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
  – Client- & Server-initiated caching
Introduction

• DS essential in everyday life but has unique challenges, e.g. synchronizing data & resolving conflicts.
• Must replicate content but such replicas must be kept consistent with each other.
• Saw above how processes communicate – related to this is how they cooperate & synchronize with each other.
• Here, mainly look at how processes can synchronize:
  – So, vital 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. message from process P sent before/after another 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*$p$ 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, $\frac{dc}{dt} = 1$

*incs s/w clock counting no. of ticks since some (agreed on) time in the past
Time/Clocks (/3)

- **Physical clocks:**
  
  - In practice: \( 1 - \rho \leq \frac{dc}{dt} \leq 1 + \rho \)
  
  \( \rho \) is the clock’s **skew**

- From the figure:
  
  - 2 clocks drifting from UTC in opposite directions in time \( \Delta t \), may be \( \leq 2\rho\Delta t \) apart

- Goal:
  
  - Don’t let 2 clocks differ by more than \( \delta \) time units
  
  => synchronise every \( \delta/(2\rho) \) secs

  - \( \delta \) termed the **rate of drift**

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)

Measured distance, \( d_i = c \text{(time for light to go from satellite to ship)} \)

So \( d_i = c \Delta_i \)

But \( \Delta_i = (T_{\text{now}} - T_i) + \Delta_r \) (\( T_i \) is a satellite’s timestamp)

\[ \Rightarrow d_i = c \Delta_i - c \Delta_r \]

(\( \Delta_i \) is measured time diff, \( \Delta_r \) is correction for clock deviation)

\[ \Rightarrow d_i = c \Delta_i - c \Delta_r = \sqrt{(x_i-x_r)^2 + (y_i-y_r)^2 + (z_i-z_r)^2} \]

i.e. with 4 satellites, now have 4 equations in 4 unknowns

---

Time/Clocks (/6)

**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 must measure round trip delay, incl interrupts & processing incoming messages.

---

**Principle II**

- Time server scans all machines periodically, averages, informs each how to 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 \( \Rightarrow \) smooth adjustments.
Time/Clocks (/7)
• **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 (/8)
• **Logical Clocks:**
  — **Problem:** How to keep 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:
  • \( P1 \) If \( a \) and \( b \) are two events in the same process, and \( a \rightarrow b \), then require \( C(a) < C(b) \).
  • \( P2 \) 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 \( C(a), C(b) \).
Time/Clocks (/9)

• **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. On receipt of message $m$ 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 (/10)

• **Logical Clocks:** Example

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 (/11)

- **Logical Clocks:**
  - Adjustments take place in the middleware layer:

![Diagram of application and middleware layers](image)

The positioning of Lamport’s logical clocks in distributed systems

Time/Clocks (/12)

- **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 and updates](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 (/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 $P_i$ is queued in queue $j$, according to its timestamp, and acknowledged to every other process.

  $P_i$ 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 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.
  - e.g. acknowledgement.

Time/Clocks (/14)

- Logical Clocks: Example
  - Observation:
    - Lamport's clocks don't guarantee that if $C(a) < C(b)$ that $a$ causally preceded $b$

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

  Know $T_{rcv}(m_3), T_{snd}(m_3)$ correspond to events that took place at $P_2$ but also know $T_{rcv}(m_1) < T_{snd}(m_2)$ but no causality there
**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 $i$)
  - 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 $VC_j[i]$ 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 $b$

---

**Vector Clocks:**

**A Digression on Message Timestamps**

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

- Hence, when $P_j$ gets a message from $P_i$ timestamped $ts(m)$, it knows how many events 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 $m$ by $P_i$
Time/Clocks (/17)

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

* Not as strong as Totally Ordered Multicasting.

Lecture 8: Safe Access to Dist’d Shared Resources

---

Time/Clocks (/18)

- **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 $m$.

