LECTURE 8: SAFE ACCESS TO DISTRIBUTED SHARED RESOURCES: TIME, SYNCHRONIZATION, REPLICATION & CONSISTENCY

Lecture Contents

• Introduction
• Time in Distributed Systems:
  – Physical Clocks; Logical Clocks: Totally Ordered Multicast
  – Lamport’s Algorithm: Vector Clocks: Causally Ordered Multicast
• Mutual Exclusion in Distributed Systems:
  – Centralized & Decentralized Solutions
  – Election Algorithms
• Consistency Algorithms:
  – Sequential & Continuous Consistency
  – Causal Consistency
  – Client-Centric Consistency
• Replication & Caching
Introduction

- DS essential in everyday life but come with set of unique challenges, e.g. synchronizing data & resolving conflicts.
- Saw above how processes communicate – related to this is how they cooperate & synchronize with each other.
- Here, mainly look at how processes can synchronize

Examples of synchronization:
- Thus important that multiple procs don’t simultaneously access shared resource, but cooperate to grant each other temporary *exclusive* access.
- Multiple processes may also need to agree on *event orderings*, e.g. if message from process $P$ was sent before/after another one from process $Q$.

- Synchronization in DS thus much harder than synchronization in uniprocessor or multiprocessor systems.
- The problems & solutions are, by their nature, rather general, and occur in many different situations in DS.

**SECTION 8.1**: TIME IN DISTRIBUTED SYSTEMS
Time/Clocks

- **Physical clocks**:  
  - **Problem**: Often simply need exact time, not just an ordering.
    - Previously solved by time in terms of *Sun Transits*.
  - **Solution**: Universal Coordinated Time (UTC):  
    - Based on number of transitions per second of caesium 133 atom.
    - At present, real time is taken as average of ~50 caesium-clocks worldwide.
    - Introduces a *leap second* from time to time to account for fact that days are getting longer (e.g. due to tidal drag, orbital wobbles etc).
  - Note: UTC is broadcast through SW radio & satellite. Satellites can give an accuracy of about ±0.5 ms.

---

Time/Clocks (/2)

- **Physical clocks**:  
  - **Problem**: Suppose have distributed system with a UTC-receiver in it ⇒ we still have to distribute its time to each machine.
  - **Basic principle**:  
    - Each machine has a timer generating interrupt \( H \) times per second.
    - There is a clock in machine \( p \) that ticks* on each timer interrupt.
    - Denote the value of that clock by \( C_p(t) \), where \( t \) is UTC time.
    - Ideally, we have that for each machine \( p \), \( C_p(t) = t \), or, \( \frac{dc}{dt} = 1 \)

*Adds one to a s/w clock keeping track of no. of ticks since some (agreed on) time in the past
Time/Clocks (/3)

- **Physical clocks:**

  - In practice: \( 1 - \rho \leq \frac{dc}{dt} \leq 1 + \rho \)

  - From the figure:
    - If 2 clocks drift from UTC in opposite directions in time period \( \Delta t \), may be up to \( 2\rho \Delta t \) apart

  - Goal:
    - Never let 2 clocks differ by more than time units
    - \( \Rightarrow \) synchronise every \( \delta/(2\rho) \) secs
    - \( \delta \) termed the rate of drift

![Relation btw clock time & UTC when clocks tick at different rates](image)

Time/Clocks (/4)

- **Global positioning system**

  - **Basic idea:** Can get accurate account of time as side-effect of GPS
  
  - **Problem:** Assuming satellite clocks are accurate & synchronized:
    - Takes time before a signal reaches receiver
    - Receiver’s clock is definitely out of synch with satellite
Time/Clocks (/5)

• **Clock synchronization principles**
  
  — **Principle I**
  
  • Every machine asks a time server for accurate time min every \( \delta / (2 \rho) \) seconds (Network Time Protocol).
  
  • Ok, but need to measure round trip delay, including interrupts and processing incoming messages.

  ![Getting current time from a time server](image)

  — **Principle II**
  
  • Time server scans all machines periodically, averages, and inform each machine how it should adjust its time wrt. its present time.
  
  • Ok, probably get every machine in sync. Needn’t even propagate UTC time.
  
  — **Fundamental**: Have to take into account that setting time back never allowed ⇒ smooth adjustments.

Time/Clocks (/6)

• **Logical Clocks: The Happened-before** relationship

  — **Problem**: First must introduce notion of ordering before can order anything.

