Fault-Tolerant Broadcasts - Motivation

- We have seen that if some kind of broadcast primitive existed in asynchronous systems, Consensus would be solvable!

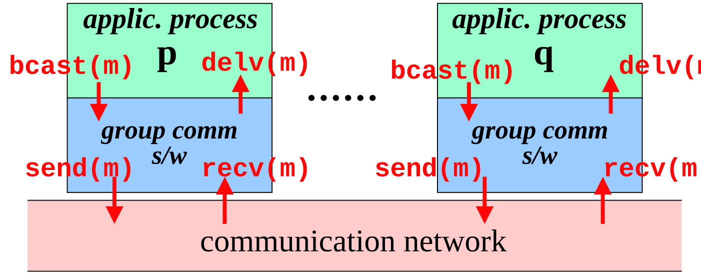
  Broadcasts are important for fault-tolerance in distributed systems.
- Broadcasts are communication primitives that simplify the design of distributed systems (replication, group-ware, …).

Hard to design/implement certain types of broadcasts. Problem complicated by process/link failures. Usually, the stricter (and more useful), the harder broadcasts are… So, what exactly do we need?

pt2pt comm primitives (send/recv)	Group comm primitives(bcast/delv)
+ easy to support	– hard to support
+ cheap to provide	– expensive to provide
– hard to work with	+ easy to work with

Fault-Tolerant Broadcasts - Architecture

Our goal is to provide the "group communication s/w" that implements **`bcast/delv`** using the **`send/recv`** provided by the underlying network



<u>Assume</u>:
- Fixed group of (application) processes; senders from within group.
- Each broadcast message **m** is unique by tagging with two fields:
  - ***sender(m***) : the identity of its sender
  - ***seq#(m)*** : sequence no. of m in its sender

Fault-Tolerant Broadcasts

SYSTEM MODEL:

Asynchronous distributed systems

Failure assumptions:
- Processes may crash
- link failures possible

Point-to-point networks (represented as graphs - nodes: processes, edges: bi-directional comm. links)

Definitions for fault-tolerant broadcasts:

p broadcasts m : p invokes **`bcast(m)`**
[may not complete it due to a crash]

p delivers m : p <u>completes</u> execution of **`delv(m)`**

Fault-Tolerant Broadcasts

METHODOLOGY - **Modular protocol design:**

- Various broadcast protocols presented as a **hierarchy** of specifications and corresponding algorithms.
- Obtain algorithm for a stronger variation by using given weaker broadcast primitive as a "**black box**" **-** based on that primitive's specifications and not actual implementation! ("**transformations**")
- We'll describe **generic transformations**, which given <u>any</u> algorithm for some type of fault-tolerant broadcast, will produce an algorithm for a stronger type of fault-tolerant broadcast by:

  preserving the properties of the given (weak) broadcast

  introducing some additional properties

  Application processes must use the "group communication software/layer" as black box too - based on its properties (specs), not actual implementation in a certain system model!

Reliable broadcast - 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 eventually all correct processes deliver **m**.

} Liveness

- **Integrity**: For any message **m**, a *(?)* process delivers **m** at most once and only if ***sender*(m)** has previously broadcast **m**.

} Safety

Informally:

- the same (perhaps infinite) **set** of msgs is delivered by all correct processes [Agreement]
- that set includes all msgs broadcast by correct processes [Validity+Agreement]
- "spurious" msgs are not included in that set [Integrity]

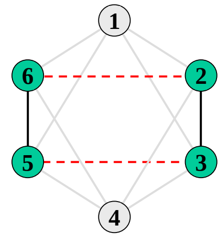
What is the possible outcome when a process fails while broadcasting m?

Reliable broadcast - Algorithm

The Reliable Broadcast algorithm to be presented here is the basis for all other algorithms to be presented later on… others use it directly or indirectly. So, it is important to make clear when this algorithm works!

No-partition assumption: Any two correct processes are connected via a path consisting only of correct processes and correct links

I.e. network connections have enough redundancy, so that failures do not disrupt communication between correct processes. Assumption necessary; otherwise, Reliable bcast and, hence, any other type of bcast is unsolvable.



In this network, the *no-partition assumption* is

- ***satisfied***, if we know that ≤ 2 processes and ≤ 1 link may be faulty.
- ***violated*** if we know that 2 processes and 2 links may be faulty.

