CS505: Distributed Systems

Lecture 6: Reliable Broadcasts
Overview

- From Best-Effort to Reliable Broadcast
- Ordering Guarantees
Network provides one-to-one communication primitives
- Sometimes one-to-many also
  - Membership opaque
  - Fuzzy guarantees

Need one-to-many communication primitives
- E.g., replication, peer discovery
- With reliability guarantees!
  - And possibly ordering
Example: Reliable Broadcast

Informally
- A process $p_1$ wants to send a message $m$ to several processes $p_2 \ldots p_n$

Scenario
- Chat, mailing list, ...

Assume
- Asynchronous system
- Reliable channels
- Crash-stop failures of processes
p1

broadcast

m

m

deliver

p2

deliver

p3
Layers

“Application”

“Broadcast”

“Channel”

broadcast deliver

send receive
Best-effort Broadcast

Generalized
- Any process in a group can broadcast
- Primitive broadcast (and deliver) behave as follows

I. No duplication
- No message is delivered more than once

II. No creation
- No message is delivered unless it was broadcast

III. Validity
- If \( p_i \) and \( p_j \) are correct, then every message broadcast by \( p_i \) is eventually delivered by \( p_j \)
Implementation

- Simple algorithm
  - broadcaster sends \( m \) to every process including itself
  - Every process which receives \( m \) delivers \( m \)

- What if
  - \( p_1 \) sends to \( p_2 \), and then crashes, or shuts off?

- E.g., “hand in solutions for assignment x by …”
Reliable Broadcast

I. No duplication
   - No message is delivered more than once

II. No creation
    - No message is delivered unless it was broadcast

III. Validity
    - If $p_i$ and $p_j$ are correct, then every message broadcast by $p_i$
      is eventually delivered by $p_j$

IV. Agreement
    - If one correct process delivers a message $m$, every correct
      process eventually delivers $m$
Simple algorithm

- $p$ sends $m$ to every other process including itself
- Every process which receives $m$ for the first time sends it to every other process (except the sender) and delivers it
crash

deliver

deliver
I. and II. by corresponding properties of reliable channels

By first line of algo, broadcasting process $p_i$ sends to every other process; if it is correct, by Validity of reliable channels every correct process eventually receives the message

By second line of algo, every correct process $p_j$ which receives the message eventually delivers it

If some correct process delivers the message, it sends it (before) to all processes, and the correct ones eventually receive and deliver it
Fault Tolerance and Complexity

- How many faults can be tolerated?

- What is the complexity of the algorithm?
  - Messages
  - “Communication steps”
But

▶ What if a process delivers and then crashes?
  - sends possibly haven’t completed
  - Not correct?
  - It can initiate next action before crashing that affects entire system

▶ (General problem)
  - No timing assumptions
  - Algorithm runs terminate eventually only
  - Correctness defined with respect to algorithm runs
Uniform Reliable Broadcast

I. No duplication
   - No message is delivered more than once

II. No creation
   - No message is delivered unless it was broadcast

III. Validity
   - If $p_i$ and $p_j$ are correct, then every message broadcast by $p_i$ is eventually delivered by $p_j$

V. Uniform agreement
   - If a (correct or not) process delivers a message $m$, every correct process delivers $m$
A Simple Algorithm?

- Avoid that some process delivers and subsequently crashes
  - Can not keep processes from crashing, can not foresee crashes
  - However, can make sure everybody receives message before even thinking about delivering
  - Processes need acknowledgements from every process before delivering
    - Every process?

- Uniform Reliable Broadcast
  - Not implementable in asynchronous system
  - Need a failure detector
Uniform Properties

Properties which range over all involved processes
- As opposed to only correct ones

Intuition
- Correctness is bound by algorithm termination, and eventual (liveness) properties have requirements on correct processes
- Algorithm termination on individual processes is not same
- Some processes can terminate their “active” part of the algorithm, move on to subsequent (causally) tasks and fail amidst
  - E.g., message still in buffer of outgoing channel
  - The failure affects also the seemingly terminated previous algorithm
  - Often algorithm runs follow each other

Sometimes “for free”
How About

Uniform variants of:

I. No duplication
   - No message is delivered more than once

II. No creation
   - No message is delivered unless it was broadcast

(usually summarized as Integrity)?
Terminating Reliable Broadcast

- **Message delivered iff broadcaster delivers it**
  - Otherwise, SF (sender failure) may be delivered
  - All processes do deliver something
  - Cf. passive replication

- **I., III., IV., and**

- **VI. No creation’**
  - No message other than SF is delivered unless it was broadcast

- **VII. Termination**
  - Every correct process eventually delivers some message
Implementable?

Processes need to know if broadcaster delivered
  – If correct, need to deliver \( m \)
  – If not, may deliver SF

How to know?
  – Acknowledgement
    ▪ What if does not arrive?
  – Need to accurately detect failure of broadcaster
  – If failed, need to decide between SF and \( m \)
Ordered Broadcasts

- Concurrency/parallelism underlies distribution
  - Concurrent activities need to be perceived by all processes in the same order
    - Cf. causality relationship
  - Ordering guarantees required depend on application

- FIFO
  - “Messages from same process are delivered in the order broadcast”

- Causal Order
  - “Messages broadcast are delivered after any messages that causally affected them”

- Total Order
  - “Processes deliver messages in same order”
I., II., III., IV., and VIII.

