Business and Logistics

- Problem sets
  - Answers posted to problem set #1
  - Problem set #2 available

- Term paper proposal
  - Got feedback to most of you

- Term projects
  - Keep meeting regularly
  - Stay on top of it
Acknowledgements

☐ Some material taken from author’s teaching slides
☐ Some figures taken from Distributed Systems: Concepts and Design book
Lecture Objectives

- Discuss how data is replicated in a distributed system
- Look at how transparency is provided
- Discuss fault tolerance and availability
- Understand correctness for replicated objects
  - Linearizability and sequential consistency
- Discuss five stages of operation in replicated system
  - Request, coordination, execution, agreement, response
- Discuss active and passive replication
- Gossip architecture
- Network partitioning
Introduction to Replication

- Replication is the maintenance of copies of data at multiple computers
  - *Logical* objects implemented by a collection of *physical* copies called *replicas*
  - Not necessarily consistent all the time

- Replication is a common distributed technique for:
  - Performance enhancement
    - multiple servers used to balance workload
  - Availability
    - multiple servers increase probability of data availability
    - network partitions and disconnected operation
  - Fault-tolerant service
    - guarantees correct behavior in spite of certain faults
Requirements for Replicated Data

☐ Any system based on replications has two requirements:

☐ *Transparency* (fundamental requirement)
  ❖ Clients see logical objects (not several physical copies)
    ➢ they access one logical item and receive a single result
  ❖ Should not know that the data is replicated or where

☐ *Consistency*
  ❖ Specified to suit the application
    ➢ connected clients using different copies should get consistent results
    ➢ when a user disconnects, their local copy may be inconsistent with the others, but this can be reconciled when they connect again
What is so hard about replication?

☐ Consider static data (does not change)
  ☐ Copy the data to multiple places
  ☐ Each place where data is replicated maintains the data
  ☐ But the data does not change, so there is no problem
  ☐ Benefits?

☐ Consider dynamic data (can change)
  ☐ Maintaining consistency is a challenge
  ☐ Think about a single updater
  ☐ Then think about multiple updates at the same time
  ☐ Need to maintain a logical data view for users
  ☐ Would like to perform better than a central store
Replication Examples

- We all have some experience with distributed applications that do replication
- Can you name some?
- Recent developments in version control systems have definitely taken on a more sophisticated distributed flavor
  - RCS, CVS, SVN, Git, Mercurial, …
Replication Basics

- We have a set of objects we want to manage
  - Think of these as *logical* objects

- We maintain a set of instances of these objects stored on servers
  - These are the *physical* objects
  - Also call *replicas*

- A replication system allows *clients* to work on the logical objects by hiding the translation of the logical objects to the corresponding physical instances
  - Clients interact with *front-end* nodes
  - Front-ends talk with the servers managing the physical objects (*replica managers*)
Replicated Data Management Model

- Assume asynchronous system
  - Processes fail only by crashing, no network partition

- Replica managers
  - Processes that contain replicas (physical copies)
  - Perform operations on replica data directly
  - May be fixed (static) or join and leave (dynamic)
  - Think of operation implemented by a state machine
    - applies operation atomically
    - current state is a consequence of a deterministic function applied to an initial state and all past operations
    - all replicas start identical and carry out same operations
    - operations must not be affected by clock readings, …
Replicated Data Management Model (continued)

- **Clients**
  - Make requests on logical data items
  - See a service that provides access and operations
    - read-only and update

- **Front ends**
  - Handle client requests
  - Responsible for transparency
  - Communicates with replica managers (transparency)
    - direct interaction with one replica manager, which in turn interacts with other managers
    - multicast to a set of replica managers
  - Different FE’s communicate with differ RM’s
  - Interaction method is application dependent
Basic Architectural Model
Interactions and Coordination

- Interactions with the replica manager come in two flavors:
  - Read operations
    - A request is made for an object
  - Update operations
    - Provide a new version of an object to the replica manager
    - Requires the set of replica managers to coordinate ordering when multiple updates occur simultaneously

- The problem is in coordinating multiple updates

- Replica managers use different ordering disciplines:
  - FIFO: ordered determined as appear at front-end
  - Causal: happened-before ordering of front-end requests
  - Total: A before B at one replica manager implies A before B at all replica managers
Five Phases in Performing a Request

☐ *Issue request*
   - FE either sends to specific RM or multicasts to all

☐ *Coordination*
   - RM decides whether to apply request and order
     - FIFO, causal, total orderings

☐ *Execution*
   - RM executes the request

☐ *Agreement*
   - RMs agree on the effect of the request
     - lazy or immediate