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

$\text{At } P_0$ delivers $m^* \cos \theta = VC_0[1] + 1$

After $m$ arrives, $P_1$ sends $m^*$ to $P_0, P_2$

Delivery of $m^*$ delayed by $P_2$ until $m$ is received & delivered by $P_2$’s application layer

$ts(m) = (1,0,0) \Rightarrow VC_1(1,1,0)$

$ts(m^*) = (1,1,0) \Rightarrow VC_0(1,1,0)$
Time/Clocks (/19)

- **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:
    - $P_i$ postpones delivery of $m$ until:
      a) $ts(m)[i] = VC_i[i] + 1$ (i.e. $m$ is next message $P_i$ expects from $P_i$)
      b) $ts(m)[k] \leq VC_i[k]$ for $k \neq i$. (i.e. $P_i$ has seen all messages sent by $P_i$ when $P_i$ sent $m$)
  - For a) $ts(m)[0] = VC_2[0] + 1 \checkmark$
  - For b) $ts(m)[1] \leq VC_2[1] \Rightarrow 3 \leq 2 \times ; ts(m)[2] \leq VC_2[2] \Rightarrow 0 \leq 2 \checkmark$
  - $P_2$ will adjust $VC_2[0]$ to 1, inc $VC_2[2]$ to 2 & wait for msg from $P_1$
  - $VC_2 = (1, 3, 2)$ eventually

---

**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**

  (a) Process 1 asks coordinator permission to access shared resource. Granted.

  (b) Process 2 then asks permission to access same resource. Coordinator doesn’t reply.

  (c) When process 1 releases resource, tells coordinator, which then replies to 2.

- **Problem:**
  - What happens if the Coordinator crashes?
  - Or if process blocks waiting to hear back from coordinator on requesting a resource, how to tell a wait from processor crash?
**Mutual Exclusion (/2)**

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

- **Principle:**
  - Assume every resource is replicated $n$ times (i.e. P2P approach), each replica has 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 recovers quickly, but forgets about permissions it had granted.

**Mutual Exclusion (/3)**

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

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

*Access time of 10s over 3 hours period*
Mutual Exclusion (/4)

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

  - **Problem:**
    Often, probably correct algm no good. Need deterministic dist’d ME.

  - **Principle:**
    Same as Lamport’s (clock synchronization) but no acks sent. Instead, replies (i.e. grants) are sent only when:
    - Receiving process has no interest in shared resource; or
    - Interested, but has lower priority (knows by comparing timestamps)
    - In all other cases, reply is deferred, implying some more local admin.

![Diagram](a) 2 procs want to access shared resource at same time.
![Diagram](b) Process 0 has lowest timestamp, so it wins.
![Diagram](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 ✓; starvation ✓. But 1.’s single point of failure now replaced by n points (ie if any process crashes, can’t reply).

  - **Essence:**
    - Organize processes in logical ring, let token be passed btw them.
    - Process holding token allowed to enter CS (if it wants to).

![Diagram](a) An unordered group of processes on a network.
![Diagram](b) Logical ring Constructed in software.

- Ring is initialized. Process 0 given a token. Token circulates.
- Goes from k to k + 1 (mod ring size) in point-to-point messages.
- Process gets token, checks if needs CS. If wants CS does work.
- After done, passes token along ring. Can’t re-access with same token
- If process gets token neighbour & doesn’t want CS, passes token.
Mutual Exclusion (/6)

- **Centralized algorithm** is simplest and also most efficient.
  - Needs only 3 messages to enter/leave CS: request, grant to enter, release to exit.

- **Decentralized case**, messages need to be sent
  - Three for each of m coordinators, but maybe many attempts needed (hence k).

- **Distributed (Ricart & Agrawala’s) Algorithm**
  - 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 & exit=>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>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 n</td>
<td>0 to n - 1</td>
<td>Lost token, process crash</td>
</tr>
</tbody>
</table>

---

Mutual Exclusion (/7)

- **Election algorithms**

  - **Principle**
    - Algorithms above often require one process act as coordinator.
    - How to select this special process dynamically?

  - **Note**
    - In many systems coordinator chosen by hand (e.g. file servers).
    - Leads to centralized solutions ⇒ single point of failure.