  — The happened-before relation

  • If \( a, b \) are 2 events in same process, \( a \) comes before \( b \), then \( a \rightarrow b \)
  
  • If \( a \) is the sending of a message, and \( b \) is the receipt of that message, then \( a \rightarrow b \)
  
  • If \( a \rightarrow b \) and \( b \rightarrow c \), then \( a \rightarrow c \)

  — **Note**: This introduces a **partial ordering** of events in a system with concurrently operating processes

  • For such a system, \( x \rightarrow y \) is not true but neither is \( y \rightarrow x \)

*Read: “\( a \) happens before \( b \)”*
Time/Clocks (/7)

• **Logical Clocks:**
  
  — **Problem:** How to maintain a global view on system behaviour that is consistent with the *happened-before* relation?
  
  — **Solution:**
  
  — Attach timestamp \( C(e) \) to each event \( e \), with following properties:
    
    • If \( a \) and \( b \) are two events in the same process, and \( a \rightarrow b \), then require \( C(a) < C(b) \).
    
    • If \( a \) corresponds to sending a message \( m \), and \( b \) to the receipt of that message, then also \( C(a) < C(b) \).
    
    • Everybody agrees on the values of \( C(a) \) and \( C(b) \).

Time/Clocks (/8)

• **Logical Clocks:** Lamport’s Algorithm
  
  — **Problem:**
  
  — How to attach a timestamp to an event when there's no global clock?
  
  ⇒ maintain a consistent set of logical clocks, one per process.
  
  — **Solution:**
  
  — Each process \( P_i \) has local counter \( C_i \), adjusts it as per following rules:
    
    1. For any 2 successive events taking place within \( P_i \), \( C_i \) is incremented by 1.
    
    2. Each time a message \( m \) is sent by process \( P_i \), the message receives a timestamp \( ts(m) = C_i \)
    
    3. Whenever a message \( m \) is received by process \( P_j \), \( P_j \) adjusts its local counter \( C_j \) to \( \max(C_j, ts(m)) \) then executes step 1 before passing \( m \) to the application.
  
  — **Notes**
  
  • Property \( P1 \) is satisfied by (1); Property \( P2 \) by (2) and (3).
  
  • Can still occur that 2 events happen simultaneously.
  
  • Avoid this by breaking ties thro process IDs.
**Time/Clocks (/9)**

- **Logical Clocks: Example**

  ![Diagram of logical clocks](image1)

  Three processes, each with its own clock. Lamport’s algorithm corrects the clocks. The clocks run at different rates.

  - **Impossibility:** In (a) \( m_3 \) arrives at \( P_2 \) before it was sent from \( P_3 \)
  - **Lamport’s Algorithm:**
    - \( P_2 \) adjusts its clock to \( 1 + \) sending time (\( =60 \)) on arrival of \( m_3 \) from \( P_3 \)

---

**Time/Clocks (/10)**

- **Logical Clocks:**

  - Adjustments take place in the middleware layer:

![Diagram of logical clocks in distributed systems](image2)

  The positioning of Lamport’s logical clocks in distributed systems
Time/Clocks (/11)

- **Logical Clocks:**
  
  **Example of Totally Ordered Multicast**
  
  - **Problem:**
    - Sometimes must ensure that concurrent updates on a replicated DB are seen in the same order everywhere:
      - P1 adds $100 to an account (initial value: $1000)
      - P2 increments account by 1% interest in New York
    - Two replicas

  ![Diagram of replicated database update](image)

  Updating a replicated database & leaving it in an inconsistent state.

  - **Result:** In absence of proper synchronization:
    - replica #1 ← $1111, while replica #2 ← $1110.

---

Time/Clocks (/12)

- **Logical Clocks:**

  **A Digression on Message Timestamps**

  - If an event $a$ has timestamp $ts(a)$ then $ts(a)[i] - 1$ denotes the number of events processed at $P_i$ that causally precede $a$.

  - Hence, when $P_j$ receives a message from $P_i$ with timestamp $ts(m)$, it knows the number of events that have occurred at $P_i$ that causally preceded the sending of $m$.

  - This way, it knows how many events have occurred at other processes prior to the sending of $m$. 

---

Lecture 8: Safe Access to Dist'd Shared Resources
Time/Clocks (/13)

• Logical Clocks: Example Totally Ordered Multicast
  
  **Solution:**
  
  • Process \( P_i \) sends timestamped 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.

\[ P_j \text{ passes a message } msg_i \text{ to its application if:} \]

1. \( msg_i \) is at the head of queue \( j \)
2. For each process \( P_k \), there is a message \( msg_k \) in queue \( j \) with a larger timestamp. This means that \( msg_i \) is at the head of \( j \)’s queue and has been acknowledged by other processes.

