the Availability Digest

Time Synchronization in Distributed Systems – Part 3

February 2008

In distributed systems, it is often imperative that all nodes in the system have the same view of time; and that time must be synchronized with real civil time according to some standard time reference. In the Internet world, this is typically done with the Network Time Protocol, NTP. NTP is one of the earliest Internet protocols used and is probably one of the most used protocols today. In Parts 1 and 2 of this series,[1] we described the processes by which NTP accurately synchronizes every node in a distributed system with a reference civil-time source.

However, in many cases, we can relax the time requirements provided by NTP. Specifically, though it is imperative that the clocks of all nodes be synchronized with each other, it is often not necessary that the network be synchronized with civil time. If the network time is a few seconds or even more in error, this may not cause a problem so long as all nodes agree upon a common time.

This occurs when relative time is necessary to properly order events but when the measurement of absolute time for each event need not be terribly accurate.

Logical clocks, first proposed by Leslie Lamport in a 1978 paper published in the Communications of the ACM (Association for Computing Machinery), [2] provide just this function in a much simpler way than NTP. Lamport was awarded the prestigious Dijkstra Prize by the ACM for his work on logical clocks.

Logical Clocks

A logical clock is a clock that keeps accurate time independent of any precise reference time. Therefore, though it may measure time intervals correctly, its relationship to actual time may be in error.

Many distributed transaction processing systems depend upon all nodes seeing the same time so that they may properly order events (for instance, which transaction occurred first?). However, if there are no critical ties with other independent systems, then the absolute time of these events may not be terribly important so long as the timestamps are not out of reason. For instance, how many nondistributed transaction processing systems depend only upon their own internal clock, which may not have been accurately set by the operator initially and which may drift over time?

Lamport's logical clocks solve this problem with a simple algorithm.

[1] Time Synchronization for Distributed Systems – Part 1, *Availability Digest*; November, 2007.
Time Synchronization for Distributed Systems – Part 2, *Availability Digest*; December, 2007.
[2] Lamport, L., Time, Clocks, and the Ordering of Events in a Distributed System, Communications of the ACM; July, 1978

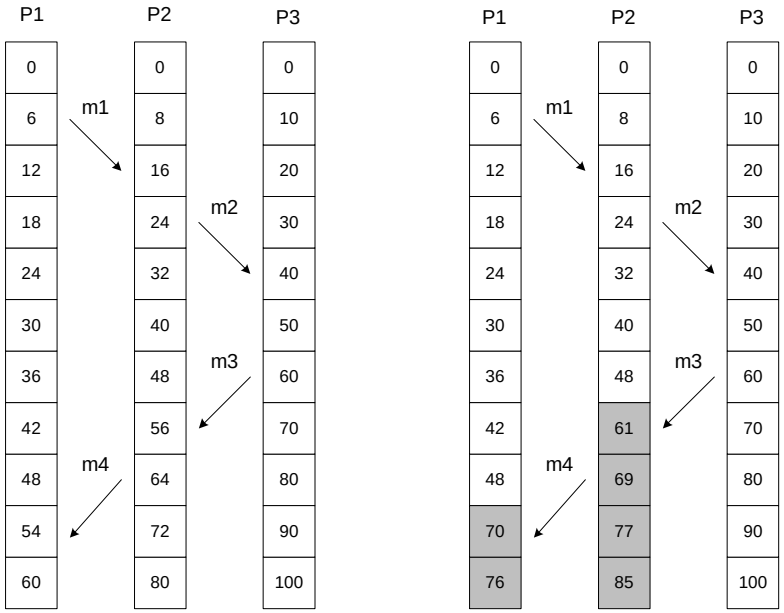
Lamport's Logical Clocks

As with any clock, a logical clock keeps track of time via clock ticks. The greater the value of the clock tick, the later the time. Lamport's algorithm depends only upon two obvious rules:

Rule 1: Within a system, there is a minimum of one clock tick between any two events.

Rule 2: If Node A sends a message to Node B, Node B receives that message after Node A sent it.

Figure 1 shows an example in which three nodes, P1, P2, and P3, are cooperating with each other. Each has its own clock, in which the frequencies are slightly off (greatly off in the example for purposes of illustration). P1's clock increases by a factor of 6 on each tick. P2's clock increases by a factor of 8 on each tick. P3's clock increases by a factor of 10 on each tick. Left alone over a period of time, they would diverge, as shown in Figure 1a.



Clocks Running at Different Rates
(a)

Lamport's Algorithmic Correction
(b)

Lamport's Logical Clocks
Figure 1

Using Lamport's algorithm, these clocks can be forced into synchronization, as shown in Figure 1b. Node P1 is shown sending a timestamped message m1 to P2. The message leaves P1 at time 6 and arrives at P2 at time 16. This satisfies Lamport's second rule. The same holds for message m2 sent by P2 to P3.

However, when P3 returns a message to P2 at time 60, it is received by P2 at time 56. This violates Rule 2 as the message must arrive after it was sent. Therefore, P2's clock is advanced so that its clock tick 56 is advanced to 61, one tick after the message was sent by P3. All further ticks of P2 are similarly adjusted, incrementing by 8 on each tick.

