Replication

Overview

- Introduction to replication
- System model and group communication
- Fault-tolerant services
- Highly available services
- Gossip architecture
Introduction to replication

(1)

- Replication of data: the maintenance of copies of data at multiple computers

- Replication can provide the following
  - performance enhancement
    - e.g. several web servers can have the same DNS name and the servers are selected in turn. To share the load.
    - replication of read-only data is simple, but replication of changing data has overheads

Introduction to replication

(2)

- fault-tolerant service
  - guarantees correct behaviour in spite of certain faults (can include timeliness)
  - if $f$ of $f+1$ servers crash then 1 remains to supply the service
  - if $f$ of $2f+1$ servers have byzantine faults then they can supply a correct service

- availability is hindered by
  - server failures
    - replicate data at failure-independent servers and when one fails, client may use another. Note that caches do not help with availability (they are incomplete).
  - network partitions and disconnected operation
    - Users of mobile computers deliberately disconnect, and then on re-connection, resolve conflicts
Requirements for replicated data

- Replication transparency
  - clients see logical objects (not several physical copies)
    - they access one logical item and receive a single result

- Consistency
  - specified to suit the application,
    - e.g. when a user of a diary disconnects, their local copy may be inconsistent with the others and will need to be reconciled when they connect again. But connected clients using different copies should get consistent results. These issues are addressed in Bayou and Coda.

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 conveyed to the others)
- we assume an asynchronous system where processes fail only by crashing and generally assume no network partitions
- replica managers
  - an RM contains replicas on a computer and access them directly
  - RMs apply operations to replicas recoverably
    - i.e. they do not leave inconsistent results if they crash
  - objects are copied at all RMs unless we state otherwise
  - static systems are based on a fixed set of RMs
  - in a dynamic system: RMs may join or leave (e.g. when they crash)
A basic architectural model for the management of replicated data

Clients request operations: those without updates are called read-only requests the others are called update requests (they may include reads). Clients requests are handled by frontends. A frontend makes replication transparent.

Figure 14.1

Clients request operations: those without updates are called read-only requests the others are called update requests (they may include reads). Clients requests are handled by frontends. A frontend makes replication transparent.

Five phases in performing a request

- issue request
  - the FE either
    - sends the request to a single RM that passes it on to the others
    - or multicasts the request to all of the RMs (in state machine approach)
- coordination
  - the RMs decide whether to apply the request; and decide on its ordering relative to other requests (according to FIFO, causal or total ordering)
- execution
  - the RMs execute the request (sometimes tentatively)
- agreement
  - RMs agree on the effect of the request, e.g. perform 'lazily' or immediately
- response
  - one or more RMs reply to FE, e.g.
    - for high availability give first response to client.
    - to tolerate byzantine faults, take a vote
Fault-tolerant services

- provision of a service that is correct even if $f$ processes fail
  - by replicating data and functionality at RMs
  - assume communication reliable and no partitions
  - RMs are assumed to behave according to specification or to crash
  - intuitively, a service is correct if it responds despite failures and clients can’t tell the difference between replicated data and a single copy
  - but care is needed to ensure that a set of replicas produce the same result as a single one would.

Example of a naive replication system

- initial balance of $x$ and $y$ is $0$
  - client 1 updates $X$ at B (local) then finds B has failed, so uses A
  - client 2 reads balances at A (local)
    - as client 1 updates $y$ after $x$, client 2 should see $1$ for $x$
  - not the behaviour that would occur if A and B were implemented at a single server

- Systems can be constructed to replicate objects without producing this anomalous behaviour.

- We now discuss what counts as correct behaviour in a replication system.
The passive (primary-backup) model for fault tolerance

- There is at any time a single primary RM and one or more secondary (backup, slave) RMs.
- FEs communicate with the primary which executes the operation and sends copies of the updated data to the result to backups.
- If the primary fails, one of the backups is promoted to act as the primary.

![Diagram of primary-backup model](image)

Passive (primary-backup) replication. Five phases.

- The five phases in performing a client request are as follows:
  1. Request:
     - A FE issues the request, containing a unique identifier, to the primary RM.
  2. Coordination:
     - The primary performs each request atomically, in the order in which it receives it relative to other requests.
     - It checks the unique id; if it has already done the request it re-sends the response.
  3. Execution:
     - The primary executes the request and stores the response.
  4. Agreement:
     - If the request is an update the primary sends the updated state, the response and the unique identifier to all the backups. The backups send an acknowledgement.
  5. Response:
     - The primary responds to the FE, which hands the response back to the client.
Active replication for fault tolerance (1)

- the RMs are *state machines* all playing the same role and organised as a group.
  - all start in the same state and perform the same operations in the same order so that their state remains identical
- If an RM crashes it has no effect on performance of the service because the others continue as normal
- It can tolerate byzantine failures because the FE can collect and compare the replies it receives

