University of Bologna

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

Engineering Bologna Campus

Class of

Computer Networks M

Group issues and policies

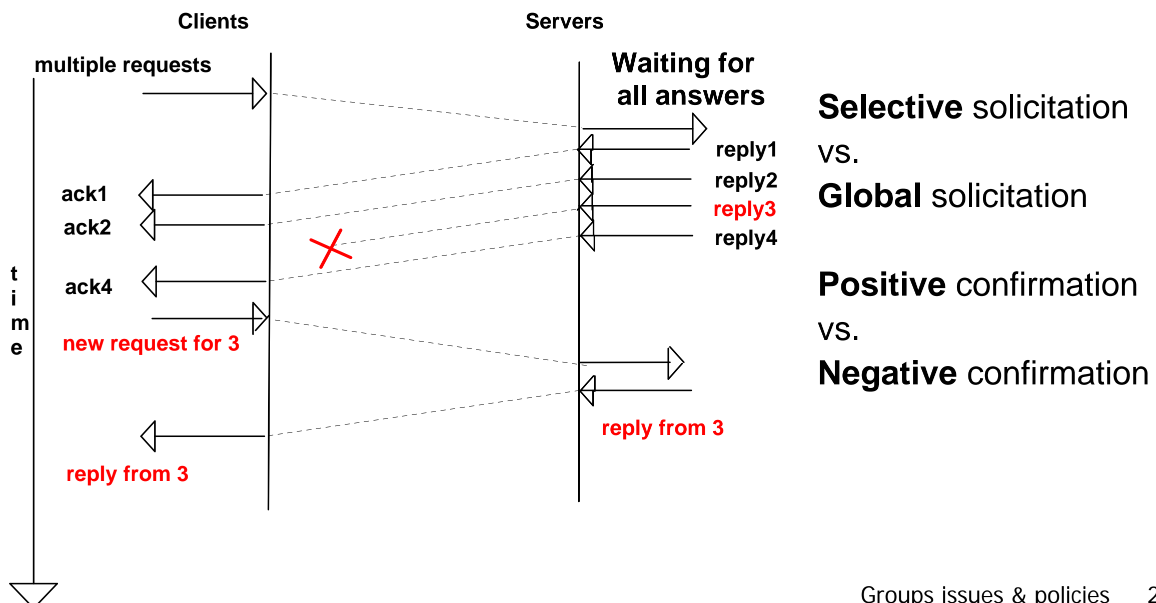
Antonio Corradi

Academic year 2015/2016

GROUP COMMUNICATION

Semantics: use of selective retransmissions? How many times?

primitive semantic depends on these choices



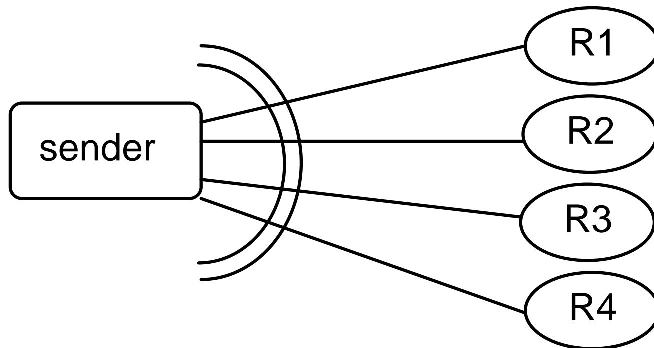
Selective solicitation vs. **Global** solicitation

Positive confirmation vs. **Negative** confirmation

GROUP COMMUNICATION

MULTICAST SEMANTIC

The **multicast** action could make the multiple group sending operations atomic, but they can try to associate a different and more suitable meaning



Motivations of the interest

- *object copy location inside a system*
- *fault tolerance*
- *use of data replication and streaming*
- *multiple changes on group entities*

GROUP COMMUNICATION

TWO aspects of MULTICAST SEMANTICS are intertwined and can be untangled

Reliability

group members message reception
reliable ⇨ guaranteed delivery
unreliable ⇨ only 1 attempt (Chorus)

Atomicity

message reception for all group members
with possible **different ordering** for different actions

The two aspects can and must be considered in separation

Essential Element

We must think not only to the **semantics of a single action**, but also to **message ordering in a multiple action occurrence (and consider their synchronization)**

RELIABLE MULTICAST

Reliability ***can be achieved if*** **some occurrences cause no problems**

- **sender crash**
- **receiver crash**
- **message omission**

Necessity of fault **identification** and **recovery** through ***monitoring*** of **multicast** and **group** actions

- **check of every ongoing communication**
- **eventual retransmissions**
- **removal of failed components**
- **protocol to re-enter in the group**

