How Often Do Nodes Fail?

Example: disk drives

*Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You?* Schroeder & Gibson, USENIX FAST 2007

Typical drive replacement rate is 2–4% annually
1 PB = 1,000 × 1 TB drives ~ 20–40 dead drives annually
→ A failure every couple of weeks

Replication Model

Reliable network, fail-stop nodes
Replication Model

Assume single fragment (for now) → Data replication increases availability

Outline

Basic replication algorithms
Improved algorithms
Multiple fragments

Basic Replication

Simple concurrency control solution
Treat each copy as an independent data item

Example
Object X has 3 copies X₁, X₂, and X₃

Basic Replication

read(X)
Get shared X₁ lock
Get shared X₂ lock
Get shared X₃ lock
Read one of X₁, X₂, X₃
At the end of the transaction, release X₁, X₂, X₃ locks
Basic Replication

write(X)
Get exclusive X₁ lock
Get exclusive X₂ lock
Get exclusive X₃ lock
Write new value into X₁, X₂, X₃
At the end of the transaction, release X₁, X₂, X₃ locks

→ Read lock all, write lock all replication: RAWA

Correctness OK
2PL ⇒ serializability
2PC ⇒ atomic transactions

Problem
Low availability

Node is down
X is not accessible

Improved solution
Readers lock and access a single copy
Writers lock and update all copies

→ Read lock one, write lock all replication: ROWA

Reminder
Using standard 2PL
Using standard commit protocols

Good availability for reads
Poor availability for writes
Primary Copy Replication

Select primary node for $X$ (static)
Readers lock and access primary copy
Writers lock primary copy and update all copies

→ Read lock primary, write lock primary: **RPWP**

Local commit (RPWP-LC)
write($X$)
Get exclusive $X_i$ (primary) lock
Write new value into $X_i$
Commit at primary, get sequence number for transaction
Perform $X_2$ and $X_3$ updates in sequence number order

Example

<table>
<thead>
<tr>
<th>$t=0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$: 0</td>
</tr>
<tr>
<td>$Y_1$: 0</td>
</tr>
<tr>
<td>$Z_1$: 0</td>
</tr>
</tbody>
</table>

$T_1$: $X \leftarrow Y \leftarrow 1$
$T_2$: $Y \leftarrow 2$
$T_3$: $Z \leftarrow 3$

<table>
<thead>
<tr>
<th>$t=1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$: 1</td>
</tr>
<tr>
<td>$Y_1$: 0</td>
</tr>
<tr>
<td>$Z_1$: 3</td>
</tr>
</tbody>
</table>

$T_1$: $X \leftarrow Y \leftarrow 1$ (Active at node 1)
$T_2$: $Y \leftarrow 2$ (Waiting for lock at node 1)
$T_3$: $Z \leftarrow 3$ (Active at node 1)
Primary Copy Replication

Example

$t = 2$

\[
\begin{array}{c|c|c}
X & Y & Z \\
1 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
T_1 & X \leftarrow Y \leftarrow 1 & Committed \\
T_2 & Y \leftarrow 2 & Committed \\
T_3 & Z \leftarrow 3 & Committed \\
\end{array}
\]

Primary Copy Replication

What good is RPWP-LC?

Primary Copy Replication

Example

$t = 3$

\[
\begin{array}{c|c|c}
X & Y & Z \\
1 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
T_1 & X \leftarrow Y \leftarrow 1 & Committed \\
T_2 & Y \leftarrow 2 & Committed \\
T_3 & Z \leftarrow 3 & Committed \\
\end{array}
\]

Primary Copy Replication

What good is RPWP-LC?

Can read *out-of-date* backup copy

→ Could be handy, more about it later
Primary Copy Replication

Distributed commit (RPWP-DC)

write(X)
Get exclusive X; (primary) lock
Compute new value for X;
Prepare to write new value at all nodes
When all nodes are prepared, write new value and commit

Basic Replication

Read lock all, write lock all (RAWA)
Read lock one, write lock all (ROWA)
Read and write lock primary (RPWP)
Local commit (LC)
Distributed commit (DC)

