CS 5523 Operating Systems: Synchronization in Distributed Systems

Instructor: Dr. Tongping Liu

Thank Dr. Dakai Zhu and Dr. Palden Lama for providing their slides.
Outline

- Physical clock/time in distributed systems
  - No global time is available
  - Network Time Protocol
  - Berkeley Algorithm

- Logical clock/time and ‘Happen Before’ Relation
  - Lamport’s logical clock → total ordering multicast
  - Vector clocks → Causally ordering

- Mutual Exclusion: Distributed synchronizations
  - De/Centralized algorithms
  - Distributed algorithms (Ricart & Agrawala)
  - Logical token ring
Objectives

- To understand synchronization and related issues in DS
- To learn about clocks and how to sync them
Misconceptions about Distributed Systems

- The same globe time
- **Perfect** network/communication
  - Latency is zero
  - Bandwidth is infinite
  - The network is reliable
  - The network is secure
  - The network is homogeneous
- The **topology** does not change
- There is one **administrator**
Time in Physical World

- What is a “second”? 
- Time it takes the cesium 133 atom to make exactly 9,192,631,770 transitions.

- International Atomic Time is based on very accurate physical clocks (drift rate $10^{-13}$)

- It is based on atomic time, but occasionally adjusted to astronomical time
Computer Clocks and Timing Events

- Each computer has its own internal clock: **quartz crystal**
  - Used by local processes to obtain current time value

- **Problems with quartz**
  - *Drift rate*: the difference per unit of time from some **ideal** reference
  - Ordinary **quartz** clocks drift by about 1 sec in 11-12 days (10^{-6} secs/sec).
  - High precision quartz clocks drift rate is about 10^{-7} or 10^{-8} secs/sec

- **Clock Skew**: difference between times on two clocks (at any instant)
Computer Clocks and Timing Events

- Processes on different computers can timestamp their events using their own clocks
  - Clocks on different computers may give different times
  - Computer clocks drift from perfect time and their drift rates differ from one another
How to sync N clocks with a global clock?

Let each computer have a UTC (Universal Coordinated Time) receiver.

- Ordinary quartz: ±10ms might be too much for some applications (e.g., GPS)
- It might be costly (e.g., in case of sensor nodes)
- Indoor equipment may not get the UTC signals

We may have some nodes with a UTC receiver, then can we sync others with those nodes?

What if none have UTC receiver, can we sync them with each other?
Clock Synchronization Algorithms

All algorithms have the same system model:

- Each machine has a timer causing $H$ interrupts/sec.
- The interrupt handler adds 1 to software clock $C$
- $C$ keeps track of the number of ticks since some agreed-upon time in the past

- Let $C_p(t)$ be the clock at $p$ when the UTC time is $t$,
  - In a perfect world, $C_p(t) = t$ (i.e., frequency $C'_p(t) = dC/dt = 1$)
- The **skew** of a clock is $C'_p(t) - 1$
- The **offset** relative to a specific time is $C_p(t) - t$
Clock Synchronization Algorithms

- Real timers do not tick exactly $H$ times per second. For example, $H=60$ should generate 216,000 ticks per hour but it may range 215,998 to 216,002 per hour.

- So if there exists a constant $\rho$:
  
  \[
  1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho
  \]

  then, timer is working within its specifications.

  $\rho$ (maximum drift rate) is given by the manufacturer.

- How often two clocks should be synchronized?
Clock Synchronization Algorithms

If two clocks are drifting from UTC in the opposite directions, they would be apart as much as $2\rho \Delta t$

■ So if want to guarantee that no two clocks ever differ by more than $\delta$ (i.e., $2\rho \Delta t < \delta$)
then we should sync them $\Delta t < \delta/2\rho$ seconds

■ Various algorithms differ in precisely how to do this re-sync!
  - NTP (Network Time Protocol)
  - The Berkeley algorithm
  - Clock sync in wireless networks
NTP (Network Time Protocol)
At least one machine has a UTC receiver

- Suppose we have a server with UTC receiver.
- The server has an accurate clock.
- So clients can simply contact it and get the accurate time (every $\delta/2\rho$ sec).

A gets T1, T2, T3, T4.

How should A adjust its clock?

The problem is the delay which causes inaccuracy.
NTP: basic idea

Suppose propagation delay is the same in both ways?

