Distributed systems

Causal Broadcast

Prof R. Guerraoui
Distributed Programming Laboratory
Overview

- **Intuitions:** why causal broadcast?

- **Specifications of causal broadcast**

- **Algorithms:**
  - A *non-blocking* algorithm using the *past* and
  - A *blocking* algorithm using *vector clocks*
Broadcast

A \rightarrow \text{broadcast} \rightarrow m \rightarrow \text{deliver} \rightarrow B

C \rightarrow \text{deliver} \rightarrow m \rightarrow \text{broadcast} \rightarrow A
Intuitions (1)

- So far, we did not consider ordering among messages; In particular, we considered messages to be independent.
- Two messages from the same process might not be delivered in the order they were broadcast.
- A message m1 that causes a message m2 might be delivered by some process after m2.
Consider a system of news where every new event that is displayed in the screen contains a reference to the event that caused it, e.g., a comment on some information includes a reference to the actual information.

Even uniform reliable broadcast does not guarantee such a dependency of delivery.

Causal broadcast alleviates the need for the application to deal with such dependencies.
Modules of a process

Applications

(R-U) Causal broadcast

Failure detector

(R-U) Reliable broadcast

Channels
Overview

- **Intuitions:** why causal broadcast?
- **Specifications** of *causal broadcast*
- **Algorithms:**
  - A *non-blocking* algorithm using the *past* and
  - A *blocking* algorithm using *vector clocks*
Causal broadcast

**Events**
- Request: `<coBroadcast, m>`
- Indication: `<coDeliver, src, m>`

**Property:**
- **Causal Order (CO)**
Causality

Let $m_1$ and $m_2$ be any two messages: $m_1 \rightarrow m_2$ (m1 causally precedes m2)

iff

C1 (FIFO order). Some process $p_i$ broadcasts $m_1$ before broadcasting $m_2$

C2 (Local order). Some process $p_i$ delivers $m_1$ and then broadcasts $m_2$

C3 (Transitivity). There is a message $m_3$ such that $m_1 \rightarrow m_3$ and $m_3 \rightarrow m_2$
Causal broadcast

**Events**

- Request: `<coBroadcast, m>`
- Indication: `<coDeliver, src, m>`

**Property:**

- **CO:** If any process pi delivers a message m2, then pi must have delivered every message m1 such that m1 -> m2
Causality?
Causality ?
Causality ?

p1

m1

delivery

m2
delivery

m2

delivery

m1
delivery

p2

p3
Reliable causal broadcast (rcb)

*Events*
- Request: <rcoBroadcast, m>
- Indication: <rcoDeliver, src, m>

- **Properties:**
  - **RB1, RB2, RB3, RB4** +
  - **CO**
Uniform causal broadcast (ucb)

**Events**
- Request: <ucoBroadcast, m>
- Indication: <ucoDeliver, src, m>

**Properties:**
- URB1, URB2, URB3, URB4 +
- CO
Overview

- **Intuitions:** why causal broadcast?
- **Specifications** of *causal broadcast*
- **Algorithms:**
  - A *non-blocking* algorithm using the *past*
  - And
  - A *blocking* algorithm using *vector clocks*
Algorithms

- We present **reliable causal broadcast algorithms** using **reliable broadcast**

- We obtain **uniform causal broadcast algorithms** by using instead an underlying **uniform reliable broadcast**
Algorithm 1

**Implements:** ReliableCausalOrderBroadcast (rco).

**Uses:** ReliableBroadcast (rb).

**upon event** `< Init > do`
- delivered := past := ∅;

**upon event** `< rcoBroadcast, m> do`
- **trigger** `< rbBroadcast, [Data,past,m]>;`
- past := past U {[self,m]};
Algorithm 1 (cont’d)

upon event <rbDeliver,pi,[Data,pastm,m]> do

if m \notin delivered then

(*) forall [sn, n] in pastm do

if n \notin delivered then

trigger < rcoDeliver,sn,n>;
delivered := delivered U \{n\};
past := past U \{[sn, n]\};
Algorithm 1 (cont’d)

(*)

... 

... 

... 

... 

trigger <rcoDeliver,pi,m>;
delivered := delivered U {m};
past := past U {[pi,m]};
Algorithm 1

p1

m1

m2

m2(m1)

p2

m1

m2

m2(m1)

p3

m1

m2
Algorithm 1
Uniformity

- Algorithm 1 ensures causal reliable broadcast

- If we replace reliable broadcast with uniform reliable broadcast, Algorithm 1 would ensure uniform causal broadcast
Algorithm 1’ (gc)

- **Implements:** GarbageCollection (+ Algo 1).
- **Uses:**
  - ReliableBroadcast (rb).
  - PerfectFailureDetector(P).
- **upon event** < Init > **do**
  - delivered := past := ∅;
  - correct := S;
  - ackm := ∅ (for all m);
Algorithm 1’ (gc – cont’d)

- **upon event** \(<\) crash, \(\pi\) \(>)\ do
  - correct := correct \(\setminus\) \(\{\pi\}\)

- **upon** for some \(m \in\) delivered: self \(\notin\) ack\(m\) do
  - ack\(m\) := ack\(m\) \(\cup\) \(\{\text{self}\}\);
  - **trigger** \(<\) rbBroadcast, [ACK,\(m\)]>;}
Algorithm 1’ (gc – cont’d)

upon event <rbDeliver, pi, [ACK, m]> do
  ack_m := ack_m U {pi};
  if forall pj ∈ correct: pj ∈ ack_m do
    past := past \ {[sm, m]};
Algorithm 2

- **Implements**: ReliableCausalOrderBroadcast (rco).
- **Uses**: ReliableBroadcast (rb).

**upon event** `< Init >` **do**

- **for all** $pi \in S$: $VC[pi] := 0$;
- pending := $\emptyset$
Algorithm 2 (cont’d)

upon event < rcoBroadcast, m> do
  trigger < rcoDeliver, self, m>;
  trigger < rbBroadcast, [Data,VC,m] >;
  VC[self] := VC[self] + 1;
Algorithm 2 (cont’d)

upon event <rb Deliver, pj, [Data, VCM, m]>

if pj ≠ self then

  pending := pending ∪ (pj, [Data, VCM, m]);

  deliver-pending.
Algorithm 2 (cont’d)

procedure deliver-pending is
  While (s, [Data,VCm,m]) ∈ pending s.t.
  for all pk: (VC[pk] ≥ VCm[pk]) do
    pending := pending - (s, [Data,VCm,m]);
    trigger < rcoDeliver, self, m>);
    VC[s] := VC[s] + 1.
Algorithm 2

p1

m1

m1

m2

m2

m1

[0,0,0]

m1

[1,0,0]

m2

[0,0,0]

m2

m1

31
Algorithm 2

\begin{align*}
\text{p1} & \quad \text{m1} & \quad \text{m2} \\
\text{p2} & \quad \text{m1} & \quad \text{m2} & \quad [1,0,0] \\
\text{p3} & \quad [1,0,0] & \quad [0,0,0] & \quad \text{m1} & \quad \text{m2} \\
\end{align*}