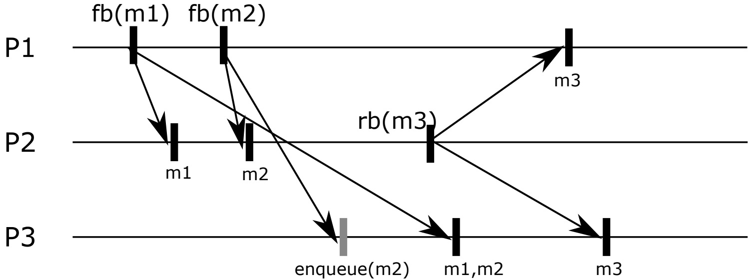
Dependable Distributed Systems – 5880V/UE

Part 3: Broadcast semantics and algorithms – 2021-10-06

Prof. Dr. Hans P. Reiser | WS 2021/22

UNIVERSITÄT PASSAU



Overview

1. Basics: Safety and Liveness
2. Broadcast
   - Best-effort broadcast
   - Reliable broadcast (rbcast)
   - FIFO broadcast (fbcast)
   - Causal broadcast (cbcast)
   - Atomic broadcast (abcast)
3. Reliable broadcast with Byzantine faults
4. Implementing atomic broadcast
   - Sequencer-based approaches
   - Token-based approaches
   - Communication history-based approaches
   - Consensus-based approaches

Safety and liveness [according to Lamport]

- Safety *(Sicherheit)*
  - **Something bad does not happen**
  - Safety property defines that some events must *never* happen
  - In other words: Safety property is a predicate P that is always true
- Liveness *(Lebendigkeit)*
  - **Something good will happen**
  - Liveness property defines good events that at some time (“eventually”) must happen
  - In other words: Liveness property is a predicate P that eventually becomes true

Safety and liveness

Beware of false friends

“eventually” is (close to) the opposite of the german word “eventuell”

- *“X eventually happens”* means that:
- *“X **will happen** within an unspecified but finite amount of time”*

Safety and liveness

Example: Factorization algorithm $\mathit{Fac}(n) \rightarrow \{p_i\}$:

- If $\mathit{Fac}(n)$ is started with an argument n that is the product of two prime numbers, *Fac* will output $p_1 > 1, p_2 > 1$ with $p_1 \cdot p_2 = n$.

Safety and liveness

Example: Factorization algorithm $Fac(n) \rightarrow \{p_i\}$:

- If $Fac(n)$ is started with an argument n that is the product of two prime numbers, Fac will output $p_1 > 1, p_2 > 1$ with $p_1 \cdot p_2 = n$.
- Safety property or liveness property?

Safety and liveness

Verification usually is easier if you use multiple minimalistic properties!

Example:

- *Liveness:* The algorithm $Fac(n)$, started with an arbitrary integer n as argument, eventually terminates.
- *Safety:* If $Fac(n)$ terminates and produces output $\{p_1, \ldots, p_k\}$, then $\forall i \in \{1..k\} : p_i$ is a prime number > 1 and $\prod_{i=1}^{k} p_i = n$

Overview

1. Basics: Safety and Liveness
2. Broadcast
   - Best-effort broadcast
   - Reliable broadcast (rbcast)
   - FIFO broadcast (fbcast)
   - Causal broadcast (cbcast)
   - Atomic broadcast (abcast)
3. Reliable broadcast with Byzantine faults
4. Implementing atomic broadcast
   - Sequencer-based approaches
   - Token-based approaches
   - Communication history-based approaches
   - Consensus-based approaches

Broadcasts

Fault-Tolerant Broadcasts

- Very fundamental abstraction for distributed systems
- A large variety of broadcasts with different **semantics**
- Algorithms highly depend on **system model**

Broadcasts

Fault-Tolerant Broadcasts

- Very fundamental abstraction for distributed systems
- A large variety of broadcasts with different **semantics**
- Algorithms highly depend on **system model**

This chapter is heavily based

- on the publication *"A modular approach to fault-tolerant broadcasts and related problems"* by Vassos Hadzilacos and Sam Toueg (1994)
  - All following definitions (green boxes) are direct citations from the publication; you should read the article at least up to page 16.
- and on RSDP

