Fault-tolerant broadcasts

Part II: Algorithms

Fault-tolerant broadcasts

- Reliable broadcast
  - System model
    - $\prod = \{ p_1, p_2, \ldots, p_n \}$
    - Asynchronous system (i.e., no timing assumptions)
    - Crash-stop failure model (at most f failures)
    - Quasi-reliable channels
      - If p sends a message m to q and both processes are correct, then q eventually receives m

Fault-tolerant broadcasts

- Non-uniform reliable broadcast

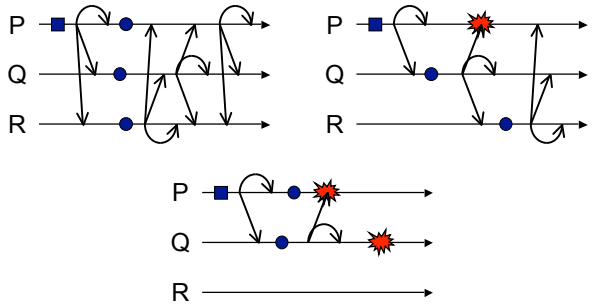
Agreement: if a correct process delivers m, then all correct processes eventually deliver m.

broadcast(R, m) works as follows:
send(m) to all

upon receive(m) for the first time do
deliver(R, m)
send(m) to all

Fault-tolerant broadcasts

- Non-uniform reliable broadcast (cont'd)



Fault-tolerant broadcasts

– Uniform reliable broadcast

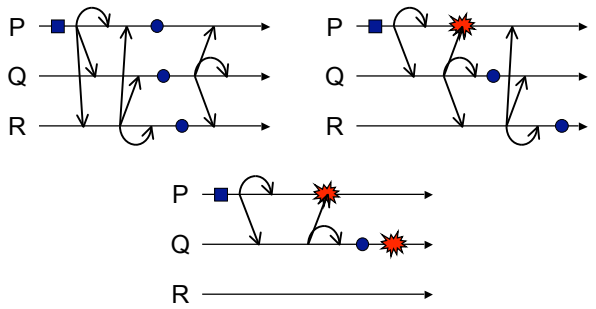
Uniform agreement: if a process delivers m, then all correct processes eventually deliver m.

broadcast(UR, m) works as follows:
    send(m) to all

upon receive(m)
    if [for f+1 processes q: received(m) from q] then
        deliver(UR, m)
    if [didn't send m yet] then send(m) to all

Fault-tolerant broadcasts

– Uniform reliable broadcast (cont'd) – f = 1



Fault-tolerant broadcasts

– Cost of reliable broadcast algorithms

	Number of Messages	Latency
Reliable Broadcast (optimized)	n^2	δ
Uniform Reliable Broadcast	n^2	2δ

Fault-tolerant broadcasts

– Reducing the number of messages

- Asynchronous system + $\diamond S$

broadcast(R, m) works as follows:
    send(m) to all

upon receive(m) for the first time do
    deliver(R, m)
    start Task Propagate(m, sender(m))

Task Propagate(m, q)
    if suspect q then send(m) to all

Fault-tolerant broadcasts

- FIFO reliable broadcast
  - Transforms Reliable Broadcast into FIFO Uniform Reliable Broadcast
  - Assumptions
    - If m is the i-th message broadcast by p, then m is tagged with sender(m) = p and seq#(m) = i
    - Each process keeps a local vector of counters next[1..n]

Fault-tolerant broadcasts

```
Initialization
    msgSet ← Ø
    next[q] ← 1, for each process q

To execute broadcast(F, m):
    broadcast(R, m)

deliver(F,–) occurs as follows:
    upon deliver(R, m) do
        q ← sender(m)
        msgSet ← msgSet ∪ { m }
        while (∃m' ∈ msgSet: sender(m') = q and next[q] = seq#(m'))
            deliver(F, m')
            next[q] ← next[q] + 1
```