Reliable broadcast - Algorithm

Recall (from models of distrib systems) the properties of **send/recv** primitives:

- *Safety*: q receives m from p at most once and only if p previously sent m to q.
- *Liveness*: if p sends m to q and q takes infinitely many steps (i.e. q correct), then q eventually receives m from p.

Diffusion Algorithm

To broadcast, a process p executes…
R_bcast(m):
  tag m with *sender(m)* and *seq#(m)*;
  send(m) to all neighbours including p;

R_delv(m) occurs as follows (every process p executes this)...
upon recv(m) **do**
  if p has not previously executed **R_delv**(m) **then**
    if *sender*(m)≠p **then send**(m) to all neighbours;
    R_delv(m);

any obvious optimisations?

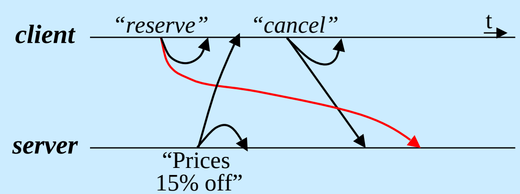
Correctness?

- *Validity*: by liveness of send/recv.
- *Agreement*: By no-partition assum + liveness of send/recv + induction.
- *Integrity*: By safety of send/recv + induction.

FIFO broadcast - Motivation

In Reliable broadcast, there are **no** requirements on **order** in which messages are delivered. This may result in "anomalies"...

Example: Delivery of a message canceling a flight reservation (on airline's server) before delivery of the original message making the reservation - airliner's server application may get "confused"!



Broadcast messages from the *same sender* must be delivered in some order consistent with the order they were generated (for delivery to reflect potential dependencies on sender).

FIFO broadcast - Specifications

FIFO Broadcast = Reliable Broadcast + FIFO Order

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**.

A Safety property.

Note:

*Suppose a process **p** broadcasts messages **m_1**, **m_2** and **m_3** in that order. Due to a transient failure of process **p** while it broadcasts **m_2**, a correct process **q** delivers **m_1** and **m_3** (in that order) but not **m_2**.*

Is this behaviour permitted by the specification of FIFO broadcast?

FIFO broadcast - Specifications

Consider the following alternative definition ...

FIFO Order: All messages broadcast by the same process are delivered to all processes in the order they were sent.

Is this definition correct?

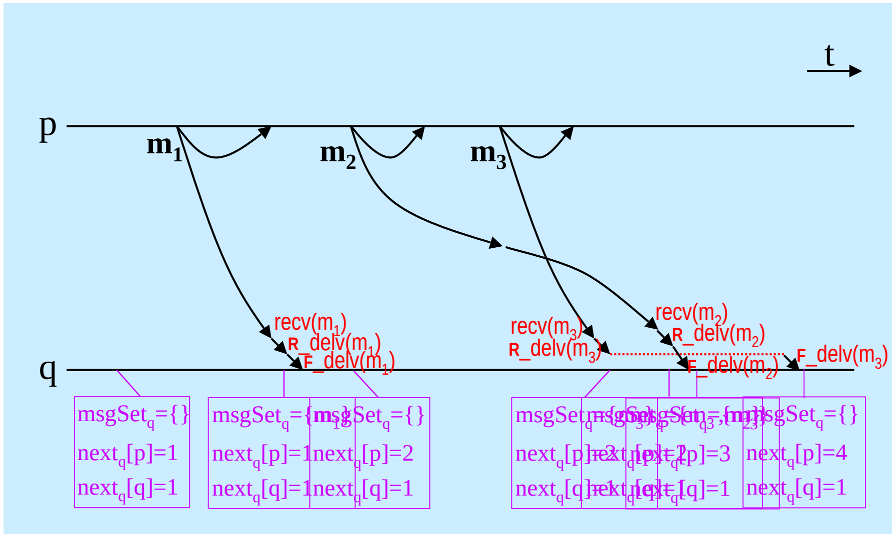
FIFO broadcast - Algorithm

We present a **generic transformation**, which given any algorithm for **Reliable broadcast** will provide **FIFO broadcast**: preserves the three properties of Reliable broadcast and adds FIFO delivery order.

```
Every process p executes the following:
Initialisation:
    msgSet := ∅;  // set of messages R_delv'ed but not F_delv'ed
    next[q] := 1 forall q;  // seq# of next m from q that p will F_delv
F_bcast(m):
    R_bcast(m);
upon R_delv(m) do
    s := sender(m);
    msgSet := msgSet ∪ {m};
    while ( ∃m' ∈ msgSet : sender(m')=s and seq#(m')=next[s] ) do
        F_delv(m');
        next[s] := next[s] + 1;
        msgSet := msgSet - {m'};
```

Relies only on *correctness* of R_bcast - needs no system model assumptions.