The additional costs for **identification** and **recovery** must be considered ***and they apply in case of failures***

RELIABLE MULTICAST

Implementation

- Dispatch all message to the group members support and **delay** before passing them to the applications
  timeout and **retransmission** (who checks the protocol?)
- **How long** to wait? problems with efficiency
- If **controller** fails?
  Quis custodiet ipsos custodes? **(Juvenal / Giovenale)**

hold-back → the support holds a message until it is sure that all previous others reached the destination in order

In case of dense numbering, a message is delayed until all previous ones appeared (if is the number 3, it must appear after and 2)

negative ack → the support sends an ack only in case of losses, to highlight those events (in selective way)

ATOMICITY VS. NO ORDERING

The other aspect of ATOMICITY is connected to semantic connected properties

with **atomicity** we focus on the **reception order** of messages by **any alive members of the group**

In distributed systems sometimes we are not so interested in obtaining a very tight synchronization of copies

No Ordering → the multicast messages coming from any sending process to all receivers can present a different ordering in any copy

The No ordering policy is very nice to support

- It has no cost and you do not have to synchronize copies in any way and they are free of operating on their own

FIFO ORDERING

We have many situations in which we want to require some connections between copy scheduling

FIFO Ordering → from the same sending process to all receivers for a sequence of successive multicast messages

- In case of FIFO ordering, **two multicast messages from the same sender** reach **any group member in the same order**
- For example, m1 and m2 from S1, and m3 and m4 from S2 reach everyone respecting sending order of the two senders
- many sequences are compatible m1 m2 m3 m4, m1 m3 m2 m4, m1 m3 m4 m2, m3 m4 m1 m2, m3 m1 m2 m4, …

We can use supports that already guarantee FIFO

Otherwise→ we need to achieve it

An easy way is message numbering for that specific sender

FIFO ORDERING LIMITATIONS

Compliance with FIFO ordering guarantees that every message to the group from the same sender (and its requests) are received in the same order in which are sent from the group (only related with same sender multicasts)

Compliance with FIFO ordering do not guarantees a feature that we tend to consider considering more than one sender

- A sends a news Na
- B receives the news and sends a response to Nb
- C receives first Nb then Na (Nb before Na)
- D receives first Na then Nb (Na before Nb)

We need to consider cause/effect relationships between different (two or more) senders

CAUSAL ORDERING

CAUSE-EFFECT ordering can connect events from different senders process

CAUSAL ordering → events that are correlated with a cause-effect relationship outside the group must be acknowledged by the group and the group must achieve consistency about them (to be delivered to everyone)

first the cause than the effect (Cause before Effect)

In case of CAUSAL ordering, two multicast messages in the **causal relationship** must be considered in the right order from everyone

For example, m1 and m2 from S1, and m3 and m4 from S2, and m1 causes m3. So they must reach copies respecting FIFO and CAUSAL ordering. Many sequences are compatible

m1 m2 m3 m4, m1 m3 m2 m4, m1 m3 m4 m2 NOT m3 m1 m4 m2

There are no supports that guarantee CAUSAL ordering

How can we guarantee it?

CAUSAL ORDERING

Compliance with ***CAUSAL ordering*** guarantees that messages from different senders in cause-effect relationship are received in the causal order by the group

Compliance with ***CAUSAL ordering*** for just one sender is similar to FIFO and it is easy to implement

Compliance with ***CAUSAL ordering do not*** catch **real world situations** that we tend to take for granted in case of more then one operations

- A requests an action to Na; B requests an action to Nb
- These actions are not related
- C receives first Nb and then Na, D receives first Na then Nb

So copies have different internal decisions of scheduling

ATOMIC ORDERING

No external relations imposes a scheduling, but the group should act in a coordinated and reasonable way, with all the members the operate in the same order

- ***ATOMIC ordering*** guarantees that all messages are received in the same order by all group members (so related actions can occur in the same order in all copies)
- Often no predetermined order is likely, **but it is necessary to agree on one and it should be the same for all**

If a copy C decides to receive first Nb then Na, all copies must follow that decision

- *Nb may ask to compute an interest on a bank account,*
  Na intends to make a withdrawal

Obviously, many different **atomic orderings exists** that we can consider with group operations

ATOMIC ORDERING

In a distributed environment the introduction and the and enforcing of orderings is costly

(coordination between group entities or numbering support)
and tend to enforce it only when necessary
Minimum cost: no ordering → each one group member work in a free and independent way

FIFO and CAUSAL ordering ore orderings that we tend to enforce only for some specific events in the system

Partial orderings

ATOMIC ordering is an ordering that we tend to enforce on every event within the group in the system

Total or global ordering

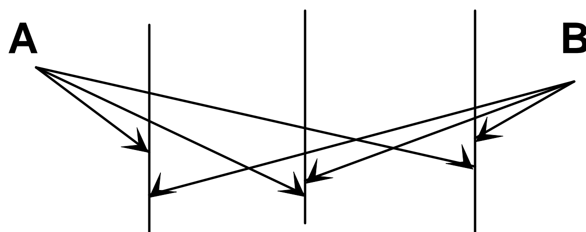
ATOMIC ORDERING

Obviously, given a group and a set of events coming from outside, we may have many different atomic orderings
How many?