Likewise, when message m4 is sent to P1 by P2, it leaves P2 at its new time 69 and is received by P1 at time 54. This again violates Rule 2, so P1's clock is advanced so that its old tick 54 is now one tick later than when P2 sent the message, or 70.

As can be seen from Figure 1, following this exchange of messages, the clocks are more closely synchronized than they were before the Lamport adjustments. Over time, this procedure will bring the clocks even more in synchronism until they are synchronized arbitrarily accurately, especially for high message traffic.

Thus, Lamport logical clocks will be maintained in close synchronism across the network, though their relation to real civil time is indeterminate.

Effect of Channel Latency

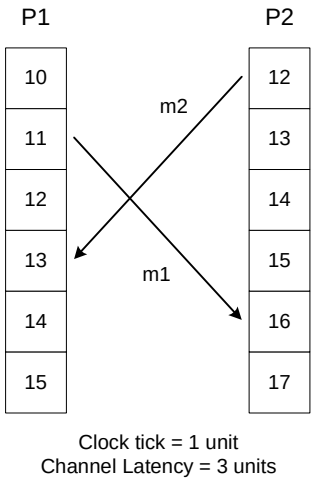
Figure 1 has used the example of clock ticks being greater than the channel latency. Each clock tick represents 6 or more units of time, and the channel latency is one unit of time. In this case, the clocks will approach perfect synchronism over time.

However, synchronization between two clocks can be no better than the latency of the communication channel connecting the two systems (a situation not considered by Lamport). To understand this, consider Figure 2. Figure 2 shows a pair of logical clocks in which each clock tick is one unit of time, and the channel latency is three units of time (just the opposite situation from that of Figure 1).

P1 is shown as being behind P2 by two clock ticks. P2 sends message m2 at its time 12. On the next clock tick, P1 sends P2 its message m1. Clearly, m1 was sent *after* m2.

Message m2 will arrive at P1 three time ticks later at P1's time 13. P1 is happy with this sequence. P2's message indicates that it was sent at time 12, and it is received by P1 at time 13, thus satisfying Lamport's second rule. Likewise, P1's message sent at time 11 is received by P2 at time 16, also satisfying this rule. Based on the message timestamps, P1 would conclude that its message, m1, was sent before P2's message m2. Furthermore, no adjustment in time is made; and the system clocks remain in relative error by two clock ticks.

If the channel latency had been two clock ticks rather than three, m2 sent at time 12 would be received by P1 at time 12. P1's clock would be advanced by one, but there would be no further corrections. The clocks would remain in relative error by one clock tick.



Effect of Channel Latency
Figure 2

If the channel latency had been one clock tick, m2 would be received by P1 at time 11. P1's clock would be advanced to time 13 (one tick later than when m2 was sent), putting it in synchronism with P2's clock. In this case, the clock tick interval is equal to the channel latency.

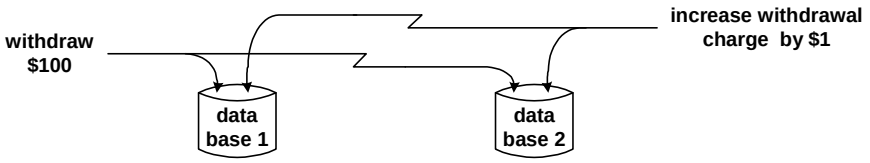
The lesson here is that there is no advantage to making the clock tick any shorter than the channel latency time, since the channel latency time will be the limit of synchronization accuracy. If the systems are 1,000 miles apart (representing a channel latency of about 10 milliseconds), there is no advantage to having a clock tick any shorter than 10 milliseconds. The resolution of event times is limited to the channel latency separating the systems.

Total Ordering of Events

Anomalous Behavior When Events Can't Be Ordered

The messages in Figure 1 may be related, or they may simply be independent messages. However, in many cases, the messages represent events that also must be ordered. For instance, consider two banking nodes in a distributed system, as shown in Figure 3, each with its own database that is intended to be an exact copy of the other. Each transaction is applied directly to the database that is closest to where it originated and is then applied across the network to the database of the remote node.

In this example, an ATM user withdraws $100 from an ATM near database 1. At nearly the same time, the bank increases the ATM charge by $1; and this is applied to database 2. The $100 withdrawal is propagated across the network to database 2, where it is applied after the increase in the withdrawal charge. Likewise, the withdrawal charge increase is propagated across the network to database 1, where it is applied after the withdrawal.



Event Ordering
Figure 3

Though the order in which these two events occur is not terribly important from an application viewpoint, it can be disastrous from the database's viewpoint. In this case, if the original withdrawal charge was $1, then database 1 would show a decrement to the user's account of $101, whereas database 2 would show a decrement of $102. The databases are now inconsistent. This situation must be avoided.

The Lamport Event-Ordering Algorithm

A simple extension of the Lamport algorithm[3] described above can provide complete ordering of events in addition to keeping the clocks synchronized. This is done by simply specifying that no event occurring at a node can be processed until that node has received messages from every other node with a later timestamp.

