Principles of Reliable Distributed Systems

Lecture 2: Fault-Tolerant Broadcasts

Idit Keidar

Today’s Agenda

• Motivation: Replication
  – Primary-backup
  – State machine
• Models of distributed systems
  – Failure, timing models
• Broadcast service implementations
  – Reliable
  – FIFO
  – Causal – LTS, Vector Clock
  – Atomic – LTS
• Multicast, group membership

Today’s Material

• Distributed Systems 2nd edition, Sape Mullender (Editor)
  – Causal order, Ch. 4
  – Fault-tolerant broadcasts, Ch. 5
  – Vector clocks, Ch. 4
  – State machine replication, from Ch. 7
• Attiya-Welch
  – Vector clocks, Ch. 6

Broadcast

• Sending a message to all the nodes in a system
  – E.g., our course mailing list
• What’s it good for?
• Allows for redundancy
  – In storage, processing
• Broadcast Service - building block (abstraction) for replication

Motivation: Replication

2 Paradigms:
• Primary-Backup – Passive
• State Machine Replication (SMR) – Active

Availability 101

• Overcome independent failure with redundancy
• Spatial redundancy: multiple servers for the same service
  – Failing independently
  – Degree of replication defines availability
• Temporal redundancy: repeat operations
Primary-Backup Replication aka Passive

- **“Hot” standby**
- Client talks to primary server
- Primary updates backup(s)
  - Using timeout
  - Performs failover to backup server
- On failover, client may need to repeat last operation(s)
  - Why?

Primary-Backup Replication

- Clients detect primary failure
- Only backups detect primary failure, clients are oblivious to failure
  a) Directed to current primary using dispatcher; or
  b) Using IP takeover

Primary-Backup Variants

1. Clients detect primary failure
2. Only backups detect primary failure, clients are oblivious to failure
   a) Directed to current primary using dispatcher; or
   b) Using IP takeover

Primary-Backup Example

- Picture from IBM web site
- Highly available Web Server Cluster
- Primary-backup with network dispatcher

Primary-Backup Limitations

- Does not work with network partitions, or “false suspicions”
- Works with benign (correct) servers only
  - No resilience to bugs, intrusions

State Machine Replication (SMR) aka Active

- Replicas are identical deterministic finite state machines
- Process operations in the same order → remain consistent
SMR Architecture: Option I

Client A

Client B

broadcast

SMR Architecture: Option II

Client A

Client B

broadcast

State Machine Replication Summary

• Send updates to all servers
• All servers are identical deterministic state machines
  – Servers begin in the same initial state
  – Perform operations in the same order to remain consistent
• May be slower than primary backup
  – but fail-over is quicker, smoother
• Can overcome network partitions and false suspicions
• Can tolerate malicious servers

Replica Coordination Requirements

• What does replication need from its broadcast service?
  • Agreement: all correct replicas receive all client requests
  • Order: replicas process requests in the same order

We’ll start with Agreement

Before We Begin …

• Define the model where we want to solve the problem
• Define the service interface
• Specify the services
  – Using properties

Model: Correct vs. Faulty Processes

• Look at a complete run
  – External observer’s view
• A process that does not fail in a run is correct in that run
• Otherwise, the process is faulty in the run
  – A process that fails any time in the run is faulty throughout the entire run
• Today we will consider only crash failures
  – A failed process permanently crashes
  – Later in the course: crash-recovery, Byzantine failures
Threshold Failure Model

- \( t \) out of \( n \) processes may fail
- \( t \) is usually given as a function of \( n \), e.g.,
  - \( t < n \)
  - \( 2t < n \)
  - \( 3t < n \)

Model: Synchronous vs. Asynchronous

- **Synchrony:**
  - Bounded latency, clock drift, processing time
  - Process crash can be accurately detected
- **Asynchrony:** non-assumption
  - Process crash failures cannot be accurately detected
- **Failstop**
  - Time-free, but crash failures accurately detected
- **Unreliable failure detectors** – later in the course

