Chapter 15: Replication

Introduction
System model and group communication
Fault-tolerant services
Highly available services
Transactions with replicated data
Summary

A basic architectural model for the management of replicated data

Replication for distributed service

- A logical object is implemented by a collection of physical copies called replicas
- Replication is a key technology to enhance service from three aspects:
  - Performance enhancement
    - Example
      - caches in DNS servers
      - replicated web servers
    - Load-balance
    - Proximity-based response

- Increase availability
  - Factors that affect availability
    - Server failures
    - Network partitions
  - $1 - p^n$
    - The availability of the service that have $n$ replicated servers each of which would crash in a probability of $p$

- Fault tolerance
  - Guarantee strictly correct behavior despite a certain number and type of faults
  - Strict data consistency between all replicated servers
  - replication of read-only data is simple, but replication of mutable data incurs overheads in form of protocols

- Replication transparency
  - clients see a single logical object
  - they are not aware that it is composed of several physical copies
  - they access one logical item and receive a single result

- Consistency
  - The consistency model specifies a contract between the replicated system and its clients
  - The contract describes how operation performed upon a collection of replicated objects produce “correct” results
  - The concept of correctness depends on the application

- Replication for distributed service continued

- A basic architectural model for the management of replicated data
System model

- **Replica manager**
  - One replica manager per replica
  - Receive FE’s request, apply operations to its replicas atomically

- **Front end**
  - One front end per client
  - Receive client’s request, communicate with RM by message passing

An operation executed on a replicated object

- **Request**
  - The front end issues the request to one or more replica managers

- **Coordination**
  - The replica managers coordinate in preparation for executing the request consistently
  - Different ordering

- **Execution**
  - The replica managers execute the request (perhaps tentatively)

An operation executed on the replicated object (2)

- **Agreement**
  - The replica managers reach consensus on the effect of the request

- **Response**
  - One or more replica managers responds to the front end

Group communication

- **Basic services seen so far (reliable broadcast, consensus):**
  - are potentially sufficient to implement replicated services...
  - ... but are not flexible enough

- **What they lack?**
  - Management of dynamic groups
  - Voluntarily join, leave operation
  - Monitoring of the status of the system
  - Detection, Reaction
  - Integration of
    - reliable communication primitives
    - agreement protocols

- **Multicast in a dynamic group**
  - Processes may join and leave the group as the system executes
  - A group membership service
  - Manage the dynamic membership of groups
  - Multicast communication
**Services provided for process groups**

- Group address expansion
- Multicast communication
- Group send
- Fail
- Leave
- Join

**Role of the group membership service**

- **Provide an interface for group membership changes**
  - Create and destroy process groups
  - Add or withdraw a process to or from a group
- **Implement a failure detector**
  - Mark processes as suspected or unsuspected
  - No messages will be delivered to the suspected process
  - It excludes a process from a membership if it is suspected to have failed or to have become unreachable

**Role of the group membership service (2)**

- **Notify members of group membership changes**
  - *Group view*: a list of identifiers of all active processes in the order of *join*
- **Perform group address expansion**
  - A process multicasts a message addressed by a group identifier rather than a list of processes

**View delivery**

- **Group view**
  - The lists of the current group members
- **Deliver a view**
  - When a membership change occurs, the application is notified of the new membership
  - Group management service delivers to any member process \( p \in g \) a series of views \( v_0(g), v_1(g), v_2(g), \ldots \)
- **View delivery is distinct from view receiving**

**View-synchronous group communication**

- **Integrity**
  - If a process delivers message \( m \) in view \( v(g) \) and subsequently delivers the next view \( v'(g) \), then all processes that survive to deliver the next view \( v'(g) \), that is the members of \( v(g) \cap v'(g) \), also deliver \( m \) in the view \( v(g) \)

**Basic requirements for view delivery**

- **Order (totally order)**
  - If a process \( p \) delivers view \( v(g) \) and then view \( v'(g) \), then no other process \( q \neq p \) delivers \( v'(g) \) before \( v(g) \)
- **Integrity**
  - If process \( p \) delivers view \( v(g) \) then \( p \in v(g) \)
- **Non-triviality**
  - If process \( q \) joins a group and is or becomes indefinitely reachable from process \( q \neq p \), then eventually \( q \) is always in the views that \( p \) delivers
  - If the group partitions and remains partitioned, then eventually the views delivered in any one partition will exclude any processes in another partition
View-synchronous group communication (2)