Basic Replication

Comparison

<table>
<thead>
<tr>
<th>System</th>
<th>Probability that can read</th>
<th>Probability that can write</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAWA</td>
<td>p^N</td>
<td>p^N</td>
</tr>
<tr>
<td>ROWA</td>
<td>1 - (1 - p)^N</td>
<td>p^N</td>
</tr>
<tr>
<td>RPWP-LC</td>
<td>p</td>
<td>p</td>
</tr>
<tr>
<td>RPWP-DC</td>
<td>p</td>
<td>p^N</td>
</tr>
</tbody>
</table>

N = number of nodes with copies
p = probability that a node is operational
### Probability that can read

<table>
<thead>
<tr>
<th>N = 5</th>
<th>P = 0.99</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAWA</td>
<td>0.9510</td>
</tr>
<tr>
<td>ROWA</td>
<td>-1.0000</td>
</tr>
<tr>
<td>RPWP-LC</td>
<td>0.9900</td>
</tr>
<tr>
<td>RPWP-DC</td>
<td>0.9510</td>
</tr>
</tbody>
</table>

### Probability that can write

<table>
<thead>
<tr>
<th>N = 5</th>
<th>P = 0.99</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAWA</td>
<td>0.9510</td>
</tr>
<tr>
<td>ROWA</td>
<td>-1.0000</td>
</tr>
<tr>
<td>RPWP-LC</td>
<td>0.9900</td>
</tr>
<tr>
<td>RPWP-DC</td>
<td>0.9510</td>
</tr>
</tbody>
</table>

### Probability that can read

<table>
<thead>
<tr>
<th>N = 5</th>
<th>P = 0.90</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAWA</td>
<td>0.5905</td>
</tr>
<tr>
<td>ROWA</td>
<td>-1.0000</td>
</tr>
<tr>
<td>RPWP-LC</td>
<td>0.9000</td>
</tr>
<tr>
<td>RPWP-DC</td>
<td>0.5905</td>
</tr>
</tbody>
</table>

### Probability that can write

<table>
<thead>
<tr>
<th>N = 5</th>
<th>P = 0.90</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAWA</td>
<td>0.5905</td>
</tr>
<tr>
<td>ROWA</td>
<td>-1.0000</td>
</tr>
<tr>
<td>RPWP-LC</td>
<td>0.9000</td>
</tr>
<tr>
<td>RPWP-DC</td>
<td>0.5905</td>
</tr>
</tbody>
</table>

### Probability that can read

<table>
<thead>
<tr>
<th>N = 100</th>
<th>P = 0.99</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAWA</td>
<td>0.3660</td>
</tr>
<tr>
<td>ROWA</td>
<td>-1.0000</td>
</tr>
<tr>
<td>RPWP-LC</td>
<td>0.9900</td>
</tr>
<tr>
<td>RPWP-DC</td>
<td>0.3660</td>
</tr>
</tbody>
</table>

### Probability that can write

<table>
<thead>
<tr>
<th>N = 100</th>
<th>P = 0.99</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAWA</td>
<td>0.3660</td>
</tr>
<tr>
<td>ROWA</td>
<td>-1.0000</td>
</tr>
<tr>
<td>RPWP-LC</td>
<td>0.9900</td>
</tr>
<tr>
<td>RPWP-DC</td>
<td>0.3660</td>
</tr>
</tbody>
</table>
Outline

Basic replication algorithms ✔

Improved algorithms
Mobile primary
Available copies

Multiple fragments

Mobile Primary

Improvement over RPWP

General approach
1. Elect new primary
2. Ensure new primary has all previously committed transactions
3. Resolve pending transactions
4. Resume processing

Mobile Primary

1. Election

Can be tricky

One idea
Nodes have IDs
Largest ID wins

Mobile Primary

1. Election

Algorithm for each node
Broadcast proposal to become primary along with own ID
Wait long enough so anyone with larger ID can stop takeover
If received proposal with smaller ID, kill that takeover
After wait without seeing a larger ID declare self as new primary
Mobile Primary

1. Election

   It is useful to attach an epoch number to messages
   Avoids confusion if stale messages linger
   An epoch starts with the election of a new primary

   Epoch: 1   Primary: N₃
   Epoch: 2   Primary: N₅
   P[E=3, ID=4], P[E=2, ID=5], P[E=3, ID=2] ⇒ D[E=3, ID=4]
   Epoch: 3   Primary: N₄

Mobile Primary

2. Ensure new primary has previously committed transactions

   E.g., assume RPWP-LC

   ![Diagram showing the state transition in RPWP-LC]