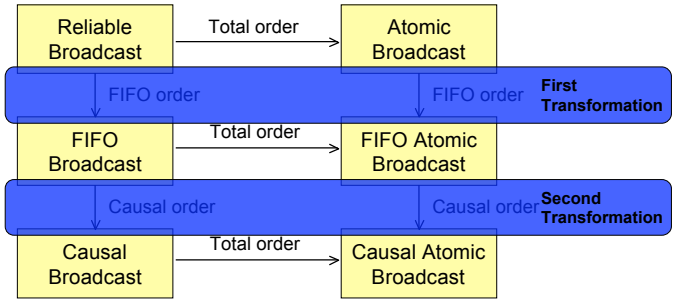

Fault-tolerant broadcasts

- Causal reliable broadcast
  - Transforms Uniform Reliable Broadcast into Causal Reliable Broadcast
  - Notation
    - Message sequences: $\langle m_1, m_2, \ldots \rangle$
    - $\perp$ is the empty sequence
    - $\oplus$ is the concatenation operator

Fault-tolerant broadcasts

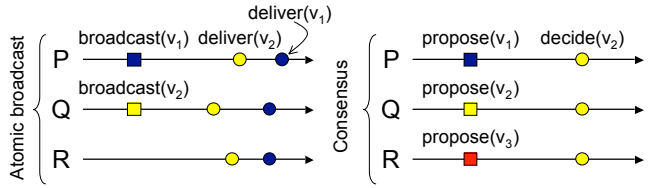
```
Initialization
    rcntDlvrs ← ⊥

To execute broadcast(C, m):
    broadcast(F, rcntDlvrs ⊕ ⟨ m ⟩ )
    rcntDlvrs ← ⊥

deliver(C,–) occurs as follows:
    upon deliver(F, ⟨ m1, m2, ..., ml ⟩) do
      for i from 1 to l do
          if p has not previously executed deliver(C, mi) then
              deliver(C, mi)
              rcntDlvrs ← rcntDlvrs ⊕ ⟨ mi ⟩
```

Fault-tolerant broadcasts



Fault-tolerant broadcasts

- Atomic broadcast
  - From atomic broadcast to consensus



To execute **propose**(v):
broadcast(v)

Let v be the first value delivered
decide(v)

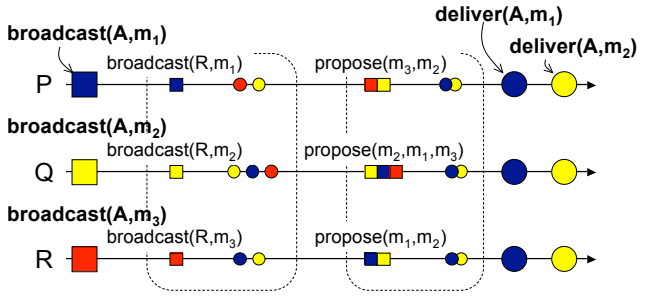
Fault-tolerant broadcasts

- From consensus to atomic broadcast

```
Initialization
    R-delivered ← Ø
    A-delivered ← Ø
    k ← 0
To execute broadcast(A, m):
    broadcast(R, m)
deliver(A,–) occurs as follows:
    upon deliver(R, m) do
        R-delivered ← R-delivered ∪ { m }
    upon R-delivered \ A-delivered ≠ Ø
        k ← k + 1
        A-undelivered ← R-delivered \ A-delivered
        propose(k, A-undelivered)
        wait until decide(k, msgSet^k)
        A-deliver^k ← msgSet^k \ A-delivered
        deliver all messages in A-deliver^k in some deterministic order
        A-delivered ← A-delivered ∪ A-deliver^k
```

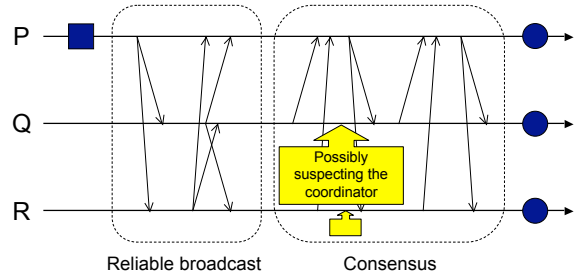

Fault-tolerant broadcasts

- From consensus to atomic broadcast (cont'd)



