each other over a period of time. A ***clock synchronization algorithm*** is used in a distributed system to ensure that the skew that develops between clocks remain bounded. Usually it is assumed that local clocks have known lower and upper bounds on their rate of progress with respect to real time. These bounds are called as ***drift bounds.*** In addition, it is also assumed that there are known lower and upper bounds on the time required to transmit a message. These bounds are called as the ***message latency bounds***. The essence of all clock synchronization problems is how to use these bounds to obtain tight synchronization.

This paper is organized as follows: Section 2 discusses some of the algorithms and the approaches used. Section 3 lists some of the practical uses of synchronized clocks in distributed system. Finally, Section 4 concludes on Clock Synchronization.

2. Approaches and Algorithms:

2.1. Background:

Synchronized clocks are crucial for many fault-tolerant real time systems. Clocks can be ***externally*** or ***internally synchronized*** [9]. A clock is ***externally synchronized*** if at any point in real-time the distance between its value and reference time is bounded by an a priori given constant called *maximum external deviation*. A set of clocks is ***internally synchronized*** if at any point in real-time the distance between the values of two correct clocks in the set is bounded by an a priori given constant called *maximum internal deviation* and each clock runs within a linear envelope of real time. Usually loosely coupled systems use external synchronization and tightly coupled systems use internal synchronization.

Clock skews that are significantly smaller than the theoretical limit described by Lundelius and Lynch [5] can be achieved if the requirement of determinism is relaxed and accept a *probabilistic* guarantee. This technique is called as the ***Probabilistic approach*** for Clock Synchronization proposed by Cristian [14]. In this approach, the transmission times of messages are assumed to adhere to some probability distribution and the transmission times of other messages are assumed to be independent. Under these assumptions, some guarantees can be made by the synchronization protocol. A *guarantee* is said to be probabilistic if it fails to hold for sometimes, but with a *failure probability* that can be determined or bounded. A clock synchronization algorithm that provides a probabilistic guarantee on the maximum clock skew is referred to as a *probabilistic clock synchronization algorithm*.

2.2. Cristian's Probabilistic Algorithm [14][15]*:*

The idea of probabilistic clock synchronization was proposed by Cristian. This algorithm is well suited to systems in which one machine acts as a *time server* and the goal is to have all the other machines stay synchronized with it.

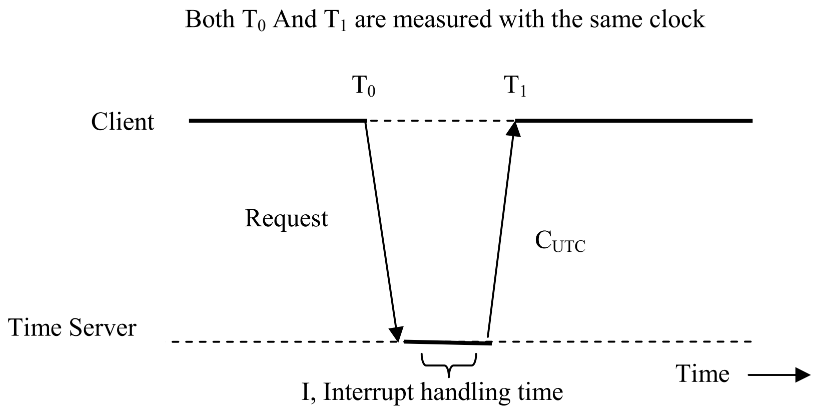

Both T_0 And T_1 are measured with the same clock
T_0
T_1
Client
Request
C_{UTC}
Time Server
I, Interrupt handling time
Time

Fig 2.1

The Cristian algorithm is based on *Remote Clock Reading [RCR].* Remote Clock Reading is used by a machine [Client] to read the clock at a remote machine [Time server] with a specified minimum accuracy. RCR involves querying a target node for the time on it clock, i.e. periodically, certainly no more than every $\delta/2\rho$ seconds, each machine sends a message to the time server asking for its current time. That machine responds as fast as it can with a message containing its current time, C_{UTC}, as shown in the Fig 2.1. The querying node then estimates time on the target nodes clock from the response received.

