Chapter 7: Replication and Consistency
Introduction to Replication

Replication is…

- **Replication of data**: the maintenance of copies of data at multiple computers
- **Object replication**: the maintenance of copies of whole server objects at multiple computers

Replication can provide …

- **Performance enhancement, scalability**
  - Remember *caching*: improve performance by storing data locally, but data are incomplete
  - Additionally: several web servers can have the same DNS name. The servers are selected by DNS in turn to share the load
  - Replication of read-only data is simple, but replication of frequently changing data causes overhead in providing current data
Introduction to Replication

Increased availability

- Sometimes needed: nearly 100% of time a service should be available
- In case of server failures: simply contact another server with the same data items
- Network partitions and disconnected operations: availability of data if the connection to a server is lost. But: after re-establishing the connection, (conflicting) data updates have to be resolved

Fault-tolerant services

- Guaranteeing correct behaviour in spite of certain faults (can include timeliness)
- If \( f \) in a group of \( f+1 \) servers crash, then 1 remains to supply the service
- If \( f \) in a group of \( 2f+1 \) servers have byzantine faults, the group can supply a correct service
When to do Replication?

*Replication at service initialisation*

- Try to estimate number of servers needed from a customer's specification regarding performance, availability, fault-tolerance, …
- Choose places to deposit data or objects
  - Example: root servers in DNS

*Replication 'on demand'*

- When failures or performance bottlenecks occur, make a new replica, possibly placed at a new location in the network
  - Example: web server of an online shop
- Or: to improve local access operations, place a copy near a client
  - Example: (DNS) caching, disconnected operations
Why Object Replication?

- It is useful to consider objects instead of only considering data
- Objects have the benefit of encapsulating data and operations on data. Thus, object-specific operation requests can be distributed
- But: now one has to consider internal object states! This topic is related to mobile agents, but it becomes more complicated in the case of consistent internal states!
Requirements for Replication

For all of these application areas, one requirement holds: the client should not be aware of using a group of computers!

*Replication transparency*

- Clients do not work on multiple physical copies, they only see one logical object which they request to do an action
- Clients only expect a single result, not results of each data copy
- Needed: propagation of updates

*Consistency*

- How to ensure all data copies to be consistent?
- Specific degrees depending on the application:
  - Temporarily inconsistencies could be tolerated
  - When multiple clients are connected using different copies, they should get consistent results

*Performance, availability, fault-tolerance ⇔ consistency, up-to-date information*
Management of Replicated Data

*Needed: replication transparency and consistency*

- Client requests are handled by *Front Ends*. A front end provides replication transparency.
- By the front ends, clients see a service that gives them access to logical objects, which are in fact replicated at several *Replica Managers* which guarantee consistency.
- Client request operations:
  - *read-only* requests: calls without making updates
  - *update* requests: calls executing write operations (but could also execute read operations)
System Model

- Each *logical* object is implemented by a collection of *physical* copies called *replicas* (the replicas are not necessarily consistent all the time; some may have received updates not yet delivered to the others)

- **Replica managers**
  - Contain replicas on a computer and access them directly
  - Replica managers apply operations to replicas recoverably, i.e. they do not leave inconsistent results if they crash
  - *Static* systems are based on a fixed set of replica managers
  - In a *dynamic system*, replica managers may join or leave (e.g. when they crash)
  - A replica manager can be a *state machine* which has the following properties:
    a) Operations are applied atomically
    b) The current state is a deterministic function of the initial state and the applied operations
    c) All replicas start identically and carry out the same operations
    d) The operations must not be affected by clock readings etc.
Performing a Request

In general: five phases for performing a request on replicated data

- **Issue request.** The front end either:
  - sends the request to a single replica manager that passes it on to the others, or
  - multicasts the request to all of the replica managers

- **Coordination.** For consistent execution, the replica managers decide
  - whether to apply the request (e.g. because of failures)
  - how to order the request relative to other requests (according to FIFO, causal or total ordering)

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

- **Agreement.** The replica managers *agree* on the effect of the request, e.g. perform it 'lazily' or immediately

- **Response.** One or more replica managers reply to the front end, which combines the results:
  - For high availability, give first response to client
  - To tolerate faults, take a vote
Replication Example

How to provide fault-tolerant services, i.e. a service that is provided correct even if some processes fail?