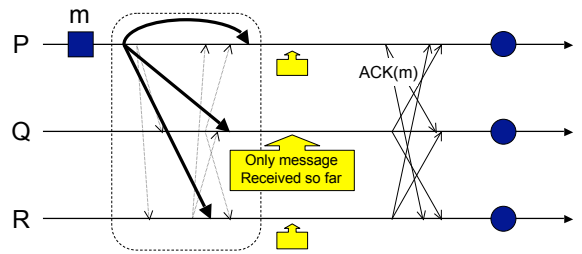
Fault-tolerant broadcasts

– From consensus to atomic broadcast (cont'd)



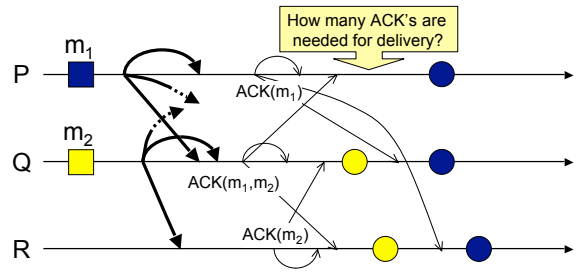
Fault-tolerant broadcasts

- Generic broadcast
  - Single-message case



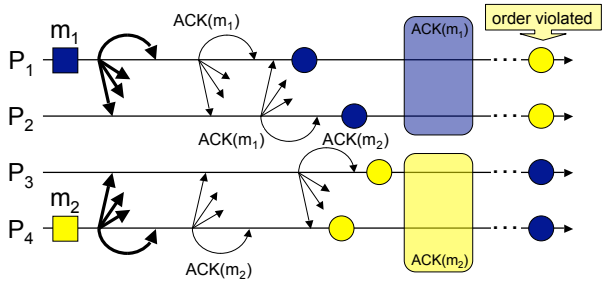
Fault-tolerant broadcasts

– Two non-conflicting messages



Fault-tolerant broadcasts

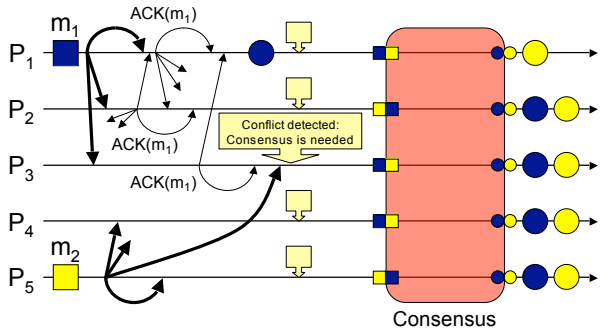
– The general case (m_1 and m_2 conflict)



$n_{ACK} \leq n/2 \Rightarrow$ order violated

Fault-tolerant broadcasts

– The general case (m_1 and m_2 conflict)



Fault-tolerant broadcasts

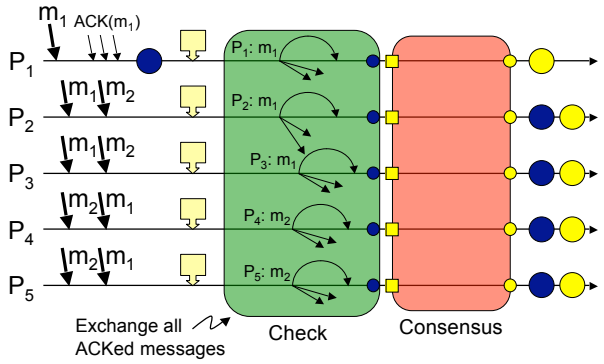
– The general case (m_1 and m_2 conflict)

- Conflicts should be detected: $n_{ACK} > n / 2$
- For conflicting messages:
  "If m_1 and m_2 conflict, and m_1 (or m_2) has been delivered before consensus, then consensus cannot contradict this order."

How to ensure this???

