Agenda

- Consistency protocols
- Clocks, logical clocks, state vectors
- Optimistic replication
  - CVS, Subversion
  - Duplicated databases
Consistency protocols

• Describe implementation of a specific consistency model

• Primary-based protocols
  ▪ Each data item x has an associated primary responsible for coordinating write operations on x
  ▪ Remote-write protocols
  ▪ Local-write protocols

• Replicated-write protocols
  ▪ Write operations can be carried out at multiple replicas
  ▪ Active replication
  ▪ Quorum-based protocols
Remote-Write Protocols (1)

- **Primary-based remote-write protocol with a fixed server to which all read and write operations are forwarded.**

  - W1. Write request
  - W2. Forward request to server for x
  - W3. Acknowledge write completed
  - W4. Acknowledge write completed
  - R1. Read request
  - R2. Forward request to server for x
  - R3. Return response
  - R4. Return response
Remote-Write Protocols (2)

- The principle of primary-backup protocol.

W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read
Remote-Write Protocols (3)

• Primary-backup protocol implements update as a blocking operation

• Alternative solution: non-blocking protocol
  ▪ As soon as primary updated the local copy of x, it returns an acknowledgement
  ▪ After that ask backup servers to perform the update
  ▪ Fault tolerance concerns

• Implementation of sequential consistency
Local-Write Protocols (1)

- Primary-based local-write protocol in which a single copy is migrated between processes
- Disadvantage: keeping track where each data item currently is

1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server
Local-Write Protocols (2)

- Primary-backup protocol in which the primary migrates to the process wanting to perform an update
- Advantage if nonblocking protocol: write operations carried locally, while reading can access local copies
- Protocol suitable for mobile computers

W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read
Replicated write protocols

Active Replication (1)

• Operations sent to each replica
• Operations have to be carried out in the same order everywhere
  - Need of totally-ordered multicast
    - Using Lamport timestamps
    - Using a central coordinator called sequencer
• Deal with replicated invocations
Active Replication (2)

Client replicates invocation request

All replicas see the same invocation

Replicated object

Object receives the same invocation three times
Active Replication (3)

a) Forwarding an invocation request from a replicated object.

b) Returning a reply to a replicated object.
Quorum-Based Protocols (1)

- Use voting: clients request and acquire permission of multiple servers before reading/writing a replicated object
- Example distributed file system
  - File replicated on N servers
  - For an update a client must contact a majority of servers (half +1)
  - If agreement file changed and version number updated
  - For a read a client must contact at least half of servers+1 and ask them to send version numbers of the file
  - Choose the most recent version
Quorum-Based Protocols (2) - Gifford scheme

- A file with N replicas
- A read quorum ($N_R$ servers) for reading the file
- A write quorum ($N_W$ servers) for modifying the file
- $N_R + N_W > N$
- $N_W > N/2$
Quorum-Based Protocols (3)

- a) A correct choice of read and write set
- b) A choice that may lead to write-write conflicts
- c) A correct choice, known as ROWA (read one, write all)
Pessimistic vs. optimistic replication (1)

- Pessimistic replication
  - Give the illusion of one replica (no divergence)
  - Block access to a replica unless it is up-to-date
  - Example: primary-copy algorithms
    - Elect a primary replica
    - After an update primary writes the change to secondary replicas
    - If primary crashes elect a new replica
  - Bad performance and availability
Pessimistic vs. optimistic replication (2)

- Optimistic replication
  - Allows replicas to diverge
    - Commit modifications immediately and propagate later
    - Observers can see different values on different sites
  - Eventual consistency
  - Mandatory for offline access
  - Better scaling
Eventual Consistency

- **Eventual delivery**: An update executed at some correct replica eventually executes at all correct replicas.

- **Termination**: All update executions terminate.

- **Convergence**: Correct replicas that have executed the same updates eventually reach equivalent state (and stay).