- Simple replication system: e.g. two replica managers $A$ and $B$ each managing replicas of two accounts $x$ and $y$. Clients use the local replica manager if possible, after responding to a client the update is transmitted to the other replica manager

<table>
<thead>
<tr>
<th>Client 1:</th>
<th>Client 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$setBalance_B(x,1)$</td>
<td></td>
</tr>
<tr>
<td>$setBalance_A(y,2)$</td>
<td></td>
</tr>
<tr>
<td>$getBalance_A(x) \rightarrow 0$</td>
<td>$getBalance_A(y) \rightarrow 2$</td>
</tr>
</tbody>
</table>

- Initial balance of $x$ and $y$ is $0$
  - Client 1 first updates $x$ at $B$ (local). When updating $y$ it finds $B$ has failed, so it uses $A$ for the next operation.
  - Client 2 reads balances at $A$ (local), but because $B$ had failed, no update was propagated to $A$: $x$ has amount 0.

*Be careful when designing replication algorithms! You need a consistency model*
## Consistency Models

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strict</td>
<td>Absolute time ordering of all shared accesses matters</td>
</tr>
<tr>
<td>Linearisability</td>
<td>All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a global timestamp</td>
</tr>
<tr>
<td>Sequential</td>
<td>All processes see all shared accesses in the same order. Accesses are not ordered in time</td>
</tr>
<tr>
<td>Causal</td>
<td>All processes see causally-related shared accesses in the same order</td>
</tr>
<tr>
<td>FIFO</td>
<td>All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order</td>
</tr>
</tbody>
</table>
Distinction between mechanisms: replication…

- …for *fault tolerance*
- …for *highly available services*
- …in *transactions*
Replication for fault tolerance: *passive* or *primary-backup model*

- There is at any time a *single primary* replica manager and one or more secondary replica managers (backups, slaves)

- Front ends only communicate with the primary replica manager which executes the operation and sends copies of the updated data to the backups
- If the primary fails, one of the backups is promoted to act as the primary
- This system implements linearisability, since the primary sequences all the operations on the shared objects
- Variation: clients can read from backups, which reduces the work load for the primary, but does only achieve sequential consistency
Active Replication

The replica managers are *state machines all playing the same role* and are organised as a group:

- A front end multicasts each request to the group of replica managers
- All replica managers start in the same state and perform the same operations in the same order so that their state remains identical (notice: totally ordered reliable multicast would be needed to guarantee the identical execution order!)
- If a replica manager crashes it has no effect on the performance of the service because the others continue as normal
- Failures in a few replicas can be tolerated because the front end can collect and compare the replies it receives
How to provide services with high availability, i.e. how to use replication to achieve an availability of (ideally) 100%?

- Reasonable response times for as much of the time as possible
- Even if some results do not conform to sequential consistency
- e.g. a disconnected user may accept temporarily inconsistent results if he can continue to work and fix inconsistencies later

**Difference to fault tolerant systems**

- For fault tolerance, all replica managers have to agree on a result – *'eager'* evaluation, not acceptable for reaching reasonable response times
- Instead for high availability:
  - Only contact a minimum number of replica managers necessary to reach an acceptable level of service
  - Client should tied up for a minimum time while managers coordinate their actions
  - Weaker consistency generally requires less agreement and makes data more available. Updates are propagated *'lazily'*. 
The Gossip Architecture