Assume \( dT_{\text{req}} = dT_{\text{res}} \)

A can estimate offset value to B(\( \theta \))

\[
\theta = T_3 + ((T_2-(T_1+\theta)) + ((T_4+\theta)-T_3))/2 - T_4 \\
= ((T_3-T_4) + (T_2-T_1))/2
\]

Confuse: the object file is earlier than the source

\( \theta > 0 \), A is slower

\( \theta < 0 \), A is faster, but time cannot run backward?

Introduce the difference gradually
NTP

At least one machine has a UTC receiver

- Use this basic idea in a pairwise manner to distribute time information over the Internet.

Objectives

- Enable clients on Internet to synchronize to UCT
- Reliable service through redundant servers/paths
- Provide protection against interference with the time service, whether malicious or accidental

Need: accurate measure of round trip delay, interrupt handling & processing messages
NTP (cont.)

- Provided by a network of **servers** located across the Internet
- Primary servers are connected to UCT sources and time server is passive
- Secondary servers are synchronized to primary servers
- Synchronization subnet - lowest level servers in users’ computers
Outline

- Physical clock/time in distributed systems
  - No global time is available
  - Network Time Protocol
  - Berkeley Algorithm

- Logical clock/time and ‘Happen Before’ Relation
  - Lamport’s logical clock → total ordering multicast
  - Vector clocks → Causally ordering

- Mutual Exclusion: Distributed synchronizations
  - De/Centralized algorithms
  - Distributed algorithms (Ricart & Agrawala)
  - Logical token ring

Election Algorithms
Berkeley Algorithm
No machine has UTC receiver

- Operator manually sets the time at the time server (daemon)

- Time server is active and does the followings:
  - periodically poll all machines
  - compute the average and
  - tell other machines to adjust their times
    ✓ gradually slow down or advance the clock
Berkeley Algorithm
No machine has UTC receiver

- Time does not need to be the actual time...
- As long as all machines agree, then that is OK for many applications
  Gradually advance or slow down the clock...
Outline

- Physical clock/time in distributed systems
  - No global time is available
  - Network Time Protocol
  - Berkeley Algorithm

- Logical clock/time and ‘Happen Before’ Relation
  - Lamport’s logical clock → total ordering multicast
  - Vector clocks → Causally ordering

- Mutual Exclusion: Distributed synchronizations
  - De/Centralized algorithms
  - Distributed algorithms (Ricart & Agrawala)
  - Logical token ring

Election Algorithms
Time in Distributed Systems

- There is **no global clock** in a distributed system

- **Logical time** is an alternative
  - Order of events - also useful for consistency of replicated data

- Algorithms for clock synchronization are useful for
  - concurrency control based on timestamp ordering
  - Consistency in distributed transactions
  - checking the authenticity of requests
Logical Time

The order of two events occurring at two different computers cannot be determined based on their “local” time.

Problem: How do we maintain a global view on the system’s behavior that is consistent with the happened-before relation.

The notion of logical time/clock is fairly general and constitutes the basis of many distributed algorithms.
Why Logic Clock – from Lamport

- If two processes do not interact, it is not necessary that their clocks be synchronized because the lack of synchronization would not be observable and thus could not cause problems.
- What matters is that they agree on the order in which events occur.
“Happened Before” Relation

- Lamport first defined a “happened before” relation (\(\rightarrow\)) to capture the causal dependencies between events.

- **Same process**: A \(\rightarrow\) B, if A and B are events in the **same process** and A occurred before B.

- **Different processes**: A \(\rightarrow\) B, if A is the event of sending a message \(m\) in a process and B is the event of the receipt of the same message \(m\) by another process.

- If A \(\rightarrow\) B, and B \(\rightarrow\) C, then A \(\rightarrow\) C (happened-before relation is transitive).
“Happened Before”: Partial Order

- $a \rightarrow b$ (at $p1$); $c \rightarrow d$ (at $p2$); $b \rightarrow c$; also $d \rightarrow f$
- Not all events are related by the “$\rightarrow$” relation
  - $a$ and $e$ (different processes and no message chain)
  - they are not related by “$\rightarrow$”
  - they are said to be concurrent (written as $a \parallel e$)
Solution: attach a timestamp $C(e)$ to each event $e$, satisfying the following properties

- **P1**: If $a$ and $b$ are two events in the same process, and $a \rightarrow b$, then we demand that $C(a) < C(b)$.

- **P2**: For different processes, if $a$ corresponds to sending a message $m$, and $b$ to the receipt of that message, then also $C(a) < C(b)$.