- Consensus moved to the background.
Strong Eventual Consistency

- **Eventual delivery**: An update executed at some correct replica eventually executes at all correct replicas

- **Termination**: All update executions terminate

- **Strong convergence**: Correct replicas that have executed the same updates have equivalent states

- No consensus in background, no need to rollback
**Pessimistic vs. optimistic replication (3)**

- **Basic principles of optimistic replication**
  - N sites replicate an object
  - An object is modified by applying an operation
  - Local operations applied immediately
  - Operations broadcast to the other sites
  - Remote operations integrated and executed
  - System is correct if when it is idle all replicas are identical
Clock Synchronisation

- Time is unambiguous in a centralised system
- There is no global agreement on time in a distributed system
- Example
  - Program consisting of 100 files
  - Use of `make` to recompile only changed source files
  - If `input.c` has time 2151 and `input.o` has time 2150, then recompilation needed
Clock Synchronization

- make does not call the compiler
Logical clock

• Sufficient that all machines agree on the same time (not necessarily real time)

• Lamport 1978 – rather than agreeing on what time it is, sufficient to agree on the order in which events occur

• Previous example: if input.c is older or newer than input.o
Lamport timestamps

• Happens-before relation

• $a \rightarrow b$ ($a$ happens before $b$)

• Two situations:
  - If $a$ and $b$ are events in the same process and $a$ occurs before $b$, then $a \rightarrow b$
  - If $a$ is the event of a message being sent by one process and $b$ is the event of the message being received by another process, then $a \rightarrow b$. A message cannot be received before or at the same time it is sent

• If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$

• If neither $a \rightarrow b$ nor $b \rightarrow a$ then $a$ is concurrent with $b$
Lamport timestamps

• For every event $a$ assign $C(a)$ on which all processes agree
• If $a \rightarrow b$ then $C(a) < C(b)$
• Clock time must always increase
• Lamport solution
  ▪ Each message carries the sending time
  ▪ If receiver clock $< \text{time of the arrived message}$, then receiver forwards its clock to $1 + \text{sending time}$
Lamport timestamps

(a)  

(b)
Lamport timestamps

- If $a$ happens before $b$ in the same process then $C(a)<C(b)$
- If $a$ and $b$ represent the sending and receiving of a message, $C(a)<C(b)$
- For all distinctive events $a$ and $b$, $C(a)\neq C(b)$
  - Attach the number of the process to the lower order of the time
  - If $a$ generated by process 1 at time 40 and $b$ generated by process 2 at time 40, then $C(a)=40.1$ and $C(a)=40.2$
Vector timestamps

- Lamport timestamps limits
  - if \( C(a) < C(b) \) does not imply that \( a \rightarrow b \)
  - \( a \parallel b \) does not imply \( C(a) = C(b) \)
- Example: posting articles and reactions to posted articles
- Lamport timestamps do not capture causality
- Vector timestamps capture causality
  - If \( VT(a) < VT(b) \), then \( a \) causally precedes \( b \)
  - Each process \( P_i \) maintains \( V_i \)
    - \( V_i[i] \) = the no. of events that occurred so far at \( P_i \)
    - If \( V_i[j] = k \) then \( P_i \) knows that \( k \) events occurred at \( P_j \)
Vector timestamps

- Comparison of two vectors
  - $V=W$ iff $\forall i V[i]=W[i]$
  - $V<W$ iff $\forall i V[i] \leq W[i]$ and $\exists i V[i]<W[i]$
  - $[1,2,0] < [3,2,1]$
  - $[0,1,1] \not< [1,0,1]$
Vector timestamps – computation rules