FIFO broadcast - Algorithm execution

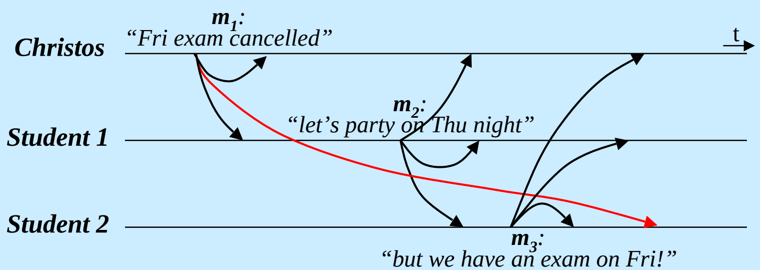


Causal broadcast - Motivation

FIFO Order does not preclude all anomalies due to bizarre order of delivery...

Example: The “**newsgroup anomaly**”

Use group communication primitives to implement newsgroup software. To post an article, a user **F_bcasts** it to the group. The article is delivered to the user’s newsreader application as soon as it arrives at his/her local site.



- FIFO order is satisfied (trivially)
- What is wrong then? m_2 depends on m_1, yet Student 2 delivers m_2 before delivering m_1. **m_1 *causally precedes* m_2**, i.e. **m1 → m2**

Causal broadcast - Specifications

Causal Broadcast = Reliable Broadcast + Causal Order

Causal Order: If the broadcast of a message **m** causally precedes the broadcast of message **m’**, then no *correct* process delivers **m’** unless it has previously delivered **m**.

A Safety property.

Causal Order ⇒ FIFO Order , but

FIFO Order ⇏ Causal Order

So, Causal Order = FIFO Order + ?

Causal broadcast - Specifications

Causal Order = FIFO Order + Local Order

Local Order: If a process delivers a message **m** before broadcasting a message **m’**, then no correct process delivers **m’** unless it has previously delivered **m**.

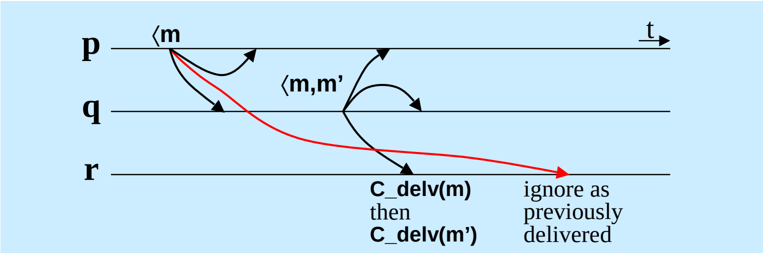
A Safety property.

Causal broadcast - Algorithm

Again, this is a **generic transformation**, which given any algorithm for **FIFO broadcast** will provide **Causal broadcast**.

```
Every process p executes the following:
Initialisation:
    rcntDlvs := ⊥;  // sequence of msgs that p C_delv'ed since its
                    // previous C_bcast
C_bcast(m):
    F_bcast( ⟨rcntDlvs||m ⟩);  // append m at end of rcntDlvs
    rcntDlvs := ⊥;
upon F_delv(⟨m1,m2,…,mn ⟩) do
    for i := 1 .. n do    // order : important!
        if p has not previously executed C_delv(mi) then
            C_delv(mi);
            rcntDlvs := rcntDlvs || mi;
```

