Further readings:
• *From Total Order to Database Replication* ICDCS 2002
• *Paxos Made Simple*, Leslie Lamport ACM Sigact News 2001
• *Paxos for System Builders: An Overview* LADIS 2008
  www.dsn.jhu.edu/publications web page.
Replication

• Benefits of replication:
  – **High Availability**.
  – **High Performance**.

• Costs of replication:
  – **Synchronization**.

• Requirements from a generic solution:
  – **Strict consistency** – one copy serializability.
  – Sometimes too expensive so requirements are tailored to applications.

Replication Methods

• Two phase commit, three phase commit
• Primary and backups
• Weak consistency (weaker update semantics)
• Primary component.
  • What happens when there is no primary component?
• Congruity: Virtual Synchrony based replication.
• Paxos
**Two Phase Commit**

- Built for updating distributed databases.
- Can be used for the special case of replication.
- Consistent with a generic update model.
- Relatively expensive.
Primary and Backups

Possible options:

- Backups are maintained for availability only.
- Backups can improve performance for reads, updates are sent to the primary by the user.
  - What is the query semantics? How can one copy serializability be achieved?
- The user interacts with one copy, and if it is a backup, the updates are sent to the primary
  - What is the query semantics with regards to our own updates?

Primary and Backups (1)
Primary and Backups (2)

- P → B → B
- B → P

Primary and Backups (3)

- P ↔ B ↔ B
- P → B → P

Weak Consistency
(weaker update semantics)

The Anti-Entropy method: Golding 92

- State kept by the replication servers can be weakly consistent. i.e. copies are allowed to diverge temporarily. They will eventually come to agreement.
- From time to time, a server picks another server and these two servers exchange updates and converge to the same state.
- Total ordering is obtained after getting one message from every server.
- Lamport time stamps are used to order messages.

The Anti-Entropy method

<table>
<thead>
<tr>
<th>Knowledge at Server A</th>
<th>Knowledge at Server B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 1 3 5 12</td>
<td>A 1 3</td>
</tr>
<tr>
<td>B 2</td>
<td>B 2 5 6 9 11</td>
</tr>
<tr>
<td>C 2 3 4</td>
<td>C 2</td>
</tr>
</tbody>
</table>

Summary A

Summary B

Numbers refer to Lamport time stamps.
The Anti-Entropy Method (cont.)

Knowledge at Server A

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Knowledge at Server B

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Summary A

12
2
4

Summary B

3
11
2

Summary After merge

12
11
4

The Anti-Entropy Method (cont.)

Knowledge at Server A

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Knowledge at Server B

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summary A

12
11
4

Summary B

12
11
4
Eventual Path Propagation

Partitioned system

Further partitioning

(cont.)
Eventual Path Propagation (cont.)

Merging

\[ mx \quad my \]

\[ mx \quad my \]

\[ mx \quad my \]

Further merging

\[ ✔️ mx \quad my \]

\[ ✔️ mx \quad my \]

\[ ✔️ mx \quad my \]

\[ ✔️ mx \quad my \]
Going back to 2 Phase Commit

Send decision

- Forced disk write
- Lazy disk write

Primary Component

- A quorum can proceed with updates.
  - Remember that for distributed transactions, every DM had to agree
  - But in the more specific problem of replication, a quorum can continue (not all DM have to agree)
- When the network connectivity changes, if there is a quorum, the members can continue with updates
- Dynamic methods will allow the next quorum to be formed based on the current quorum
  - Dynamic Linear Voting: the next quorum is a majority of the current quorum
  - Useful to put a minimum cap on the size of a viable quorum to avoid relying on too few specific remaining replicas, which can lead to potential vulnerability
Going back to 2 Phase Commit

Send decision

- Forced disk write
- Lazy disk write

What can be improved?

- Reduce number of forced writes to disk
- Reduce number of messages
  - Aggregate acknowledgements
- Avoid end-to-end (application to application) acknowledgements
- Robustness
Group Communication “Tools”

- Efficient message delivery
  - Group multicast
- Message delivery/ordering guarantees
  - Reliable
  - FIFO/Causal
  - Total Order
- Partitionable Group Membership
- Strong semantics (what is actually needed?)

