**CSE 513: Distributed Systems**  
*(Global State and Mutual Exclusion)*

Guohong Cao  
Department of Computer Science & Engineering  
The Pennsylvania State University  
http://www.cse.psu.edu/~gcao

---

**Inherent Limitation**

- No global clock  
  - Due to unpredictable message delay, it is difficult to synchronize physical clocks  
  - Distributed schedule is difficult to implement
- No shared memory  
  - It is difficult to obtain a coherent global view, which includes local state and messages in transmission.

---

**Different Clocks**

- Physical clocks  
  - Make file, input.c has time 2152, input.o has time 2150, then recompile input.c  
  - What happens when input.c was written as 2144 by a computer?
- Logical clocks  
  - Do not care whether the time is 10:00 or 10:02, but the logical relation is important.

---

**Physical Clocks**

- Clock skew  
  - The difference between two computers clocks
- Clock drift  
  - Computers count time at different rates. The underlying oscillators are subject to physical variations such as temperature  
  - For ordinary clocks based on quartz crystal, the drift is $10^{-7}$ seconds/second.
- Coordinated Universal Time (UTC)  
  - Atomic oscillator: drift rate is about one part in $10^{13}$  
  - Broadcasted by radio or satellite and accessed through global positioning systems (GPS)  
  - Accuracy in the order of 0.1 ms
Cristian’s Algorithm

- Also called Network Time Protocol (NTP)
- A time server can access UTC
- Clients check with the server to synchronize

Sending machine

\[
T_0 \quad \text{Request} \quad \text{Time server} \quad T_1 \quad \text{Reply}
\]

I, Interrupt handling time

Major Problem

- Clocks never run back
  - What about \( T_1 < T_0 \)
- Solution
  - Changes should be introduced gradually
  - Suppose the timer is set to generate 100 interrupts per second, then each interrupt adds 10ms to the time
  - To slow down, each interrupt adds only 9ms
  - To be faster, each interrupt adds 11ms.

Minor Problems

- How about the communication delay
  - The delay may be large and vary with network load
- One solution is to use \( (T_1 - T_0)/2 \) as the communication delay
  - \( (T_1 - T_0 - I)/2 \)
  - Take several measurements, and discard those values which exceed some threshold value
- Other problems
  - Single point of failure

The Berkeley Algorithm

- A time server (master) is active, polling every machine (slaves) periodically to ask what time it is there
  - The master may not be able to access UTC
- Based on the answers, it computes an average time and tells slaves to adjust their time
  - Instead of sending the updated current time, the server only sends the amount to be adjusted.
- When the master fails, elect a new one
  - Delay is not bounded.
Clock Synchronization in Wireless Networks

- Reference broadcast synchronization (RBS)
- Use a sender to have the receivers synchronize with each other
- After a node broadcasts reference message $m$, each node $p$ records time $T_{p,m}$ that it received $m$. Two nodes $p$ and $q$ can exchange each other’s delivery times to estimate their mutual.

$$\text{offset}[p,q] = \frac{\sum_{k=1}^{M} (T_{p,k} - T_{q,k})}{M}$$

Where $M$ is the total number of reference messages sent.

The critical path in RBS

- What does “happened before” mean without a global clock?
- Each processor usually has a local physical clock. Denote the value of the local clock at process $P_i$ when event $a$ occurs as $C_i(a)$.
- Each event in a process can be timestamped in that process
- $C_i$ may have no relation to real world time; e.g., just a monotonically increasing counter.
**Happened before**

**Definition**: A happened before relation $\rightarrow$ is defined as follows:

- $a \rightarrow b$, if $a$ and $b$ are events in the same process and $a$ occurred before $b$
- $a \rightarrow b$, if $a$ is the event of sending a message $m$ in a process and $b$ is the event of receipt of the same message by another process
- If $a \rightarrow b$ and $b \rightarrow c$, $a \rightarrow c$, transitive