  - **Questions**
    - If Coordinator chosen on the fly, to what extent can refer to **centralized** or **distributed** solution?
    - Is a fully distributed solution, i.e. no 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**

![Diagram of Election By Bullying Example](image-url)
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 message body

SECTION 8.3: CONSISTENCY IN DISTRIBUTED SYSTEMS
Introduction to Consistency

- Need replication in DS to enhance reliability/performance.
- In Multicasting Example, major problems to keep replicas consistent.
- Must ensure all copies kept updated else replicas won’t be the same.
- Consistency models assume that multiple procs access shared data.
- Look at consistency as what processes expect when reading/updating shared data, knowing others are accessing it too.
- Also must consider how to implement consistency.
- There are two independent issues we need to consider:
  1. Managing replicas (handle placing replica servers, content distribution)
  2. Ensuring replica consistency (ie update one, must update other copies)
- Hard implementing efficiently on large-scale DS, use simpler models

Performance and scalability Consistency (/2)

- **Main issue:**
  - For replica consistency, must ensure all conflicting operations done in same order everywhere – ‘Tight Consistency’.
- **Conflicting operations:**
  - From the world of Transactions:
    - *Read–write conflict:* read & write operations act concurrently
    - *Write–write conflict:* two concurrent write operations.
  - **Ideal:** Atomic update – but causes b/w problems in large-scale n/ws:
    - $P$ accesses local replica $N$ times/s, but replica updated $M$ times/s.
    - Assume that update totally refreshes previous version of local replica.
    - If $N \ll M$, i.e. very low access-to-update ratio, $P$ won’t access many updated replica versions => useless/waste of bandwidth
    =>better installing local replica close to $P$
- **Issue**
  - Ensuring global ordering on conflicting operations can be costly, downgrading scalability
  - **Solution:** weaken consistency requirements so that hopefully global synchronization can be avoided
Consistency (/3)

- **Data-centric** consistency models:
  
  - **Consistency model:**
    
    - A contract between a (distributed) data store and processes,
    
    - Have a range of consistency models:
      
      - Those with major restrictions on what *read* results of the last *write* operation are easy to use,
      
      - Those with minor restrictions are sometimes difficult.
    
    - Data store specifies precisely what results of R/W operations are in the presence of concurrency

---

Consistency (/4)

- **Data-centric** consistency models: **Continuous Consistency**
  
  - **Observation:** Can actually talk about a *degree* of consistency:
    
    1. Replicas may differ in their *numerical value*
       
       - E.g. replicas of stock data should not deviate by more than $0.02
       
       - i.e. a maximum numerical deviation
    
    2. Replicas may differ in their relative *staleness*
       
       - E.g. normal users only need weather data O(hours) old usually
       
       - But other weather use-cases (e.g. mountaineering) need O(mins)
    
    3. Also possibly differences in number & order of performed update operations
       
       - Applied first to local copy, awaiting global agreement from all replicas.
       
       - A result is some updates may need to be rolled back and applied in a different order before becoming permanent.
       
       - Ordering deviations are harder to grasp than other two metrics.
Consistency (/5)

- **Data-centric** consistency models: **Sequential consistency**
  - **Definition**
    - Result of any execution same as if operations of all processes were executed in some sequential order, and
    - Operations of each individual process appear in this sequence in order specified by its program.
    - i.e. when procs run concurrently on different machines, any valid operation (r/w) is acceptable if all procs see same interleaving.

Consistency (/6)

- **Data-centric** consistency models: **Causal consistency**:
  - **Definition**
    - Writes that are potentially causally related must be seen by all processes in the same order.
    - Concurrent writes may be seen in different order by different procs.
    - Weaker variant to Sequential Consistency as distinguishes between events that are potentially causally related and those that aren’t.
Consistency (/7)

- **Grouping operations:**
  - **Definition**
    - Granularity imposed by consistency models above frequently does not match granularity provided by applications.
    - Normally accesses to shared variables provided by synchronization variables on critical sections `Acq / Rel` are sequentially consistent.
    - No access to a synchronization variable can be performed until all previous writes have completed everywhere.
    - No data access is allowed to be performed until all previous accesses to synchronization variables have been performed.

- **Basic idea**
  - Don’t care that r/w of a series of operations are immediately known to other processes, just that effect of the series itself to be known.

```
P1: Acq(Lx) W(x)a Acq(Ly) W(y)b Rel(Lx) Rel(Ly)
P2: Acq(Lx) R(x)a R(y)b NIL
P3: Acq(Ly) R(y)b
```

A valid event sequence for entry consistency.