- The gossip architecture is a framework for implementing *highly available services*
  - Data is replicated *close to the location of clients*
  - Replica managers periodically exchange ‘gossip’ messages containing updates they have received from clients
- Two basic types of operations are provided:
  - *queries* - read only operations
  - *updates* - modify (*but do not read*) the state
- Front ends choose any replica manager to send queries and updates to, selected by availability and response times
- *Two guarantees* are made (even if managers are temporarily unable to communicate with one another):
  - *Each client gets a consistent service over time* (i.e. the data reflects at least the updates seen by the client so far, even if it uses different replica managers). Vector timestamps are used – with one entry per manager.
  - *Relaxed consistency between replicas*. All replica managers eventually receive all updates and use ordering guarantees to suit the needs of the application (generally causal ordering). A client may observe stale data.
Query & Update Operations

- The service consists of a collection of replica managers that exchange gossip messages.
- Queries and updates are sent by a client via a front end to any replica manager.

The front end converts operations containing both, reading and writing, in two separate calls.

For ordering operations, the front end sends a timestamp `prev` with each request to denote its latest state.

A new timestamp `new` is passed back in a read operation to mark the data state the client has seen last.
Timestamps

- Each front end keeps a vector timestamp that reflects the latest data value seen by the front end (prev).
- Clients can communicate with each other. This can lead to causal relationships between client operations which has to be considered in the replicated system. Thus, communication is made via the front ends including an exchange of vector timestamps allowing the front ends to consider causal ordering in their timestamps.
Gossip Processing of Queries and Updates

Phases in performing a client request:

1. **Request**
   - Front ends normally use the same replica manager for all operations
   - Front ends may be blocked on queries

2. **Update response** - The replica manager replies as soon as it has received the update

3. **Coordination** - The replica manager receiving a request waits to process it until the ordering constraints apply. This may involve receiving updates from other replica managers in gossip messages

4. **Execution** - The replica manager executes the request

5. **Query response** - If the request is a query the replica manager now replies

6. **Agreement** - Replica managers update one another by exchanging gossip messages with the most recent received updates. This has not to be done for each update separately
Gossip Replica Manager

Other replica managers

Gossip messages

Replica manager

Replica log

Replica timestamp

Timestamp table

Update log

Stable updates

Executed operation table

Value timestamp

Value

OperationID Update Prev

FE

Updates
Replica Manager State

Main state components of a replica manager:

- **Value**: reflects the application state as maintained by the replica manager (each manager is a state machine). It begins with a defined initial value and is applied update operations to.
- **Value timestamp**: the vector timestamp that reflects the updates applied to get the saved value.
- **Executed operation table**: prevents an operation from being applied twice, e.g. if received from other replica managers as well as the front end.
- **Timestamp table**: contains a vector timestamp for each other replica manager, extracted from gossip messages.
- **Update log**: all update operations are recorded immediately when they are received. An operation is held back until ordering allows it to be applied.
- **Replica timestamp**: a vector timestamp indicating updates accepted by the manager, i.e. placed in the log (different from value’s timestamp if some updates are not yet stable).
Query & Update Operations

- **Query** includes a timestamp. The operation is marked as pending until the manager's timestamp is larger than the operation's timestamp.

- **Update** operations are processed in causal order. A front end can send an update operation containing timestamp and identifier \( id \) to one or several replica managers.
  - When replica manager \( i \) receives an update request, it checks whether it is new, by looking for the \( id \) in its executed ops table and its log.
  - If it is new, the replica manager:
    - increments by 1 the \( i \)th element of its replica timestamp,
    - assigns the result as new timestamp \( ts \) to the update, and
    - stores the update in its log.
  - The replica manager returns \( ts \) to the front end, which merges it with its vector timestamp (Note: by sending a request to several managers the front end gets back several timestamps which have to be merged).
  - Depending on the timestamp for an operation, it can be delayed till gossip messages from other replica managers arrive. When the timestamp allows, the replica manager applies the operation and makes an entry in the *executed operation table*.
Gossip Messages