☐ *Response* (one or more RMs) to FE
Execution and Agreement

- Replica managers execute requests from front-ends
- Tentative execution is possible if it is necessary to later undo the requests
  - If replica managers obey a transactional discipline, this is necessary
  - Why?
    (Remember: Two-phase commit protocol in distributed transaction requires tentative commits during first voting phase to store potential commit to nonvolatile store before second phase, where actual commit occurs if unanimous.)
- Agreement amongst replica managers is consensus amongst the set on the effect of a request
Replication and Group Communication

- Replication is an inherently group-based operation
  - Group is both client(s) and replica managers
- A key concern is membership in the group
  - Membership can change (participants come and go)
  - Failures may occur (participants may vanish)
  - Notification (participants may wish to be notified upon changes to replicated data)
    - think subscription model discussed a couple weeks ago
  - Address expansion (map logical group name to actual participant addresses)
- IP multicast does not quite cut it for this membership model mainly because of dynamic membership
We would like to have fault tolerant support in the replicated system.

Service correct even if $f$ processes fail:
- How? by replicating data and functionality at RMs
- Assume communication reliable and no partitions
- RMs are assumed to behave correctly 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
Bulletin Board Reader

- Bulletin board for message posting
  - Items are numbered with sender name and dates
  - Original messages (Vote) and responses (Re: Vote)

- Replicated bulletin boards
  - Reading
  - Updates
  - Posting and response numbering and ordering

- Asynchronous model
  - Client requests processed by local replica server
  - Possible inconsistencies can arise
Totally synchronous model
- All update requests are totally ordered
- Return to client after update processed at all servers
- Can deliver worse performance than single server

Replications schemes fall between the two models
- Need some replica consistency (how much?)
- Need good response time and throughput
- Trade-off between consistency, availability, response
- Two approaches
  - quorum-based
  - causality
Example of a Naive Replication System

<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></td>
<td>getBalance(_A)(y) →2</td>
</tr>
<tr>
<td></td>
<td>getBalance(_A)(x) →0</td>
</tr>
</tbody>
</table>

- 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
  - Different if A and B were implemented at a single server
- Anomalous behaviour

RMs at \(A\) and \(B\) maintain copies of \(x, y\)
- clients use local RM when available, otherwise the other one
- RMs propagate updates to one another after replying to client
Linearizability and Sequential Consistency

- Linearizability and sequential consistency are properties of a system.

- In both cases, we start with the constraint that:
  - An interleaved sequence of operations meets the specification of a (single) correct copy of the objects.
  - This means that the interleaved sequence yields an end result equivalent to only one process executing the sequence of operations.

- **Linearizability:** Order of operations in interleaving is consistent with *real times at which the operations occurred in actual execution*.

- **Sequential consistency:** Order of operations in interleaving is consistent with the *program order in which each individual client executed them*.
Linearizability

- Strictest criterion for a replication system … why?
- Consider a replicated service with two clients
  - Interleaved read and update operations
  - Each operation is specified by the operation type and the arguments and return values as occurred at run time
  - Every operation is synchronous (completes before next)
- Single server would serialize
- Correctness criteria for replicated objects are defined by referring to a virtual (correct) interleaving
  - Does not necessarily physically occur at any particular replica manager
  - But establishes the correctness of the execution
A replicated object service is linearizable if for any execution there is some interleaving of clients’ operations that satisfies the two following criteria:

- The interleaved sequence 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

Definition captures the idea that for any set of client operations there is a virtual canonical execution against a virtual single image of the shared objects

Each client sees a view that is consistent with this
Linearizability (continued)

- Linearizability concerns only the interleaving of individual operations and is not intended to be transactional.
- A linearizable execution may break application-specific notions of consistency if concurrent control is not applied.
- The real-time requirement is desirable because it captures our notion that clients should receive up-to-date information.
- The presence of real time in the definition raises the issue of practicality.
Sequential Consistency

- Weaker correctness condition (and more practical) since it relaxes the constraints a bit
- Try to capture an essential requirement concerning the order in which requests are processed without appealing to real time (but with respect to each client)
- A replicated shared object service is *sequentially consistent* if for any execution there is some interleaving of clients’ operations such that:
  - Interleaved sequence of operations meets the specification of a (single) correct copy of the objects
  - Order of operations in the interleaving is consistent with program order in which each client executed them
Sequential Consistency (continued)

- Absolute time does not appear in the definition
- Nor does any other total order on all operations
- The only order that is relevant is the order of events at each separate client – the program order
- The interleaving of operations can shuffle the sequence of operations from a set of clients in any order, as long as
  - Each client’s order is not violated
  - Result of each operation is consistent
- Every linearizable service is also sequentially consistent, but the converse does not hold
Sequential Consistency (continued)

☐ Linearizable? Sequentially consistent?