ATOMIC orderings

Among many atomic orderings, some of them can follow **CAUSAL** and **FIFO** ordering, some only **FIFO**, some only **CAUSAL**, and some of other none of them

Costs for atomic orderings can be very different



MULTICAST ORDERING

FIFO multicast with ordering only from the same sender

messages from the same sender arrive with sender and ordering number and are delivered in that order (only for the same sender)

CAUSAL multicast with causal ordering (logic)

messages arrive and are delivered to the group in order to respect the relationship if the event **A** causes the event **B**

- **Lamport ordering**

ATOMIC multicast requires the same order (**any order**) for all messages to the group members

- **ATOMIC multicast imposes a total or global ordering for messages** that arrive to the group and must be delivered with the same order to all correct members of the group

We stress that an atomic order does not necessarily subsume the other two or anyone of them: any order can be decided inside the group

MULTICAST IMPLEMENTATION

ATOMIC multicast needs the same order

A simple way to implement it is to have an **entry element (a front end)** inside the group and it is the one that orders messages while they arrive and imposes an increasing numbers before sending them to all other members of the group

- **Every member receives messages and must send them in the agreed order to the application layers**

Disadvantages of this implementation are

Problems in case of ***coordinator fault*** (that defines and rules the policy)

Unfair management: the coordinator neighbors or preferred ones can be favored and scheduled always before others (that have made their requests before)

Low cost solution, but very unfair

→ ***mobile*** *coordinator (****circulating token****)*

Many other solution even more dynamic (Lamport, ring, etc.)

SYNCHRONIZATION

Synchronization means to impose orderings on events, typically …there are

Constraints on temporal ordering of some events inside a distributed system

It is necessary to provide a **consistent view** of the system to the **entire set** of communicating processes

Communication and Synchronization are often correlated, for example:

- *synchronizing sender / receiver* of a message
- check on cooperating activities
- *serialization of access to shared resources*
- N processes in access to a resource (mutually exclusive)

so, ***ordering on important events*** *must be enforced*

CLOCK SYNCHRONIZATION

Synchronization by using PHYSICAL TIME and PHYSICAL CLOCK

Unique time can be determined, if we assume that either

1) a **unique clock** is available on every node or

2) one **clock** for any node, and all of them perfectly in sync

This work assumption is perfectly admissible in concentrated or limited systems, but absolutely not feasible and easy to be granted in distributed and global environments

It has been defined **Universal Coordinated Time (UTC)** that is based on the transmission of the value and on local correction

Some systems are based on **coordination clock**

> A node verifies the time of all group members, computes the average and distributes it to all as the group time (Berkeley time)

CLOCK SYNCHRONIZATION

NTP - Network **T**ime **P**rotocol introduces a protocol based on UTC and on synchronization to achieve an **agreement on clocks**

NTP tries to overcome possible transmission delay of the common time through **statistical filtering policies** based on historic behavior of servers

- Starts with a higher **server hierarchy**, where every node transmit time to **lower-level neighbors** (its subtree)
- The **primary** nodes are more accurate and going farther from the **root**, accuracy decreases, of course
- The NTP tries to make actions to recover from **server** fault

The problem that can occur, by using clocks not perfectly in synch, is that an event happened afterwards maybe labeled and considered before an event that precedes it in time (that may produce a wrong time synchronization)

SYNCHRONIZATION

Synchronization via PHYSICAL TIME clashes with the difficulties of guaranteeing synching of clocks and a high implies a high overhead and also may present errors

- **Precision requires to coordinate continuously the clocks, and it is impossible to avoid conflicts and clock drifting with limited overhead**

Typically distributed ***synchronization is*** **not** based on complex algorithms of **physical clock** agreement but based on different strategies that can restrict the sync requirements and focus only on a **subset** of global system **events**

- The idea is to work ***on a subset of events*** (considering only some interesting events) and to create an agreement only on them
- The assumption of a limiting focus and a reduced group can **limit the overhead and protocol cost**