  **Note:** We are assuming that communication is **reliable & FIFO ordered.**

Time/Clocks (/14)

• Logical Clocks: Example

  **Observation:**

  • Lamport's clocks don't guarantee that if \( C(a) < C(b) \) that \( a \) causally preceded \( b \)

  ![Diagram of Logical Clocks Example]

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

  • From diagram, know that for \( P_2 \), \( T_{rcv}(m_1) < T_{snd}(m_3) \) but what can be concluded in general from this statement?

  • Know \( T_{rcv}(m_1), T_{snd}(m_3) \) correspond to events that took place at \( P_2 \) but also know \( T_{rcv}(m_1) < T_{snd}(m_2) \) but no causality there
Time/Clocks (/15)

- **Logical Clocks:**

  - **Problem with Lamport’s Clocks:**
    - No guarantee that if \( C(a) < C(b) \) that \( a \) causally preceded \( b \)
  
  - **Solution: Vector Clocks:**
    - Each process \( P_i \) has an array \( VC_i[1 \ldots n] \), where \( VC_i[j] \) denotes no. of events that process \( P_i \) knows have taken place at process \( P_j \).
    - When \( P_i \) sends message \( m \), it adds 1 to \( VC_i[i] \) & sends \( VC_i \) along with \( m \) as vector timestamp \( ts(m) \).
      - Result: on arrival, recipient knows \( P_i \)'s timestamp (i.e. the number of events at \( P_i \) that causally precede \( P_j \)).
    - When a process \( P_j \) delivers a message \( m \) that it received from \( P_i \) with vector timestamp \( ts(m) \), it (1) updates each \( VC_j[k] \) to \( \max\{VC_j[k], ts(m)[k]\} \)
      (2) increments \( VC_j[k] \) by 1.
    - Put another way, \( ts(m)[k] \) is a tuple consisting of a process’s logical time & its last known time of process \( k \) in terms of no. of events that occurred at \( k \).
    - So with Vector Clocks know that if \( VC(a) < VC(b) \) ie \( a \) causally preceded \( b \)

Time/Clocks (/16)

- **Vector Clocks:** Causally Ordered Multicasting*

  - **Observation:**
    - Can now ensure that a message is delivered only if all causally preceding messages have already been delivered.
    - Note, in terms of messages sent and received \( VC_i[j] = k \) means that \( P_i \) knows that \( k \) events have occurred at \( P_j \)

  - **Adjustment:**
    - \( P_i \) increments \( VC_i[i] \) only on sending a message, & \( P_j \) “adjusts” \( VC_j[k] \) (to \( \max\{VC_j[k], ts(m)[k]\} \)) on receiving a message (i.e., effectively doesn’t change \( VC_j[j] \)).

    \( P_j \) postpones delivery of \( m \) until:
    - \( ts(m)[i] = VC_j[i] + 1 \) i.e. \( m \) is next message \( P_j \) expects from \( P_i \)
    - \( ts(m)[k] \leq VC_j[k] \) for \( k \neq i \). (i.e. \( P_j \) has seen all messages sent by \( P_i \) when \( P_j \) sent \( m \))

* Not as strong as **Totally Ordered Multicasting.**

Lecture 8: Safe Access to Dist’d Shared Resources

CA4006 Lecture Notes (Martin Crane 2013)
Time/Clocks (/17)

- **Vector Clocks**: Example 1
  - Recall each time message \( m \) is sent by process \( P_i \), the message receives a timestamp \( ts(m) = C_i \) (\( C_i \) denotes no. of events at occurred at \( P_i \)).
  - Thus when \( P_j \) receives \( m \) from \( P_i \), it knows about the number of events that have occurred at \( P_i \) before the sending of \( m \).

\[
P_0 \text{ delivers } m^* \text{ cos } ts(m^*) = VC_0[1] + 1
\]

At \((1, 0, 0)\) local time \( P_0 \) sends message \( m \) to \( P_1, P_2 \)

\[
\begin{align*}
P_0 &\quad VC_0 = (1,0,0) \\
P_1 &\quad VC_1 = (1,1,0) \\
P_2 &\quad VC_2 = (0,0,0) \quad VC_2 = (1,0,0)
\end{align*}
\]

\[
\text{After } m \text{ arrives, } P_1 \text{ sends } m^* \text{ to } P_0, P_2
\]

\[
\begin{align*}
&ts(m) = (1,0,0) \Rightarrow VC_1(1,1,0) \\
&ts(m^*) = (1,1,0) \Rightarrow VC_0(1,1,0)
\end{align*}
\]