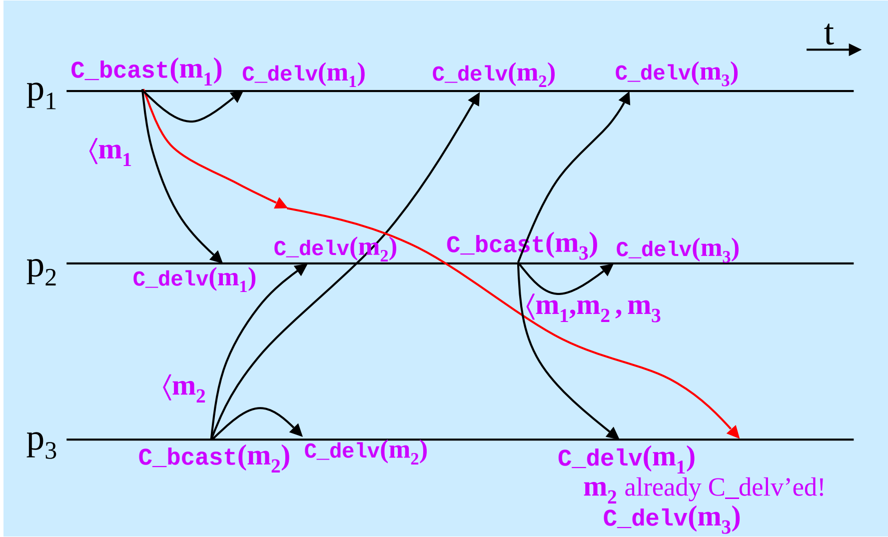
Causal broadcast - Example



- (+) This is a **non-blocking** algorithm (transformation), i.e. C_delivery of messages is never postponed until some condition is satisfied.
- (−) This is obviously not a practical protocol due to the **size of messages** transmitted (sequences of msgs). This the price to pay for not blocking!

  Practical protocols (e.g. ISIS - see later on) transmit not sequences of msgs, but sequences of msg IDs. However, they delay C_delivery of a msg until all its causal predecessors have arrived and been delivered.

Causal broadcast - Example

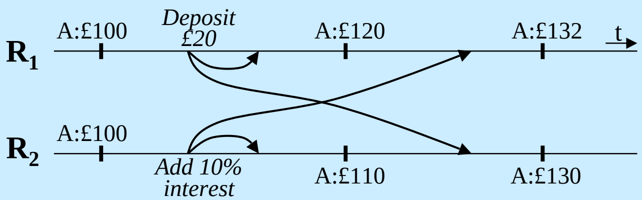


Atomic broadcast - Motivation

Even Causal Order is not enough to ensure absence of anomalies...

Example: "**Replicated bank account**"

Use group communication primitives to implement a replicated database for a bank, in two sites. Bankers may work on any of the sites. A request to update an account in the database is broadcast to both replicas.



Although replicas identical at start, they diverge at the end.

- Causal Order satisfied (trivially).
- Problem: to guarantee identical replicas at the end, must ensure that all updates are delivered in **same order**, even if not causally related.

Atomic broadcast - Specifications

Atomic Broadcast = Reliable Broadcast + Total Order

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'**.

In Atomic broadcast…

- the same (perhaps infinite) **sequence** of msgs is delivered by all correct processes [Agreement + Total Order]

Compare with specifications of Reliable broadcast…
The only difference is: "**sequence**" instead of "**set**"

This innocuous-seeming difference makes a huge difference in the kind of systems in which these two types of broadcasts can be implemented!

Atomic broadcast & Consensus

We have seen that Consensus can be solved using some kind of broadcast. In fact, that is Atomic broadcast. In other words, the problem of **Consensus** can be **reduced** to the problem of **Atomic broadcast**.

Consensus impossible in asynchronous systems ⇒ Atomic broadcast impossible in asynchronous systems

In addition, it has been shown (by Chandra & Toueg) that the problem of **Atomic broadcast** can be **reduced** to the problem of **Consensus**. I.e. given an algorithm for Consensus, Atomic broadcast can be implemented.