**Concurrent**

- Note that $a \rightarrow a$ does not hold
- If $a$ happened before $b$, it is possible that $a$ causally affects $b$.
- If neither $a \rightarrow b$ nor $b \rightarrow a$, then we say $a$ and $b$ are concurrent and write as $a \parallel b$
- Happened before is a partial order
- Note that concurrency is not transitive
  
  $a \parallel b$ and $b \parallel c$ does not imply $a \parallel c$

**Space-Time Diagram**

**Lamport’s Logical Clocks**

- Goal: implement the happens before relation in a distributed system without global clock
- Assume each process has a clock (counter)
- A system of clocks is correct if $a \rightarrow b$ implies $C(a) < C(b)$
- Happens before can be realized if the following clock conditions are met:
  
  - **[C1]** For any two events $a$ and $b$ in a process $P_i$, if $a \rightarrow b$ then $C_i(a) < C_i(b)$.
  - **[C2]** If $a$ is the event of sending a message in process $P_i$, and $b$ is the event of receiving the same message at $P_j$, then, $C_i(a) < C_j(b)$. 
Implement the Logical Clock

[IR1] Clock $C_i$ is incremented between any two successive events in process $P_i$.

\[
C_i := C_i(a) + d \ (d > 0)
\]

- It’s easy to see if $a$ and $b$ are two successive events in $P_i$ and $a \rightarrow b$, then $C_i(b) = C_i(a) + d$.

[IR2] If $a$ is the event of sending a message $m$, with timestamp $t_m := C_i(a)$, by process $P_i$. On receiving $m$ by process $P_j$, $C_j$ is set to a value greater than or equal to its present value and greater than $t_m$.

\[
C_j := \max(C_j, t_m + d) \ (d > 0)
\]

- Note that the message receipt event at $P_j$ increments $C_j$ as per rule IR1.

Lamport’s Logical Clock

Total Order

- Order events by their local time, but Lamport’s logical clock only defines a partial order.
- Break ties by a total ordering on processes
- Total ordering of events ($a \Rightarrow b$):
  - If $a$ is any event at process $P_i$ and $b$ is any event at process $P_j$, then $a \Rightarrow b$ if either
    \[
    C_i(a) < C_j(b) \text{ or } C_i(a) = C_j(b) \text{ and } P_i \ll P_j
    \]
  - $\ll$ is any arbitrary relation that totally orders the processes to break ties. A simple way is to implement $\ll$ by using unique identification numbers.
Limitations of Lamport’s Clocks

- If $a \rightarrow b$ then $C(a) < C(b)$. However the reverse is not necessarily true, but we know if $C(a) < C(b)$ then what?
- See figure, $C(e_{11}) < C(e_{22})$ and $C(e_{11}) < C(e_{32})$, however, $e_{11}$ is causally related to event $e_{22}$ but not to event $e_{32}$.

Vector Clock

- Vector clock is proposed to decide whether two events are causally related or not by simply looking at their timestamps.
- Vector clock is an array of integers instead of just one integer. In a vector, each entry is used to represent the knowledge about the clock of other processes.
- $C_i[j]$ denotes $P_i$’s knowledge about the logical time at $P_j$.

Implementation Rules

[IR1] Clock $C_i$ is incremented between any two successive events in process $P_i$:
$$C_i := C_i(a) + d \quad (d>0)$$

[IR2] If $a$ is the event of sending a message $m$, with timestamp $t_m := C_i(a)$, by process $P_i$. On receiving $m$ by process $P_j$, $C_j$ is updated as follows:
$$\forall k, C_j[k] := \max(C_j[k], t_m[k])$$

Assertion: At any time, $\forall i, \forall j: C_i[i] \geq C_j[i]$
Causal Ordering of Messages

- Causal ordering: If send(m1) happens before send(m2), then every recipient of both messages m1 and m2 must receive m1 before m2.
- Applications: replicated database systems, the order may be important; e.g., account adjustment
- The Birman-Schiper-Stephenson protocol was implemented in ISIS
- Basic Idea: deliver a message to a process only if the message immediately preceding it has been delivered to the process. Otherwise, buffer it until the message immediately preceding it is delivered.