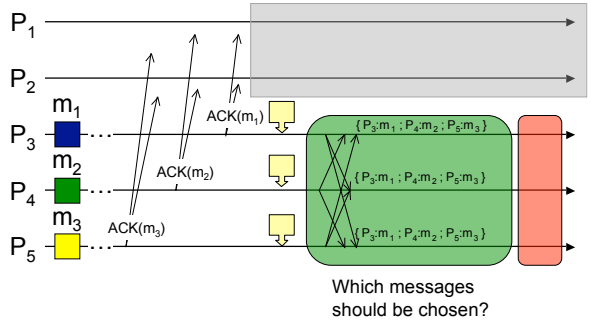
Fault-tolerant broadcasts

– The general case (m_1 and m_2 conflict)



Fault-tolerant broadcasts

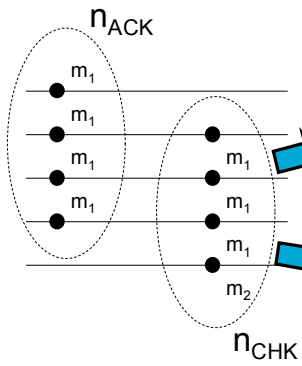
– The general case (m_1 and m_2 conflict)



Fault-tolerant broadcasts

- If $n_{ACK} = \lceil (n+1) / 2 \rceil$ (i.e., majority), how much should n_{CHK} be?
- How to choose a message in the "check phase"?
- Problem: we can't tolerate failures
- What can be done?
- Increase n_{ACK}… for example, if $n_{ACK} = 4$ and $n = 5$, could we have $n_{CHK} = 4$?

Fault-tolerant broadcasts

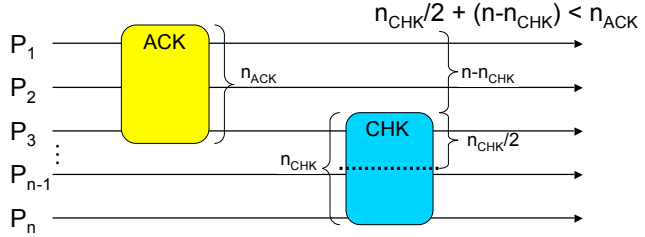


Could m_2 be delivered in the "ack phase"?

Can we be sure that m_1 was delivered in the "ack phase"?

Fault-tolerant broadcasts

- What is the relation between n_{ACK} and n_{CHK}?



Fault-tolerant broadcasts

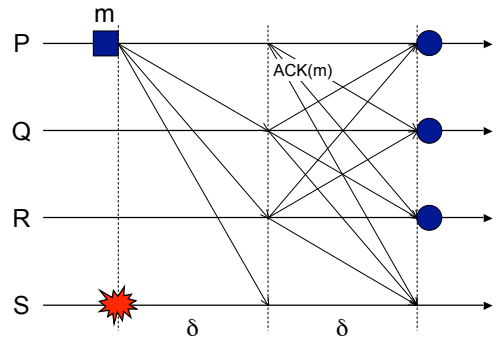
- The general case (cont'd)

$n_{CHK}/2 + (n-n_{CHK}) < n_{ACK}$
$n_{CHK} + 2n - 2n_{CHK} < 2n_{ACK}$
$-n_{CHK} + 2n < 2n_{ACK}$
$2n_{ACK} + n_{CHK} > 2n$

Best case: $n_{ACK} = n_{CHK} = x$
$2x + x > 2n$
$x > 2n / 3$ (Optimal for ACK with 2δ)

Fault-tolerant broadcasts

– Optimality (i.e., lower bound)



Fault-tolerant broadcasts

```
Initialization
    R-delivered ← G-delivered ← Ø
    pending¹ ← g-Deliver¹ ← Ø
    k ← 1

To execute broadcast(m):
    broadcast(R, m)

when deliver(R, m) do
    R-delivered ← R-delivered ∪ { m }

when (R-delivered \ (G-delivered ∪ pendingᵏ) ≠ Ø)
    if [ for all m, m' ∈ (R-delivered \ G-delivered): m doesn't conflict with m' ] then
        pendingᵏ ← R-delivered \ G-delivered
        send (k, pendingᵏ, ACK) to all
    else
        Handle conflict (next slide)

when receive (k, pendingᵏ, ACK) from pⱼ
    while ∃m such that [ for n_ACK pᵢ: received (k, pendingᵏ, ACK) from pⱼ and
                                        m ∈ (pendingᵏ \ g-Deliverᵏ) ] do
        g-Deliverᵏ ← g-Deliverᵏ ∪ { m }
        deliver(m)
```


