

# Distributed Software Development

## Replication

Chris Brooks

Department of Computer Science  
University of San Francisco

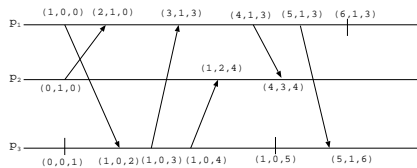
### 8-2: Previously on cs682: Causal delivery

- *causal delivery* says that if  $send(m_1) \rightarrow send(m_2)$ , then  $deliver(m_1) \rightarrow (m_2)$  when different processes are sending  $m_1$  and  $m_2$ .
- Logical clocks aren't enough to give us causal delivery.

### 8-3: Previously on cs682: Vector clocks

- Solution: keep a "logical clock" for each process.
- these are stored in a vector  $VC$ .
  - Assumes number of processes is known and fixed.
- Update rule:
  - $VC(e)[i] = VC[i] + 1$  for send and internal.
  - $VC(e) = \max(VC, TS(m))$  for receive; then  $VC(e)[i] = VC[i] + 1$
- On receive, the vector clock takes the max on a component-by-component basis, then updates the local clock.

### 8-4: Previously on cs682: Vector Clock example



### 8-5: Previously on cs682: Consensus

- If we don't have reliable communication, consensus is impossible, even without failures.
- With reliable communication, we can solve consensus for crash failures.
- In asynchronous systems, it is impossible to guarantee that we will reach consensus, even in the presence of a single crash failure.
- This means that we can't do:
  - Asynchronous Byzantine generals
  - Asynchronous totally ordered multicast

### 8-6: Replication

- Replication is the maintenance of copies of data at multiple computers.
- Enhances a service by providing:
  - Fault tolerance
  - Improved performance
  - Increased availability
  - Information redundancy

### 8-7: Why is replication useful?

- Increased performance.
  - By moving data closer to a client, latency is reduced.
  - Web caching, proxy servers are an example of this.
- Performance is improved most effectively with immutable data.
  - If the client is going to change the data and send it back, performance gains are reduced.

### 8-8: Why is replication useful?

- Increased availability.
- Many services need to be highly available
- Replication provides a way of overcoming server failures.
- If a server will fail with probability  $p$ , then we can determine how many servers are needed to provide a given level of service:
  - $Avail = 1 - p^n$
- For example, if a server has a 5% chance of failure (i.i.d) over a given time period, and we want 99.9% availability, we need at least 4 replicas.

### 8-9: Client-side replication

- Note that replication is not limited to servers.
- Multiple clients may need to replicate data.
  - Shared code or documents being edited.
  - Meeting scheduling
  - Conferencing or whiteboard software.

### 8-10: Fault tolerance

- Highly available data may not be correct data.
  - For example, in the presence of network outages.
- Fault tolerance guarantees correct behavior in the presence of a given number of faults.
- Similar to availability, but a coordination element is also required.
- We may also want to ensure against corruption of data.

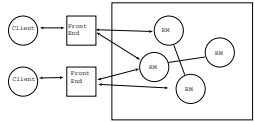
### 8-11: Outline

- Passive Replication
  - What problems must be solved for this?
- Active Replication
  - What problems must be solved for this?
- Lazy Replication
  - What problems must be solved for this?

### 8-12: Client Requirements

- Single logical copy.
- Multiple physical copies.
- Consistency
  - The details of this will depend on the application.

### 8-13: System Model



- Clients interact with replica managers via a front end.
- Front end hides replication from the client.
- Front end may interact with all replica managers, or just a subset.
- Replica managers interact to ensure consistency.

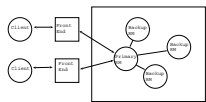
### 8-14: Fault tolerance example

- Consider this example:
  - We have a bank with two replicated servers: a user's front end may connect to either one.
  - First, a user updates account A via server 1 to contain \$10.
  - Next, the user transfers \$5 from A to account B via server 2.
  - Server 1 fails before data is propagated.
  - The bank manager logs into server 2 and sees that account A has a balance of 0 and account B has a balance of \$5.
- Problem: even though the transfer happened after the deposit, it is the only operation seen.