Birman-Schiper-Stephenson protocol

- Before broadcasting a msg m, a process P_i increments the vector time C_i[i] and timestamps m.
  - Note that (C_i[i] – 1) indicates how many messages from P_i preceded m.
- A process P_j ≠ P_i, upon receiving message m timestamped with C_m from P_i, delays its delivery until both the following conditions are satisfied.
  - C_j[i] = C_m[i] – 1
  - C_j[k] ≥ C_m[k] ∀k ∈ {1, 2, ..., n} – {i}
  - Where n is the total number of processes. Delayed msgs are queued
- When a message is delivered at a process P_p, C_j is updated according to the vector clock rule IR2

Assertion: a → b iff t^a < t^b
Global State

- The recorded global state may be inconsistent if \( n < n' \) or \( n > n' \),
  - where \( n \) is the number of messages sent by A along the channel before A’s state was recorded.
  - \( n' \) is the number of messages sent by A along the channel before the channel’s state was recorded.

- Examples:
  - Record state of A at state 1, and state of channel and B at state 2, \( n = 0, n' = 1 \)
  - Channel at state 1, state of A and B at state 2, \( n' = 0, n = 1 \).

Local States

- Local states
  - \( \text{send}(m_{ij}) \in LS_i \iff \text{time}(\text{send}(m_{ij})) < \text{time}(LS_i) \)
  - \( \text{rec}(m_{ij}) \in LS_j \iff \text{time}(\text{rec}(m_{ij})) < \text{time}(LS_j) \)
  - Transit: \( \text{transit}(LS_i, LS_j) = \{m_{ij} | \text{send}(m_{ij}) \in LS_i \land \text{rec}(m_{ij}) \notin LS_j\} \)
  - Inconsistent: \( \text{inconsistent}(LS_i, LS_j) = \{m_{ij} | \text{send}(m_{ij}) \notin LS_i \land \text{rec}(m_{ij}) \in LS_j\} \)

Global States

- Consistent Global states:
  - a global state \( GS = \{LS_1, LS_2, ...LS_n\} \) is consistent iff
    \[ \forall i, \forall j: 1 \leq i, j \leq n :: \text{inconsistent}(LS_i, LS_j) = \emptyset \]
- Transitless global states:
  - a global state \( GS = \{LS_1, LS_2, ...LS_n\} \) is transitless iff
    \[ \forall i, \forall j: 1 \leq i, j \leq n :: \text{transit}(LS_i, LS_j) = \emptyset \]
- Strongly consistent global states:
  - It is consistent and transitless
**Global State**

**Chandy-Lamport Algorithm**

- **Assumptions:** FIFO channel
- **Marker sending rule for a process P**
  - P records its state
  - For each outgoing channel C from P on which a marker has not been already sent, P sends a marker along C before P sends further messages along C
- **Marker receiving Rule for a process Q.** On receipt of a marker along a channel C:
  - If Q has not recorded its state
    - Record the state of C as empty,
    - Follow the marker sending rule.
  - Else (record the state of C as the sequence of messages received along C after Q’s state was recorded and before Q received the marker along C)

---

**Local state**

**State 1**

- S1: A
- C1: Empty
- C2: Empty
- S2: B

**State 2**

- $450
- C1: +$50 M
- C2: Empty
- $200

**State 3**

- $450
- C1: +$50 M
- C2: + $40
- $160

**State 4**

- $450
- C1: + $50
- C2: +$40 M
- $160

**State 5**

- $490
- C1: empty
- C2: empty
- $210

**Recorded state: A ($500), B ($160), c1 (empty), c2 (+$40)**
A Note on the Collected Global State

- It’s possible that the collected global state is not identical to the real global state
  - The state may change after the marker is sent out and before it is received
- There exists a sequence of actions which can start from the initial state and reach the collected state by permutation.
- Useful to detect stable properties, deadlock detection, and termination detection.