Fault-tolerant broadcasts

```
Handling conflicts:

when (R-delivered \ (G-delivered ∪ pendingᵏ) ≠ Ø)
    if [ for all m, m' ∈ (R-delivered \ G-delivered): m doesn't conflict with m' ] then
        ...
    else
        send (k, pendingᵏ, CHK) to all
        wait until [ for n_CHK pⱼ : received (k, pendingᵏ, CHK) from pⱼ ]
        msgSet = { m | for ⌈(n_CHK+1)/2⌉ pⱼ : received (k, { …, m, … }, CHK) from pⱼ

        propose(k, msgSet, (R-delivered \ (G-delivered ∪ pendingᵏ)))
        wait until decide (k, NCset, Cset)

        for each m ∈ NCset \ (G-delivered ∪ g-Deliverᵏ) do deliver(m)
        in ID order: for each m Cset \ (G-delivered ∪ g-Deliverᵏ) do deliver(m)
        G-delivered ← G-delivered ∪ NCset ∪ Cset

        k ← k + 1
        pendingᵏ ← g-Deliverᵏ ← Ø
```


Paxos

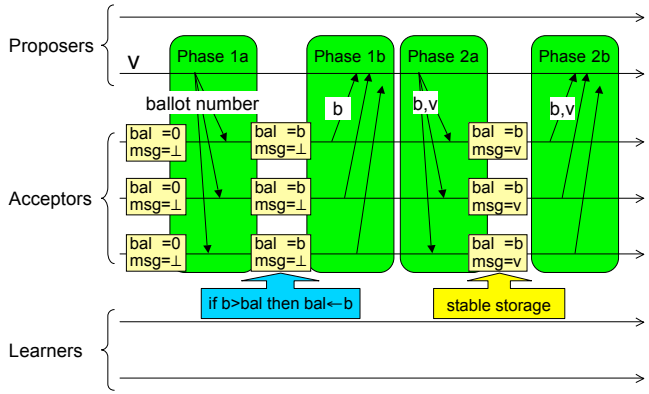
- The part-time parliament
  - Leslie Lamport
  - Consensus algorithm
  - SRC Research Report 49, Sep. 1, 1989
  - *ACM TOCS,* May 1998
    - Submitted in 1990

Paxos

- The protocol
  - System model
    - $\prod = \{ p_1, p_2, \ldots, p_n \}$
    - Asynchronous system, plus…
    - Leader-election oracle (Ω)
    - Crash-recovery failure model
    - Unreliable channels
    - Stable storage