- Process $P_i$
  - Initialisation: $\forall k \ V_i[k]=0$
  - Local event: $V_i[i] = V_i[i]+1$
  - Sending message $m$: $V_i[i] = V_i[i]+1$, then send $(m, V_i)$
  - Receiving message $(m, V_j)$:
    - $\forall k \ V_i[k] = \max(V_i[k], V_j[k])$
    - $V_i[i] = V_i[i]+1$
Vector timestamps – example

\[ \begin{align*}
P_1 &: a \quad [1,0,0] \\
   &: b \quad [2,0,0] \\
   &: c \quad [3,0,0] \\
P_2 &: d \quad [0,1,0] \\
   &: e \quad [2,2,0] \\
   &: f \quad [2,3,0] \\
P_3 &: g \quad [0,0,1] \\
   &: h \quad [0,0,2] \\
   &: i \quad [2,3,3]
\end{align*} \]
State vector

The diagram illustrates the state vector for three sites labeled site 0, site 1, and site 2, with time progression on the vertical axis. The state vector at each site is represented by a vector in a 3-dimensional space, indicating the state changes over time. The lines connecting the sites show the propagation of state changes, with arrows indicating the direction and timing of updates. For example, site 0 transitions from [0,0,0] to [1,0,0] to [1,1,0] to [1,1,1] to [1,2,1] and so on, with similar transitions at site 1 and site 2. The state vector at site 0 is updated before or at the same time as site 1, and site 1 updates before site 2, with a delayed update shown by the dashed line from [1,1,1] to [1,1,1].
State vector based timestamping scheme

- State vector $SV^k$ at site $k$
  - Initially $SV^k[i] = 0$, $\forall i \in \{0, \ldots, N-1\}$
  - Updating rule 1: after executing a local operation, $SV^k[k] = SV^k[k] + 1$
  - After executing a local operation and updating $SV^k$, the local operation is timestamped with $SV^k$ and broadcast to all remote sites
  - Updating rule 2: after executing a remote operation $O$ with $SV_O$, $SV^k[i] = \max(SV^k[i], SV_O[i])$, $\forall i \in \{0, \ldots, N-1\}$
State vector: causality preservation

- $O_i$ generated at site $i$ and timestamped by $SV_{Oi}$
- $O_i$ not allowed to be executed at site $k$ ($k \neq i$) until:
  - $SV_{Oi}[i] = SV^k[i] + 1$
  - $SV_{Oi}[j] \leq SV^k[j]$, $\forall j \in \{0, \ldots, N-1\}$, $j \neq i$
Version control systems: CVS, Subversion

- Two users read the same file:
  - Repository
    - Read
    - A
      - Harry
      - Sally
  - They both begin to edit their copies:
    - Repository
      - A
      - Harry
      - Sally
  - Harry publishes his version first:
    - Repository
      - A'
      - Write
      - A'
        - Harry
      - A''
        - Sally
    - Sally accidentally overwrites Harry's version:
      - Repository
        - A''
        - Write
        - A'
          - Harry
        - A''
          - Sally
Lock-modify-unlock solution
Copy-modify-merge solution
Copy-modify-merge solution
Duplicated databases (Thomas Write Rule 1975) (*)

- **Model**
  - A set of independent DBMPs
  - Each DBMP has its own copy of the database
  - DBMPs communicate via messages
  - Communications are subject to failures
  - Messages between two sites are delivered in the same order they were sent
  - No use of global timestamps

- **The system is correct if it eventually converges**

Duplicated databases (Thomas Write Rule 1975)

DBMP$_1$  DBMP$_2$

Not possible

DBMP$_1$  DBMP$_2$  DBMP$_3$

Possible

op$_1$

op$_2$

op$_1$

op$_2$
Duplicated databases (Thomas Write Rule 1975)

- The database = collection of (selector, value) pairs

- Operations:
  - Selection:
    - `get(selector)` returns the current associated value
  - Assignment:
    - `set(selector, new_value)` replaces associated value with `new_value`
  - Creation:
    - `new(selector, initial_value)` adds (selector, initial_value) entry
  - Deletion:
    - `delete(selector, value)` deletes existing (selector, value) pair
Duplicated databases (Thomas Write Rule 1975)