CSE 513: Distributed Systems
(Mutual Exclusion)

Guohong Cao
Department of Computer Science & Engineering
The Pennsylvania State University
http://www.cse.psu.edu/~gcao

Performance Metrics

- Message complexity
- Synchronization delay
- Response time
- System throughput = \(1/(sd + E)\)
  - Where \(E\) is the average CS execution time, \(sd\) is the synchronization delay

Mutual exclusion in Distributed Systems

- No shared memory, then no semaphores
- No common physical clock
- Unpredictable network delays
- Requirements
  - Free of deadlock
  - Free of starvation
  - Fairness
  - Fault tolerance
A simple solution

- Use a control site
  - Send request to the control site
  - Control site queues requests, and grants permission one by one.

- Drawbacks
  - Single point of failure,
    - election algorithm to select another coordinator
  - Bottleneck and congestion

Election Algorithms

- The Bully Algorithm. The process with the highest process id wins. A process detects the coordinator problem starts this election.
  1. \( P \) sends an \textit{ELECTION} message to all processes with higher numbers.
  2. If no one responds, \( P \) wins the election and becomes coordinator.
  3. If one of the higher-ups answers, it takes over. \( P \)’s job is done.

(a) Process 1 asks the coordinator for permission to access a shared resource. Permission is granted. (b) Process 2 then asks permission to access the same resource. The coordinator does not reply. (c) When process 1 releases the resource, it tells the coordinator, which then replies to 2.
The Bully Algorithm (1)

- The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election.

The Bully Algorithm (2)

- The bully election algorithm. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

A ring algorithm, but no token.
- Each process knows its successor or more.
- When a process detects coordinator failure, it builds an ELECTION message containing its own process id and sends the message to its successor.
- If the successor is down, send to another.
- Finally, the message gets back to the initiator, which can recognize its own id. Change the message to COORDINATOR and circulate once again.

Election in Wireless Environments

- Previous election algorithms assume that the message passing is reliable, and the network topology does not change. Also, process with the highest number wins. Not true in wireless environments. The coordinator should have more resources such as battery power.
- The initiator sends an election to its immediate neighbors.
- When a node receives election for the first time, it designates the sender as its parent, and then sends out an election to all neighbors, except the parent.
- When a node receives an election from a node other than its parent, sends ack. A node sends ack to its parent after it receives all acks from which it sends elections.
Figure 6-22. Election algorithm in a wireless network, with node a as the source. (a) Initial network. (b)–(e) The build-tree phase (last broadcast step by nodes f and I not shown). (f) reporting of best node to source.

Lamport’s M.E

- Non-token based algorithm
- Requirements:
  - A process in the CS must release it before it can be granted to another.
  - Requests to enter CS should be granted in the order in which they were made.
  - If each use of CS is finite, each request will eventually be met.
- Network assumptions
  - FIFO channel and the channel is reliable
  - Complete connection
- N sites $S_1, S_2, ..., S_n$ each keeps a queue ($request_queue_i$) of requests for entering the CS.

- **Requesting the CS**
  - When a site $S_i$ wants to enter the CS, it sends a $REQUEST(ts_i, i)$ message to all sites and places the request on its own $request_queue_i$.
  - When a site $S_j$ receives the $REQUEST(ts_i, i)$ from $S_i$, it returns a timestamped $REPLY$ to $S_i$, and places $S_i$’s request on $request_queue_j$.
- **Executing the CS**: site $S_i$ can enter the CS when the following conditions hold
  - $S_i$ has received a message with timestamp larger than $(ts_i, i)$ from all other sites.
  - $S_i$’s request is on the top of $request_queue_i$.
- **Releasing the CS**
  - Site $S_i$, upon exiting the CS, removes its request from the top of its request queue and sends a timestamped $RELEASE$ to all the sites in its request set.
  - When $S_j$ receives a $RELEASE$ from $S_i$, it removes $S_i$’s request from its request queue.
**Making Requests**

- S1 requests (2, 1)
- S2 requests (1, 2)