Consistency (/8)

- **Client-Centric** Consistency Models:
  - **Definition**
    - Avoid *Global Consistency* models, concentrate on consistency from view of a single (mobile?) client.
  - **Consistency for mobile users**
    - Consider a distributed database with access through your notebook.
    - Assume your notebook acts as a front end to the database.
      - At point A you access DB doing reads and updates.
      - At point B you continue your work, but unless you access same server as that at point A, you may detect inconsistencies:
        » your updates at A may not have yet been propagated to B
        » you may be reading newer entries than the ones available at A
        » your updates at B may eventually conflict with those at A
Consistency (/9)

• **Client-Centric Consistency** Models:
  
  — **Note**
  
  • Must ensure entries updated/read at A, are in B as per last seen at A.
  • Here DB will appear **consistent** to you, (e.g. eventual consistency)
  • Consider a distributed database with access through your notebook.
  • Assume your notebook acts as a front end to the database.

  ![Diagram of distributed database with replication and consistency](image)

  **Problem:** Mobile user accesses different replicas of distributed DB over a short period of time
  **Solution:** Alleviated by CCC. Guarantees consistency for a single client but not concurrent access

Consistency (/10)

• **Client-Centric Consistency** Models:

  — **Monotonic Reads**

  — **Notation**

  • $WS(x_1[t_1])$ is set of writes (at $L_1$) that lead to version $x_1$ of $x$ (at time $t$)
  • $WS(x_1[t_1]; x_j[t_2])$ indicates that it is known that $WS(x_1[t_1])$ is part of $WS(x_j[t_2])$

  — **Example**

  • Automatically reading personal calendar writes from different servers.
  • Monotonic Reads guarantees that user sees all updates, no matter from which server the automatic reading takes place.

  — **Example**

  • Reading (not modifying) incoming mail while you are on the move.
  • Each time you connect to a different e-mail server, that server fetches (at least) all the updates from server you previously visited.
Consistency (/11)

- **Client-Centric Consistency** Models:
  - **Monotonic Reads**
    - If a process reads value of a data item \( x \) any successive read on \( x \) by that process always returns that same value or a more recent value.

(a) A monotonic-read consistent data store.

(b) Data store not providing monotonic reads.

```
L1: \[WS(x_1) \rightarrow R(x_1)\]
L2: \[WS(x_2) \rightarrow R(x_2)\]

Write set \( WS(x_1) \) is part of \( WS(x_2) \).
```

Reads performed by a single process \( P \) at 2 different local copies of the same data store.

```
L1: \[WS(x_1) \rightarrow R(x_1)\]
L2: \[WS(x_2) \rightarrow R(x_2)\]
```

Reads performed by a single process \( P \) at 2 different local copies of the same data store.

Consistency (/12)

- **Client-Centric Consistency Models**:
  - **Monotonic Writes**
    - Often writes must be propagated in *correct order* to all DS copies
    - Write operation by a process on a data item \( x \) is completed before any successive write operation on \( x \) by the same process...

(a) A monotonic-write consistent data store.

(b) Data store not providing monotonic-write consistency.

```
L1: \[W(x_1)\]
L2: \[WS(x_1) \rightarrow W(x_2)\]

Write Set at \( L_1 \) propagated at \( L_2 \) before \( W(x_2) \).
```

```
L1: \[W(x_1)\]
L2: \[W(x_2)\]
```

```
L1: \[W(x_1)\]
L2: \[W(x_2)\]
```

Writes performed by a single process \( P \) at 2 different local copies of same data store.

Write store not providing monotonic-write consistency.
Consistency (/13)

- **Client-Centric Consistency** Models:
  - **Monotonic Writes** (cont’d)
    - **Example**
      - Updating a program at server $S_2$, ensuring that all components on which compilation/linking depends, are also placed at $S_2$.
    - **Example**
      - Maintaining versions of replicated files in correct order everywhere
      - i.e. propagate the previous version to server where newest version is installed.

Consistency (/14)

- **Client-Centric Consistency** Models:
  - **Read Your Writes**
    - Write is always completed before successive read by same process no matter where read happens (closely related to **Monotonic Reads**)
    - Effect of a write operation by a process on data item $x$, will always be seen by a successive read operation on $x$ by the same process...
    - **Example**
      - Updating Webpage ensuring browser shows updates not cached copy.
Consistency (/15)

- **Client-Centric Consistency** Models:
  - **Writes follow Reads**
    - Write on data item \( x \), following previous read on \( x \) by same process, will always occur on same or more recent value of \( x \) that was read

<table>
<thead>
<tr>
<th>L1:</th>
<th>WS((x_1))</th>
<th>R((x_1))</th>
<th>W((x_2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2:</td>
<td>WS((x_1);(x_2))</td>
<td>(a)</td>
<td>W((x_2))</td>
</tr>
</tbody>
</table>

- **Example**
  - See replies to posted articles only if you have original posting (a read “pulls in” the corresponding write operation).