As a first approximation, when the sender gets the reply, it can just set its clock to C_{UTC}. However, this algorithm has two problems. The *major problem* is that the time must never run backward. If the senders clock is fast, C_{UTC} will be smaller than the sender's current value of clock. Just taking over C_{UTC} could cause serious problems such as an object file compiled after the clock change having a time earlier than the source that was modified before the clock change. Such a clock change must be introduced gradually. Suppose that the timer is set to generate 100 interrupts per second. Normally, each interrupt would add 10msec to the time. When slowing down, the interrupt routine adds 9msec each time until the correction has been made. Similarly, the clock can be advanced by adding 11msec at each interrupt instead of jumping it forward all at once. The *minor problem* is that it takes a non-zero amount of time for the time server's reply to get back to the sender, yet this delay may be large and vary with the network load. A good way of dealing with it would be to measure it. Both the starting time T_0 and the ending time T_1 are measured with the same clock and the sender can record accurately the interval between sending the request to the time server and the arrival of the reply. The best estimate of the message propagation time is $(T_0 - T_1)/2$. When the reply comes in, the value in the message can be increased by this amount to give an estimate of the server's current time. The time estimate can be further improved if it is known approximately how long it takes the time server to handle the interrupt and process the incoming message. Then the amount of the interval from T_0 to T_1 that was devoted to message propagation is $T_1 – T_0 – I$, where I is the interrupt handling time. RCR guarantees that the maximum estimation error is approximately $T – T_{min}$, where T is the half the response time and T_{min} is the minimum response time.

To improve the accuracy, Cristian suggested making a series of measurements. RCR repeatedly queries the target node until rapport is achieved. Any measurements in which $T_1 – T_0$ exceeds some threshold value are discarded as being victims of network congestion and thus are unreliable. The estimates derived from the remaining probes can then be averaged to get a better value. If there is any constraint on the number of attempts to achieve rapport, it may never be achieved. Alternatively, the message that came back fastest can be taken to be the most accurate since it presumably encountered the least traffic underway and therefore is the most representative of the pure propagation time. This algorithm is a master-slave algorithm that makes use of RCR to achieve synchronization. However, resynchronization is not guaranteed, because RCR may fail to achieve rapport. The Cristain's Algorithm is a probabilistic algorithm and can hence guarantee much smaller clock skews than deterministic algorithms.

2.3. Network Time Protocol [NTP][3][13]:

2.3.1 Background:

The Network Time Protocol [NTP], now established as Internet Standard Protocol, is used to organize and maintain a set of time servers and transmission paths as a synchronization subnet.

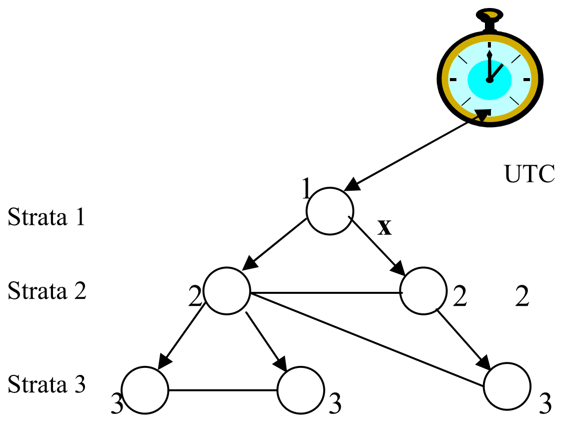


Fig 2.2(a)

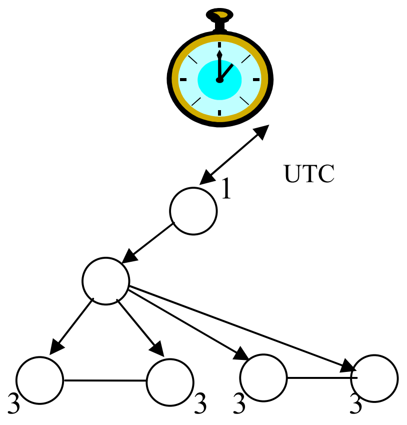


Fig 2.2(b)

NTP is built on the *Internet Protocol [IP]* and *User Datagram Protocol [UDP],* however it is readily acceptable to other protocol suites. It is specifically designed to maintain accuracy and reliability.