<table>
<thead>
<tr>
<th>Client 1:</th>
<th>Client 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{setBalance}_B(x,1)</td>
<td></td>
</tr>
<tr>
<td>\textit{setBalance}_A(y,2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\textit{getBalance}_A(y) \to 2</td>
</tr>
<tr>
<td></td>
<td>\textit{getBalance}_A(x) \to 0</td>
</tr>
</tbody>
</table>

☐ Linearizability and sequential consistency are types of \textit{consistency properties}

▪ Address issues of concurrency with respect to operations on shared objects

▪ There are weaker consistency properties used in distributed shared memory
Passive Model for Fault Tolerance

- Single primary RM and secondary (backup) RMs
- FEs communicate with the primary which executes the operation and sends copies of updated data to backups
- If the primary fails, one of the backups is promoted to act as the primary
Passive Replication - Five Request Phases

1. *Request*:
   - FE issues request w/unique identifier to primary RM

2. *Coordination*:
   - Primary performs each request atomically, in the order in which it receives it relative to other requests
   - Checks the unique id for duplicates (re-sends response)

3. *Execution*:
   - Primary executes the request and stores the response

4. *Agreement*:
   - If request is an update, the primary sends the updated state, the response, and the unique identifier to all the backups
   - Backups send an acknowledgement

5. *Response*:
   - Primary responds to FE, which hands response back to client
Discussion of Passive Replication

- This system implements linearizability, since the primary sequences all the operations on shared objects.
- If the primary fails, the system is linearizable, if a single backup takes over exactly where the primary left off:
  - The primary is replaced by a unique backup.
  - Surviving RMs agree which operations had been performed at take over.
- To survive \( f \) process crashes, \( f+1 \) RMs are required:
  - It cannot deal with byzantine failures because the client can't get replies from the backup RMs.
Active Replication for Fault Tolerance

- 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

- Replica managers for a group
  - FE multicasts each request to the group of RMs
- 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 Request Phases

1. Request
   - FE attaches a unique id and uses totally ordered reliable multicast to send request to RMs

2. Coordination
   - Multicast delivers requests to all RMs in same (total) order

3. Execution
   - Every RM executes the request with same effect
   - State machines and receive requests in the same order

4. Agreement
   - No agreement is required

5. Response
   - FEs collect responses from RMs
   - FE may just use one or more responses
   - If only trying to tolerate crash failures, gives first response
Discussion of Active Replication

- RMs are state machines - sequential consistency
  - Due to reliable totally ordered multicast, the RMs collectively do the same as a single copy would do
  - It works in a synchronous system
  - In an asynchronous system reliable totally ordered multicast is impossible
  - Failure detectors can be used to work around this problem

- This replication scheme is not linearizable
  - Total order is not necessarily the same as real-time order

- To deal with byzantine failures
  - For up to $f$ byzantine failures, use $2f+1$ RMs
  - FE collects $f+1$ identical responses

- To improve performance
  - FEs send read-only requests to just one RM
Summary

- Replicating objects helps services to provide good performance, high availability and fault tolerance

System model
- Each logical object is implemented by physical replicas

Linearizability and sequential consistency can be used as correctness criteria
- Sequential consistency is less strict but more practical

Fault tolerance can be provided by:
- Passive replication
  - using a primary RM and backups
- Active replication
  - all RMs process all requests identically
  - needs totally ordered and reliable multicast
  - can be achieved in a synchronous system
Distributed vs. Primary Updates

☑ How to apply replication techniques to make services highly available

☑ Clients and replica managers are separate processes

☑ Distributed update
  ☑ Replica managers exchange messages periodically to convey updates
  ☑ “Gossip” metaphor

☑ Primary update
  ☑ All front ends communicate with “primary” server
  ☑ Primary propagates updates to “slave” servers
  ☑ Read operations performed on any server
Different Architectures

- Design considerations
  - Separation of front ends and replica managers
  - Responsibility for update propagation
  - Choice of communication pattern

- Application requirements
  - Availability, consistency, response

- Consistency requirement
  - Lead to ordering constraints for request processing
Consistency and Request Ordering

- Order of requests processing at different replicas
  - Important for correctness
  - Ordering requirements impose performance costs
- Requests $r1$ and $r2$ are commutative if:
  - Order of processing does not matter
  - Use knowledge of commutativity to avoid ordering
- Ordering examples
  - Bulletin Boards
  - Multi-user document editing
Ordering