Congruity: Virtual-Synchrony based replication

- Application
  - Request
  - Reply
  - DB
- Replication Server
  - Generate
  - Apply
  - Messages
- Group Communication
  - Send
  - Receive
  - Medium
- Application
  - Actions
  - DB
- Replication Server
  - Messages
  - Deliver
- Group Communication
State Machine Replication

- Servers **start in the same state**.
- Servers change their state only when they execute an update.
- State changes are deterministic. **Two servers in the same state will move to identical states, if they execute the same update.**
- If servers **execute updates in the same order**, they will progress through exactly the same states. **State Machine Replication!**

State Machine Replication Example

- Our State: one variable
- Operations (cause state changes)
  - Op 1) $+n$: Add $n$ to our variable
  - Op 2) $?v:n$: If variable = $v$, then set it to $n$
- Start: All servers have variable = 0
- If we apply the above operations in the same order, then the servers will remain consistent

<table>
<thead>
<tr>
<th>Operation</th>
<th>State Change</th>
<th>State Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v=9$</td>
<td>$v=9$</td>
<td>$v=9$</td>
</tr>
<tr>
<td>$?3:9$</td>
<td>$v=3$</td>
<td>$v=3$</td>
</tr>
<tr>
<td>$v=3$</td>
<td>$+1$</td>
<td>$v=4$</td>
</tr>
<tr>
<td>$v=4$</td>
<td>$?3:9$</td>
<td>$v=3$</td>
</tr>
<tr>
<td>$v=3$</td>
<td>$+1$</td>
<td>$v=4$</td>
</tr>
<tr>
<td>$v=4$</td>
<td>$?3:9$</td>
<td>$v=3$</td>
</tr>
<tr>
<td>$v=3$</td>
<td>$+1$</td>
<td>$v=4$</td>
</tr>
<tr>
<td>$v=4$</td>
<td>$?3:9$</td>
<td>$v=3$</td>
</tr>
<tr>
<td>$v=3$</td>
<td>$+1$</td>
<td>$v=4$</td>
</tr>
<tr>
<td>$v=4$</td>
<td>$?3:9$</td>
<td>$v=3$</td>
</tr>
<tr>
<td>$v=3$</td>
<td>$+1$</td>
<td>$v=4$</td>
</tr>
<tr>
<td>$v=4$</td>
<td>$?3:9$</td>
<td>$v=3$</td>
</tr>
<tr>
<td>$v=3$</td>
<td>$+1$</td>
<td>$v=4$</td>
</tr>
<tr>
<td>$v=4$</td>
<td>$?3:9$</td>
<td>$v=3$</td>
</tr>
</tbody>
</table>
State Machine Replication

Congruity: The Basic Idea

- Reduce database replication to **Global Consistent Persistent Order**
  - Use group communication ordering to establish the **Global Consistent Persistent Order** on the updates.
  - Deterministic + serialized = consistent
- Group Communication membership + quorum = **primary** partition.
  - Only replicas in the **primary** component can commit updates.
  - Updates ordered in a primary component are marked **green** and applied. Updates ordered in a non-primary component are marked **red** and will be delayed.
Action Ordering

- Order is unknown
- Order is known
- Order is known to all

Congruity:
Conceptual State Machine

- Prim
  - Update (Green)
  - Membership Change
  - Last Create Primary Message
- Exchange
  - Membership Change
- Non Prim
  - Update (Red)
  - Membership Change
  - No Primary
- Non Prim
  - Recover
Not so simple…

- **Virtual Synchrony**: If $s_1$ and $s_2$ move directly from membership $M_1$ to $M_2$, then they deliver the same ordered set of messages in $M_1$.
  - What about $s_3$ that was part of $M_1$ but is not part of $M_2$?

\[
\begin{align*}
S_1: & \quad M_1 \xrightarrow{u_1} u_2 \xrightarrow{M_2} \\
S_2: & \quad M_1 \xrightarrow{u_1} u_2 \xrightarrow{M_2} \\
S_3: & \quad M_1 \xrightarrow{u_1} ? \xrightarrow{M_3}
\end{align*}
\]

- **Total (Agreed) Order with no holes** is not guaranteed across partitions or server crashes/recoveries!