- **Validity**
  - Correct processes always deliver the messages that they send
  - If the system fails to deliver a message to any process $q$, then it notifies the surviving processes by delivering a new view with $q$ excluded, immediately after the view in which any of them delivered the message (like Non-triviality)

- Let $p$ be any correct process that delivers message $m$ in view $v(g)$, if some process $q \in v(g)$ does not deliver $m$ in view $v(g)$, then the next view $v'(g)$ that $p$ delivers has $q \notin v'(g)$

Discussion of view-synchronous group communication

- **The basic idea**
  - Extend the reliable multicast semantics to take account of changing group views
- **Significance**
  - A process knows the set of messages that other correct processes have delivered when it delivers a new view
- **Implementation**
  - ISIS [Birman 1993] originally developed it

Chapter 15: Replication

- **Introduction**
- **System model and group communication**
- **Fault-tolerant services**
- **Highly available services**
- **Transactions with replicated data**
- **Summary**

Replication for fault-tolerance

- **Service replication is an effective measure for fault-tolerance**
  - Provide a single image for users
  - Strict consistency among all replicas
- **Inconsistency between replicas make the property of fault-tolerance fail**

An example of inconsistency between two replications

- Each of computer A and B maintains replicas of two bank accounts $x$ and $y$
- Client accesses any one of the two computers, updates synchronized between the two computers
Inconsistency between two replications

<table>
<thead>
<tr>
<th>Client1:</th>
<th>Client2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>setBalanceB(x,1)</td>
<td></td>
</tr>
<tr>
<td>Server B failed…</td>
<td></td>
</tr>
<tr>
<td>setBalanceA(y,2)</td>
<td></td>
</tr>
<tr>
<td>getBalanceA(y)=2</td>
<td></td>
</tr>
<tr>
<td>getBalanceA(x)=0</td>
<td></td>
</tr>
</tbody>
</table>

- Inconsistency happens since computer B fails before propagating new value to computer A

Linearizability

- The interleaved sequence of operations
  - Assume client i performs operations: 
    $o_{i1}, o_{i2}, o_{i3}, …$
  - Then a sequence of operations executed on one replica that issued by two clients may be:
    $o_{i1}, o_{i2}, o_{i3}, o_{j1}, o_{j2}, o_{j3}, …$

- Linearizability criteria
  - The interleaved sequence of operations meets the specification of a (single) correct copy of the objects
  - The order of operations in the interleaving is consistent with the real times at which the operations occurred in the actual execution
  - The most strict consistency between replicas, hard to achieve

Linearizability … continued

<table>
<thead>
<tr>
<th>Client1:</th>
<th>Client2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>setBalanceB(x,1)</td>
<td></td>
</tr>
<tr>
<td>Server B failed…</td>
<td></td>
</tr>
<tr>
<td>setBalanceA(y,2)</td>
<td></td>
</tr>
<tr>
<td>getBalanceA(y)=2</td>
<td></td>
</tr>
<tr>
<td>getBalanceA(x)=0</td>
<td></td>
</tr>
</tbody>
</table>

Second update has been observed, but first one not

Sequential consistency

- Weaker consistency than linearizability
- Sequential consistency criteria
  - The interleaved sequence of operations meets the specification of a (single) correct copy of the objects
  - The order of operations in the interleaving is consistent with the program order in which each individual client executed them

An example of sequential consistency

<table>
<thead>
<tr>
<th>Client1:</th>
<th>Client2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>setBalanceB(x,1)</td>
<td></td>
</tr>
<tr>
<td>setBalanceA(y,2)</td>
<td></td>
</tr>
</tbody>
</table>

An interleaving operations at server A: getBalanceA(y)=0, getBalanceA(x)=0, setBalanceB(x,1), setBalanceA(y,2)

Not satisfy linearizability
Satisfy sequential consistency
Passive (primary-backup) replication

- One primary replica manager, one or more secondary replica manager
  - When the primary replica manager fail, one of the backups is prompted to act as the primary

