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
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.
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

(R-U) Reliable broadcast

Failure detector

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 m1 and m2 be any two messages: m1 -> m2 (m1 causally precedes m2) iff

C1 (FIFO order). Some process pi broadcasts m1 before broadcasting m2

C2 (Local order). Some process pi delivers m1 and then broadcasts m2

C3 (Transitivity). There is a message m3 such that m1 -> m3 and m3 -> m2
Causal broadcast

**Events**

- Request: \(<\text{coBroadcast}, m>\>
- Indication: \(<\text{coDeliver}, \text{src}, m>\>

**Property:**

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

p1  delivery  delivery

p2  m1  delivery  m2  delivery

p3  m2  delivery  m1
Causality ?

p1

m1

p2

m2

p3

m2

m1
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 \textless rcoDeliver,pi,m\textgreater ;
delivered := delivered U \{m\};
past := past U \{[pi,m]\};
Algorithm 1

\text{p1} \quad \text{m1} \quad \text{m2} \quad \text{m2(m1)}

\text{p2} \quad \text{m1} \quad \text{m2} \quad \text{m2(m1)}

\text{p3} \quad \text{m1} \quad \text{m2}
Algorithm 1

```
p1
  m1
  m2(m1)  m2

p2
  m1
  m1
  m2(m1)  m2

p3
  m1
  m2
```
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 := \emptyset;`
- `correct := S;`
- `ackm := \emptyset` (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 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 ∈ S: VC[pi] := 0;

pending := ∅
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

m2

p2

m1

m2

[1,0,0]

p3

[0,0,0]
Algorithm 2