Delicate Points

- $s_3$ receives update $u$ in $\text{Prim}$ and commits it right before a partition occurs, but $s_1$ and $s_2$ do not receive $u$. If $s_1$ and $s_2$ will form the next primary component, they will commit new updates, without knowledge of $u$!!

\[
\begin{align*}
S_1: & \quad M_1 \xrightarrow{i} M_2 \xrightarrow{u'} M_3 \xrightarrow{?} M_4 \\
S_2: & \quad M_1 \xrightarrow{i} M_2 \xrightarrow{u'} M_3 \xrightarrow{?} M_4 \\
S_3: & \quad M_1 \xrightarrow{u} M_3 \xrightarrow{?} M_4
\end{align*}
\]

- $s_1$ receives all CPC messages in $\text{Construct}$, and moves to $\text{Prim}$, but one of the servers that were with $s_1$ in $\text{Construct}$ does not receive the last CPC message. A new primary is created possibly without having the desired majority!!

\[
\begin{align*}
S_1: & \quad M_1 \xrightarrow{i} M_2 \xrightarrow{u'} M_3 \xrightarrow{?} M_4 \\
S_2: & \quad M_1 \xrightarrow{i} M_2 \xrightarrow{u'} M_3 \xrightarrow{?} M_4 \\
S_3: & \quad M_1 \xrightarrow{u} M_3 \xrightarrow{?} M_4
\end{align*}
\]
Extended Virtual Synchrony

- **Transitional/Regular membership notification**
- **Safe** message = Agreed plus every server in the current membership will deliver the message unless it crashes.
- **Safe** delivery breaks the two-way uncertainty into 3 possible scenarios, the extremes being mutually exclusive!

\[
\begin{align*}
S_1: & \quad M_1 \rightarrow_u M_2 \quad 1 \\
S_2: & \quad M_1 \rightarrow_u M_3 \quad ? \\
S_3: & \quad M_1 \rightarrow M_4 \quad 0
\end{align*}
\]

Action Ordering

- (Red) Order is unknown
- (Yellow) Transitional membership
- (Green) Order is known
- (White) (I know that) Order is known to all
**Congruity State Diagram**

- **States:** Reg, Trans, Non-Prim, Exchange States, Exchange Messages
- **Messages:** Reg Memb, Trans Memb, Update (Green), Update (Yellow), Update (Red)
- **Transitions:**
  - Reg Prim to Trans Prim
  - Trans Prim to Reg Prim
  - Exchange States to Exchange Messages
  - Non Prim

**Throughput Comparison (LAN)**

- **Graph:**
  - X-axis: number of clients (14 replicas on LAN)
  - Y-axis: update transactions / second
  - Lines:
    - Our Engine
    - CORel
    - Upper bound 2PC

**Comments:**
- CORel – by Dolev and Keidar, using Agreed order + additional round.
The CAIRN Wide-Area Network

- A real experimental network (CAIRN).
- Was also modeled in the Emulab facility.

Throughput Comparison (WAN)

![Throughput Comparison Graph]

- Update transactions / second
- Number of clients (7 replicas on wide area)

- **Our Engine**
- Upper bound 2PC
Congruity Recap

- Knowledge propagation
  - Eventual Path Propagation
- Amortizing end-to-end acknowledgments
  - Low level Ack derived from Safe Delivery of group communication
  - End-to-end Ack upon membership changes
- Primary component selection
  - Dynamic Linear Voting

What about Dynamic Networks?

- Group communication requires stable membership to work well
  - If membership is not stable, group communication based scheme will spend a lot of time synchronizing
- A more robust replication algorithm is needed for such environments – **Paxos**
  - Requires a stable-enough network to elect a leader that will stay stable for a while
  - Requires a (potentially changing) majority of members to support the leader (in order to make progress)
Simple Replication

Can we use a **Leader** to establish an **order**?

- **Server** sends update, \( u \), to **Leader**
- **Leader** assigns a sequence number, \( s \), to \( u \), and sends the update to the non-leader servers.
- Servers order update \( u \) with sequence number \( s \).

Is this resilient?

If leader fails, then the system is not live!

How can we improve resiliency?

Elect another **leader**.
Use more **messages**.
Assign a sequence number to each leader. (**Views**)
Use the fact that two sets, each having at least a majority of servers, must **intersect**!