- The architecture

The sequence of events when a client issue a request

- Request
  - The front end issues the request, containing a unique identifier, to the primary replica manager

- Coordination
  - The primary takes each request atomically, in the order in which it receives it

- Execution
  - The primary execute the request and stores the response

The sequence of events when a client issue a request

- Agreement
  - If the request is an update then the primary sends the updated state, the response and the unique identifier to all the backups
  - The backups send an acknowledgement

- Response
  - The primary responds to the front end, which hands the response back to the client

Linearizability of passive replication

- If the primary is correct
  - The system implements linearizability obviously

- If the primary fail, linearizability retains
  - Requirements
    - The primary is replaced by an unique backup
    - The replica managers that survive agree on which operations had been performed at the point when the replacement primary takes over
  - Approach
    - The primary uses view-synchronization group communication to send the updates to the backups

Active replication

- Front end multicast request to replication managers

- The architecture

Active replication scheme

- Request
  - The front end attaches a unique identifier to the request and multicasts it to the group of replica managers, using a totally ordered, reliable multicast primitive, and blocked.

- Coordination
  - The group communication system delivers the request to every correct replica manager in the same order

- Execution
  - Every replica manager have the same request queue, they executes the request

- Agreement (no)

- Response
  - Each replica manager sends its response to the front end, FE delivers the first arrived result.
Active replication performance
- Achieve sequential consistency
  - Reliable multicast
    • All correct replica manager process the same set of requests: reliable multicast
  - Total order
    • All correct replica manager process requests in the same order
    • FIFO order
      • Because the FE awaits response before making the next request.
- No linearizability
  - The total order is not same as the real-time order

Chapter 15: Replication
- Introduction
- System model and group communication
- Fault-tolerant services
- Highly available services
- Transactions with replicated data
- Summary

Fault tolerance
- “eager” consistency
  • all replicas reach agreement before passing control to client

High availability vs. fault tolerance
- Fault tolerance
  - “eager” consistency
    • all replicas reach agreement before passing control to client
- High availability
  - “lazy” consistency
    • Reach consistency until next access
    • Reach agreement after passing control to client
  - Gossip, Bayou, Coda

High availability vs. fault tolerance
- Fault tolerance
  - “eager” consistency
    • all replicas reach agreement before passing control to client
- High availability
  - “lazy” consistency
    • Reach consistency until next access
    • Reach agreement after passing control to client
  - Gossip, Bayou, Coda

The gossip architecture
- The architecture
  - Front end connects to any of replica manager
  - Query/Update
  - Replica managers exchange “gossip” messages periodically to maintain consistency
- Two guarantees
  - Each client obtains a consistent service over time(FIFO,get the value observed, though not lastest)
  - Relaxed consistency between replicas
    • All replica managers eventually receive all updates and they apply updates with ordering guarantees

Queries and updates in a gossip service
- Request
  - The front end sends the request to a replica manager
    • Query: client may be blocked
    • Update: unblocked
- Update response
  - Replica manager replies immediately
- Coordination
  - Suspend the request until it can be apply
    • May receive gossip messages that sent from other replica managers
Queries and updates in a gossip service … continued

- **Execution**
  - The replica manager executes the request
- **Query response**
  - Reply at this point
- **Agreement**
  - Exchange gossip messages which contain the most recent updates applied on the replica
    - Exchange occasionally
    - Ask the particular replica manager to send when some replica manager finds it has missed one

Front ends propagate their timestamps whenever clients communicate directly

A gossip replica manager

- **Value**: value of the object maintained by the RM.
- **Value timestamp**: the timestamp that represents the updates reflected in the value. Updated whenever an update operation is applied. \{2, 3, 5\}
- **Update log**: Record all received updates; *stable* update
- **Replica timestamp**: Represents the updates that have been accepted by the replica manager
- **Executed operation table**: Filter duplicated updates that could be received from front end and other replica managers
- **Timestamp table**: Contain a vector timestamp for each other replica manager to identify what updates have been applied at these replica managers