Atomic Broadcast ⇔ Consensus

Atomic broadcast

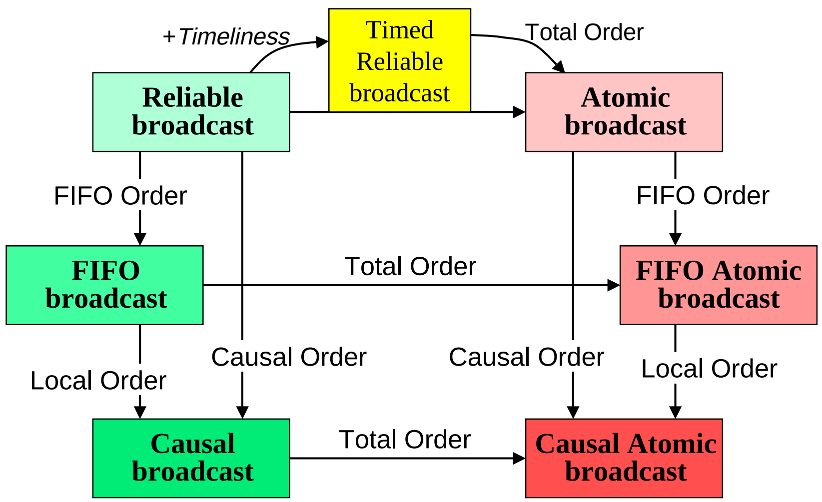
- **Reliable bcast** implementable in asynchronous systems
  [for any # of process/link failures, given no-partition]
- **Atomic bcast** not implementable in asynchronous systems
  [even for one process failure]
- We **cannot** use the "Diffusion Algorithm" for Reliable broadcast (as it is) to transform it into an Atomic broadcast algorithm!

Total Order $\not\Rightarrow$ **Causal Order**
$\not\Rightarrow$ **FIFO Order**

So, we have two more, even **stronger**, broadcasts:

- **FIFO Atomic** bcast: Reliable bcast + FIFO Order + Total Order
- **Causal Atomic** bcast: Reliable bcast + Causal Order + Total Order

Relationship among Broadcast types



Algorithm transformations?

Timed Reliable Broadcast

To construct an Atomic broadcast algorithm (by transformation), we need **Timed Reliable Broadcast = Reliable broadcast + Timeliness**.

Timeliness: There is a known constant Δ such that if a message **m** is broadcast at time **t**, then no correct process delivers **m** after time **t+Δ**.

Timeliness can be achieved in **synchronous point-to-point** networks, where processes/links may **crash**.

The "Diffusion" Algorithm for Reliable Broadcast presented earlier does, in fact, satisfy Timeliness when executed in synchronous networks.

What is the value of Δ?

Timed Reliable Broadcast

Recall (from "models" lecture): properties of **synch. point-to-point** networks:

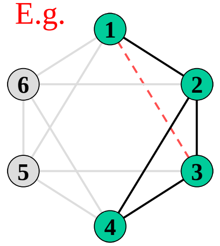
- There is known upper bound on msg **transmission delay** over a comm link which connects <u>directly</u> two processes: δ
- There is known upper bound on time required for a process to execute a **local step**. Here, we consider the time to process a msg as negligible: **0**

Recall: "*Diffusion Algorithm*" requires the **no-partition** assumption - still required in the case of synchronous systems. To estimate the value of Δ...

<u>Assume:</u>

- ***f*** : max # of faulty processes
- ***k*** : max # of faulty links
- **d** : worst shortest path between any two correct processes, when ***f*** faulty processes and ***k*** faulty links

E.g.



f=2, k=1 $\Rightarrow$ **d = 3**

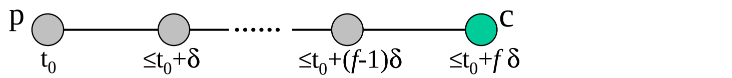
[For <u>all</u> failure combinations, calculate shortest possible paths between <u>any two</u> correct processes;
***d** = longest of them!]*

Timed Reliable Broadcast

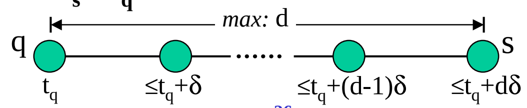
In a synchronous network where a max of ***f*** processes may crash and a max of ***k*** links may fail, the **"Diffusion" algorithm** for Reliable Broadcast satisfies **Timeliness** with $\Delta = (f+d)\delta$.