- How to guarantee that copies are consistent?
Thomas Timestamps

• In the face of concurrent modifications to an entry, how to select the « most recent » change?
• Thomas timestamps before Lamport timestamps!
• A timestamp is a pair \((T,D)\)
  - \(T\) is a network time standard (time-of-day)
  - \(D\) is a DBMP identifier
• Timestamps comparison
  - \((T_1,D_1)>\>(T_2,D_2)\) iff \((T_1>T_2)\) or \((T_1=T_2\) and \(D_1>\>D_2)\)
• If \(D_1=D_2\) and \(T_1=T_2\), then the same operation
**Database entry**

- $E ::= (S, V, T)$
  - $S$ is the selector
  - $V$ is the value
  - $T$ is the timestamp = (Time, DBMP id) of the last change to the entry
Thomas write rule = last writer wins

Database:
- new(x,5)
- set(x,8)
- set(x,9)

DBMP\textsubscript{1}:
- (x,5,(10h,1))
- (x,8,(10h02,1))
- (x,9,(10h03,2))

DBMP\textsubscript{2}:
- (x,5,(10h,1))
- set(x,9)
- (x,9,(10h03,2))

Constraints:
- (10h02,1) < (10h03,2)
- (10h03,2) > (10h02,1)
Creation/update

- Assume the creation will arrive and create the entry right away
- Creation operation ignored at arrival
Creation/update

DBMP$_1$: new(x,2)

DBMP$_2$: set(x,3) → (x,2,…)

DBMP$_3$: (x,3,…)

ignored
Deletion

- Solution: never remove an entry, mark "deleted" flag
Tombstones

- $E ::= (S, V, F, T)$
  - $S$ is the selector
  - $V$ is the value
  - $F$ is the deleted/not-deleted flag
  - $T$ is the timestamp = $(Time, DBMP id)$ of the last change to the entry

- $F = t$ if deleted
- $F = f$ if not-deleted
Tombstones

DBMP₁
(x,3,f,(10h,2))
delete(x)
(x,3,t,(10h01,1))

DBMP₂
(x,3,f,(10h,2))
set(x,5)
(x,5,f,(10h02,2))

DBMP₃
(x,3,f,(10h,2))

DBMP₄
(x,3,f,(10h,2))

(x,3,t,(10h01,1))

(x,5,f,(10h02,2))

(x,3,t,(10h01,1))

(x,3,t,(10h01,1))

Tombstones prevent recreation
Tombstones

- **DBMP1 cannot distinguish in which of the two cases DBMP2 is**

- **Solution:** Associate to an entry the creation timestamp
Tombstones

• E ::= (S, V, F, CT, T)
  ▪ S is the selector
  ▪ V is the value
  ▪ F is the deleted/not-deleted flag
  ▪ CT is the timestamp for creation
  ▪ T is the timestamp = (Time, DBMP id) of the last change to the entry

• If F = f and CT = T, then creation
• If F = f and CT < T, then assignment
• If F = t, then deletion
Tombstones

DBMP₁

(x,3,f,(10h,2),(10h,2))

delete(x)

(x,3,t,(10h,2),(10h₀1,1))

(x,3,t,(10h,2),(10h₀1,1))

Same creation time =>

delete

DBMP₂

(x,3,f,(10h,2),(10h,2))

set(x,5)

(x,5,f,(10h,2),(10h₀2,2))

(x,3,t,(10h,2),(10h₀1,1))

Same creation time =>

delete
Tombstones

DBMP$_1$

\[(x,3,f,(10h,2),(10h,2))\]

\[\text{delete}(x)\]

\[(x,3,t,(10h,2),(10h01,1))\]

\[(x,5,f,(10h03,2),(10h03,2))\]

DBMP$_2$

\[(x,3,f,(10h,2),(10h,2))\]

\[\text{set}(x,5)\]

\[(x,3,t,(10h,2),(10h01,1))\]

\[(x,5,f,(10h03,2),(10h03,2))\]