Service Interface: Separating Reception from Delivery

<table>
<thead>
<tr>
<th>Application (e.g., state machine)</th>
</tr>
</thead>
<tbody>
<tr>
<td>broadcast</td>
</tr>
</tbody>
</table>

Broadcast Algorithm: delivery waits for messages that should be acted on first

<table>
<thead>
<tr>
<th>send</th>
<th>receive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network</td>
<td></td>
</tr>
</tbody>
</table>

Broadcast Service: Distributed View

- Primitives: \( broadcast(m) \), \( deliver(m) \).
  - For simplicity, assume \( m \) is unique.

<table>
<thead>
<tr>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>broadcast</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>deliver</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Broadcast Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>send</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Broadcast Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>receive</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>send</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>receive</td>
</tr>
</tbody>
</table>

Broadcast Services

- Reliable Broadcast
- FIFO Broadcast
- Causal Broadcast
- Atomic Broadcast

Reliable Broadcast (RBCast) Specification

- **Validity:** if a correct process broadcasts \( m \) then all correct processes eventually deliver \( m \)
- **Agreement:** if a correct process delivers \( m \) then all correct processes eventually deliver \( m \)
  - Uniform agreement: if any process delivers \( m \) then all correct processes eventually deliver \( m \)
- **Integrity:** \( m \) is delivered by a correct process at most once, and only if it was previously broadcast
Model for RBCast Implementation

- Asynchronous
- Process crash failures
  - Note: cannot be detected
- Pair-wise reliable links between correct processes
  - If a correct process p sends a message m to a correct process q, then q receives m
    - Safety or liveness?
    - Message m is received at most once, and only if it was previously sent
    - Safety or liveness?
  - Typically implemented using timeouts/retransmissions

RBCast Interface

- Primitives: broadcast(m), deliver(m)

<table>
<thead>
<tr>
<th>Application</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>broadcast</td>
<td>broadcast</td>
</tr>
<tr>
<td>deliver</td>
<td>deliver</td>
</tr>
</tbody>
</table>

Network – Reliable Links between Correct Processes

RBCast Algorithm

- Simple algorithm
  - On the board
- Proof – Validity, Agreement, Integrity
  - On the board
- Complexity?
- Does it solve Uniform Reliable Broadcast?
- What if there are some link failures?
  - Under what condition does the protocol still work?

Replica Coordination Requirements

✓ Agreement: all correct replicas receive all client requests
- Order: replicas process requests in the same order

Possible Reception Orders in Point-to-Point Communication

Process p₁: send “1” receive message send “2” receive message
Process p₂: send “a” receive message send “b” receive message

Space-time diagram

P₁ 1 2
P₂ a b

Probably should not deliver yet

Possible Reception Orders in Point-to-Point Communication

Process p₁: send “1” receive message send “2” receive message
Process p₂: send “a” receive message send “b” receive message

Space-time diagram

P₁ 1 2
P₂ a b
FIFO Order on Broadcast

- If some process broadcasts message $m$ before message $m'$, then every correct process that delivers $m'$ delivers $m$ beforehand.
- Trivial to implement (How?)

FIFO Broadcast

- Reliable Broadcast + FIFO
- Simple implementation on top of reliable broadcast (using sequence numbers)

What Delivery Orders Make Sense?

![Diagram showing Alice, Bob, and Carol with messages being broadcast and delivered](image)

Happened Before or Causal Order on Events [Lamport 78]

- Event $e$ happens before (causally precedes) $f$, denoted $e \rightarrow f$, if:
  1. The same process executes $e$ before $f$; or
  2. $e$ is send/bcast($m$) and $f$ is receive/deliver($m$) (with whatever communication layer is used); or
  3. Exists $h$ so that $e \rightarrow h$ and $h \rightarrow f$
- Two events are concurrent, denoted $e \parallel f$, if:
  $$(e \rightarrow f \lor f \rightarrow e)$$