- The timestamp table contains a vector timestamp for each other replica, collected from gossip messages.
- A replica manager uses entries in this timestamp table to estimate which updates another manager has not yet received. These information are sent in a gossip message.
- A gossip message $m$ contains the log $m.log$ and the replica timestamp $m.ts$.
- A manager receiving gossip message $m$ has the following main tasks:
  - Merge the arriving log with its own.
  - Apply in causal order updates that are new and have become stable.
  - Remove redundant entries from the log and executed operation table when it is known that they have been applied by all replica managers.
  - Merge its replica timestamp with $m.ts$, so that it corresponds to the additions in the log.
Update Propagation

The given architecture does not specify when to exchange gossip messages. To design a robust system in which each update is propagated in reasonable time, an exchange strategy is needed.

The time which is required for all replica managers to receive a given update depends upon three factors:

1. The frequency and duration of network partitions
   This is beyond the system’s control

2. The frequency with which replica managers send gossip messages
   This may be tuned to the application

3. The policy for choosing a partner with which to exchange gossip messages
   • Random policies choose a partner randomly but with weighted probabilities so as to favour some partners over others
   • Deterministic policies give fixed communication partners
   • Topological policies arrange the replica managers into a fixed graph (mesh, circle, tree, …) and messages are passed to the neighbours
   • Other strategies: consider transmission latencies, fault probabilities, …
Properties of the Gossip Architecture

• The gossip architecture is designed to provide a highly available service
  + Clients with access to a single replica manager can work even when other managers are inaccessible
    - but this is not suitable for data such as bank accounts
    - it is inappropriate for updating replicas in real time

• Scalability
  - as the number of replica managers grows, so does the number of gossip messages
  - for $R$ managers collecting $G$ updates in a gossip message, the number of messages per request is $2 + (R-1)/G$
    ➢ Variation: increase $G$ and improve the number of gossip messages, but make latency worse
    ➢ For applications where queries are more frequent than updates, use some read-only replicas placed near the client, which are updated only by gossip messages
Operational Transformation Approach

- The so-called *Bayou system* provides data replication for high availability with weaker guarantees than sequential consistency.
- Bayou replica managers cope with variable connectivity by exchanging updates in pairs (like in gossip architecture), but it adopts a markedly different approach in that it enables domain specific conflict detection and resolution to take place.
- All updates are applied and recorded at whatever replica manager they reach.
- Replica manager detect and resolve conflicts when exchanging information by using domain-specific policies. The effect of undo or alter conflicting operations to resolve them is called *operational transformation*.
- Bayou update is a *special case of a transaction*. It is carried out with the ACID guarantees. Bayou may undo and redo updates to the database as execution proceeds.
- The Bayou guarantee is that, eventually, every replica manager receives the same set of updates and applies those updates in such a way that the replica managers' databases are identical.
- In practice, there may be a continuous stream of updates, and the databases may never become identical; but they would become identical if the updates ceased.
Committed and Tentative Updates

- Updates are marked as *tentative* when they are first applied to the database. While they are in this state, they can be undone and re-applied if necessary.
- Tentative updates are eventually placed in a canonical order and marked as *committed*.
- The committed order can be achieved by designating some replica manager as the primary replica manager deciding an order by receiving date.
- A tentative update $t_i$ becomes the next committed update and is inserted after the last committed update $c_N$.

![Diagram showing the order of committed and tentative updates]

All updates $t_0$ to $t_{i-1}$ are to be reapplied after it.
Every Bayou update contains a dependency check and merge procedure in addition to the operation's specification because of the possibility that an update may conflict with other operation that has already been applied:

- A replica manager calls the dependency check procedure before applying the operation.
- It would check whether a conflict would occur if the update was applied and it may examine any part of the database to do that.
- If the dependency check indicates a conflict, then Bayou invokes the operation's merge procedure.
- That procedure alerts the operation that will be applied so that it achieves something similar to the intended effect but avoids a conflict.
- The merge procedure may fail to find a suitable alteration of the operation, in which case the system indicates an error.
- The effect of a merge procedure is deterministic. However – Bayou replica managers are state machines.
Problem with Bayou

Bayou is different from other approaches because it makes replication **non-transparent** to the application:

- Increased complexity for the application programmer: provide dependency checks and merge procedures.
- Increased complexity for the user: getting tentative data and alteration of user-specified operations.