The front end’s version timestamp

- **Access the gossip service** (Client exchange data)
  - Communicate directly
- **A vector timestamp at each front end \{3, 4, 5\**
  - Contain an entry for each replica manager
    - Attached to every message sent to the gossip service or other front ends
  - When front end receives a message
    - Merge the local vector timestamp with the timestamp in the message
- **The significance of the vector timestamp**
  - Reflect the version of the latest data values accessed by the front end

Rm.st<1,0,0>
Value.st<1,0,0>
Value ‘A’

- logRecord = \(\langle i, ts, u.op, u.prev, u.id \rangle\)
- \(\langle 0, \langle 1,0,0 \rangle, ‘A’, \langle 0,0,0 \rangle, 01 \rangle\)
- FE1 first updates the replica 0 with ‘A’, it sets the u.prev = \(\langle 1,0,0 \rangle\), at next time, FE1 tries access the object via RMI, it is *pend*.
- After rm0 transfers the update ‘A’ to rm1, and operation is executed, ValueTS at rm1 is set as \(\langle 1,1,0 \rangle\), then FE1 can get the response

Rm.st<0,1,0>
Value.st<0,1,0>
Value ‘B’

- logRecord = \(\langle i, ts, u.op, u.prev, u.id \rangle\)
- \(\langle 0, \langle 0,0,0 \rangle, ‘B’, \langle 0,0,0 \rangle, 01 \rangle\)
Query operations in gossip service

- When the query reach the replica manager
  - If \( q.prev \leq valueTS \)
    - Return immediately
  - The timestamp in the returned message is \( valueTS \)
  - Otherwise
    - Pend the query in a hold-back queue until the condition meets
    - E.g. \( valueTS = (2,5,5), q.prev=(2,4,6) \): one update from replica manager 2 is missing
- When query return
  - \( frontEndTS := merge(frontEndTS, new) \) \{(2,5,6)\}

Gossip service

- Update log: Rm keeps update in a log for two reasons:
  - Rm can not yet apply the update because it it not yet stable.
  - Even the update has become stable and has been applied to the value, the rm has not received confirmation that this update has been received at all other rm.

Gossip messages

- Exchange gossip message
  - Estimate the missed messages of one replica manager by its timestamp table
  - Exchange gossip messages periodically or when some other replica manager ask

- The format or a gossip message
  - \( m.log \): one or more updates in the source replica manager’s log
  - \( m.ts \): the replica timestamp of the source replica manager
When receiving a gossip message ...continued

- **Update the timestamp table**
  - If the gossip message is from replica manager \( j \), then
    \[
    tableTS[j] = m.ts
    \]
- **Discard any record \( r \) in the log for an update that has been received everywhere.**
  - if \( tableTS[j][c] \geq r.ts[c] \), for all \( i \), then discard \( r \)
  - \( c \) is the replica manager that created \( r \)

\[
\text{logRecord } \{ 1, ts=[2,2,5], u.op, u.prev=[2,1,5], u.id \}
\]

---

**Update propagation**

- How often to exchange gossip messages?
  - Minutes, hours or days
  - Depend on the requirement of application
- How to choose partners to exchange?
  - Random
  - Deterministic
    - Utilize a simple function of the replica manager’s state to make the choice of partner
      - Topological
    - Mesh, circle, tree

---

**The Coda file system**

- **Limits of AFS**
  - Read-only replica
- **The objective of Coda**
  - Constant data availability
- **Coda: extend AFS on**
  - Read-write replica
    - Optimistic strategy to resolve conflicts
    - Disconnected operation

---

**The Coda architecture**

- **Venus/Vice**
  - Vice: replica manager
  - Venus: hybrid of front end and replica manager. they also need to manage the a local cache of file.
- **Volume storage group (VSG)**
  - The set of servers holding replicas of a file volume
- **Available volume storage group (AVSG)**
  - Venus know AVSG of each file, AVSG varies as server become accessible or are made inaccessible.
- **Access a file**
  - The file is serviced by any server in AVSG, and cached at client computer.
  - Call back distribute to other replica manager.