In NTP one or more primary severs synchronize directly to external reference sources such as timecode receivers. Secondary time receivers synchronize to the primary servers and others in the synchronization subnet. A typical subnet is shown in Fig 2.2(a). The nodes represent *subnet servers*, with normal *stratum*, the accuracy of each time server, numbers determined by the hop count to the root and the arrows represent the active synchronization paths and the direction of timing information flow. The lines represent backup synchronization paths where timing information is exchanged, but not necessarily used to synchronize the local clocks.

2.3.2. Implementation model:

NTP time servers can operate in one of the three service classes: *multicast, procedure-call* and *symmetric*. These classes are distinguished by the number of peers involved, whether synchronization is to be given or received and whether state information is retained. In the typical scenario one or more time servers operating in *multicast mode* send periodic NTP broadcasts. The workstation peers operating in the client mode then determine the time on the basis of an assumed delay in the order of a few milliseconds. By operating in multicast mode the server announces its willingness to provide synchronization to many other peers, but to accept NTP messages from none of them. The multicast mode is intended for use on high speed LANs with numerous workstations and where the highest accuracies are needed.

In a *procedure call mode*, a time server operating in a client mode sends a NTP message to a peer operating in server mode, which then interchanges the addresses, inserts the required timestamps, recalculates the checksum and optional authenticator and returns the message immediately. By operating in a client mode a server announces its willingness to be synchronized by, but not provide synchronization to a peer. By operating in server mode a server announces its willingness to be synchronized to, but not be synchronized by a peer. The full generality of NTP requires distributed participation of a number of time servers arranged in a dynamically reconfigurable, hierarchically distributed configuration. This is when *symmetric mode [active and passive]* is used. By operating in these modes a server announces its willingness to synchronize to or to be synchronized by a peer, depending on the peer-selection algorithm. Symmetric active mode is designed for use by servers operating near the leaves of the synchronization subnet. Symmetric passive mode is designed for use by servers operating near the root.

The transmit process, driven by independent timers for each peer, collects information in the data base and sends NTP messages to the peers. Each message contains the local timestamp, together with previously received timestamps and other information necessary to determine the hierarchy and manage the association. The message transmission rate is determined by the accuracy of the local clock, as well as accuracies of its peers.

The receive procedure is called upon arrival of NTP messages, which is then matched with the association indicated by its addresses and ports. This results in the creation of a persistent association for a symmetric mode or a transient one for the other modes. When an NTP message is received, the offset between the peer clock and the local clock is computed and incorporated into the database along with other information useful for error determination and peer selection.