- The operational transformation approach used by Bayou appears in systems for computer supported cooperative work (CSCW).
- This approach is limited in practice to situations where only few conflicts arise, users can deal with tentative data and where data semantics are simple.
Transactions with Replicated Data

Till now: consideration of only single operations on a set of replicas.

But: objects in transactional systems can be replicated to enhance availability and performance. How to deal with atomic sequences of operations?

• The effect of transactions on replicated objects should be the same as if they had been performed one at a time on a single set of objects

• This property is called *one-copy serialisability*

• Each replica manager provides concurrency control and recovery of its own objects

• Assumption for presented methods: two-phase locking is used

• Replication makes recovery more complicated… when a replica manager recovers, it restores its objects with information from other managers
Read One/Write All (ROWA)

- Used for transactions
- One replica manager is required for a *read* request, all replica managers for a *write* request
  - Every write operation must be performed at all managers, each of which applies a write lock
  - Each read operation is performed by a single manager, which sets a read lock
- Consider pairs of operations by different transactions on the same object:
  - Any pair of write operations will require conflicting locks at all of the managers
  - A read operation and a write operation will require conflicting locks at a single manager
Available Copies Replication

- The simple read one/write all scheme is not realistic:
  - It cannot be carried out if some of the replica managers are unavailable either because they have crashed or because of a communication failure.

- The available copies replication scheme is designed to allow some managers to be temporarily unavailable:
  - A read request can be performed at any available replica manager.
  - Write requests are performed by the receiving manager and all other available managers in the group.
  - As long as the set of available managers does not change, local concurrency control achieves one-copy serialisability in the same way as read one/write all.
  - Problems with this occur if a manager fails/recovers during the progress of conflicting transactions.
Read One/Write All Available (ROWA-A)

- T’s `getBalance` is performed by X whereas T's `deposit` is performed by M, N and P.
- At X, T has read A and has locked it. Therefore U's `deposit` is delayed until T finishes.
- Local concurrency control achieves one-copy serialisability provided the set of replica managers does not change.
- …but we have managers failing and recovering.
Network partitions

- Part of the network fails creating *sub-groups*, which cannot communicate with one another.
- Replication schemes assume partitions will be repaired.
  - Operations done during a partition must not cause inconsistency.
  - Optimistic schemes (e.g., available copies with validation) allow all operations and resolve inconsistencies when a partition is repaired.
  - Pessimistic schemes (e.g., quorum consensus) prevent inconsistency e.g., by limiting availability in all but one sub-group.
Available Copies with Validation

• Optimistic approach
• The algorithm is applied within each partition
• Maintains the normal level of availability for read operations, even during partitions
• When a partition is repaired, the possibly conflicting transactions that took place in the separate partitions are validated:
  - if the validation fails then some steps must be taken to overcome the inconsistencies
  - if there had been no partition, one of a pair of transactions with conflicting operations would have been delayed or aborted
  - As there has been a partition, pairs of conflicting transactions have been allowed to commit in different partitions – then the only choice after the event is to abort one of them, this requires making changes in the objects and in some cases, compensating effects in the real world
  - The optimistic approach is only feasible with applications where such compensating actions can be taken
Quorum Consensus Methods

