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 \quad m \quad B

broadcast \quad deliver

m

C

deliver
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 $m_1$ that causes a message $m_2$ might be delivered by some process after $m_2$. 
Intuitions (2)

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: $m_1 \rightarrow m_2$ ($m_1$ causally precedes $m_2$) 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: \(<\text{coBroadcast}, m>\>
- Indication: \(<\text{coDeliver}, \text{src}, m>\>\

**Property:**
- **CO:** If any process \(p_i\) delivers a message \(m_2\), then \(p_i\) must have delivered every message \(m_1\) such that \(m_1 \rightarrow m_2\)
Causality ?

p1

m1

delivery

delivery

m2

p2

delivery

delivery

m2

m1

p3

delivery

delivery
Causality?
Causality ?
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:

- \textit{URB1, URB2, URB3, URB4 +}
- \textit{CO}
Overview

**Intuitions:** why causal broadcast?

**Specifications of causal broadcast**

**Algorithms:**
- A *non-blocking* algorithm using the *past*
- 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,pastₘₘₘ,m]>

if \( m \not\in \text{delivered} \) then

(*) forall \([sn, n]\) in pastₘ do

if \( n \not\in \text{delivered} \) then

trigger < rcoDeliver,sn,n>;

delivered := delivered \cup \{n\};

past := past \cup \{[sn, n]\};
Algorithm 1 (cont’d)

(*)

... 
... 
... 

trigger <rcoDeliver,pi,m>;

delivered := delivered U \{m\};
past := past U \{[pi,m]\};
Algorithm 1
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**

```plaintext
delivered := past := \emptyset;
correct := S;
ack_m := \emptyset \ (\text{for all } m);
```
Algorithm 1’ (gc – cont’d)

upon event < crash, pi > do
  correct := correct \ {pi}

• upon for some m ∈ delivered: self ∉ ackm do
  • ackm := ackm U {self};
  • trigger < rbBroadcast, [ACK,m]>;
Algorithm 1’ (gc – cont’d)

upon event $<rbDeliver, pi, [ACK, m]>$ do

$ack_m := ack_m \cup \{pi\}$;

if forall $pj \in correct$: $pj \in ack_m$ do

$past := past \setminus \{[sm, m]\}$;
**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 <rbDeliver, pj, [Data,VCm,m]> do
  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

m2

[0,0,0]

m2

[1,0,0]

m1

m1

[0,0,0]
Algorithm 2

p1

m1

[1,0,0] m2

p2

m1

m2

m2

p3

[1,0,0]

m1

[0,0,0]

m1

m2