- Total ordering of requests $r1$ and $r2$
  - Either $r1$ processed before $r2$ at all replica managers
  - Or $r2$ processed before $r1$ at all replica managers
- Causal ordering of requests $r1$ and $r2$
  - Happened-before ordering
  - Potential causal relationship between two events
  - One event happened-before another if information flowed (message or action) between the two events
  - $r1$ happened-before $r2$, then $r1$ processed before $r2$ at all replica managers
- Total ordering is not necessarily also causal
Stronger Ordering

- *Sync-ordering* of requests *r1* (*sync*) and *r2*
  - Forces order of requests processed at replica managers to be “in sync”
  - Every other request is consistently processed before it or after it at all replica managers
  - Effectively flushes any outstanding requests issued but not yet processed everywhere
  - All later requests are processed after the sync
  - *r2* is processed either before or after *r1* at all replica managers
Implementing Request Ordering

- **Hold-back**
  - Receive requests are not processed by replica manager until ordering constraints can be met
  - Hold-back queue keeps requests until their order has been determined

- **Stable request message**
  - All prior messages (defined according to type of ordering) have been processed
  - Placed on process (delivery) queue
  - Addresses safety and liveness properties
Implementing Total Ordering

- Basic approach is to assign totally ordered IDs
- Two methods
  - Sequencer
    - all requests sent to sequencer and RM sites
    - sequencer assigns consecutive IDs
    - IDs are forwarded to RM sites
    - sequencer can become the bottleneck
  - Distributed agreement
    - RM sites store largest final and proposed IDs
Implementing Total Ordering (continued)

- 1. FE site send request with largest yet ID
- 2. RM sites generate proposal for final ID
- 3. FE site chooses largest and informs RM sites
- 4. RM sites attach ID to request and reorder hold-back queue

➤ expensive algorithm – three messages per request
Implementing Causal Ordering

☐ BB example
  ✓ No item that refers to another can be posted earlier
  ✓ Items should be posted in causal order

☐ Need suitable ordered identifiers
  ✓ Logical timestamps will not do
  ✓ Require timestamps (used as IDs) that carry more information to determine causal ordering

☐ BB example
  ✓ Timestamp represented by counts of the update events (direct or forwarded) that led to the current state
Vector Timestamps

☐ BB timestamps represented by list of counts of update events, one for each replica manager

☐ Maintenance of causal order
  ☐ Ensure each FE reads from a BB version at least as advanced as the version last read
  ☐ New item should be added to a replica only when the replica already reflects all causally prior updates
  ☐ RM’s maintain own timestamp of their BB version
  ☐ FE’s maintain vector timestamps reflecting latest BB version they have read
Vector Timestamp Formalism

- Each process $p_i$ has a vector timestamp $VT_i$
- Each $VT_i$ is a vector of integer values of length $n$
- $VT_i[k]$ represents a count of events that have occurred at $p_k$ and that are known at $p_i$
- Vector clock update algorithm
  1. $VT_i$ initialized to 0
  2. $VT_i[i]++$ when $p_i$ generates a new event
  3. request with timestamp $vt$ received at $p_j$
     $VT_j = \text{merge}(VT_j, vt)$, where
     $\text{merge}(u, v)[k] = \max(u[k], v[k]), k=1..n$
- Implements happened-before
The Gossip Architecture

- Framework for implementing highly available services by replicating data close to points where groups of clients need it
- “Gossip” messages used to convey updates
- System makes two guarantees
  - Each client obtains a consistent service over time
  - Relaxed consistency between replicas
- Architecture based on earlier work on databases
The Gossip Architecture

- Clients requests operations processed by FEs
- FE typically communicates with one RM
- RMs update one another by gossip messages
  - Contain most recent updates received
  - Lazy updates
- Three strengths of update ordering
  - Causal (e.g., posting)
  - Forced (total and causal) (e.g., add user)
  - Immediate (sync-ordered) (e.g., delete user)
- Specified for each type of update operations
Query and Update Operations in Gossip Service

(prev, new) are vector timestamps

Update id is a vector timestamp
Operations and Ordering in a Gossip Service

- Two basic types of operations
  - *Queries*: read-only operations
  - *Updates*: modify-only operations

- Clients are blocked on queries

- Clients continue after updates pass to FE

- Each FE keeps vector timestamp *prev*
  - Reflects latest values accessed by FE (client)
  - *prev* sent in every request message to RM

- New timestamp returned as a result of query

- *Update id* vector timestamp returned with update
Returned timestamps are merged with FE previous to record version of replicated data seen by client