How to get timestamp $\rightarrow$ consistent logical clocks
Logical Clocks (Lamport, 1978)

- Each process $Pi$ maintains a logical clock, which is a monotonically increasing **software counter** (no relation to physical clock)

- Update the logical clock/counter following
  - For any two successive events that take place within $Pi$, $Ci$ is incremented by 1;
  - Each time a message $m$ is sent by process $Pi$, the message receives a timestamp $ts(m) = Ci$;
  - Whenever a message $m$ is received by a process $Pj$, $Pj$ adjusts its local counter $Cj$ to $\max\{Cj, ts(m)+1\}$
Logical Clock: Example

(a) P_1: 0 6 12 18 24 30 36 42 48 54 60
    P_2: 0 8 16 24 32 40 48 56 64 72 80
    P_3: 0 10 20 30 40 50 60 70 80 90 100

(b) P_1: 0 6 12 18 24 30 36 42 48 54 60
    P_2: 0 8 16 24 32 40 48 56 64 72 80
    P_3: 0 10 20 30 40 50 60 70 80 90 100

P_2 adjusts its clock
P_1 adjusts its clock
The positioning of Lamport’s logical clocks in distributed systems
Logical Clock: Properties

- $e \rightarrow e'$ implies $L(e) < L(e')$
- The converse is not true, that is $L(e) < L(e')$ does not imply $e \rightarrow e'$.
- Lamport’s “happened before” relation defines an irreflexive partial order among the events in the distributed system.
An Example: Logical Clock Application

- Updating a replicated database (Initially $1000)

Add $100
Add 1% interest

Result $1111
Result $1110

Two updates should perform in the same order. But which one is first is not important!
Outline

- Physical clock/time in distributed systems
  - No global time is available
  - Network Time Protocol
  - Berkeley Algorithm

- Logical clock/time and ‘Happen Before’ Relation
  - Lamport’s logical clock → total ordering multicast
  - Vector clocks → Causally ordering

- Mutual Exclusion: Distributed synchronizations
  - De/Centralized algorithms
  - Distributed algorithms (Ricart & Agrawala)
  - Logical token ring
How to sync N clocks with a global clock?

Let each computer have a UTC (Universal Coordinated Time) receiver.

- ±10ms might be too much for some applications (e.g., GPS)
- It might be costly (e.g., in case of sensor nodes)
- Indoor equipment may not get the UTC signals

We may have some nodes with a UTC receiver, then can we sync others with those nodes?

What if none have UTC receiver, can we sync them with each other?
Clock Synchronization Algorithms

If two clocks are drifting from UTC in the opposite directions, they would be apart as much as $2\rho \Delta t$

- So if want to guarantee that no two clocks ever differ by more than $\delta$ (i.e., $2\rho \Delta t < \delta$)
  then we should sync them $\Delta t < \delta/2\rho$ seconds

- Various algorithms differ in precisely how to do this re-sync!
  - NTP (Network Time Protocol)
  - The Berkeley algorithm
  - Clock sync in wireless networks
NTP: basic idea

Suppose propagation delay is the same in both ways?

- Assume $T_2 - T_1 = T_4 - T_3$
- A can estimate its offset value to B as $\theta$

$$\theta = T_3 + \frac{(T_2 - T_1) + (T_4 - T_3)}{2} - T_4$$

$$\theta = \frac{(T_3 - T_4) + (T_2 - T_1)}{2}$$

Confuse: the object file is earlier than the source

$\theta > 0$, A is slower

$\theta < 0$, A is faster, but time cannot run backward?

Introduce the difference gradually (e.g., if time generate 100 interrupts, instead of 10ms, we add 9ms for each interrupt.)
Berkeley Algorithm

No machine has UTC receiver

- Time does not need to be the actual time...
- As long as all machines agree, then that is OK for many applications
- Gradually advance or slow down the clock...
Time in Distributed Systems

- There is **no global clock** in a distributed system

- **Logical time** is an alternative
  - Order of events - also useful for consistency of replicated data

- Algorithms for clock synchronization are useful for
  - concurrency control based on timestamp ordering
  - Consistency in distributed transactions
  - checking the authenticity of requests
Logical Time

- The order of two events occurring at two different computers cannot be determined based on their “local” time.

- **Problem:** How do we maintain a global view on the system’s behavior that is consistent with the *happened-before* relation

- *The notion of logical time/clock is fairly general and constitutes the basis of many distributed algorithms*
“Happened Before” Relation

- Lamport first defined a “happened before” relation (→) to capture the causal dependencies between events.