Why ? **(Δ represents the "worst case" delay scenario)**

❶ If a process **p** **bcasts m** at time t_0, then the **first correct** process **c** that delivers **m** (if one exists), does so at time $t_c \le t_0 + f\,\delta$



If a **correct** process **q delivers m** at time t_q, then <u>every</u> correct process **s** does so at time $t_s \le t_q + d\delta$



Atomic Broadcast - Algorithm

Given an algorithm for **Timed Reliable Broadcast** in synchronous systems, we can use a simple transformation to get **Atomic Broadcast**...

Every process p executes the following:

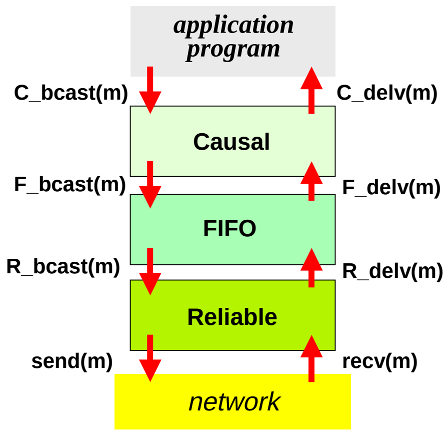
A_bcast(m)**:**
    tag **m** with *ts*(m) := current real time;
    RΔ_bcast(m**)**;

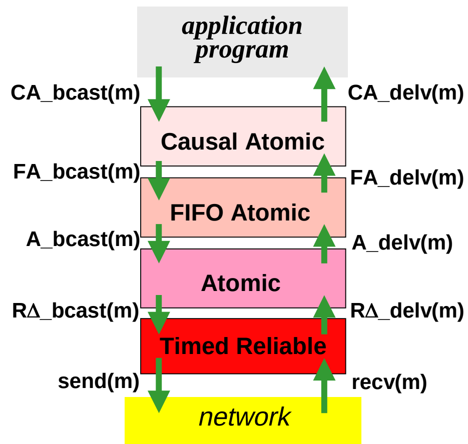
upon RΔ_ delv(m**) do**
    schedule A_ delv(m**) at time** *ts*(m)+Δ;

<u>Note:</u> if two deliveries scheduled for the same time, then deliver in order of sender's identity: *sender*(m)

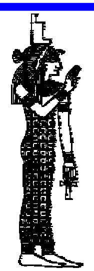
The above algorithm transforms <u>any</u> algorithm that satisfies **Timeliness** into an Atomic Broadcast **preserving** Agreement, Validity, Integrity and also **FIFO Order** and **Causal Order**.

Broadcast algorithms (transformations)





ISIS - practical Group Communication



ISIS is a toolkit developed by Ken Birman and others at Cornell Univ. It facilitates the construction of **fault-tolerant** distributed applications by providing a range of **group communication** primitives. It is now marketed commercially. Has been used for the development of s/w for the NY and Zurich Stock Exchanges. It supports the following protocols:

- **FBCAST** : FIFO Broadcast (group **multi**-cast)
- **CBCAST** : Causal Broadcast (group **multi**-cast)
- **ABCAST** : Atomic Causal Broadcast (group **multi**-cast)

ISIS gives to the application programmer the abstraction of virtual synchrony: Application behaviour perceives group communication activities (broadcasts, process failures) as if scheduled in sequential order, the same in all processes. In fact, ISIS is designed for asynchronous systems and processes are executed concurrently and asynchronously.

Ref: *"Lightweight Causal and Atomic Group Multicast", ACM Trans. on Computer Sys., 9(3), 1991*

ISIS - System Model

- Processes form **groups** which are the destination for multicasts. A process has to explicitly **join** a group (can be member of >1 groups).
- Processes **multi-cast** messages to groups they are members of.
- Processes fail by crashing detectably **- failstop**. A faulty process is removed from the group(s) it is member of.
- Processes learn of group membership through the **view** mechanism. A view of a process group ***g*** is a list of its members' IDs. A **view "history"** for group ***g*** is an infinite sequence

  $view_0(\mathbf{g}), view_1(\mathbf{g}), \ldots, view_n(\mathbf{g}), \ldots$ where:

  - $view_0(\mathbf{g}) = \varnothing$
  - $\forall i>0$, $view_i(\mathbf{g}) \subseteq P$ (set of all processes in the system) $view_i(\mathbf{g})$ and $view_{i+1}(\mathbf{g})$ differ by the addition / subtraction of exactly one process

If correct process's **p** current view of **g** is $\mathbf{v_p(g)}$, then $\mathbf{p} \in \mathbf{v_p(g)}$. If $\mathbf{q} \in \mathbf{v_p(g)}$, then **p** and **q** have "seen" the same sequence of views of group **g** from the moment they where both members of **g** up to (including) $\mathbf{v_p(g)}$.