Clients exchange information in two ways:

- Accessing replicated services
  - happened-before relationships between operations maintained
- Direct communication
  - occurs via FEs who piggy-back vector timestamps on client messages
  - communication can lead to causal ordering
Front Ends Propagate Their Timestamps
Replica Manager State

- **Replica manager contains main state components:**
  - Value of application state resulting from updates
  - Value timestamp representing updates applied
  - Update log recording all updates received
    - RM holds back update requests until stable
    - Hold updates until confirmed from all RMs
  - Replica timestamp represents logged updates
  - Identifiers of executed calls kept in list to prevent update from being performed twice
    - List checked before executing a stable update
Replica Manager State (continued)

☐ Vector timestamp at RM $i$
  ☐ $i$th element corresponds to updates from FEs received by $i$
  ☐ $j$th element equals number of updates received by $j$ and propagated to $i$ in gossip messages
Gossip Replica Manager State Components

- **Replica timestamp**
- **Update log**
- **Value timestamp**
- **Stable updates**
- **Executed operation table**
- **OperationID**
- **Update**
- **Prev**

Other replica managers

Gossip messages

Replica manager
Processing Query Operations

- Query request \( q \) contains timestamp \( q.prev \)
- Replica manager should return a value at least as recent as this
  - If \( valueTS \) is the replica’s value timestamp, then \( q \) can be applied to the replica’s value if:
    \[ q.prev \leq valueTS \]
  - RM keeps \( q \) on list of pending queries until this condition is fulfilled
  - Once the query can be applied, value \( TS \) is returned to the front end as \( new \)
    \[ frontEndTS = merge(frontEndTS, new) \]
- Suppose \( valueTS \) is \( (2,5,5) \) and \( q.prev \) is \( (2,4,6) \)
Processing Updates in Causal Order

- On receipt of update request $u$, RM checks to see if it processed this request before (if so, discard)
  - If not, increments own element in its replica timestamp to keep count of number of updates received from FE
- $u$ is assigned a unique vector timestamp, $TS$, and logged, $logRecord = <TS, u.op, u.prev>$
- $TS$ is derived from $u.prev$ by replacing its $i$th element by that of the replica timestamp for RM $i$
- $TS$ is passed back to the front end which merges it
- Stability condition: $u.prev \leq valueTS$
  - All the updates on which this update depends have already been applied (same as for query)
When the stability condition is met for an update record $r$, the RM checks to see whether the call identifier $r.cid$ appears in the list of executed calls.

Applying the update record $r$

\[
\begin{align*}
\text{value} &= \text{apply}(\text{value}, r.op) \\
\text{valueTS} &= \text{merge}(\text{valueTS}, r.ts) \\
\text{executed} &= \text{executed} \cup \{r.cid\}
\end{align*}
\]
Gossip Messages

- Sent by RMs to help bring other RMs up to date
- Gossip message $m$ consists of two items:
  - Its log $m.\log$
  - Its replica timestamp $m.ts$
- Receipt of $m$ by a replica manager (main tasks)
  1. Merge arriving log with own
     - check timestamps in log (may contain updates not seen)
  2. Apply any updates that have become stable
  3. Eliminate records from executed call log when it is known that updates have been applied everywhere
  4. Merge timestamp of gossip message with $replicaTS$
Gossip Messages – Merging Logs

- Straightforward

- Let \( replicaTS \) denote the recipient’s replica timestamp

- A record \( r \) in \( m.log \) is added to receiver’s log unless \( r.ts \leq replicaTS \)
  
  In this case it is already in the log or has been applied to the value and then discarded

- Replica manager merges the timestamp of the incoming gossip message with its own replica timestamp
  
  \[ replicaTS := \text{merge}(replicaTS, m.ts) \]

  Updates are sorted according to partial order between vector timestamps
Processing Forced and Immediate Updates

☐ Unique sequence numbers are appended to timestamps for forced updates
   ♦ Obtained from primary RM sequencer

☐ Immediate updates are sync-ordered
   ♦ Easily ordered with respect to forced updates
   ♦ Primary determines which causal updates precede via agreement with other RMs
Discussion

- Gossip architecture aimed at high availability
- Rate of flow of gossip messages unspecified
  - Tuning parameter
- Requests sent to one or more RM\textsubscript{s}
  - Parameter depending on load and failure conditions
  - Tradeoff between latency and bandwidth
- Increase in replica managers results in:
  - Increase size of timestamps and \# gossip messages
- Make more RM\textsubscript{s} read-only to aid in scalability