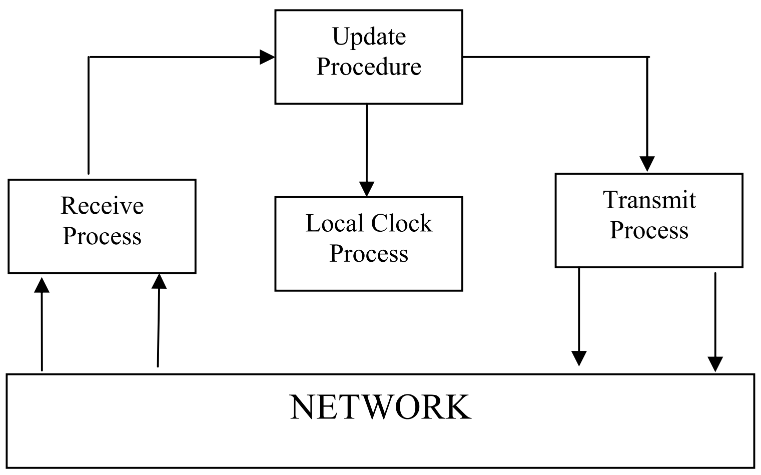


Fig 2.3: Implementation Model

The update procedure is called when a new set of estimates becomes available. A weighted voting procedure determines the best peer, which may result in a new synchronization source. This may involve many observations of a few peers or a few observations of many peers, depending on the accuracies required. The local clock process operates upon the offset data produced by the update procedure and adjusts the phase and frequency of the local clock using different mechanisms. This may result in either a step-change or a gradual phase adjustment of the local clock to reduce the offset to zero. The local clock provides a stable source of time information to other users of the system and for subsequent reference by NTP itself.

NTP provides the mechanisms to synchronize and coordinate time in a large, diverse Internet operating at different rates. The purpose of NTP is to convey timekeeping information from a standard primary reference server to other time servers via the Internet and also to cross check clocks and mitigate errors due to equipment or propagation failures. Experiments have shown that accuracies of 10s msec over Internet and 1msec on LAN using NTP can be achieved.

2.4. Lamport's Logical Clock Synchronization [4][15]*:*

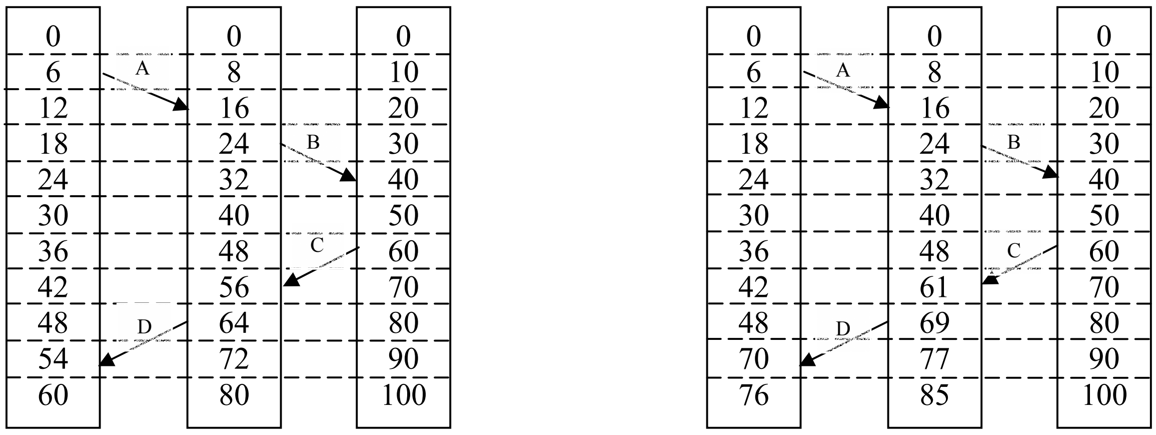
For many purposes, it is sufficient that all machines agree on the same time. It is not essential that this time also agree with the real time. For certain class of algorithms, it is the internal consistency of the clocks that matters, for which it is conventional to speak of the clocks as *Logical Clocks*. Clock synchronization [internal and external] cannot be sufficiently precise in order to use timestamping for the determination of total orderings in different processes in a distributed system. Lamports algorithm is examined here for synchronizing a system of logical clocks that can be used to totally order the events. Lamport's approach to logical clocks is used in many situations in distributed systems where ordering is important but global time is not required.

To synchronize logical clocks, Lamport defined a relation called **happens-before.** The expression $a \longrightarrow b$ is read " a happens before b" and means that all processes agree that first event '*a'* occurs, then afterward, event '*b*' occurs. The happens-before relation can be observed directly in two situations:

- If '*a'* and '*b'* are events in the same process and '*a'* occurs before '*b'*, then $a \longrightarrow b$ is true.
- If '*a'* is the event of a message being sent by one process and '*b'* is the event of the message being received by another process, then $a \longrightarrow b$ is also true.