Broadcasts: layered architecture

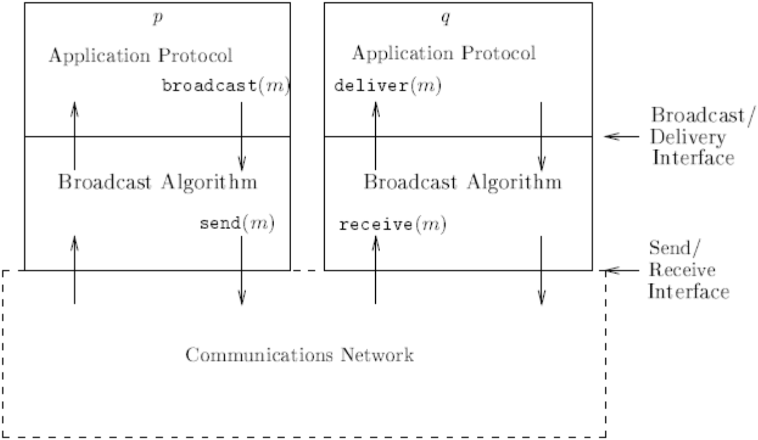


Figure 3: Application/Broadcast Layering

Broadcast semantics

- Best-effort broadcast
- **rbcast**: Reliable broadcast
- **fbcast**: FIFO broadcast
- **cbcast**: Causal broadcast
- **abcast**: Atomic (total-order) broadcast
- **fabcast**: FIFO atomic broadcast
- **cabcast**: Causal atomic broadcast
- Timed broadcast
- Uniform broadcast

Best-effort broadcast

Implementation

- Send message to all nodes using perfect point-to-point links
- (Or use existing low-level broadcast mechanisms, such as Ethernet broadcast)

Characteristics

- Simple, low latency
- No message order guarantees
- Reliability?

Best-effort broadcast

Implementation

- Send message to all nodes using perfect point-to-point links
- (Or use existing low-level broadcast mechanisms, such as Ethernet broadcast)

Characteristics

- Simple, low latency
- No message order guarantees
- Reliable delivery if sender does no crash
  - No guarantees if sender crashes! ($\rightarrow$ see next slide)

Best-effort broadcast

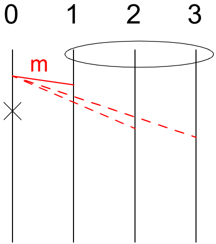
Best-effort broadcast – *P* denotes the set of all processes:

```
Uses: PerfectPointToPointLink (ppp)

Upon event < BEBroadcast, m > do:
   forall p in P do:
      trigger < ppp.Send, p, m >;

Upon event < ppp.Receive, m > do:
   trigger < BEDeliver, m >;
```

Best-effort broadcast

Simple algorithm (best-effort broadcast):

- Works fine as long as sender does not crash
- If sender crashes, some but not all processes might receive message

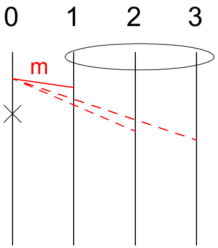


Best-effort broadcast

Simple algorithm (best-effort broadcast):

- Works fine as long as sender does not crash
- If sender crashes, some but not all processes might receive message

- In some situations, this can be a problem . . .



Overview

1. Basics: Safety and Liveness
2. Broadcast
   - Best-effort broadcast
   - Reliable broadcast (rbcast)
   - FIFO broadcast (fbcast)
   - Causal broadcast (cbcast)
   - Atomic broadcast (abcast)
3. Reliable broadcast with Byzantine faults
4. Implementing atomic broadcast
   - Sequencer-based approaches
   - Token-based approaches
   - Communication history-based approaches
   - Consensus-based approaches

Reliable broadcast

Basic idea (very informal!):

- Make sure that messages are not lost: all (nonfaulty) processes shall deliver all messages
- No guarantees on message order

Reliable broadcast

Basic idea (very informal!):

- Make sure that messages are not lost: all (nonfaulty) processes shall deliver all messages
- No guarantees on message order

- Before defining a “correct” algorithm, let’s be a bit more formal. . .