---

**When receiving a gossip message ...continued**

- **Update the timestamp table**
  - If the gossip message is from replica manager \( j \), then
    \[
    tableTS[j] = m.ts
    \]
- **Discard any record \( r \) in the log for an update that has been received everywhere.**
  - if \( tableTS[j][c] \geq r.ts[c] \), for all \( i \), then discard \( r \)
  - \( c \) is the replica manager that created \( r \)

\[
\text{logRecord } \{ 1, ts=[2,2,5], u.op, u.prev=[2,1,5], u.id \}
\]

---

**The Coda architecture ... continued**

- **On close a file**
  - Copies of modified files are broadcast in parallel to all of the servers in the AVSG
  - Allow file modification when the network is partitioned (optimistic), relied on the all file cached.
  - When network partition is repaired, new updates are reapplied to the file copies in other partition
    - Meanwhile, file conflict is detected
**The Coda architecture . . . continued**

- **Disconnected operation**
  - When the file’s AVSG becomes empty, and the file is in the cache
  - Updates in the disconnected operation apply on the server later on when AVSG becomes nonempty
  - if there are conflicts, resolve manually

- **The principle of design of CODA**
  - Copies of files residing on the server are more reliable than in the caches of client computers.

**The replication strategy**

- **Coda version vector (CVV)**
  - Attached to each version of a file
  - Each element of the CVV is an estimate of the number of modifications performed on the version of the file that is held at the corresponding server

- **Example: CVV = (2,2,1)**
  - The file on server1 has received 2 updates
  - The file on server2 has received 2 updates
  - The file on server3 has received 1 update

**How to construct a CVV**

- **File F is replicated at 3 servers: s1,s2,s3**
  - VSG={s1,s2,s3}
  - F is modified at the same time by c1 and c2
  - Because network partition, AVSG of c1 is {s1,s2}, AVSG of c2 is {s3}

- **When a modified file is closed**
  - Venus broadcast the file with current CVV to AVSG
  - Each server in AVSG check the CVV, if it greater than one currently held, store the new one and return a positive AKW
  - The Venus computer new CVV by incrementing one to element corresponding all servers giving AKW , and distribute it to AVSG

- **Initially**
  - The CVVs for F at all 3 servers are [1,1,1]

- **C1 updates the file and close**
  - the CVVs at s1 and s2 become [2,2,1]
  - There is an update applied on s1 and s2 since beginning
When the network failure is repaired
– C2 modify AVSG to \{s1, s2, s3\} and requests the CVVs for F from all members of the new AVSG
– v1: CVV of a file at server1, v3: CVV of the file at server3
– v1>=v3, or v1<=v3 : [2,2,1] vs [1,1,1], no conflict
– s3 replace its CVV with CVV at s2.

How to construct a CVV

Initially
– The CVVs for F at all 3 servers are [1,1,1]

C1 updates the file and close
– the CVVs at s1 and s2 become [2,2,1]
  • There is an update applied on s1 and s2 since beginning

C2 updates the file and close twice
– The CVV at s3 become [1,1,3]
  • There are two updates applied on s3 since beginning

Example … continued

• Initially
  – The CVVs for F at all 3 servers are [1,1,1]
• C1 updates the file and close
  – The CVVs at s1 and s2 become [2,2,1]
    • There is an update applied on s1 and s2 since beginning
• C2 updates the file and close twice
  – The CVV at s3 become [1,1,3]
    • There are two updates applied on s3 since beginning

Example … continued

• C2 modify AVSG to \{s1, s2, s3\} and requests the CVVs for F from all members of the new AVSG
• v1: CVV of a file at server1, v3: CVV of the file at server3
• Neither v1>=v3, nor v3>=v1: conflict
• C2 find [2,2,1]<>[1,1,3], that means conflict happens
  • Conflict means concurrent updates when network happens
  • C2 manually resolve the conflict

Chapter 14: Replication

• Introduction
• System model and group communication
• Fault-tolerant services
• Highly available services
• Transactions with replicated data
• Summary

One-copy serializability

• What is one-copy serializability
  – The effect of transactions performed by clients on replicated objects should be the same as if they had been performed one at a time on a single set of objects
• Architecture of replicated transactions
  – Where to forward a client request?
  – How many replica managers are required to complete an operation?
  – Consideration abort the commitment
• Different replication schemes
  – Available copies, Quorum consensus, Virtual partition
Architectures for replicated transactions