   Committed T₁, T₃: May need to get and apply T₁, T₃

Mobile Primary

3. Resolve pending transactions

   E.g., assume RPWP-DC with 3PC

   ![Diagram showing the state transition in RPWP-DC with 3PC]

Mobile Primary

Bad node-failure scenario

   ![Diagram showing the bad node-failure scenario]
Bad node-failure scenario
RPWP-DC with 3PC takes care of the problem

(1) Finish T
(2) Send data
(3) Get acks
(4) Prepare
(5) Get acks
(6) Commit

All transactions have commit sequence number
Active nodes save updates as long as necessary
E.g., since last checkpoint spanning all nodes
Recovering node asks active primary for missed updates and
applies them in order

Majority commit example

X1 fails
X2 new primary
X2 commits T1, T2, T3 and aborts T4
X2 resumes processing
X2 commits T5, T6
X1 recovers and asks X2 for latest state
X2 sends committed and pending transactions
X2 involves X1 in any future transactions
**Mobile Primary**

**RPWP-DC guarantee**
After transaction $T$ commits at current primary, any future primary will see $T$.

---

**Mobile Primary**

**RPWP-DC performance hit**

3PC is very expensive
Many messages
Locks held longer $\Rightarrow$ less concurrency

Could use 2PC instead
May be blocking
2PC is still expensive

---

**Mobile Primary**

**Alternative: RPWP-LC**
Commit transactions unilaterally at primary
Send updates to backups as soon as possible

---

**Mobile Primary**

**Lost transactions**
May happen with RPWP-LC
Mobile Primary

Lost transactions
Claim: the problem is tolerable
- Failures are rare
- Only a few transactions are lost

Outline

Basic replication algorithms ✔

Improved algorithms
- Mobile primary ✔
- Available copies

Multiple fragments

Available Copies

Locks
- Primary
- Available copies
- Transaction write lock at all available copies
- Transactions read lock at any available copy
- Primary site (static) manages set of available copies
Available Copies

Updates
1. Get $U$ from primary
2. Get write locks at $U$ nodes
3. Commit at $U$ nodes

Potential problem

Primary
$U = (X_1, X_2)$

Updates, commit

$T @ U = (X_1, X_3)$

Available Copies

Solution

Initially for transaction $T$ get copy $U_T$ of $U$ from primary
Can use cached value instead

At commit of $T$, compare $U_T$ with current $U$ at primary
If different, abort $T$
**Available Copies**

**Solution**

Let all nodes have a copy of \( U \) (not just primary).

To modify \( U \), run a special atomic transaction at all available sites.

Use commit protocol.

E.g., \( U_1 = \{ X_1, X_2 \} \rightarrow U_2 = \{ X_1, X_2, X_3 \} \)

\( X_3 \) initiates transaction, but only \( X_1, X_2 \) participate.

E.g., \( U_2 = \{ X_1, X_2, X_3 \} \rightarrow U_3 = \{ X_1, X_2 \} \)

Only \( X_1, X_2 \) participate in this transaction.

Who initiates?

---

**Available Copies**

**No primary**

Can get tricky.

What if the \( U \)-update transaction blocks?

How much update information must be remembered by whom?

---

**Outline**

Basic replication algorithms ✔

Improved algorithms ✔

Mobile primary

Available copies

Multiple fragments
Correctness

\[ S_1: r_1[X_1] \rightarrow r_2[X_2] \rightarrow w_1[X_1] \rightarrow w_2[X_2] \]
Is this schedule serializable?

Correctness

\[ S_1: r_1[X_1] \rightarrow r_2[X_2] \rightarrow w_1[X_1] \rightarrow w_2[X_2] \]
Is this schedule serializable?

One idea: require transactions to update all copies

\[ S_1: r_1[X_1] \rightarrow r_2[X_2] \rightarrow w_1[X_1] \rightarrow w_2[X_2] \Rightarrow w_1[X_2] \rightarrow w_2[X_1] \]

Correctness

\[ S_1: r_1[X_1] \rightarrow r_2[X_2] \rightarrow w_1[X_1] \rightarrow w_2[X_2] \]
Is this schedule serializable?