Happens-before is a *transitive relation.* If two events x and y, happen in different processes that do not exchange messages, then $x \longrightarrow y$ is not true, but neither is $y \longrightarrow x$. These events are said to be *concurrent.* A way of measuring time is needed such that for every event, '*a*', we can assign it a time value C(a) on which all processes agree. These time values must have the property that if $a \longrightarrow b$, then C(a) < C(b). Similarly, if '*a*' is the sending of a message by one process and '*b*' is the reception of that message by another process, then C(a) and C(b) must be assigned in such a way that everyone agrees on the values of C(a) and C(b) with C(a) < C(b). In addition, the clock time C, must always go forward (increasing), never backward (decreasing). Corrections to time can be made by adding a positive value, never by subtracting one.

Consider three processes as shown in the ***Fig 2.4***



The processes run on different machines, each with its own clock, running at its own speed. As can be seen from the figure, when the clock has ticked 6 times in process 0, it has ticked 8 times in process 1 and 10 times in process 2. Each clock runs at a constant rate, but the rates are different due to differences in the crystals. At time 6 process 0 sends message A to process 1. The arrival rate of this message depends on the clock that you choose. In any event, the clock in process 1 reads 16 when it arrives. If the message carries the starting time, 6, in it, process 1 will conclude that it took 10 ticks for the message to reach its destination. Accordingly, message B from process 1 to process2 takes 16 ticks, which is also a plausible value. Message C from process 2 to process1 leaves at 60 and arrives at 56. Similarly, message D from process 1 to process 0 leaves at 64 and arrives at 54. These values are clearly impossible. It is this type of situation that should be prevented.

Lamport's solution follows directly from the happens-before relation. Since C left at 60, it must arrive at 61 or later. Therefore, each message carries the sending time according to the sender's clock. When a message arrives and the receiver's clock shows a prior to the time the message sent, the receiver fast forwards its clock to be one more than the sending time. In fig 2.4(b) we can see that C now arrives at 61. Similarly D arrives at 70. With one small addition, this algorithm meets our requirements for global time. The addition is that between every two events, the clock must tick at least once. If a process sends or receives two messages in quick succession, it must advance its clock by at least one tick in between them. In some situations, an additional requirement is desirable: no two events ever occur at exactly the same time. To achieve this goal, we can attach the number of the process in which the event occurs to the low-order end of time, separated by a decimal point. Thus if events happen in processes 1 and 2, b0th with time 25, the former becomes 25.1 and the latter becomes 25.2

This algorithm gives way to provide a total ordering of all events in the system. Many other distributed algorithms need such an ordering to avoid ambiguities.

3. Uses of Synchronized Clocks:

3.1. At-most-once Messages

Synchronized Clocks is used in the *SCMP protocol* that guarantees at-most-once delivery of messages [6]. Implementing at-most-once semantics is typically done by having each message receiver maintain a table containing information about "active" senders that have communicated with the receiver recently. When a message arrives, if there is information about the sender in the table it is used to determine whether or not the message is a duplicate. If there is no information, there are two choices: *either accept the message or reject it*. If the message is accepted, there is a chance of accepting a duplicate. This chance can be made arbitrarily small by keeping information about senders long enough. The alternative of rejecting the message guarantees that no duplicates will ever be accepted. However, it gives rise to a problem that can be solved by means of a *handshake*.

The SCMP protocol avoids the handshake between the sender and receiver by using synchronized clocks. The idea is the receiver remembers all “recent” communications. If a message from a particular sender is “recent,” the receiver will be able to compare it with the stored information and decide accurately whether the message is a duplicate, If the message from the sender is “old,” it will be tagged as a duplicate even though it may not be, but this case is very unlikely. Thus the system will never accept a duplicate but it may occasionally reject a non-duplicate. For the scheme to work, receivers need to know whether a message is “recent.” When a node sends a message, it timestamps the message with the current time of its clock. When the message arrives at the receiver, it is considered recent if its timestamp is later than the receiver’s local time minus the message lifetime interval ρ; otherwise it is old. The message lifetime interval must be big enough so that almost all messages will arrive within ρ time units of when they were