- **primary-backup**
  - All front ends communicate with a “primary” replica manager to perform an operation
  - The replica manager keeps the backups up to date
- **Cooperation of the replica managers**
  - [Read-one/write-all]
  - Quorum consensus
- **Updates propagation**
  - Lazy approach
    - forwards the updates to other replica managers until after a transaction commits
  - Eager approach
    - forwards the updates to other replica managers within a transaction and before it commits

The passive model for fault tolerance

- Cooperation of the replica managers
- Read-one/write-all scheme example

The two-phase commit protocol

- Two-level nested two phase commit protocol
  - Top level subtransaction: for the primary object
  - The second level subtransaction: for the other objects

Read-one/write-all scheme example

- A simple replication scheme
- How to obtain one-copy serializability?
  - Write lock
    - When applying a write operation, set a write lock on each object
  - Read lock
    - When applying a read operation, set a read lock on any of object
- Deadlock may happen
  - But one-copy serializability is maintained

Available copies replication

- Read-one/write-all is not realistic
  - Some of the replica managers may be unavailable
- Available copies replication
  - Read: be performed by any of available object
  - Write: be performed by all available objects
- Example
- How to obtain one-copy serializability?
  - Can local locking scheme work?

Replica manager failure

- Inconsistency due to server crash
  - RM may crash during a transaction
- Example
  - X fails after T has getBalance but before U deposit
  - N fails after U has getBalance but before U deposit
  - The concurrency control on A at RM x does not prevent transaction U from updating A at RM x, so that inconsistency happen
- Local concurrency control is not sufficient to ensure one-copy serializability
Concurrency control in addition to locking
- Ensure that any failure or recovery event does not appear to happen during the progress of a transaction.
- Such dependencies cannot arise if the failure and recoveries of replicas of objects are serialized with respect to transactions.

Example
- Since $T$ has read from an object at $X$ and observes the failure of $N$ when it attempts to update, so if the transaction is valid, the relation of failures and the transaction $T$ must be
  - $N$ fails $\rightarrow T$ reads object $A$ at $X$; $T$ writes object $B$ at $M$ and $P \rightarrow T$ commits $\rightarrow X$ fails
- Similarly, the relation of failures and transaction $U$ must be
  - $X$ fails $\rightarrow U$ reads object $B$ at $N$; $U$ writes object $A$ at $Y$ $\rightarrow U$ commits $\rightarrow N$ fails
- Find conflict, so if $T$ is validated firstly, then $U$ is aborted, and vice versa.

Network partition
- Network partition for replicated transactions
  - May lead to inconsistency
- Deal with partition in available copies scheme
  - Assumption: partitions will eventually be repaired
  - Compensate scheme: if find conflict when partition is repaired, abort some transactions
  - Precedence graph: detect inconsistencies between partitions

Avoid inconsistency in the case of partition
- Conflicting operations can be carried out within only one of the partitions
- Quorum
  - A subgroup whose size gives it the right to carry out operations

Votes
- Each object is assigned an integer that can be regarded as a weighting related to the desirability of using a particular copy

Quorum scheme
- Read quorum
  - Before read: must obtain a read quorum of $R$ votes
- Write quorum
  - Before write: must obtain a write quorum of $W$ votes
  - $W > \frac{1}{2}$ the total votes, $R + W >$ total number of votes for the group
    - any conflicting operations pair must be performed on at least one common copies

Different performance or reliability
- Decrease $W$ (or $R$): increase the performance of write (or read)
- Increase $W$ (or $R$): increase the reliability of write (or read)

Weak representatives
- Local cache at client computers
- Vote = 0
- A read may be performed on it, once a read quorum has been obtained and it is up-to-date
**An example from Gifford**

- **Example 1**
  - A file with a high read-to-write ratio
  - Replication is used to enhance the performance of the system, not the reliability
- **Example 2**
  - A file with a moderate read-to-write ratio
  - Read can be satisfied from the local RM
  - Write must access one remote RM
- **Example 3**
  - A file with a very high read-to-write ratio
  - Read-one/write-all

**Virtual partition algorithm**

- **Combination of available copies algorithm and quorum consensus algorithm**
  - Quorum consensus algorithm
    - work correctly in the presence of partitions
  - Available copies algorithm
    - Less expensive for read operations
- **Virtual partition**
  - A partition which has enough RMs to meet the quorum criteria
  - Perform available copies algorithm in a virtual partition