Virtual Synchrony

- Address expansion: **Group ids** are used as the destination for **multicasts**. The protocols expand group ids into **destination lists** and deliver messages in such a way that:
  - Delivery atomicity and order

    The protocols obey the *Validity*, *Agreement* and *Integrity* properties of Reliable broadcasts (within a group - multicasts). Either all correct processes in the group eventually deliver a message or (only if the sender fails) none of them does. In addition, CBCAST provides Causal order and ABCAST provides *Total order consistent with Causality*.
  - Virtual Synchrony

    If process **p** (correct or faulty) multicasts **m** to **g** "in view" $\mathbf{v_i(g)}$ and there is correct process **q** that delivers **m** in view $\mathbf{v_{i+k}(g)}$ ($k \geq 0$), then every correct process in **g** delivers **m** in $\mathbf{v_{i+k}(g)}$; in that case $\mathbf{p} \in \mathbf{v_{i+k}(g)}$. *What if p faulty?*

  These properties require that processes must not deliver multicasts from a process which is not member of their **current view** (removed because failed).

CBCAST Protocol - Vector Clocks

Assume: process participates in single group **g**.

Each process $\mathbf{p_i}$ maintains a vector clock $\mathbf{VT(p_i)[j]}$, for all p_j in g. Initially, $\forall j : VT(p_i)[j] = 0$;

Before each event ***send***(m) at $\mathbf{p_i}$, $\mathbf{VT(p_i)[i] := VT(p_i)[i] + 1}$ and **m** is timestamped with $\mathbf{VT(p_i)}$. **[What does $VT(p_i)[i]$ represent ?]**

After ***C_deliver***(m) at $\mathbf{p_j}$, the process updates its local vector clock:

$$\forall \mathbf{k} : \mathbf{VT(p_j)[k] := max\{ VT(p_j)[k],\ VT(m)[k] \}} \qquad \mathbf{(*)}$$

[What does $VT(p_j)[k]$ represent ?]

Recall that vector clocks represent causality precisely:

$\mathbf{m \rightarrow m'}$ ***if and only if*** $\mathbf{VT(m) < VT(m')}$

where

$\mathbf{VT_1 \leq VT_2}$ ***if and only if*** $\mathbf{\forall i: VT_1[i] \leq VT_2[i]}$

$\mathbf{VT_1 < VT_2}$ ***if and only if*** $\mathbf{VT_1 \leq VT_2}$ **and** $\mathbf{\exists i: VT_1[i] < VT_2[i]}$

CBCAST Protocol - Algorithm

Every process p_i executes the following:

C_multicast(m):

$VT(p_i)[i] := VT(p_i)[i] + 1;$

tag **m** with *VT*(m) := $VT(p_i)$;

send(m**) to** *g*;

upon recv(m**) do**

s := *sender*(m);

if $p_i = s$ **then C_deliver(**m**)**;

else delay delivery of **m** until the following hold:

a) $VT(p_i)[s] = VT(m)[s] - 1$

b) $VT(p_i)[k] \geq VT(m)[k]$, $\forall k \in \{1,2,\ldots,n\} - \{s\}$

When m delivered by p_i, update $VT(p_i)$ as in (*)

- Delayed messages are kept in a **CBCAST delay queue**. This queue is **sorted by vector time**. Concurrent messages are ordered by id of sender.
- ISIS is designed on top of TCP (n*n connections per group); assumes that msg diffusion (required for reliability) is implemented at that level.

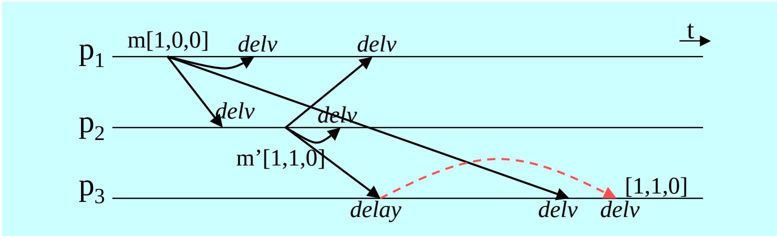
CBCAST Protocol - Comments

The main functionality of the protocol is implemented on receipt of a message:

<u>condition (a):</u> ensures that $\mathbf{p_i}$ has delivered all messages from **s** that precede **m**.