Synchronized clocks allow the protocol to establish a system-wide notion of “recent.” Clocks are used to avoid communication and to save storage at receivers (only timestamps of recent messages need be saved). Timestamps identify messages that have already arrived, the identification is only approximate, since a single timestamp stands for all earlier messages, and therefore sometimes a message that is not a duplicate will be rejected. If clocks get out of synch, there is no danger of a duplicate message being accepted, but recent messages may be flagged as duplicates. If a node’s clock is slow, its messages are more likely to be flagged as duplicates by other modules; if its clock is fast; it is more likely to flag messages from other modules as duplicates.

3.2. Cache Consistency:

The next use of Synchronized Clock concerns systems in which servers provide persistent storage for objects and programs that use those objects run at client workstations [6][8]. To provide reasonable response time to clients, copies of persistent objects are cached at the workstations so that clients can use them locally when there is a cache hit. As is the case in any system with cached copies, cache consistency is a concern. One possibility is to use “leases”. This concept is used in a file system, so the objects in question are files. In the initial use of leases, the caches were writethrough, newer systems use write-behind. This cache behavior leads to a difference in how leases are used. The systems in fact only require synchronized clock rates.

In the case of the *write-through cache*, leases work as follows. Each client workstation obtains a lease for a file when the file is copied into its cache. The lease contains an expiration time E, when E has been reached, i.e., when E > time(client) - ε the client stops using the file. The client can then request that a lease be renewed by asking the server for a new expiration time. When a client modifies a file, the modification goes directly to the server (since this is a write-through cache). The server can do the modification immediately if there are no other outstanding leases on the file. Otherwise it communicates with the clients holding the leases, requesting them to give them up. The modification is done when all leases have been relinquished. Of course, it is possible that a current holder of a lease might not respond, either because of a network problem, or because of a crash of its node. In this case, the system will wait until the expiration time of the lease, and then do the modification. In *write-behind caches* there are two kinds of leases, read leases and write leases, and a client must use the file in accordance with its lease. Thus a client with a read lease can only read the file, while a client with a write lease can both read and write the file. Each lease has an expiration time as discussed above. The only difference is that competition for leases now occurs when a client requests a lease (rather than when a file is written), If a client needs a lease that conflicts with leases held by other clients, the server sends messages to the other clients requesting them to relinquish their leases.

The invariant of interest in a system with leases is, each time a client uses a file, it has a valid lease for that file. Validity is determined by using the client's clock as an approximation of the time of the server's clock. If the client's clock is slow, or the server's clock is fast the invariant will not hold. In this case, the client may continue to use the file after its lease has expired at the server. In the absence of the use of synchronized clocks there are two possibilities for maintaining cache consistency, neither of which is desirable. One alternative is for the client to check the validity of each file use. The other alternative is for the server to not invalidate a client's lease until it hears from the client. Thus the system is presented with two unattractive choices: either depend on assumptions such as synchronized clocks that might fail causing inconsistencies, or sacrifice performance. Choosing to improve performance is a valid position, given the low probability of clocks getting out of synch.

4. Discussion and Conclusion:

Clock Synchronization is an important matter in any fault-tolerant distributed system and has been extensively studied in recent years. A clock synchronization algorithm lets processors adjust their clocks, to overcome the effects of drifts and failures. Some of the relevant works are focused here:

A number of works focus on handling processor faults, but ignore drifts, as drift rates are quite small. A model of time-adaptive self-stabilization suggested by Kutten and Patt-Shamir has the goal to recover from arbitrary faults at '*f*' processors in time that is a function of '*f*' but the main problem being it assumes periods of no faults. Among the works dealing with both processor faults and drifts, most assume that once a processor failed, it never recovers and there is a bound '*f*' on the number of failed processors throughout the lifetime of the system. The Network Time Protocol, designed by Mills, allows recoveries, but without analysis and proof. Furthermore, while authenticated versions of Mills were proposed, so far these do not attempt recovery from malicious faults. The probabilistic internal clock synchronization algorithm suggested by Fetzer and Cristian gives a probabilistic guarantee of attaining clock skews that are significantly smaller than the theoretical limit. Specifically, the algorithm tries to minimize the change made to the clocks in each synchronization operation. Using such small correction may delay the recovery of a processor with a clock very far from the correct one. The Lamports algorithm introduced the concept of "happening-before" and used logical clocks to achieve synchronization in distributed environment. The total ordering defined is somewhat arbitrary. It can produce anomalous behavior if it disagrees with the ordering perceived by the systems users.