### 8-15: Correctness

- This example violates *sequential consistency*
- Sequential consistency says that:
  - Interleaving operations across servers produces a sequence that is consistent with a correct copy of the objects.
  - The global order is consistent with the local order in which each client executed its operations.
- This is very much like our definition of causality.

### 8-16: Passive Replication



- Passive replication uses a single replica manager.
- Other replica managers act as backups (or slaves).
- Primary manager executes operations and sends copies of updated data to backups
- If primary fails, a new one is elected.

### 8-17: Passive Replication

- Sequence of events:
  1. Front end issues a request to primary
  2. Primary takes requests in FIFO order. Checks to see if request has already been serviced. If so, re-send response.
  3. Execute request and store response.
  4. If data is updated, send updates to all backups. All backups acknowledge.
  5. Response is sent to client.

### 8-18: Passive Replication

- A question:
  - How do we communicate updates to all backups?
  - Recall that we want to be able to tolerate primary crashes before, during, and after updating.

### 8-19: Group Communication

- Updating replicas is a *group communication* problem.
- If groups can change dynamically, then a group membership service is needed.
- This keeps track of the processes that are currently in a group.

### 8-20: Group membership

- A group membership process needs to:
  - Provide an interface for membership changes, creation, and destruction.
  - Provide a failure detector
  - Notify members of membership changes.
  - Perform group address expansion. When a message is sent via multicast, it is sent to the group address. The membership process must then deliver that message to all processes.

### 8-21: View-synchronous communication

- View-synchronous communication is an extension of reliable multicast.
- It uses a *view*, which is a list of all processes currently in the group.
- When membership changes, a new view is sent to all members.
- All messages that originate before a new view must be delivered before that new view is delivered.
  - Provides a cut in the message timeline.
  - Can also be thought of as providing a state transition.

### 8-22: View-synchronous communication

- View-synchronous communication guarantees that if one process delivers a message within the context of a view, all processes deliver that message with that view's context.
- Change of view is treated as another timestamped message.
- Allows us to break the timeline into states, or sequences of views.
- All processes agree on what has happened in that view.

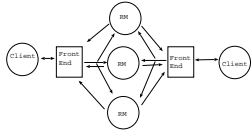
### 8-23: Comments on passive replication

- $n + 1$  replicas are needed to tolerate  $n$  crash failures.
- Very easy from the front end's point of view; all that is needed is to communicate with a single server, and possibly change servers if the primary fails.
- Problem: view-synchronous communication requires significant overhead.
  - This can lead to latency in transmitting data back to the client.

### 8-24: Failures

- Passive replication deals nicely with crash failures.
- If a backup crashes, there's no problem.
- If the primary crashes, it is replaced by a new primary. View-synchronous group communication allows the replicas to agree as to what operations have been performed.
- The new primary picks up at that point.

### 8-25: Active Replication



- In active replication, all replica managers are equal
- Front ends multicast requests to all replica managers.
- All managers process request independently
- If a replica crashes, others continue as normal.

### 8-26: Active Replication

- Sequence of events:
  1. Front end attaches a unique identifier to request and multicasts it to all RMs using totally ordered reliable multicast.
  2. Group communication system delivers messages to all RMs in the same order.
  3. Each replica manager evaluates the request.
  4. Since totally ordered multicast is used for communication, no agreement phase is needed.
  5. Each replica returns its response to the front end.

### 8-27: Active Replication

- When can the front end return a response to the client?
- Depends on what sorts of failures you want to tolerate.
  - Crash failures: can return the first response.
  - Byzantine failures: Must collect  $\frac{2}{3} + 1$  responses.

### 8-28: Active replication

- Notice that active replication assumes totally ordered reliable multicast.
- Recall that this is equivalent to the consensus problem.
- We need either:
  - A synchronous system
  - Failure detectors
- As with passive replication, totally ordered multicast may cause latencies.

### 8-29: Failures

- Failure is totally transparent here.
- Nothing new is needed if a replica manager crashes.
- Too many Byzantine failures can cause the usual problems.

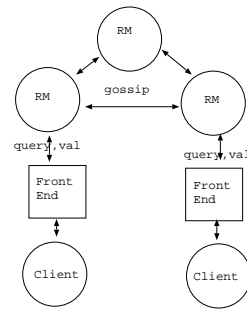
### 8-30: Lazy Replication

- Active and passive replication provide fault tolerance and sequential consistency.
- They do this by immediately synchronizing before returning data to the client.
- This is great for fault tolerance, but not for availability.
  - Some applications may not tolerate latency.
- We can trade latency for sequential consistency to get a highly available service.