Reliable broadcast (rbcast)

- Sender p executes operation $broadcast(m)$ with message m
  - m contains sender ID and unique sequence number of the sender
  - i.e. $m := (sender, seq\#, data)$
- Message m is delivered at q: Operation $deliver(m)$ is executed

Properties of rbcast

- **Validity:** correct $p : broadcast(m) \Rightarrow p : deliver(m)$ within finite time
- **Agreement:** $\exists$ correct $p : deliver(m) \Rightarrow \forall$ correct $q : deliver(m)$ within finite time
- **Integrity:** Each messages m is delivered only once by each correct process, and only if $sender(m)$ did broadcast the message

Reliable broadcast (rbcast)

- Think about an algorithm that solves the problem

Reliable broadcast (rbcast)

- Think about an algorithm that solves the problem

Echo algorithm (from paper):

Algorithm for process p:
To execute `broadcast(R,`m`)`:
    `send(`m`) to` p

`deliver(R,`m`)` *occurs as follows:*
    upon `receive(`m`)` **do**
        if p has not previously executed `deliver(R,`m`)`
        then
            `send(`m`) to` all neighbors
            `deliver(R,`m`)`

Figure 11: Reliable Broadcast for Point-to-Point Networks

Reliable broadcast (rbcast)

In our notation:

```
Uses: PerfectPointToPointLink (ppp)

upon Init:
   delivered := empty set;

upon event < RBroadcast, m > do:
   trigger < ppp.Send, self, m >;

upon event < ppp.Receive, m > do:
   if(not delivered.contains(m)) then
      forall p in P do:
         trigger < ppp.Send, p, m >;
      delivered.add(m);
      trigger < RDeliver, m >;
```

Reliable broadcast (rbcast)

Correctness verification (informal):

- Validity: RBroadcast $\rightarrow$ Send $\rightarrow$ Receive $\rightarrow$ RDeliver
  within finite time (assuming that ppp channel delivers message within finite time)
- Agreement: If some correct process trigger RDeliver, it has triggered ppp.Send for all destinations before. Thus, assuming the properties of perfect links, each correct process p_i will trigger the corresponding ppp.Receive
  - If m is not in delivered at p_i, p_i will RDeliver the message
  - If m is in delivered, it was added to delivered before, which implies RDeliver of m
- Integrity:
  - the `delivered` set prevents duplication
  - ppp.Receive happens only after ppp.Send (property of ppp link), which in turn happens only after RBroadcast

Uniform reliable broadcast

- The reliable broadcast so far places no restrictions on messages **delivered by faulty processes**
  - For instance the *agreement* property allows faulty processes to deliver different messages
  - Problematic especially if processes interact with external entities

- Properties can be strengthened ($\rightarrow$ **uniformity**):

Modified properties of uniform rbcast

- **Uniform agreement (einheitliche Übereinstimmung):**
  $\exists p : \mathit{deliver}(m) \Rightarrow \forall$ correct $q : \mathit{deliver}(m)$ within finite time
- **Uniform integrity (einheitliche Integrität):**
  Each messages m is delivered only once, and only if m was sent by $\mathit{sender}(m)$ before.

Uniform reliable broadcast

Previous echo algorithm:

- Satisfies uniform integrity
- Does **not** satisfy uniform agreement!

Uniform reliable broadcast

Previous echo algorithm:

- Satisfies uniform integrity
- Does **not** satisfy uniform agreement!

Note (without proof): It is impossible to implement uniform reliable broadcast in an asynchronous system model with an arbitrary number of faulty processes!

- Solutions exist for synchronous model
- Solutions exist for asynchronous model if a majority of processes are correct

Overview

FIFO broadcast (fbcast)

Difference: order of messages

- Reliable broadcasts: All correct processes deliver all messages, but order may be completely different (i.e., no order guarantees at all)
- FIFO: If P_i broadcasts m_1 before m_2, then no correct process will deliver m_2 unless it has previously delivered m_1
- Causal: If m_1 causally precedes m_2, then no correct process will deliver m_2 unless it has previously delivered m_1
- Total: If two correct processes, A and B, deliver two messages m_1 and m_2, then B delivers m_2 after m_1 if and only if A delivers m_2 after m_1

FIFO broadcast (fbcast)

Difference: order of messages

- Reliable broadcasts: All correct processes deliver all messages, but order may be completely different (i.e., no order guarantees at all)
- FIFO: If P_i broadcasts m_1 before m_2, then no correct process will deliver m_2 unless it has previously delivered m_1
- Causal: If m_1 causally precedes m_2, then no correct process will deliver m_2 unless it has previously delivered m_1
- Total: If two correct processes, A and B, deliver two messages m_1 and m_2, then B delivers m_2 after m_1 if and only if A delivers m_2 after m_1
- (and combinations FIFO+Total, Causal+Total)

FIFO broadcast (fbcast)

- Add another property to the definition of reliable broadcast:

Additional property of fbcast

- **FIFO order:** If a process broadcasts a message *m* before it broadcasts a message m', then no correct process delivers m' unless it has previously delivered m.