Lecture 8: Safe Access to Dist'd Shared Resources

Time/Clocks (/18)

- **Vector Clocks**: Example 2 Three processes \( P_0, P_1, P_2 \)
  - Take \( VC_2 = (0,2,2) \) & \( ts(m) = (1,3,0) \) from \( P_0 \)
    1. What information does \( P_2 \) have?
    2. What will it do when receiving \( m \) from \( P_0 \)?
  - 1. aware of 2 events that have taken place at \( P_1 \) & \( P_2 \) & none at \( P_0 \); when sent \( m, P_0 \) not aware of 2 events at \( P_2 \)-- but that doesn’t affect clock at \( P_2 \).
  - 2. To deliver \( m \) to \( P_2 \) recall rule for Causally Ordered Multicasting:
    \( P_j \) postpones delivery of \( m \) until:
    \[a) \quad ts(m)[i] = VC_j[i] + 1 \text{ (i.e. } m \text{ is next message } P_j \text{ expects from } P_i \text{ )}
    \]
    \[b) \quad ts(m)[k] \leq VC_j[k] \text{ for } k \neq i \text{. (i.e. } P_j \text{ has seen all messages sent by } P_i \text{ when } P_i \text{ sent } m \text{)}
    \]
    \[\Rightarrow \text{ For } a) \quad ts(m)[0] = VC_2[0] + 1\sqrt{ }
    \]
    \[\Rightarrow \text{ For } b) ts(m)[1] \leq VC_2[1] \Rightarrow 3 \leq 2 \sqrt{ } \quad ts(m)[2] \leq VC_2[2] \Rightarrow 0 \leq 2 \sqrt{ }
    \]
    \[\Rightarrow P_2 \text{ will adjust } VC_2[0] \text{ to } 1, \text{ } VC_2[1] \text{ to } 3 \text{ deliver } m \text{ & increment } VC_2[2] \text{ to } 2
    \]
    \[\Rightarrow VC_2 = (1,3,3)
    \]
SECTION 8.2: MUTUAL EXCLUSION IN DISTRIBUTED SYSTEMS

Introduction

- Fundamental to distributed systems is the concurrency and collaboration among multiple processes.
- In concurrent/uniprocessor systems, this produces few insurmountable issues.
- Often, similarly, distributed processes need to simultaneously access same resources.
- Have seen that in terms of Totally/Causally Ordered Multicasting above that issues of Time in terms of events must be tackled as well.
- To prevent concurrent accesses corrupting the resource, or make it inconsistent, need solutions to grant ME access by processes.
- Distributed algorithms for ME problem break down into solutions:
  - Via a centralized server.
  - Completely decentralized, using a peer-to-peer system.
  - Completely distributed, with no topology imposed.
  - Completely distributed along a (logical) ring.
Mutual Exclusion

**Approach 1: Centralized Server Solution**

- **Problem:**
  - What happens if the Coordinator crashes?
  - Alternatively, if process blocks waiting to hear back from coordinator on requesting a resource, how to tell the difference between a wait and processor crash?

**Approach 2: Lin’s Decentralized Approach**

- **Principle:**
  - Assume every resource is replicated \( n \) times (i.e. a peer-to-peer approach), with each replica having its own coordinator:
    \[ \Rightarrow \text{access requires a majority vote from } m > \frac{n}{2} \text{ coordinators.} \]
  - A coordinator always responds immediately to a request from a client to access (read/write) a replica.

- **Assumption:**
  - When a coordinator crashes, it will recover quickly, but will have forgotten about permissions it had granted.
Mutual Exclusion (/3)

- **Approach 2: Lin’s Decentralized Approach (cont’d)**

  - **Issue:** How robust is this system?
    - Let $p = \Delta t/T$ denote the probability that a coordinator crashes and recovers in a period $\Delta t$ while having an average lifetime $T$.
    - No memory after crash, so coordinator can be open to new requests.
    - Have DHT system with each node participating for ~3 hours on end.
    - Given that
      - $m$ here is number of replicas voting for a particular ME write
      - $2m - n$ coordinators need to reset in order to violate correctness of vote.
    - $\Rightarrow$ probability that $k$ out $m$ coordinators reset during same $\Delta t/T$:
      $$P[\text{violation}] = p^k = \sum_{k=2m-n}^{m} \binom{m}{k} p^k (1-p)^{m-k}$$