VIII. FIFO order

- If a process broadcasts a message $m_1$ before $m_2$, then no correct process delivers $m_2$ before $m_1$

▶ Can we build FIFO Broadcast with Reliable Broadcast?
Proposition

\[ \text{msgBag} := \emptyset \]
\[ \text{next}[p] := 1 \text{ for all } p \]

To execute \( \text{broadcast}_F(m) \):
\[ \text{broadcast}_R(m) \quad \text{\\ \( \text{m} \) tagged with sender and seq#} \]

\( \text{deliver}_F(m) \) occurs as follows:

Upon \( \text{deliver}_R(m) \) do
\[ p := \text{sender}(m) \]
\[ \text{msgBag} := \text{msgBag} \cup \{m\} \]
While (exists \( m' \in \text{msgBag}: \text{sender}(m') = p \) and \( \text{seq#}(m) = \text{next}[p] \)) do
\[ \text{deliver}_F(m') \]
\[ \text{next}[p]++ \]
\[ \text{msgBag} := \text{msgBag} \setminus \{m'\} \]
IX. Uniform FIFO order

- If a process broadcasts a message \( m_1 \) before \( m_2 \), then no process (correct or faulty) delivers \( m_2 \) before \( m_1 \).

Implementable?

- Can anyone deliver \( m_2 \) from \( p \) before \( m_1 \)?
Causal Order Reliable Broadcast

Causal order (in broadcast setting)

\[ a \rightarrow b \iff \]

1. A process executed both \( a \) and \( b \), and in that order,
2. \( a \) is a broadcast (m) and \( b \) is a deliver (m), or
3. \( \exists c: a \rightarrow c \) and \( c \rightarrow b \)

Causal order vs FIFO order?
I., II., III., IV., VIII., and X.

X. Local order
   - If a process delivers a message $m_1$ before it broadcasts a message $m_2$, then no correct process delivers $m_2$ before $m_1$

► How about

XI. Causal order
   - If the broadcast of a message $m_1$ causally precedes the broadcast of a message $m_2$, then no correct process delivers $m_2$ unless it has previously delivered $m_1$

► XI. is equivalent to VIII. and X.
with FIFO Broadcast -- Uniform FIFO order:

\[ \text{prevDlvrs} := \bot \]

\textit{to execute broadcast}_C(m):

\text{broadcast}_F(<\text{prevDlvrs} \ || \ m>)

\[ \text{prevDlvrs} := \bot \]

\textit{deliver}_C(m) \textit{ occurs as follows:}

\textit{upon deliver}_F(<m_1, \ldots, m_k>) \textit{ do}

\text{for i:= 1..k do}

\text{if p has not previously executed deliver}_C(m_i) \textit{ then}

\text{deliver}_C(m_i)

\[ \text{prevDlvrs} := \text{prevDlvrs} \ || \ m_i \]
Correct?

- Why purge prevDlvrars after a broadcast?
- Do we need uniform FIFO order?
  - Suppose a faulty process $p_i$ delivers ($\text{deliver}_F$) a message $m_2$ before a message $m_1$ (from the same broadcaster)
  - It can broadcast a message after delivering $m_2$, and then fail, ...
    - Remember: it can still be in the process of broadcasting/sending previous message $m_2$ and/or $m_1$
    - Failures propagate throughout algorithm runs
      - Particularly in ordered broadcasts, as these relate different messages/runs
- Limitations?
  - Improvements?
XII. Uniform causal order

- If the broadcast of a message $m_1$ causally precedes the broadcast of a message $m_2$, then no process (correct or faulty) delivers $m_2$ unless it has previously delivered $m_1$

▶ How about the previous algorithm?

▶ How about uniform agreement?
I., II., III., IV., and XIII.

**XIII. Total order**

- If correct processes $p_i$ and $p_j$ both deliver messages $m_1$ and $m_2$, then $p_i$ delivers $m_1$ before $m_2$ iff process $p_j$ delivers $m_2$ before $m_1$

**Can we build Atomic Broadcast with Reliable Broadcast?**

- Lamport clocks?
- Vector clocks?
Causal Atomic Broadcast

with FIFO Atomic Broadcast:

\[ \text{prevDlvrs := } \emptyset \]

\[ \text{to execute broadcast}_{CA}(m): } \]
\[ \text{broadcast}_{FA}(<m, \text{prevDlvrs}>) \]
\[ \text{prevDlvrs := } \emptyset \]

\[ \text{deliver}_{CA}(m) \text{ occurs as follows: } \]
\[ \text{upon deliver}_{FA}(<m,D>) \text{ do } \]
\[ \text{if sender}(m) \notin \text{suspects and } \]
\[ p \text{ has prev. executed } \text{deliver}_{CA}(m') \quad \forall m' \in D \]
\[ \text{then } \]
\[ \text{deliver}_{CA}(m) \]
\[ \text{prevDlvrs := prevDlvrs} \cup \{m\} \]
\[ \text{else } \]
\[ \text{discard } m \]
\[ \text{add sender}(m) \text{ to suspects} \]
suspects?

- Process can be faulty, and thus violate XIII.
  - Deliver messages out of order and broadcast before crashing

Remedy?
Interestingly, we can implement causal order in an asynchronous distributed system…
  – … but not Total Order

Intuition
  – Causal ordering is defined w.r.t. a given message
    ▪ Only depends on the deliveries preceding the broadcast on the very broadcaster (one process)
    ▪ Ordering is defined a priori, algorithm must then enforce this order
    ▪ (Same goes for FIFO)
  – Total ordering of messages is defined w.r.t. multiple messages from concurrent broadcasters
    ▪ No predefined order, algorithm must “come up” with an order. How about deterministic order?