- How to implement it?
- We present a *transformation* of reliable broadcast (any algorithm) into FIFO broadcast

FIFO broadcast (fbcast)

```
Uses: Reliable broadcast (rbcast)

upon event Init:
   msgSet := ∅
   next[s] := 1 for each process s

upon event < FBroadcast, m >:
   trigger < rbcast.RBroadcast, m >;

upon event < rbcast.RDeliver, m >:
   s := sender(m);
   if next[s] = seq#(m) then
      trigger < Fdeliver, m >;
      next[s] := next[s] + 1;
      while(∃ m′ in msgSet: sender(m′)=s and next[s]=seq#(m′)):
         trigger < Fdeliver, m′ >;
         next[s] := next[s] + 1;
   else
      msgSet := msgSet ∪ m;
```

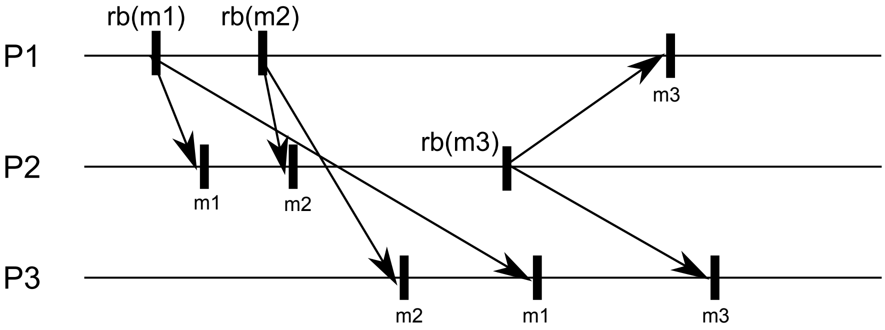
FIFO broadcast (fbcast)

The same algorithm, less formal:

- Every message has a per-sender sequence number
- If the “right” message arrives: deliver it immediately
- If a message arrives out of order (at least one message with smaller sequence number is missing): Put message in queue and deliver it only after receiving and delivering the missing messages

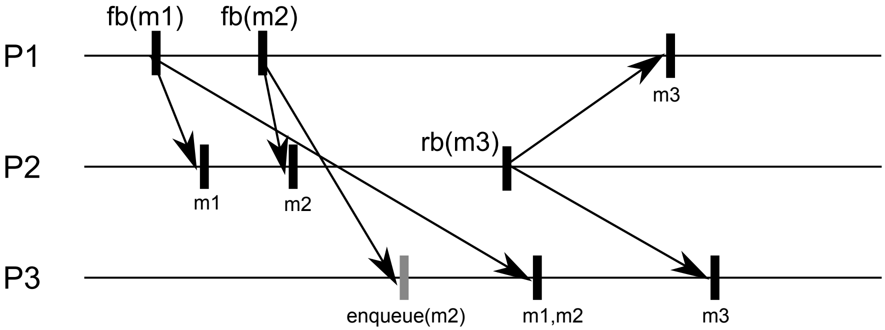
rbcast vs. fbcast

- Reliable broadcast



rbcast vs. fbcast

- FIFO broadcast



- P3 delivers *m*2 after *m*1

Overview

1. Basics: Safety and Liveness
2. Broadcast
   - Best-effort broadcast
   - Reliable broadcast (rbcast)
   - FIFO broadcast (fbcast)
   - Causal broadcast (cbcast)
   - Atomic broadcast (abcast)
3. Reliable broadcast with Byzantine faults
4. Implementing atomic broadcast
   - Sequencer-based approaches
   - Token-based approaches
   - Communication history-based approaches
   - Consensus-based approaches

Causal broadcast (cbcast)

Causal broadcast is a reliable broadcast with satisfies the following additional property:

Properties of rbcast

- **Causal order:** If the broadcast of a message m causally precedes the broadcast of a message m' (i.e., if $m \rightarrow m'$), then no correct process delivers m' unless it has previously delivered m.

Causal broadcast (cbcast)

Causal broadcast is a reliable broadcast with satisfies the following additional property:

Properties of rbcast

- **Causal order:** If the broadcast of a message *m* causally precedes the broadcast of a message m' (i.e., if $m \rightarrow m'$), then no correct process delivers m' unless it has previously delivered *m*.