- **Same process**: A → B, if A and B are events in the **same process** and A occurred before B.

- **Different processes**: A → B, if A is the event of sending a message $m$ in a process and B is the event of the receipt of the same message $m$ by another process.

- If A→B, and B → C, then A → C (happened-before relation is **transitive**).
“Happened Before”: Partial Order

- $a \rightarrow b$ (at $p1$); $c \rightarrow d$ (at $p2$); $b \rightarrow c$; also $d \rightarrow f$
- Not all events are related by the “$\rightarrow$” relation
  - $a$ and $e$ (different processes and no message chain)
  - They are not related by “$\rightarrow$”
  - They are said to be concurrent (written as $a \parallel e$)
Logical Clock: Example

(a) P_1
0 6 12 18 24 30 36 42 48 54 60

P_2
0 8 16 24 32 40 48 56 64 72 80

P_3
0 10 20 30 40 50 60 70 80 90 100

(b) P_1
0 6 12 18 24 30 36 42 48 54 60

P_2
0 8 16 24 32 40 48 56 64 72 80

P_3
0 10 20 30 40 50 60 70 80 90 100

P_2 adjusts its clock
P_1 adjusts its clock
Logical Clock: Properties

- $e \rightarrow e'$ implies $L(e) < L(e')$

- The converse is not true, that is $L(e) < L(e')$ does not imply $e \rightarrow e'$.

- Lamport’s “happened before” relation defines an **irreflexive partial order** among the events in the distributed system.
An Example: Logical Clock Application

- Updating a replicated database (Initially $1000)

  Two updates should perform in the same order. But which one is first is not important!

  Add $100
  Add 1% interest

  Result $1111
  Result $1110
Totally-Ordered Multicast

Consider a group of $n$ distributed processes, $m \leq n$ processes multicasts “update” messages

- How to guarantee that all the updates are performed in the same order by all the processes?

Assumptions

- No messages are lost (Reliable delivery)
- Messages from the same sender are received in the order they were sent (FIFO)
- A copy of each message is also sent to the sender
Totally-Ordered Multicast (cont.)

- Process $P_i$ sends time stamped message $msg_i$ to all others. The message itself is put in a local queue $queue_i$.

- Any incoming message at $P_j$ is queued in $queue_j$, according to its timestamp, and acknowledged to every other process.

- $P_j$ passes a message $msg_i$ to its application if:
  1. $msg_i$ is at the head of $queue_j$
  2. for each process $P_k$, there is an acknowledgement message $msg_k$ in $queue_j$ with a larger timestamp.
Outline

- Physical clock/time in distributed systems
  - No global time is available
  - Network Time Protocol
  - Berkeley Algorithm

- Logical clock/time and ‘Happen Before’ Relation
  - Lamport’s logical clock → total ordering multicast
  - Vector clocks → Causally ordering

- Mutual Exclusion: Distributed synchronizations
  - De/Centralized algorithms
  - Distributed algorithms (Ricart & Agrawala)
  - Logical token ring
Problem with Lamport’s Clocks

**Observation:** Lamport’s clocks do not guarantee that if \( C(a) < C(b) \) that \( a \) **causally preceded** \( b \):

- Event \( a \): \( m_1 \) is received at \( T = 16 \).
- Event \( b \): \( m_2 \) is sent at \( T = 20 \).

**We cannot** conclude that \( a \) **causally precedes** \( b \).
Vector Clocks

Vector clocks are constructed by letting each process $P_i$ maintain a vector $VC_i$ with the following two properties:

- $VC_i[i]$ is the number of events that have occurred so far at $P_i$. In other words, $VC_i[i]$ is the local logical clock at process $P_i$.

- If $VC_i[j] = k$ then $P_i$ knows that $k$ events have occurred at $P_j$. It is $P_i$’s knowledge of the local time at $P_j$.
Vector Clocks: Update

- Rule 1: Before executing an event, \( P_i \) executes \( VC_i[i] \leftarrow VC_i[i] + 1 \).

- Rule 2: When process \( P_i \) sends \( m \) to \( P_j \), it sets \( m \)'s (vector) timestamp \( ts(m) = VC_i \) after Rule 1;

- Rule 3: Upon the receipt of \( m \), process \( P_j \) adjust \( VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\} \) for each \( k \), after which it executes Rule 1 and delivers \( m \) to the application.