<u>condition (b):</u> ensures that $\mathbf{p_i}$ has delivered all those messages delivered by **s** before it sent **m**.

Since the ordering relation "→" imposed by vector clocks is <u>acyclic</u>, the protocol is **deadlock free**.



ABCAST Protocol

Uses a **token site** to impose total order.

Token holder process ***token*(g)** $\in$ ***view$_i$*(g)**

Each message is uniquely identified by ***uid*(m) = ⟨*sender*(m), *seq#*(m)**

To **ABCAST(**m**)**:

if *sender*(m) = *token*(g) **then** CBCAST(m) **else**

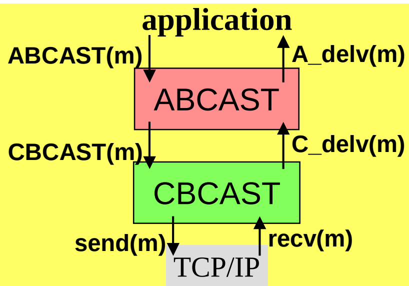
❶ CBCAST(m), but mark m "**undeliverable**". This may also delay causally following CBCASTs in delay queue of some processes.

❷ *token*(g) delivers **m** (as if it was CBCAST) and records *uid*(m).

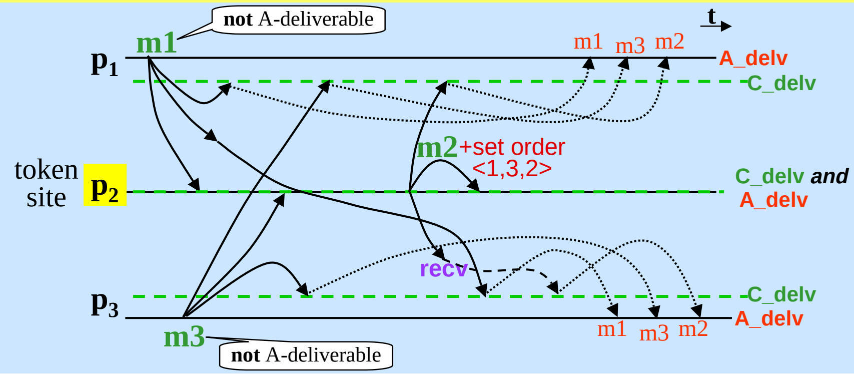
token(g) generates and CBCASTs msg typed **"set_order"** containing list of *uids* for ABCASTs it has delivered, in the order it has delivered them.

On receipt of a **m'="set_order"**, a process **p** ≠ *token*(g) places **m'** in the local CBCAST delay queue. Eventually **all** ABCASTs referred to in **m'** (its causal predecessors) are received by p. Concurrent ABCASTs are re-ordered in queue as indicated by **m'** and are marked "**deliverable**" (order must still respect VTs).

"**Deliverable**" ABCAST msgs are delivered from the **front** of the queue.



ISIS ABCAST



ABCAST Protocol - Comments

Implications of the **FLP result** :

- **Token holder** does not respond - has it *crashed* or *slow*? Correct processes cannot deliver delayed ABCAST without a “set_order” msg from token holder - **blocked**!
  - ☞ There is a **Failure Detector** in the system, which uses empirical **timeouts** (partially synchronous?) to detect (suspect) crashed processes. If a process decided faulty and is in fact correct, it is forced to re-join the group!
- Correct process **p** does not receive an ABCAST msg **m’** referenced by some “set_order” msg **-** **blocked**! Shall **p** wait *longer* for **m’** or has ***sender*(**m’**)** *crashed*? In the latter case, can **m’** be retrieved from other process(es)?
  - ☞ There is a protocol (not presented here) to update group **views** according to the **Virtual Synchrony** requirements. In the case of decided **process failure**, this protocol is initiated to **flush** any “transient” msgs of correct processes and any msgs of faulty processes that have been received by just a subset of the surviving processes; then, the new view is installed!