**Example**

- **Four RMs of a file: V, X, Y and Z**
  - R=2, W=3
- **Initially**
  - V, X, Y and Z can contact with each other
  - Conduct available copies algorithm
- **Network partition happens**
- **Create a virtual partition**
  - V keeps on trying to contact Y and Z until one or both of them replies
  - V, X and Y comprise a virtual partition since they are sufficient to form read and write quorum
  - Conduct available copies algorithm in the virtual partition

**Implementation of virtual partitions**

- **Overlapping virtual partitions**
  - E.g. when Y and Z creates virtual partition simultaneously
  - Conflict
    - Read lock on Z will not conflict with write lock in another virtual partition, so one-copy serializability is broken
- **Approach**
  - Logical timestamp of a virtual partition
    - Creation time of the virtual partition
  - If there are simultaneously creating virtual partition, create the one with higher logical timestamp

**Chapter 14: Replication**

- Introduction
- System model and group communication
- Fault-tolerant services
- Highly available services
- Transactions with replicated data
- Summary

**Summary**

- Replication for distributed systems
  - High performance, high availability, fault tolerance
- Group communication
  - Group management service – view delivery
  - View-synchronous group communication
**Summary**

- Replication for fault tolerance
  - Linearizability and sequential consistency
  - Primary-backup replication
    - Maintain linearizability
    - Use view synchronous group communication
  - Active replication
    - Maintain sequential consistency
    - Based on total-order, reliable group communication

**Summary (2)**

- Replication for high availability
  - Gossip protocol
    - Lazy consistency
    - Coda
  - Transactions with replicated data
    - Read-one/write-all
    - Available copies replication
    - Quorum consensus methods
    - Virtual partition
      - Combination of available copies replication and quorum consensus methods

**Transactions on replicated objects**

- ~End~

**Available copies**

- Concurrency control
  - At X, transaction T has read A and therefore transaction U is not allowed to update A with the deposit operation until transaction T has completed

**Network partition**
Gifford’s quorum consensus examples

<table>
<thead>
<tr>
<th>Example</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Latency (milliseconds)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replica 1</td>
<td>75</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>Replica 2</td>
<td>65</td>
<td>100</td>
<td>750</td>
</tr>
<tr>
<td>Replica 3</td>
<td>65</td>
<td>750</td>
<td>750</td>
</tr>
<tr>
<td><strong>Voting configuration</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replica 1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Replica 2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Replica 3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Quorum sizes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>W</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Derived performance of file suite:

<table>
<thead>
<tr>
<th>Read</th>
<th>Latency</th>
<th>Blocking probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replica 1</td>
<td>65</td>
<td>0.01</td>
</tr>
<tr>
<td>Replica 2</td>
<td>75</td>
<td>0.0002</td>
</tr>
<tr>
<td>Replica 3</td>
<td>75</td>
<td>0.000001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write</th>
<th>Latency</th>
<th>Blocking probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replica 1</td>
<td>75</td>
<td>0.0101</td>
</tr>
<tr>
<td>Replica 2</td>
<td>100</td>
<td>0.03</td>
</tr>
<tr>
<td>Replica 3</td>
<td>750</td>
<td>0.03</td>
</tr>
</tbody>
</table>

Two network partitions

Virtual partitions

Two overlapping virtual partitions

Phase 1:
- The initiator sends a `Join` request to each potential member. The argument of `Join` is a proposed logical timestamp for the new virtual partition.
- When a replica manager receives a `Join` request, it compares the proposed logical timestamp with that of its current virtual partition.
  - If the proposed logical timestamp is greater it agrees to join and replies `Yes`;
  - If it is less, it refuses to join and replies `No`.

Phase 2:
- If the initiator has received sufficient `Yes` replies to have read and write quorum, it may complete the creation of the new virtual partition by sending a `Confirmation` message to the sites that agreed to join. The creation timestamp and list of actual members are sent as arguments.
- Replica managers receiving the `Confirmation` message join the new virtual partition and record its creation timestamp and list of actual members.