⇒ See lecture on logical clocks for a definition of causal precedence ($m_1 \rightarrow m_2$)

⇒ Note that causal order implies FIFO order

Causal broadcast: simple algorithm

```
Uses: FIFO broadcast (fbcast)

upon event Init:
   rcntDeliver := ∅

upon event < CBroadcast, m >:
   trigger < fbcast.FBroadcast, <rcntDeliver || m> >;
   rcntDeliver := ∅


upon event < fbcast.FDeliver, <m_1,...,m_q> >:
   for i := 1...q:
       if p has not previously executed Cdeliver(m_i):
          trigger < Cdeliver, m_i >
          rcntDeliver := rcntDeliver || m_i
```

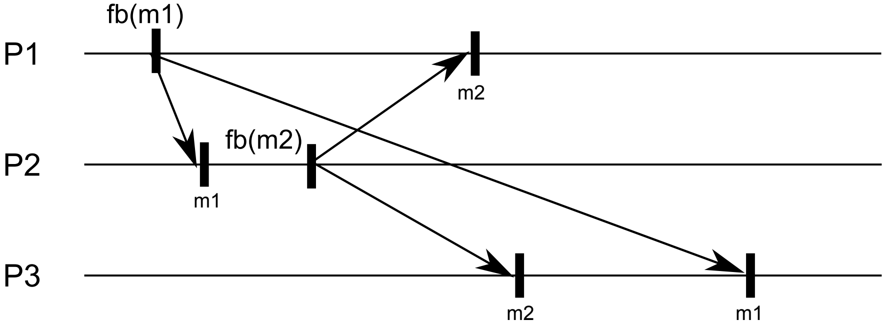
Causal broadcast: simple algorithm

Basic idea:

- Uses FIFO broadcast $\Rightarrow$ guarantees correct FIFO order
- Records all local deliveries
- Instead of sending *m*, FIFO-broadcasts all locally delivered messages since last broadcast $\Rightarrow$ guarantees correct order

fbcast vs. cbcast

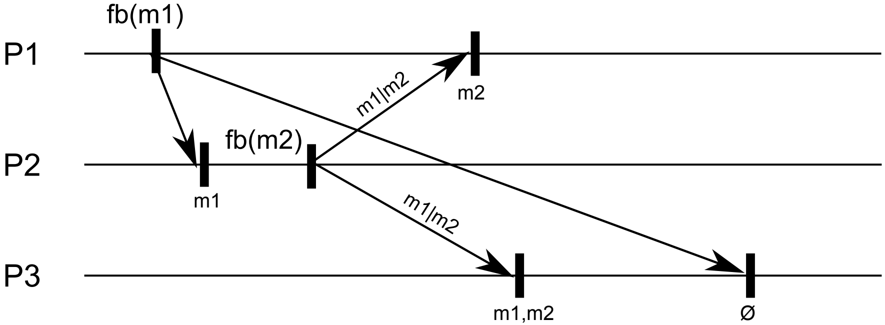
- FIFO broadcast



- P3 delivers *m*2 before *m*1, which violates causal order

fbcast vs. cbcast

- causal broadcast



- P3 delivers *m*2 after *m*1

Causal broadcast algorithms

- Previous algorithm involves a considerable overhead: each broadcast includes all causally preceding messages delivered since last broadcast
  - Benefit: algorithm is non-blocking
- Removing this overhead
  - Messages take only the IDs of those in which they depend (largest ID per process)
  - This is equivalent to labelling messages with a *vector clock*
  - Inconvenient: algorithm is blocking

Overview

Atomic broadcast (abcasts)

Atomic broadcast or total-order broadcast: fefined in terms of

- Reliable broadcast properties +
- Total order property

Additional property of abcast

Total order: If correct processes p and q both deliver messages m and m', then p delivers m before m' only if q delivers m before m'

Atomic broadcast

- Algorithm?

Atomic broadcast

- Algorithm?
- There's no (deterministic) algorithm for atomic broadcast in a system model:
  - Asynchronous communication
  - A single (a priori unknown) node my fail (crash fault)
- !? (see later)

More broadcasts

- Atomic FIFO broadcast
  - Combination of atomic broadcast with FIFO order
  - Note: Plain atomic broadcast does *not* necessarily imply FIFO order!
- Atomic causal broadcast
  - Combination of atomic broadcast with causal order

Summary of specifications

- *Validity:* If a correct process broadcasts a message m, then it eventually delivers m.
- *Agreement:* If a correct process delivers a message m, then all correct processes eventually deliver m.
- *Integrity:* For any message m, every correct process delivers m at most once, and only if m was previously broadcast by $sender(m)$.