• To prevent transactions in different partitions from producing inconsistent results, make a rule that operations can be performed in only one of the partitions.
• Replica managers in different partitions cannot communicate, thus each subgroup decides independently whether they can perform operations.
• A quorum is a sub-group of replica managers whose size gives it the right to perform operations. The right could be given by having the majority of the replica managers in the partition.
• In quorum consensus schemes, update operations may be performed by a subset of the replica managers forming a quorum:
  - The other replica managers have out-of-date copies.
  - Version numbers or timestamps can be used to determine which copies are up-to-date.
  - Operations are applied only to copies with the current version number.
Gifford’s Quorum Consensus

- Assign a number of *votes* to each data copy at a replica manager
- A vote is a weighting giving the desirability of using a particular copy
- Thus, groups of replica managers can be configured to consider different performance or reliability characteristics
- Each *read* operation must obtain a read quorum of $R$ votes before it can read from any up-to-date copy
- Each *write* operation must obtain a write quorum of $W$ votes before it can do an update operation
- $R$ and $W$ are set for a group of replica managers such that:
  - $W > \text{half the total votes}$
  - $R + W > \text{total number of votes for the group}$
- In case of a partition it is not possible to perform conflicting operations on the same data file in different partitions
- Performance and reliability of write resp. read operations are increased with decreasing $W$ resp. $R$
- Main disadvantage: performance of read operations is degraded
Three examples of the voting algorithm:

a) A correct choice of read and write set
b) A choice that may lead to write-write conflicts
c) A correct choice, coming to ROWA (read one, write all)
Gifford’s Quorum Consensus

- Before a *read* operation, a read quorum is collected:
  - Make enquiries at replica managers to find a set of copies, the sum of whose votes is not less than $R$ (not all of these copies need be up to date)
  - As each read quorum overlaps with every write quorum, every read quorum is certain to include at least one current copy
  - The *read* operation may be applied to any up-to-date copy
- Before a *write* operation, a write quorum is collected
  - Make enquiries at replica managers to find a set with up-to-date copies, the sum of whose votes is not less than $W$
  - If there are insufficient up-to-date copies, then an out-of-date file is replaced with a current one, to enable the quorum to be established
  - The *write* operation is then applied by each replica manager in the write quorum, the version number is incremented, and completion is reported to the client
  - The files at the remaining available managers are then updated in the background
- Two-phase read/write locking is used for concurrency control
## Gifford’s Quorum Consensus

### Examples

<table>
<thead>
<tr>
<th>Latency (milliseconds)</th>
<th>Example 1</th>
<th>Example 2</th>
<th>Example 3</th>
</tr>
</thead>
<tbody>
<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>Voting configuration</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>Quorum sizes</td>
<td>R</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>W</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Derived performance of file suite:

<table>
<thead>
<tr>
<th>Read</th>
<th>Latency</th>
<th>75</th>
<th>75</th>
<th>75</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Blocking probability</td>
<td>0.01</td>
<td>0.0002</td>
<td>0.000001</td>
</tr>
<tr>
<td>Write</td>
<td>Latency</td>
<td>75</td>
<td>100</td>
<td>750</td>
</tr>
<tr>
<td></td>
<td>Blocking probability</td>
<td>0.01</td>
<td>0.0101</td>
<td>0.03</td>
</tr>
</tbody>
</table>

Derived performance

- **latency**: blocking probability - probability that a quorum cannot be obtained, assuming probability of 0.01 that any single replica manager is unavailable.
Example 1

- Is configured for a file with high read to write ratio with several weak representatives and a single replica manager
- Replication is used for performance, not reliability
- The replica manager can be accessed in 75 ms and the two clients can access their weak representatives on local discs in 65 ms, resulting in lower latency and less network traffic

Example 2

- Is configured for a file with a moderate read to write ratio which is accessed mainly from one local network. Local replica manager has 2 votes and remote managers 1 vote each
- reads can be done at the local replica manager, but writes must access one local and one remote manager. If the local manager fails only reads are allowed

Example 3

- Is configured for a file with a very high read to write ratio
- reads can be done at any replica manager and the probability of the file being unavailable is small. But writes must access all managers