**Enters the CS**

- S1 enters the critical section
- S2 enters the critical section

**Exits the CS**

- S2 exits the critical section

**Enters the CS**

- S1 enters the critical section
**Correctness Proof**

- Assume that $S_i$ and $S_j$ are both in the CS, and timestamp $(S_i) >$ timestamp $(S_j)$.
- Condition 1 implies that both $S_i$ and $S_j$ had their own requests at the head of their request queues.
- Condition 2 along with FIFO order, guarantees that $S_i$ has heard about all requests that precedes its current request.
- But if $S_i$ has heard about $S_j$’s requests, $S_j$’s request would be ahead of $S_i$’s request in the queue. A contradiction.

**Performance**

- For each entry to CS
  - One request sent to each of $N-1$ processes
  - One reply from each of the $N-1$ processes
  - One release to each of the $N-1$ processes
- Message complexity: $3(N-1)$
- Synchronization delay: $T$

**Ricart Agrawala Algorithm**

- **Requesting the CS**
  - When a site $S_i$ wants to enter the CS, it sends a REQUEST message to all sites.
  - When a site $S_i$ receives the REQUEST from $S_j$, it returns a REPLY to $S_i$ if it is neither requesting nor executing the CS or if site $S_j$ is requesting and $S_j$’s request timestamp is smaller than $S_i$’s own request. Otherwise, defers the request.
- **Executing the CS**
  - Site $S_i$ enters the CS when it has received REPLY messages from all sites.
- **Releasing the CS**
  - When $S_i$ exits the CS, it sends REPLY messages to all the deferred requests.
Correctness Proof

• Assume that $S_i$ and $S_j$ are both in the CS, and timestamp $(S_i) < timestamp (S_j)$.
• Since $S_i$ is in the CS, $S_i$ has received $S_j$’s reply after it had made its own request.
• $S_j$ can concurrently execute the CS with $S_i$ only if $S_i$ returns a REPLY to $S_j$ before $S_i$ exits the CS. It’s impossible since $S_j$’s request has lower priority.

Performance

• It is an optimization of the Lamport’s algorithm, no FIFO assumption.
• Message complexity: $2(N-1)$
  – One request sent to each of $N-1$ processes
  – One reply from each of the $N-1$ processes
• Synchronization delay: $T$
• Optimization:
  – After $S_i$ has received a REPLY message from $S_j$, $S_i$ can enter the CS any number of times without requesting permission from $S_j$ until $S_i$ sends a REPLY message to $S_j$.
  – Message complexity: between 0 to $2(N-1)$.

Quorum

• Let $U$ denote set of $N$ sites. A coterie $C$ is a set of sets, where each set $g$ in $C$ is called a quorum, which satisfies condition:
  \[ \forall g, h \in C : g \cap h \neq empty \]
• $C=\{\{a, b\}, \{b, c\}\}$ is a coterie under $U=\{a, b, c\}$, and $g=\{a, b\}$ or $\{b, c\}$ is a quorum, but $g=\{a\}$ is not a quorum.
• Other examples: tree quorum
• Guarantee mutual exclusion
**Maekawa (Quorum-Based) Algorithm**

- Construct quorums so that each quorum only has $\sqrt{N}$ sites.
- **Requesting the CS:**
  - Send request to each site in its quorum.
  - When a site receives a request, it sends a reply if it hasn’t been locked; otherwise, queue it. A site is locked if it has sent a reply.

**Performance of the Maekawa Algorithm**

- Messages: $\sqrt{N}$ request, $\sqrt{N}$ reply, $\sqrt{N}$ release messages. Total $3\sqrt{N}$ messages.
- Delay is $2T$, since a site exiting the CS must first send a release to unlock the arbiter site which in turn sends a reply message to the next site to enter the CS.
- Correctness:
  - Suppose $S_i$ and $S_j$ are both in the CS, and $S_i \cap S_j = \{S_k\}$, then site $S_k$ must have sent reply messages to both $S_i$ and $S_j$ concurrently. A contradiction.