If there were no control on this, it might be a while before the node hears from all other nodes, thus delaying the processing of the event. However, this challenge can be solved by having the node experiencing the event notify the other nodes of the event and then having the other nodes acknowledge that message. This is, in fact, the way that transaction processing systems normally work. More specifically, the additional rules in the algorithm when acknowledge messages are used are:

Rule 3: Each node has a request queue of messages, which are ordered by message send time. When the node sends a message to other nodes, that message is placed in its request queue in send-time order.

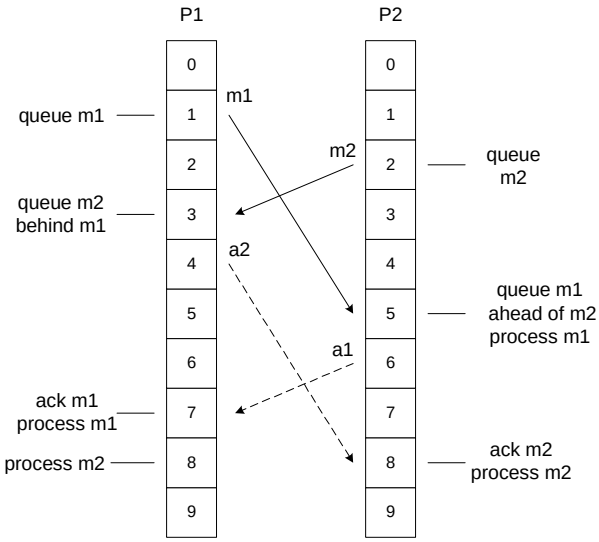
Rule 4: When a node receives a message from another node, it returns an acknowledge message to the sending node, puts the message in its request queue in send-time order, and marks the message as acknowledged.

[3] Tanenbaum, A. S.; Van Steen, M.; *Distributed Systems: Paradigms and Principals*, Pearson Education; 2007.

Rule 5: When a node receives an acknowledgement to one of its own messages from all of the nodes to which it had sent that message, it marks the message in its message queue as acknowledged.

Rule 6: If the message at the head of the queue has been acknowledged, it is sent to the application.

This process is illustrated in Figure 4. In Figure 4, there are two nodes in the system, P1 and P2. Their clocks are synchronized. P1 sends a message m1 to P2 at time 1. P2 sends a message m2 to P1 at time 2. If nothing else is done, P1 will process message m1 before message m2; and P2 will process message m2 before message m1. Therefore, there is no system-wide event ordering.



Ordered Executions
Figure 4

Let us now apply Lamport's algorithm for total ordering, as shown in Figure 4. The sequence of events is as follows:

Time 1: P1 sends message m1 to P2 and puts m1 in its message queue.

Time 2: P2 sends message m2 to P1 and puts m2 in its message queue.

Time 3: P1 receives message m2 and puts it in its message queue as an acknowledged message after m1 because its timestamp is later.

Time 4: P1 acknowledges message m2 with acknowledge message a2.

Time 5: P2 receives message m1 and puts it in its queue as an acknowledged message ahead of m2 because it is timestamped earlier. Since m1 is at the head of P2's queue and is marked as acknowledged, it is processed.

Time 6: P2 acknowledges message m1 via its message a1.

Time 7: P1 receives the acknowledgement to its message m1 and marks m1 acknowledged in its message queue. Since m1 is at the head of the queue, it can now be processed.

Time 8: P2 receives the acknowledgement to its message m2 and marks m2 acknowledged in its message queue. Since m2 is at the head of the P2 queue, it can now be

processed.

Also, m2 is now at the head of P1's message queue and can now be processed.

As a result, m1 has been processed first at both nodes; and m2 has been processed second. Total event ordering has been achieved.

Though this algorithm is an extension of Lamport's logical clocks for time synchronization, it does not depend on those sorts of clocks. It can be used with any mechanism that closely synchronizes clocks, such as NTP.

Data Collisions

The above example has avoided the issue of data collisions. What if messages m1 and m2 had been generated with the same timestamp? Neither system would have been able to determine which message was, in fact, generated first. However, it is imperative that both systems come to the same conclusion.

Thus, an algorithm for resolving data collisions is necessary. For instance, Lamport suggests that the systems have an ordered hierarchy. The system higher in the hierarchy would win the data collision at both systems.

Summary

The synchronization of time between nodes in a distributed system is a complex process. The Network Transport Protocol (NTP) has been used successfully for years to do this and has thus become a powerful and mature facility.

Lamport's logical clocks are a simpler means to synchronize time throughout the system if it is only necessary to synchronize the nodes and not to tie them precisely to civil time references. However, the accuracy of system clock synchronization is limited to the channel latency time separating the systems.

An extension to Lamport's algorithm can guarantee the total proper ordering of event processing in a distributed system within the clock resolution. This extension depends upon the node clocks being properly synchronized. Though this algorithm is an extension of Lamport's work on logical clocks, it is equally applicable to distributed systems time-synchronized by NTP or by any other means.