Different creation time => recreate
Garbage collection

• Make sure of no reception of assignments with same S and the same or older CT

• Remember assumption: Modifications of a DBMP delivered in sequential order

• Each DBMP maintains two « timestamp vectors »
  ▪ Last modifications from all DBMPs
    ○ LM[i] last timestamp from DBMP i
    ○ Modified each time an operation is received
  ▪ Oldest timestamps received by each DBMP
    ○ OT[i] oldest timestamp received by DBMP i
    ○ Sent upon reception of a delete

• Can do garbage collection if timestamp of delete <= timestamp of min(OT)
Garbage collection

DBMP_1
LM=[[]
OT=[]

new(x)
(x,1,f,(1h,1),(1h,1))

LM=[(2h,2)]
OT=[][]

LM=[(2h,2),(3h,3)]
OT=[][]

DBMP_2
LM=[[]
OT=[]

LM=[(1h,1)]
OT=[][]

LM=[(1h,1),(2h,2)]
OT=[][]

DBMP_3
LM=[[]
OT=[]

LM=[(1h,1)]
OT=[][]

LM=[(1h,1),(2h,2)]
OT=[][]

new(y)
(y,2,f,(2h,2),(2h,2))

new(z)
(z,3,f,(3h,3),(3h,3))

LM=[(1h,1),(3h,3)]
OT=[][]
Garbage collection

$\text{DBMP}_1$
$L M=[(2h,2),(3h,3)]$
$O T=[]$

$\text{delete}(z)$

$(z,3,t,(3h,3),(4h,1))$

$L M=[(2h,2),(3h,3)]$
$O T=[(3h,2)]$

$L M=[(2h,2),(3h,3)]$
$O T=[(3h,2),(2h,3)]$

$L M=[(2h,2),(3h,3)]$
$O T=[(3h,2)]$

$L M=[(4h,1),(3h,3)]$
$O T=[(3h,2)]$

$L M=[(4h,1),(3h,3)]$
$O T=[(2h,3)]$

$L M=[(1h,1),(3h,3)]$
$O T=[]$

$L M=[(1h,1),(3h,3)]$
$O T=[]$

$L M=[(1h,1),(2h,2)]$
$O T=[]$

$L M=[(4h,1),(3h,3)]$
$O T=[(3h,2)]$

$L M=[(4h,1),(2h,2)]$
$O T=[(3h,2)]$
Garbage collection

DBMP\(_1\)

\[\text{LM}=[(2h, 2), (3h, 3)] \]
\[\text{OT}=[(3h, 2), (2h, 3)] \]

LM=[(5h, 2), (3h, 3)]
OT=[(3h, 2), (2h, 3)]

DBMP\(_2\)

LM=[(4h, 1), (3h, 3)]
OT=[(2h, 3)]

LM=[(4h, 1), (3h, 3)]
OT=[(2h, 3), (5h, 2)]

(y, 2, t, (2h, 2), (5h, 2))

LM=[(4h, 1), (5h, 2)]
OT=[(3h, 2)]

LM=[(4h, 1), (3h, 3)]
OT=[(2h, 3)]

LM=[(4h, 1), (3h, 3)]
OT=[(3h, 1), (2h, 3)]

LM=[(4h, 1), (5h, 2)]
OT=[(3h, 1), (3h, 2)]

LM=[(5h, 2), (3h, 3)]
OT=[(3h, 2), (4h, 3)]

LM=[(5h, 2), (3h, 3)]
OT=[(3h, 2), (4h, 3)]

DBMP\(_3\)

LM=[(4h, 1), (2h, 2)]
OT=[(3h, 2)]

LM=[(4h, 1), (5h, 2)]
OT=[(3h, 2)]

LM=[(4h, 1), (5h, 2)]
OT=[(3h, 1), (3h, 2)]
Garbage collection

- \( z \) can be garbaged