It is possible to ensure that a message is delivered only if all messages that causally precede it have been received.
Causally-Ordered Multicasting

For $P_2$, when receive $m^*$ from $P_1$, $\text{ts}(m^*) = (1,1,0)$, but $VC_2 = (0,0,0)$; $m^*$ is delayed as $P_2$ didn’t see message from $P_0$ before;

When $P_2$ receive $m$ from $P_0$, $\text{ts}(m) = (1,0,0)$, with $VC_2 = (0,0,0) \rightarrow$ both $R1$ and $R2$ is ok, and $m$ is delivered $\rightarrow$ $VC_2 = (1,0,0)$, then $m^*$ is delivered
Causally-Ordered Multicasting

- Ensure to deliver a message only if all causally preceding messages have already been delivered

- \( P_j \) postpones delivery of \( m \) from \( P_i \) until:
  
  \[ R1: \] \( ts(m)[i] = VC_j[i] + 1; \)
  
  \( \rightarrow m \) is the next message expected from \( P_i \)

  \[ R2: \] \( ts(m)[k] \leq VC_j[k] \) for \( k \neq i \)
  
  \( \rightarrow P_j \) see all messages that have been seen by \( P_i \) when it sent out message \( m \).
Outline

- Physical clock/time in distributed systems
  - No global time is available
  - Network Time Protocol
  - Berkeley Algorithm

- Logical clock/time and ‘Happen Before’ Relation
  - Lamport’s logical clock → total ordering multicast
  - Vector clocks → Causally ordering

- Mutual Exclusion: Distributed synchronizations
  - De/Centralized algorithms
  - Distributed algorithms (Ricart & Agrawala)
  - Logical token ring
Mutual Exclusion in Distributed Systems

- To ensure exclusive access to some resource for processes in a distributed system
  - **Permission**-based vs. **token**-based approaches

**Solutions**
- **Centralized server**;
- **Decentralized**, using a peer-to-peer system;
- **Distributed**, with no topology imposed;
- Completely distributed along a (logical) ring;
Centralized Server to Grant the Permission

- Good: mutual exclusion, fair, no starvation

What is the problem with this scheme?

Coordinate is the single point of failure.
Other processes can’t distinguish from “permission denied”.
Coordinator can be performance bottleneck.
Decentralized Mutual Exclusion

Assumptions:

- Assume every resource is replicated $n$ times, with each replica having its own coordinator.
- A coordinator always responds immediately to a request.
- Access requires a majority vote from $m > n/2$ coordinators;
- When a coordinator crashes, it will recover quickly, but will have forgotten about permissions it had granted.

Probabilistic correct solution

- With $n=32$, $m=0.75n \rightarrow$ incorrect permission grant: $10^{-40}$
Distributed: Ricart & Agrawala

- Same as Lamport, except that ack aren’t sent. Instead, replies (i.e. grants) are sent only when:
  - Receive process has no interest in the shared resource; or
  - Receive process is waiting for the resource, but has lower priority (known through comparison of timestamps)

Two processes are asking for the same resource
Logical Token Ring

- Processes in a *logical* ring, and let a *token* be passed between them. The one that holds the token is allowed to enter the critical region (if it wants to).

![Logical Token Ring Diagram](image)

What if the token is lost?!
Token-based solutions

- Passing a special message between the processes
  - There is only one token
  - Who has the token is allowed to access the shared resources

- Good:
  - Ensure that every process will get a chance (no starvation)
  - No deadlocks

- Bad:
  - When token is lost, difficult to create a new one that is the only token
## Performance

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Messages per entry/exit</th>
<th>Delay before entry (in message times)</th>
<th>Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>3</td>
<td>2</td>
<td>Coordinator crash</td>
</tr>
<tr>
<td>Decentralized</td>
<td>3mk, k = 1,2,...</td>
<td>2 m</td>
<td>Starvation, low efficiency</td>
</tr>
<tr>
<td>Distributed</td>
<td>2 (n – 1)</td>
<td>2 (n – 1)</td>
<td>Crash of any process</td>
</tr>
<tr>
<td>Token ring</td>
<td>1 to ∞</td>
<td>0 to n – 1</td>
<td>Lost token, process crash</td>
</tr>
</tbody>
</table>
Summary

- Physical clock/time in distributed systems
  - No global time is available
- Logical clock/time and ‘Happen Before’ Relation
  - Lamport, 1978
  - Application: total ordering multicast → consistency
- Vector clocks
  - Causally ordering
- Distributed synchronizations
  - De/Centralized server
  - Distributed algorithms (Ricart & Agrawala)
  - Logical token ring