Cause and Effect

- There is no common clock to order events of different processes
- Instead, we use the (weaker) notion of cause
- If event $e$ caused another event $f$ to happen, then
  - $e$ happened before $f$
- By observing program behavior
  - We cannot tell whether one event causes another
  - We can tell whether it could have caused another

Causal Order on Broadcast Messages

- In a broadcast service, application-level causality is between broadcast and deliver
- For two messages, $m$ and $m'$, $m \rightarrow m'$ if bcast($m$) causally precedes bcast($m'$)
  - Causal order: transitive closure of FIFO + some process delivers $m$ before broadcasting $m'$
- Causal Delivery: If $m \rightarrow m'$ then every correct process that delivers $m'$ delivers $m$ beforehand
Does FIFO Between all Processes Guarantee Causal Order?

Causal Broadcast
- Reliable Broadcast + Causal
- Implementation on top of Reliable Broadcast

If we Could Use Clocks ….
- If processes had access to a global real-time clock RC, we could …
  - Tag every message m with a timestamp TS(m) = RC
  - Deliver messages in increasing TS order
  - If two messages have the same TS, break ties by process id (deliver the one with the lower id first)
- Clock Condition: if e → f then RC(e) < RC(f)
  - Hence, if m → m’ then TS(m) < TS(m’)
- Something missing?

When Do We Deliver a Message?
- Assume message delays are bounded by D
- Delivery Rule DR1:
  At time t,
  deliver all received messages with TSs up to t – D,
  in increasing timestamp order
- Now, what do we do without global clocks?

Lamport’s Timestamps – LTS
- Invent a logical clock that satisfies the clock condition
  - LTS (Lamport timestamp/Logical timestamp)
  - Positive-integer variable
  - Each process maintains its own local logical clock
- Tag every message m with a timestamp TS(m) equal to the sender’s LTS

LTS Broadcast Algorithm
Part I: Logical Clock Assignment
- TS_{i}[j]—0, \forall j = 0,\ldots,n /* TS_{i}[i] holds p_i’s local LTS */
- pending ← empty set
- broadcast (m)
  TS_{i}[i] ← TS_{i}[i] + 1 /* p_i’s LTS respects FIFO */
  send (m, (TS_{i}[i], i)) to all other processes
  add (m, (TS_{i}[i], i)) to pending
- upon receive (m, (t, j))
  TS_{j}[j] ← t
  add (m, (t, j)) to pending
  TS_{i}[i] ← max (TS_{i}[i], t+1) /* p_i’s LTS respects causality */
Example: Logical Clocks

![Diagram of Logical Clocks]

Does This Solve The Problem?

- When do we deliver a message?
- Deliver a message with $TS = t$ when no message with $TS < t$ can be received
- Formally:
  - A message $m$ received by $p_i$ is stable at $p_i$ if no future messages with $TS < TS(m)$ can be received by $p_i$
  - Delivery Rule DR2: Deliver all received messages that are stable at $p_i$ in increasing timestamp order

Detecting Stability

- Assume FIFO communication
- If $p_i$ receives $m$ from $p_j$ with $TS(m)$, $p_i$ cannot later receive a message $m'$ from $p_j$ with $TS(m') < TS(m)$
- Stability of $m$ at $p_i$ is guaranteed when
  - $p_i$ has received a message with timestamp greater than $TS(m)$ from all processes
- Note: the timestamp is a pair $(LTS, pid)$

LTS Broadcast Algorithm

Part II: Delivery Rule

```
let (m, (t, j)) be the entry in pending with the smallest (t, j)
if (t, j) ≤ (TS[k], k) ∀k = 0, …, n then /* DR2 */
   deliver (m)
   remove (m, (t, j)) from pending
```

The LTS Broadcast Algorithm Satisfies Causal Order

- Causal Delivery: If $m → m'$ then every correct process that delivers $m'$ delivers $m$ beforehand
- Timestamps respect the Clock Condition:
  - If $m → m'$ then $TS(m) < TS(m')$
  - DR2 + FIFO links ensure that if $m$ is in pending, all messages with lower $TS$ were received
  - Were either delivered or are pending
- Delivery from pending is by $TS$ order

Does the LTS Algorithm Implement Causal Broadcast?