The various solutions proposed are difficult to compare because they are presented under different notations and assumptions. Additional difficulty arises because of slight differences in the assumptions made by different synchronization algorithms. Hence we will classify the algorithms based on the approach namely the software or the hardware approach.

The software approach is flexible and economical. The software algorithms require nodes to exchange and adjust their individual clock values periodically. The clock values are exchanged via message passing solely for synchronization. Because they depend on message exchanges, the worst case skews guaranteed by most of these solutions are greater than the difference maximum and minimum message transit delay between any two nodes in the system.

The hardware approach, on the other hand, uses special hardware at each node to achieve a tight synchronization with minimal time overhead. However, the cost of additional hardware precludes this approach in large distributed systems unless a very tight synchronization is essential. Hardware solutions also require a separate network of clocks that is different from the interconnection network between the nodes of the distributed systems.

Because of these limitations in the software and hardware approaches, researchers have begun investigating a hybrid approach. A hardware assisted software synchronization is proposed that strikes a balance between the hardware requirement at each node and the clock skews attainable.

5. References

1. B.Barak, A.Herzberg, D.Naor and E.Shai, "*Clock Synchronization with Faults and Recoveries".*
2. F.Cristian and C.Fretzer, "*Probabilistic Internal Clock Synchronization Algorithm" Distributed Computing, 3:146-158, Press/McGraw-Hill, 1990.*
3. D.L.Mills, *"Internet Time Synchronization: The Network time Protocol". IEEE Trans. Comm, 39(10): 1482-1493, Oct 1991.*
4. L.Lamport " *Time, Clocks and the Ordering on events in a distributed system". Comm. ACM, 21(7): 558-565, July 1978.*

5. J.Lundelius and N.Lynch. "*An upper and lower bound for clock synchronization". Information and Computation, 62(2-3): 190-204, 1984.*
6. B.Liskov. "*Practical uses of synchronized clocks in distributed systems". Distributed Computing, 6: 211-219, 1993.* Invited talk at the 9th Annual ACM symposium on principles of Distributed Computing, 1990.
7. B.Patt-Shamir and S.Rajsbaum. " *A theory of clock synchronization". In proceedings of 26th Annual ACM symposium on Theory of Computing, Montreal, Canada, pages 810-819, May 1994.*
8. B.Simmons, J.L Welch and N.Lynch. " *An overview of clock synchronization". Research report RC 6505(63306), IBM, 1998.*
9. C.Fetzer and F. Cristian. " *Integrating* e*xternal and internal clock synchronization". In proceedings of the Thirteenth symposium on Reliable Distributed Systems, Dana Point, CA Oct 1994.*
10. P.Ramanathan, K.G. Shin and R.W. Butler. " *Fault-Tolerant clock synchronization in Distributed Systems". IEEE Trans. Computers, Vol C-37, No11, Nov 1988.*
11. T.K. Srikanth and S.Toueg. *"Optimal clock synchronization". Journal ACM, Vol 34, No 3, July 1987.*
12. Boaz Patt. *" A Theory of Clock synchronization".*
13. http://www.eecis.udel.edu/~ntp/database/time_pub.html
14. F.Cristian. *" Probabilistic clock synchronization". Tech report RJ 6432(62550), IBM Almaden Research Center, Sept 1988.*
15. Andrew.S.Tanenbaum and Maarten van Steen. *"Distributed Systems: Principles and Paradigms".*