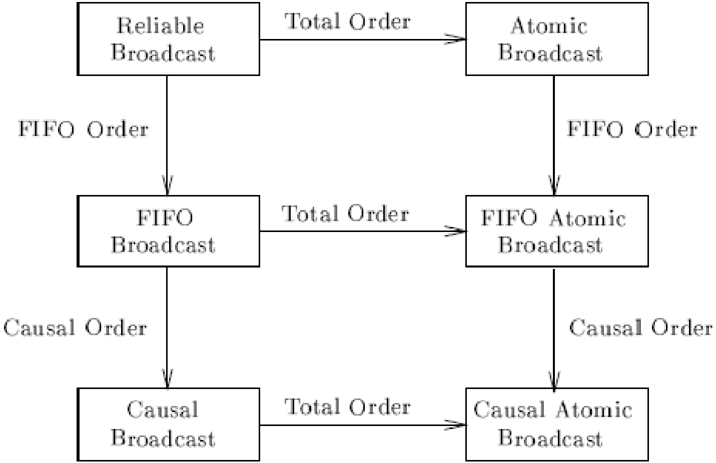
- *FIFO Order:* If a process broadcasts a message m before it broadcasts a message m', then no correct process delivers m' unless it has previously delivered m.
- *Causal Order:* If the broadcast of a message m causally precedes the broadcast of a message m', then no correct process deliveres m' unless it has previously delivered m.
- *Total Order:* If correct processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

Summary of specifications

- Reliable broadcast = Validity + Agreement + Integrity
- FIFO broadcast = Reliable broadcast + FIFO order
- Causal broadcast = Reliable broadcast + Causal order

- Atomic broadcast = Reliable broadcast + Total order
- FIFO atomic broadcast = FIFO broadcast + Total order
- Causal atomic broadcast = Causal broadcast + Total order

Summary of specifications



Overview

Reliable broadcast with Byzantine faults

(First) Problem: Faulty sender may send inconsistent values

- Simple echo algorithms (for crashes) makes sure that some value is delivered
- But not that all delivered values are equal
- . . . think about a solution!

Reliable broadcast with Byzantine faults

(First) Problem: Faulty sender may send inconsistent values

- Simple echo algorithms (for crashes) makes sure that some value is delivered
- But not that all delivered values are equal
- . . . think about a solution!

(First) Solution: Collect votes from all processes

- There are at least $n - f$ correct processes
- Accept value v if more than half of all correct processes vote in favour of v
- More than $f + \frac{n-f}{2} = \frac{n+f}{2}$ votes implies votes from more than half of all correct processes (without knowning which processes are correct)

Reliable broadcast with Byzantine faults

Implication (without synchrony):

- f faulty processes might not send anything

Reliable broadcast with Byzantine faults

Implication (without synchrony):

- f faulty processes might not send anything
- Any algorithm should to do something after receiving messages from only $n - f$ processes

Reliable broadcast with Byzantine faults

Implication (without synchrony):

- f faulty processes might not send anything
- Any algorithm should to do something after receiving messages from only $n - f$ processes
- If sender is correct, a positive vote should always be made, i.e., $n - f > \frac{n+f}{2}$ must hold
- This is equivalent to $n > 3f$

Reliable broadcast with Byzantine faults

Implication (without synchrony):

- f faulty processes might not send anything
- Any algorithm should to do something after receiving messages from only $n - f$ processes
- If sender is correct, a positive vote should always be made, i.e., $n - f > \frac{n+f}{2}$ must hold
- This is equivalent to $n > 3f$

For tolerating f Byzantine faults, we (usually) need at least $3f + 1$ *processes*

Echo broadcast with Byzantine faults

- $n = 3f + 1$ processes (or generically $f = \lfloor (n-1)/3 \rfloor$)
- What does it guarantee?

1. G sends (initial, G, m) message to all processes.
2. When a process receives the first (initial, G, m) message from G, it sends an (echo, G, m) message to all processes. All subsequent (initial, G, m') messages are ignored.
3. If a process receives an (echo, G, m) message from more than $\frac{n+f}{2}$ distinct processes, then it accepts the message *m* from G.