Active replication for fault tolerance (2)

- FE multicasts each request to the group of RMs
- the RMs process each request identically and reply
- Requires totally ordered reliable multicast so that all RMs perform the same operations in the same order
Active replication - five phases in performing a client request

- Request
  - FE attaches a unique id and uses totally ordered reliable multicast to send request to RMs. FE can at worst, crash. It does not issue requests in parallel.
- Coordination
  - the multicast delivers requests to all the RMs in the same (total) order.
- Execution
  - every RM executes the request. They are state machines and receive requests in the same order, so the effects are identical. The id is put in the response.
- Agreement
  - no agreement is required because all RMs execute the same operations in the same order, due to the properties of the totally ordered multicast.
- Response
  - FEs collect responses from RMs. FE may just use one or more responses. If it is only trying to tolerate crash failures, it gives the client the first response.

Highly available services

- we discuss the application of replication techniques to make services highly available.
  - we aim to give clients access to the service with:
    » 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 they can continue to work and fix inconsistencies later
- eager versus lazy updates
  - fault-tolerant systems send updates to RMs in an ‘eager’ fashion (as soon as possible) and reach agreement before replying to the client
  - for high availability, clients should:
    » only need to contact a minimum number of RMs and
    » be tied up for a minimum time while RMs 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
  - RMs periodically exchange 'gossip' messages containing updates
- Gossip service provides two types of operations
  - Queries - read only operations
  - Updates - modify (but do not read) the state
- FE sends queries and updates to any chosen RM
  - One that is available and gives reasonable response times
- Two guarantees (even if RMs are temporarily unable to communicate
  - Each client gets a consistent service over time (i.e. data reflects the updates seen by client, even if the use different RMs). Vector timestamps are used – with one entry per RM.
  - Relaxed consistency between replicas. All RMs eventually receive all updates. RMs use ordering guarantees to suit the needs of the application (generally causal ordering). Client may observe stale data.

Query and update operations in a gossip service

- The service consists of a collection of RMs that exchange gossip messages
- Queries and updates are sent by a client via an FE to an RM
Gossip processing of queries and updates

- The five phases in performing a client request are:
  - request
    » FEs normally use the same RM and may be blocked on queries
    » update operations return to the client as soon as the operation is passed to the FE
  - update response - the RM replies as soon as it has seen the update
  - coordination
    » the RM waits to apply the request until the ordering constraints apply.
    » this may involve receiving updates from other RMs in gossip messages
  - execution - the RM executes the request
  - query response - if the request is a query the RM now replies:
  - agreement
    » RMs update one another by exchanging gossip messages (lazily)
      - e.g. when several updates have been collected
      - or when an RM discovers it is missing an update

Front ends propagate their timestamps whenever clients communicate directly

- each FE keeps a vector timestamp of the latest value seen (prev)
  - which it sends in every request
  - clients communicate with one another via FEs which pass vector timestamps

client-to-client communication can lead to causal relationships between operations.
A gossip replica manager, showing its main state components

- Other replica manager
- Gossip messages
- Replica manager
- Timestamp table
- Replica timestamp
- Stable updates
- Value timestamp
- Value
- Executed operation table
- Operation ID
- Update
- Prev
- Updates
- FE
- FE

Processing of query and update operations

- Vector timestamp held by RM $i$ consists of:
  - $i$th element holds updates received from FEs by that RM
  - $j$th element holds updates received by RM $j$ and propagated to RM $i$

- Query operations contain $q.prev$
  - they can be applied if $q.prev \leq valueTS$ (value timestamp)
  - failing this, the RM can wait for gossip message or initiate them
    - e.g. if $valueTS = (2,5,5)$ and $q.prev = (2,4,6)$ - RM 0 has missed an update from RM 2
  - Once the query can be applied, the RM returns $valueTS (new)$ to the FE. The FE merges $new$ with its vector timestamp
Gossip update operations (1)

- Update operations are processed in causal order
  - A FE sends update operation \( u.op, u.prev, u.id \) to RM \( i \)
    » A FE can send a request to several RMs, using same id
  - When RM \( i \) 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 RM
    » increments by 1 the \( i \)th element of its replica timestamp,
    » assigns a unique vector timestamp \( ts \) to the update

Gossip messages

- An RM uses entries in its timestamp table to estimate which updates another RM has not yet received
  - The timestamp table contains a vector timestamp for each other replica, collected from gossip messages
- Gossip message, \( m \) contains log \( m.log \) and replica timestamp \( m.ts \)
- An RM receiving gossip message \( m \) has the following main tasks
  - Merge the arriving log with its own (omit those with \( ts \leq replicaTS \))
  - 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 RMs
  - Merge its replica timestamp with \( m.ts \), so that it corresponds to the additions in \( m.log \)