### 8-31: Lazy Replication

- Lazy replication is also known as *gossip* replication.
- Takes advantage of the fact that, for some applications, it's OK for replicas to be slightly out-of-sync, as long as they eventually catch up.
  - Shared bulletin board, chat room.
- Messages must be posted to replicas in causal order.

### 8-32: Lazy Replication



- Clients can issue either queries (read-only) or updates (write-only).
- System guarantees that each client receives consistent service.
  - A client should always receive data that reflects the updates it's seen.
- All replicas eventually receive all updates, but a client may observe stale data.

### 8-33: Lazy Replication

- Order of operations:
  1. Front end sends a request to a replica manager.
  2. If the request is an update,
    - (a) replica manager returns a response immediately.
    - (b) The replica then coordinates with other managers.
    - (c) The replica then executes the request.
  3. else, if the request is a query,
    - (a) The replica coordinates enough to update its state.
    - (b) It then executes the request and returns the response to the front end.

### 8-34: Coordinating with lazy replication

- Coordination is done by means of vector clocks.
- A front end keeps a vector clock with one entry for each RM.
- This clock is included in every request.
- The replica manager returns an updated timestamp with the response.
- If clients communicate directly with each other, they must also have entries in the timestamp.

### 8-35: Querying with lazy replication

- Consider a query operation sent to a replica manager.
- The replica must return a value at least as recent as the query's timestamp.
- If  $q.timestamp \leq rm.timestamp$ , then the replica manager can process  $q$  and return it to the front end. (this condition is called *stability*)
- The replica manager also returns its timestamp to the front end.
- If  $q.timestamp \not\leq rm.timestamp$ , the replica manager queues the query until it has received enough gossip messages to sufficiently update its state.

### 8-36: Updating with lazy replication

- An update is also sent to a replica manager along with a timestamp and a unique identifier.
- Upon receipt, if this update has not already been processed, it increments its timestamp and logs the update.
  - This log contains the query's timestamp, except that the RM's value is replaced with the updated one from its timestamp.
- This timestamp is returned to the front end.

#### 8-37: Updating with lazy replication

- If the query's timestamp precedes the replica manager's, the update can proceed immediately.
- Otherwise, the update is queued until the causally-previous messages arrive.
- Once this happens, the replica manager logs the update and updates its timestamp.

#### 8-38: Gossiping

- Replicas synchronize with each other through *gossip messages*.
- A gossip message contains a log of past updates and a timestamp.
- When a gossip message is received, the receiver must:
  - Merge the log with its own. (the log may reflect other updates it hasn't seen)
  - Apply stable updates
  - Remove entries from the log that are known to have been applied everywhere.

#### 8-39: Gossiping

- Update frequency (how often gossip messages are sent) is application-dependent.
  - Bulletin board: minutes, or less.
  - Inventorying systems: possibly less frequently.
  - May also depend on network availability. (for example, meeting schedulers on PDAs)
- Who updates are sent to can also be tuned to fit the application
  - Can choose randomly
  - Can choose the replica who is 'farthest behind'
  - Can use a fixed topology.
- Trade off amount of communication against latency and effects of failure.

#### 8-40: Comments on lazy replication

- Provides high availability; clients usually receive an immediate response.
- Works even when the network is partitioned.
- Not appropriate for applications where replicas must be tightly synchronized, such as video conferencing.
- Scalability is an issue, due to number of messages and size of timestamp.

#### 8-41: Failures

- The gossip architecture can also deal nicely with failure.
- Front ends can connect to more than one RM.
- It's possible for a transaction to be lost if it is submitted to an RM who immediately crashes before sending it on to any other RM.

#### 8-42: Summary

- Replication is the maintenance of copies of data at multiple computers.
  - Provides fault tolerance, increased performance, high availability.
- Passive replication: a single RM interacts with clients, and synchronizes with slaves.
- Active replication: All RMs communicate with clients and synchronize with totally ordered multicast.
- Lazy replication: RMs immediately return responses to clients and synchronize later.