(from: Sam Toueg: Randomized Byzantine Agreements, 1984)

Echo broadcast with Byzantine faults

(in Toueg-Paper: correct = “proper”, deliver = “accept”)

If a process G broadcasts the message *m* with echo broadcast then:

1. The messages delivered by correct processes are identical
2. If G is correct, then all the correct processes deliver *m*

- *Note: Property 1 guarantees only that the delivered messages are identical (i.e., if two messages are delivered, they are the same). If G is faulty, it is possible that some processes deliver m, and others do* ***not*** *deliver m.*

Reliable broadcast with Byzantine faults

For reliable broadcast:

- Requirement: If one correct delivers some message m, all correct processes must eventually deliver m (even if $sender(m)$ is faulty)

Reliable broadcast with Byzantine faults

For reliable broadcast:

- Requirement: If one correct delivers some message m, all correct processes must eventually deliver m (even if $sender(m)$ is faulty)

- Algorithm by Bracha 1984
  - Use "more than $\frac{n+f}{2}$ votes" argument to ensure that at most one message can be delivered.
  - Use additional step to guarantee "all-or-nothing" behaviour if sender is faulty.

Reliable broadcast (Bracha 1984)

- $n = 3f + 1$, operation Broadcast(v)

1. (by transmitter only): send (initial, v) to all processes.
2. Wait untill receive for some v:
   - one (initial, v) message
   - or $> \frac{n+f}{2}$ (echo, v) messages
   - or $(f + 1)$ (ready, v) messages

   send (echo, v) to all processes
5. Wait untill receive for some v:
   - $> \frac{n+f}{2}$ (echo, v) messages
   - or $(f + 1)$ (ready, v) messages

   send (ready, v) to all proc.
6. Wait untill receive for some v:
   - $2f + 1$ (ready, v) nessages

   accept v

(from: G. Bracha: An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol, 1984)

Reliable broadcast (Bracha 1984): Properties

1. If p is correct, then all the correct processes accept the value of its message.
2. If p is malicious, then either all the correct processes accept the same value, or non of them will accept any value from p.

Reliable broadcast (Bracha 1984): Proof

1. **Lemma 1:** If two correct processes *r* and *s* send *(ready,v)* and *(ready,u)* messages, respectively, then $u = v$.
2. **Lemma 2:** If two correct processes *p* and *q* accept the values *v* and *u*, respectively, then $u == v$.
3. **Lemma 3:** If a correct process *p* accepts the value *v*, then every other correct process will eventually accept *v*.
4. **Lemma 4:** If the transmitter *p* is correct and it sends *v*, then all correct processes will accept *v*.

Summary of broadcast echo algorithms

Crash faults

- Simple echo

Byzantine faults

- Simple echo + threshold
- “Two-phase echo” + thresholds

Summary of broadcast echo algorithms

Crash faults

- Simple echo for implementing **reliable broadcast**

Byzantine faults

- Simple echo + threshold for implementing **best effort broadcast**
- “Two-phase echo” + thresholds for implementing **reliable broadcast**

Overview

1. Basics: Safety and Liveness
2. Broadcast
   - Best-effort broadcast
   - Reliable broadcast (rbcast)
   - FIFO broadcast (fbcast)
   - Causal broadcast (cbcast)
   - Atomic broadcast (abcast)
3. Reliable broadcast with Byzantine faults
4. Implementing atomic broadcast
   - Sequencer-based approaches
   - Token-based approaches
   - Communication history-based approaches
   - Consensus-based approaches

Implementing atomic broadcast

Classification of well-known algorithms according to Défago et al.:

- Sequencer-based approaches
  - Selected node determines message order
  - Fixed or moving sequencer
- Token-based approaches (“privilege-based”)
  - Serialisation of all messages: only token owner may send
  - Active token / passive token
- “Communication history”
  - Sender assigns time stamps, receiver sorts messages
- Consensus-based approaches
  - Order determined by distributed consensus algorithms

Sequencer-based approaches

Sequencer defines message order

- Variant “UB”:
  - Sender sends message to sequencer (unicast)
  - Sequencer sends message to all nodes (broadcast)
  - High load / bottleneck at sequencer
- Variant “UUB”:
  - Sender sends message to sequencer (unicast)
  - Sequencer assigns sequence number to sender (unicast)
  - Sender sends message + sequence number to all nodes (broadcast)
  - Less work for sequencer, but higher latency
- Variant “BB”:
  - Sender sends message to all nodes (including sequencer; broadcast)
  - Sequencer assigns sequence number and sends it to all nodes (broadcast)

In all cases: What if sequencer crashes?
Additionally for UUB and BB: What if sender crashes?