- Causal order – see previous slide
- Anything else?
  - Reliable broadcast
  - When is it guaranteed?
Fault-Tolerant Asynchronous Causal Broadcast

Vector Clocks

- Process $p_i$ has a vector clock $VC_i[1...n]$
  - $VC_i[i]$ is the local message sequence number of the last message broadcast by $p_i$
  - For $j \neq i$, $VC_i[j]$ is the latest message $p_i$ delivered from $p_j$
- $VC_i$ is sent with each message $m$
  - For $j \neq i$, $m.VC[j]$ is $p_j$’s latest message that causally precedes $m$

Vector Clocks: Sending

- At process $p_i$, on broadcast($m$)
  - $VC_i[i] \leftarrow VC_i[i] + 1$
  - Use Reliable Broadcast to send ($m, VC_i$) to all
    - No need to send to myself
  - Deliver $m$ locally

Vector Clocks: Delivery Rule

- Upon receive $m$
  - Place in message buffer
  - Deliver $m$ from $p_j$ from buffer if
    - $VC_i[j] = m.VC[j] - 1$ and
    - For all $k \neq j$, $VC_i[k] \geq m.VC[k]$
- Upon deliver
  - $VC_i[j] := VC_i[j] + 1$

Vector Clocks Example

Correctness

- Need to prove
  - Causal order
  - Reliable broadcast
And Now, to Total Order

- If two correct processes deliver both $m$ and $m'$, they deliver them in the same order

Atomic Broadcast Services

- Atomic Broadcast:
  - Reliable Broadcast + Total Order
- FIFO Atomic Broadcast
  - FIFO + Reliable Broadcast + Total Order
- Causal Atomic Broadcast
  - Causal + Reliable Broadcast + Total Order

HW question: Are FIFO Atomic Broadcast and Causal Atomic Broadcasts equivalent?

Causal Atomic Broadcast Without Failures?

- LTS algorithm ensures total order

Causal Atomic Broadcast in Failstop Model

- Failstop = failures are accurately detected
  - No message from a faulty process arrives after its failure is detected
- Order messages by logical timestamp (LTS)
  - Recall: LTS breaks ties by process id
- Use FIFO Broadcast
- HW question: Write DR3. When is a message with LTS $t$ delivered?

Atomic Broadcast in Asynchronous Systems?

Alas, impossible if even one process can crash

Follows from reduction to consensus + [FLP 85]

Now, Back to State Machines

- We can build state machines using Atomic Broadcast
- A client can
  - Broadcast to all servers; or
  - Forward its request to one of the servers to broadcast to the others
    - Resend on timeout if the server fails
- What about client failures?
Multicast

- Processes organized in groups
- Groups have names
- Messages are sent to groups
- Like broadcast, but for group members
- Processes can \textit{join} groups
- Process can fail or voluntarily \textit{leave} groups

Group Membership

- Processes may want to know who else is a member of the group
- Want to be updated when nodes leave, fail
- Processes are notified of \textit{group membership} changes, also called \textit{views}
- Each message is delivered in a view –
  - The latest view the process was notified of before the message delivery

Virtual Synchrony

- Aka View Synchrony
- Processes moving together from view \( v_1 \) to view \( v_2 \)
  receive the same set of messages in \( v_1 \)

Summary

- Fault tolerance is achieved via replication
- To keep replicas consistent, we need fault-tolerant broadcast services with ordering guarantees
- We saw how to implement
  1. Fault-tolerant causal broadcast in asynchronous model
  2. Atomic causal broadcast in fail-stop model
- Fault-tolerant atomic broadcast is impossible in asynchronous systems
- Multicasting is broadcast for groups
  - Virtual synchrony: order on group membership events