**SECTION 8.4: REPLICATION IN DISTRIBUTED SYSTEMS**
Introduction to Replication & Caching

- Need replication in Dist’d Systems to enhance reliability/performance
- *Distribution Transparency* achieved through either replication of object/state and possible hosting at different locations.
- Also, possible multiplicity of locations that replicas are hosted at used to aide *Scalability*, another goal of Distributed Systems.
- Replication takes a number of forms:
  1. Replicated file servers/databases
  2. Mirrored Websites
  3. Web caches in browsers & proxies
  4. File caching at server and client
- Caching:
  - Special form of replication except it’s a client decision
  - A Cache normally placed on same as its client (or at least same LAN)

Caching & Replication (/2)

- *Placement Protocols*:
  - Says when/where doc copy is placed/removed - placement can be initiated either by servers or clients,
  - Distinguish three different layers of host servers holding copy of a doc:
    - **Core**: host permanent replicas, often, primary server hosts each page
      - Clusters of Web servers/servers mirroring whole sites typify many permanent replicas.
    - **Middle**: host doc-initiated replicas, mostly created by one permanent replica.
      - On internet, doc-initiated replicas appear in CDNs.
      - Here, content transferred to servers near requesting clients.
    - **Outer**: host client-initiated replicas, a.k.a. *cache servers*.
      - Creating a cached version of a doc is entirely a local decision.
      - In principle, taken independently from the replication strategy of the doc.
  - However, *decision to cache* may be subject to many constraints, e.g. client caches only docs expected not to change soon.
  - Also, may have *limited disk space* available for caching.
  - Web proxy caches typify client-initiated replicas in the Internet.
**Placement Protocols (cont’d): Server-Initiated Replicas**

- Done at data store’s initiative (for temporary needs, e.g. enhance performance)
- Example of this is www hosting services providing a collection of servers
  - provide access to www files belonging to third parties
  - replicate files close to “fussy clients”, flash crowds etc
- Issues:
  - improve response time; reduce server load; reduce data comms load
  - ⇒ bring files to servers placed in the proximity of clients - how to decide?

**Placement Protocols: Server-initiated replicas**

- Monitor access counts per file $N_F$ aggregated by considering server closest to requesting clients, $P$
- Decide action
  - $N_F$ drops below threshold $D$ ⇒ drop file from $Q$
  - $N_F$ exceeds threshold $R$ ⇒ replicate file at $P$
  - $N_F$ between $D$ and $R$ ⇒ migrate file to $P$
Caching & Replication (/2)

**Summary**

- There are 2 main reasons for replicating data in DS: improving the reliability improving performance.
- Replication introduces a consistency problem: whenever a replica is updated, that replica becomes different from the others.
- To keep replicas consistent, need to propagate updates in such a way that temporary inconsistencies are not noticed.
- Unfortunately, doing so may severely degrade performance, especially in large-scale distributed systems.
- To solve this problem various consistency models exist:
  - Data-Centric Consistency e.g. sequential (& weaker) causal consistency
  - Client-Centric Consistency v. weak model (e.g. eventual consistency)

**Parameter** | **Values** | **Meaning**
--- | --- | ---
Change distribution | - notification - full state - state differences - operation | Describes how changes between replicas are distributed: is only a notification (or invalidation) sent telling that an update is needed, is the full state sent, or only differences, or is the operation sent that is to be carried out to update the receiver’s state?
Replica responsiveness | - immediate - lazy (e.g., periodic) - passive | Describes how quickly a replica reacts when it notices it is no longer consistent with the other replicas. A passive replica will do nothing.
Replica reaction | - pull - push | Describes what a (non passive) replica does when it notices it is inconsistent with other replicas. It either sends or requests updates.
Write set | - single - multiple | This parameter gives the number of writers that may simultaneously access the document.
Coherence group | - permanent only - permanent and document initiated | Describes who implements the consistency model: permanent and/or document initiated replicas. Caches are generally not part of the coherence group.