Token-based approaches

- Only token owner may send messages
- Circulating token
  - Unused token is passed to next node automatically
  - Efficient if many nodes broadcast messages
  - Token circulates even if noone wants to broadcast messages
- Requesting the token on demand
  - Token is passed only upon request
  - Efficient if only one / few nodes want to broadcast
  - Many senders: High overhead for searching for the token

Communication history-based approaches

- Order is defined by the sender
- Messages usually carry a (physical or logical) timestamp
- Recipients observe messages generated by other processes and their timestamp
  - i.e., the “communication history”
- Process can deliver a message *m* as soon as it knows that there cannot be any other message that needs to be delivered before *m*

Communication history-based approaches

- Order is defined by the sender
- Messages usually carry a (physical or logical) timestamp
- Recipients observe messages generated by other processes and their timestamp
  - i.e., the “communication history”
- Process can deliver a message *m* as soon as it knows that there cannot be any other message that needs to be delivered before *m*

Simple examples:

- Synchronous system: Physical time stamps; delivery delayed by upper bound of clock error + transmission delay
- Asynchronous system: Round-robin approach: Deliver message 1 of $P_1, P_2, \ldots$, then message 2, etc.

Consensus-based approaches

Distributed consensus algorithms:

- Each node proposes some value
  In this case: next message ID / list of n message IDs
- Result of consensus: One of the proposed values Here: Next message / next n ordered messages

Next: transformation: abcast $\leftrightarrow$ consensus

Atomic broadcast and consensus

So far

- Simple solutations for bcast, fbcast, cbcast in asynchronous system
- Solution for uniform bcast if less than half of processes is faulty
- No solution for abcast in asynchronous system

Next step:

- Atomic broadcast and the consensus problem

Consensus

Definition of the *consensus problem:*

Each process (in a set of n processes) proposes a value, and all correct processes decide upon a common value, such that the following properties are satisfied:

- *Termination:* Every correct process eventually decides some value
- *Agreement:* No two correct processesess decide on different values v, v'
- *Validity:* If a correct process decides v, then v was previously proposed by some process.
- *Integrity:* No process decides twice

Note: some other definitions with minor differences exist!

Consensus variants

- *Uniform agreement:* If a process (whether correct or faulty) decides v, then all correct processes eventually decide v.
- *Uniform integrity:* If a process (whether correct or faulty) decides v, then v was previously proposed by some process.
- *Validity':* If all processes propose the same value v, then this is the only possible decision value
- *Validity":* If all correct processes propose the same value v, then this is the only possible decision value

Transforming atomic broadcast into consensus

- Algorithm for atomic broadcast also solves the consensus problem. . .
  - The transformation uses no timing assumption, so it works for any synchrony model

```
Uses: Atomic Broadcast (abcast)

upon Init:
   decided := false;

upon event < Propose, v > do:
   trigger < abcast.Broadcast, v >;

upon event < abcast.Deliver, u > do:
   if not decided then
      trigger < Decide, u >;
```

Transforming rbcast + consensus into abcast

```
Uses: Reliable Broadcast (rbcast), MultiConsensus (cons)

upon Init:
   R_delivered := ∅; A_delivered := ∅; k := 0

upon event < ABroadcast, m > do:
   trigger < rbcast.RBroadcast, m >;

upon event < rbcast.RDeliver, m > do:
   R_delivered := R_delivered ∪ {m},

repeat forever:
   A_undelivered := R_delivered - A_delivered
   if A_undelivered ≠ 0 then
      k := k + 1
      trigger < cons_k.Propose, A_undelivered >
      wait

upon event < cons_k.Decide, msgSet >:
   forall v in (msgSet - A_delivered) in deterministic order
      trigger < ADeliver, v >
   A_delivered := A_delivered ∪ msgSet
   resume wait
```

Summary

1. Basics: Safety and Liveness
2. Broadcast
   - Best-effort broadcast
   - Reliable broadcast (rbcast)
   - FIFO broadcast (fbcast)
   - Causal broadcast (cbcast)
   - Atomic broadcast (abcast)
3. Reliable broadcast with Byzantine faults
4. Implementing atomic broadcast
   - Sequencer-based approaches
   - Token-based approaches
   - Communication history-based approaches
   - Consensus-based approaches