Another idea: build copy-semantics into the notion of serializability

\[ S_1: r_1[X_1] \rightarrow r_2[X_2] \rightarrow w_1[X_1] \rightarrow w_2[X_2] \]
\rightarrow Not a good idea for high-availability replication
Correctness

One-copy serializability (1SR)
A schedule $S$ on replicated data is 1SR if it is equivalent to a serial history of the same transactions on a one-copy database.

Correctness

Checking for 1SR
1. Treat $r_i[X_j]$ as $r_i[X]$ and $w_i[X_j]$ as $w_i[X]$ for all $X_j$ copies of $X$
2. Compute $P(S)$
3. If $P(S)$ acyclic then $S$ is 1SR

Correctness

Example 1
$S_1 \quad r_1[X_1] \rightarrow r_2[X_2] \rightarrow w_1[X_1] \rightarrow w_2[X_2]$
$S_1' \quad r_1[X] \rightarrow r_2[X] \rightarrow w_1[X] \rightarrow w_2[X]$

$S_1$ is not 1SR

Correctness

Example 2
$S_2 \quad r_1[X_1] \rightarrow w_1[X_1] \rightarrow w_2[X_2]$
$r_2[X_1] \rightarrow w_2[X_1] \rightarrow w_2[X_2]$

$S_2' \quad r_1[X] \rightarrow w_1[X] \rightarrow w_1[X]$
$r_2[X] \rightarrow w_2[X] \rightarrow w_2[X]$

$P(S_2) \quad T_1 \rightarrow T_2$
$S_2$ is 1SR
Correctness

Example 2

\[ S_2 \quad r_1 [X_1] \rightarrow w_1[X_1] \rightarrow w_1[X_2] \]

\[ r_2[X_1] \rightarrow w_2[X_1] \rightarrow w_2[X_2] \]

\[ S_2' \quad r_1 [X] \rightarrow w_1[X] \rightarrow w_1[X] \]

\[ r_2[X] \rightarrow w_2[X] \rightarrow w_2[X] \]

Equivalent serial schedule

\[ S_2' \quad r_1 [X] \rightarrow w_1[X] \]

\[ r_2[X] \rightarrow w_2[X] \]

Correctness

Example 3

\[ S_3 \quad r_1 [X_1] \rightarrow w_1[X_1] \rightarrow w_1[X_2] \]

\[ r_2[X_1] \rightarrow w_2[X_1] \]

Is this a good schedule?

Correctness

Example 3

We need to know how \( w_2[X_2] \) is resolved

\[ S_3 \quad r_1 [X_1] \rightarrow w_1[X_1] \rightarrow w_1[X_2] \]

\( r_2[X_1] \rightarrow w_2[X_1] \)

✔

\[ S_3' \quad r_1 [X] \rightarrow w_1[X] \rightarrow w_1[X] \]

\[ r_2[X] \rightarrow w_2[X] \]

✘
### Correctness

**Example 3**

When $w_2[X_2]$ is missing because $X_2$ is down, during recovery $X_2$ will have to perform $w_2[X_2]$ in the correct order.

$S_3$

\[
 r_1[X] \rightarrow w_1[X] \\
 r_2[X] \rightarrow w_2[X]
\]

### Multiple Fragments

A transaction spanning multiple fragments must follow locking rules for each fragment. Commit with *majority in each fragment*.

### Multiple Fragments

Must be careful with update transactions that read but do not modify a fragment.

### Multiple Fragments

Assume $X_1, Y_1$ fail.
Multiple Fragments

Equivalent schedule not serializable
\[ r_1[X] \rightarrow r_2[Y] \rightarrow w_1[Y] \rightarrow w_2[X] \]

Solution
Commit at read fragments/nodes too

E.g., using available copies

\[ F_1 \rightarrow X_1 \rightarrow X_2 \rightarrow Y_1 \rightarrow Y_2 \rightarrow F_2 \]

Cannot commit on \( F_1 \) because
\( T_2 @ U = (F_2, Y_2, F_1, Y_1) \) is out of date for \( F_2 \)

Cannot commit on \( F_2 \) because
\( T_1 @ U = (F_1, X_2, F_2, Y_1) \) is out of date for \( F_1 \)

Write to \( F_1 \) at \( X_2 \)
Commit on \( F_1 \) at \( Y_1 \)

Summary

RAWA, ROWA
Primary copy
  Static primary
  Mobile primary
Available copies
Correctness
Multiple fragments