**Possible deadlocks**

- $S_i \cap S_j = \{S_{ij}\}$, $S_j \cap S_k = \{S_{jk}\}$, $S_k \cap S_i = \{S_{ki}\}$.
- It is possible that $S_{ij}$ has been locked by $S_i$ (forcing $S_j$ to wait at $S_{ij}$), $S_{jk}$ has been locked by $S_j$ (forcing $S_k$ to wait at $S_{jk}$), and $S_{ki}$ has been locked by $S_k$ (forcing $S_i$ to wait at $S_{ki}$) resulting in a deadlock involving the sites $S_i$, $S_j$, and $S_k$.
- Handling deadlock by new messages: failed, inquire, yield.
Solving the Deadlock Problem

- A failed message from $S_i$ to $S_j$ indicates that $S_i$ cannot grant $S_j$’s request because it has currently granted permission to a site with a higher priority request.
- An inquire from $S_i$ to $S_j$ indicates that $S_i$ would like to find out from $S_j$ if it has succeeded in locking all the sites in its request set.
- A yield message from $S_i$ to $S_j$ indicates that $S_i$ is returning the permission to $S_j$.

Details of handling deadlock

- When a request($t{s},i$) from $S_i$ blocks at $S_j$ because $S_j$ has currently granted permission to $S_k$, then $S_j$ sends a failed($j$) to $S_i$ if $S_i$’s request has lower priority. Otherwise, $S_j$ sends an inquire ($j$) to $S_k$.
- In response to an inquire($j$) from $S_j$, $S_k$ sends a yield ($k$) to $S_j$, provided $S_k$ has received a failed or if it sent a yield.
- In response to a yield($k$) from $S_k$, $S_j$ assumes it has been released by $S_k$, put this request into the request queue, and send reply($j$) to the top request.

Token-based schemes

- Enter the CS when holding the token.
- Issues: how to find and get the token. This distinguishes various algorithms.
- Generally do not assume FIFO message delivery.
- Proof of correctness is trivial.
- Use sequence numbers instead of timestamps
  - Differentiate old and current requests.

A Token Ring Algorithm

- (a) An unordered group of processes on a network.
- (b) A logical ring constructed in software.
Raymond’s Algorithm

- Sites are logically arranged as a directed tree such that the edge of the tree is assigned directions towards the site (root) that has the token.

![Diagram of a directed tree with sites S1 to S7]

Requesting the CS

1. To enter the CS, a site sends a request to the node along the path to the root if it does not hold the token and its request_q is empty. It adds its request to its request_q.
2. When a site on the path receives the message, it places the request in its request_q and sends a request along the path to the root if it has not sent out a request.
3. When the root receives a request, it sends the token to the site from which it received the request and sets its holder to point at that site.
4. When a site receives the token, it deletes the top entry from its request_q, sends the token to the site indicated in this entry, and sets its holder to that site. If request_q is nonempty, it sends a request to the site which is pointed at by the holder.

Executing and Exiting the CS

- Executing the CS
  - A site enters the CS when it receives the token and its own request is at the top of its request_q. In this case, remove the top entry.
- Releasing the CS
  - If its request_q is nonempty, it deletes the top entry from its request_q, sends the token to that site, and sets its holder to that site.
  - If the request_q is nonempty at this point, it sends a request to the site which is pointed at by the holder.

Requesting a Token

![Diagram showing the process of requesting a token and executing the CS]
**Receiving the Token**

- If every process is requesting the resource, the token traverses each edge twice to give every process a turn.
- Each request goes only as far as the neighbor, who has already asked for the token, so each use of the resource requires four messages.
  - One request for the token
  - One delivery of the token
  - One request for return of the token
  - One return of the token

**Performance under Light Load**

- Worst case is: \(2(N-1)\)
- Best case: \(O(\log N)\), synchronization delay: \(T \log N\)

**Heavy Load**

- Worst case is: \(2(N-1)\)
- Best case: \(O(\log N)\), synchronization delay: \(T \log N\)