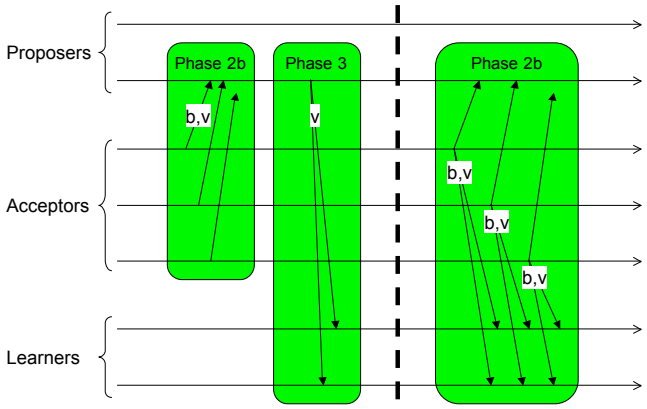
Paxos

Choosing a value

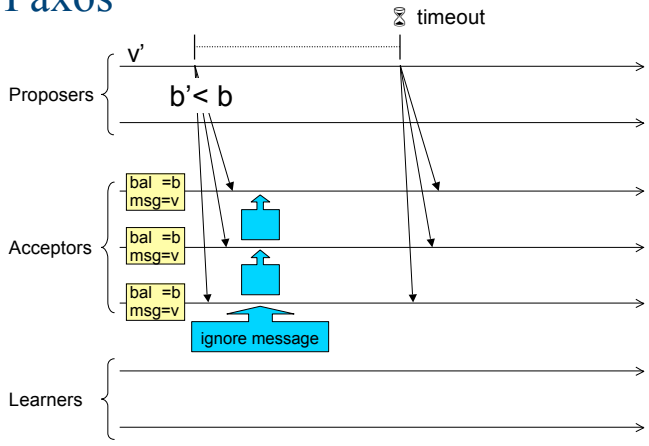


Paxos

Learning a chosen value

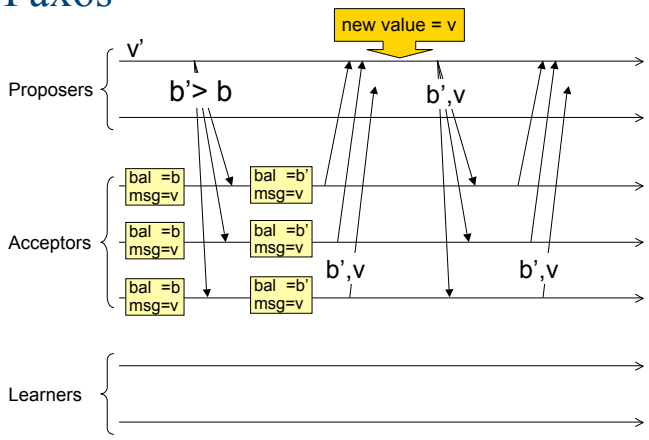


Paxos



Handling old ballot numbers

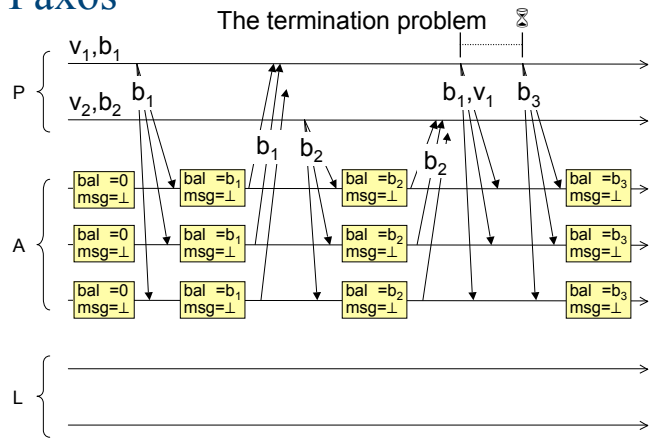
Paxos



Handling new ballot numbers

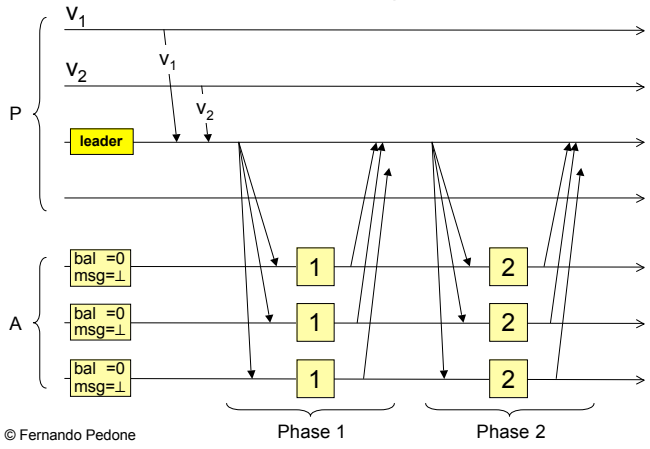
Paxos

The termination problem



Paxos

A leader-based protocol



Paxos

Ballot reservation