First... We need to describe our system model and service properties.
Paxos System Model

- **N servers**
  - Uniquely identified in \{1…N\}
- **Asynchronous communication**
  - Message loss, duplication, and delay
  - Network partitions
  - No message corruption
- **Benign faults**
  - Crash/recovery with stable storage
  - No Byzantine behavior - Not yet anyway :)

What is Safety?

- **Safety**: If two servers execute the ith update, then these updates are the same.
- Another way to look at safety:
  - If there exists an ordered update \((u,s)\) at some server, then there cannot exist an ordered update \((u',s)\) at any other server, where \(u \neq u'\).
- We will now focus on achieving safety -- making sure that we don’t execute updates in different orders on different servers.
Achieving Safety

Is this safe?

- A new leader must not violate previously established ordering!
- The new leader must know about all updates that may have been ordered.

A new leader can violate safety!
Can we fix this?

If a new leader gets information from any majority of servers, it can determine what may have been ordered!

- Leader sends \( \text{Proposal}(u,s) \) to all servers
- All servers send \( \text{Accept}(u,s) \) to all servers.
- Servers order \( (u,s) \) when they receive a majority of Proposal/\( \text{Accept}(u,s) \) messages

What does this give us?
Changing Leaders

• Changing Leaders is commonly called a **View Change**.
• Servers use **timeouts to detect failures**.
• If the current leader **fails**, the servers **elect** a new leader.
• The new leader cannot propose updates until it collects information from a **majority** of servers!
  – Each server reports any Proposals that it knows about.
  – If any server ordered a Proposal(u,s), then at least one server in any **majority** will report a Proposal for that sequence number!
  – The new server will **never violate prior ordering**!
  – **Now we have a safe protocol!!**

Changing Leaders Example

• If any server orders (u,s), then **at least majority** of servers must have received **Proposal(u,s)**.
• If a new server is elected leader, it will gather Proposals from a **majority** of servers.
• **The new leader will learn about the ordered update!!**
Is Our Protocol Live?

- **Liveness:** If there is a set, $C$, consisting of majority of connected servers, then if any server in set $C$ has a new update, then this update will eventually be executed.
- Is there a problem with our protocol? It is safe, but is it live?

Liveness Example

- Leader 3 gets conflicting Proposal messages!
- **Which one should it choose?**
- What should we add??
Adding View Numbers

- We add view numbers to the Proposal(v,u,s)!
- Leader 3 gets conflicting Proposal messages!
- Which one should it choose?
- It chooses the one with the greatest view number!!

Normal Case

Assign-Sequence()
A1.  u := NextUpdate()
A2.  next_seq++
A3.  SEND: Proposal(view, u,next_seq)

Upon receiving Proposal(v, u,s):
B1. if not leader and v == my_view
B2. SEND: Accept(v,u,s)

Upon receiving Proposal(v,u,s) and majority - 1 Accept(v,u,s):
C1. ORDER (u,s)

We use view numbers to determine which Proposal may have been ordered previously.

A server sends an Accept(v,u,s) message only for a view that it is currently in, and never for a lower view!
Leader Election

Elect Leader()

Upon Timer T Expire:
A1. my_view++
A2. SEND: New-Leader(my_view)

Upon receiving New-Leader(v):
B1. if Timer T expired
B2. if v > my_view, then my_view = v
B3. SEND: New-Leader(my_view)

Upon receiving majority New-Leader(v)
where v == my_view:
C1. timeout *= 2; Timer T = timeout
C2. Start Timer T

Let V_max be the highest view that any server has. Then, at least a majority of servers are in view V_max or V_max - 1.

Servers will stay in the maximum view for at least one full timeout period.

A server that becomes disconnected/connected repeatedly cannot disrupt the other servers.

We Have: Paxos

- The Part-Time Parliament [Lamport, 98]
- A very resilient protocol. Only a majority of participants are required to make progress.
- Works well on unstable networks.
- Only handles benign failures (not Byzantine).
Performance Results (Paxos-SB)

Update Throughput vs. Clients
Synchronous Disk Writes, Aggregation for Paxos

Local area network cluster.
Congruity: group communication-based replication.

Performance Results (Paxos-SB)

Update Latency vs. Clients
Synchronous Disk Writes, Aggregation for Paxos
Performance Results (Paxos-SB)

Update Throughput vs. Clients
No Disk Writes, Aggregation/Packing

Update Latency vs. Clients
No Disk Writes, Aggregation/Packing