  *Access time of 10s over 3 hours period*

Mutual Exclusion (/4)

- **Approach 3: Ricart & Agrawala’s (Distributed) Algorithm**

  - **Problem:**
    - Often, prob’ly correct algorithm insufficient. Need deterministic dist’d ME.
  - **Principle:**
    - Same as Lamport’s (clock synchronization) except that acks aren’t sent. Instead, replies (i.e. grants) are sent only when:
      - The receiving process has no interest in the shared resource; or
      - The receiving process is waiting for the resource, but has lower priority (known through comparison of timestamps).
    - In all other cases, reply is deferred, implying some more local admin.

- (a) 2 procs want to access shared resource at same time.
- (b) Process 0 has lowest timestamp, so it wins.
- (c) When 0 is done, sends OK also, so 2 can go ahead.
Mutual Exclusion (/5)

- **Approach 4: Token ring algorithm**
  - **Problem**
    - With 3, deadlock is ok; starvation is ok. However 1’s single point of failure now replaced by $n$ points of failure (ie if any process crashes, can’t reply).
  - **Essence:**
    - Organize processes in a logical ring, let token be passed between them.
    - Process holding token is allowed to enter critical region (if it wants to).

  ![Logical ring](image)

  - Ring is initialized, process 0 is given a token. The token circulates.
  - Passes from $k$ to $k+1$ (mod ring size) in point-to-point messages.
  - Process gets token, checks if needs shared resource. If so, process does so & releases the resources. After finishing, passes token along the ring.
  - Cannot immediately enter resource again using the same token.
  - If process gets token neighbour & doesn’t want resource, passes token.

---

Mutual Exclusion (/6)

- **A Comparison of the Four Mutual Exclusion Algorithms**
  - **Centralized algorithm** is simplest and also most efficient.
    - It requires only 3 msgs to enter/leave CS: request, grant to enter, release to exit.
  - **Decentralized case**, messages need to be sent
    - One for each $m$ coordinators, but maybe many attempts needed (hence $k$).
  - **Distributed**
    - $n-1$ requests (one to each other processes, $n-1$ grants, total of $2(n-1)$.
  - For **token ring algorithm**, the number is variable.
    - If every proc constantly wants to enter CS region each token pass will result in one entry and exit, for an average of one message per critical region entered.
    - At other extreme, token sometimes circulate for hours without any interest in it.
    - In this case, the number of messages per entry into a critical region is unbounded.

<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>$3m$, $k = 1, 2, \ldots$</td>
<td>$2m$</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 $\infty$</td>
<td>0 to $n-1$</td>
<td>Lost token, process crash</td>
</tr>
</tbody>
</table>

---

Lecture 8: Safe Access to Dist'd Shared Resources  CA4006 Lecture Notes (Martin Crane 2015)
Mutual Exclusion (/7)

- **Election algorithms**
  - **Principle**
    - Algorithms (as above) require one process act as a coordinator.
    - How to select this special process dynamically?
  - **Note**
    - In many systems coordinator chosen by hand (e.g. file servers).
    - This leads to centralized solutions ⇒ single point of failure.
  - **Question**
    - Coordinator chosen on the fly, to what extent can refer to centralized or distributed solution?
    - Is a fully distributed solution, i.e. one without a coordinator, always more robust than any centralized/coordinated solution?

---

Mutual Exclusion (/8)

- **Election By Bullying**
  - **Principle**
    - Each process has an associated priority (weight).
    - Highest priority process should always be elected as the coordinator.
  - **Issue**: How do we find the heaviest process?
    - Any process can start an election by sending election message to all other processes (assuming don’t know others’ weights).
    - If process $P_{heavy}$ gets election message from lighters $P_{light}$, sends it a take-over message ruling $P_{light}$ out of the race.
    - If a process doesn’t get a take-over message back, it wins, sends victory message to all other processes.
    - Example of this shown overleaf.
Mutual Exclusion (/9)

- Election By Bullying Example

![Election Process Diagram]

Mutual Exclusion (/10)

- Alternative: Ring Algorithm
  - Centralized algorithm is simplest and also most efficient.
  - All processes organized in ring
  - If P notices no coordinator, sends election message to successor with own process number in body of message
    - If successor is down, skip to next process, etc.
  - If Q gets election msg, adds own process number to list in msg body

![Ring Algorithm Diagram]