SYNCHRONIZATION STRATEGIES

Several Distributed Synchronization Methods

Ordering of logical time of Lamport

We can use timestamps (time indicator) to label relevant events and to order them → logical clocks and "happened before“ relationship

Token passing LeLann ring strategies

We can use authorizations, and the token can pass in a logical ring to order events

Events based on priority

We can use process priority to order correlated events

Used in real-time systems and unfair

LAMPORT RELATIONSHIP

Lamport aim at to **ordering some events in a distributed system, by excluding physical time**

Only **some events** are considered in the distributed system with a scenario constituted by **processes** that have their internal history and can exhibit a behavior based on two kinds of events:

1) local: local events

2) remote: interprocess event, generated by sending messages form one process to another process

We limit the actions of interest

The ordering must consider only some ‘relevant’ events and aims at creating a simple **ordering policy,** on which to eventually establish a **correct** synchronization with **adequate costs** and **not very expensive to implement**

HAPPENED-BEFORE RELATIONSHIP→

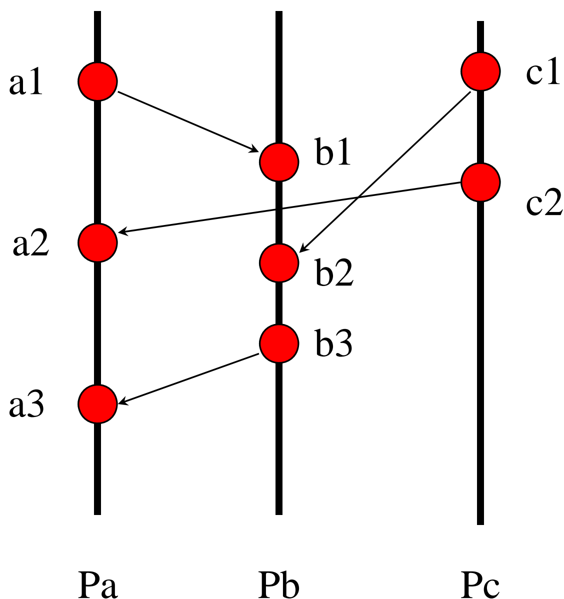
Events ordering for a **set of processes that communicate through message passing** based on cause-effect relationship introduced by process actions

1) If a and b are events of the same process and a occurs before b, then a → b (***local order***)
2) If a is the sending of a message of one process and b the receiving event within another process, then a → b (***communication interprocess order***)
3) If a → b and b → c, then a → c (***transitivity***)

The relation→ introduce a **partial ordering** in systems events and it exists only among some systems events and not available among all events (***it is not a total ordering***)

*Two events are **concurrent*** ⇈ *if* not a → b *and* not b → a

HAPPENED-BEFORE RELATIONSHIP→



a1 → a2, a1 → a3
a1 → b1, a1 → b2, a1 → b3

c1 → c2
c1 → b2, c1 → b3, c1 → a3

Concurrent events
a1⇈ c1, a1⇈c2, …
a2⇈ b2, a2 ⇈ b3, …

HAPPENED-BEFORE RELATIONSHIP →

The **happened-before** relationship allows to work in a distributed system in which only → is enough for ordering

We do not assume a unique global clock (**global time**), but allow for a set of local clocks (**local time**)

We assume also to work in an asynchronous assumption, that considers possible any **transmission delay** for messages, ***variable*** *and* ***unlimited****, in principle, so* ***higher*** than any significant possible delay

⇨ We may need several **ordering strategies, also global or total** to synchronize

We want to build a logical time system built on the → relationship that is based on logical clocks and not on physical clocks

LOGICAL CLOCKS and TIMESTAMP

We need to construct a **clock system (system timestamp)** to assign a simple indicator, a '**number**', to order events

The ***happened_before*** *relationship is only* ***partial***

We define a function TS(i), a logical time-based function (called **timestamp**) that must assign a value to any relevant event

If a → b in the system, than the logical timestamp of events must respect the law TS(a) < TS(b)

If need a **clock condition**, if we want to infer the global logical clock function **LC** for system events related to processes Pi

Clock condition (Logical Clock - LC)

Given a and b, if a → b, than LC(a) < LC(b)

NOTE: it is not true that, if LC(a) < LC(b), than a → b

LOGICAL CLOCKS and TIMESTAMP

Any process P_i *has a logical clock* $LC_i(c)$ (an integer counter)

C1. For ∀a and b, if a → b inside the same process Pi, then $LC_i(a) < LC_i(b)$

C2. For ∀a and b, if a is the sending of a message in the process P_i and b the reception in the process P_j, then $LC_i(a) < LC_j(b)$

I1. Every process P_i increments LC_i between any two events

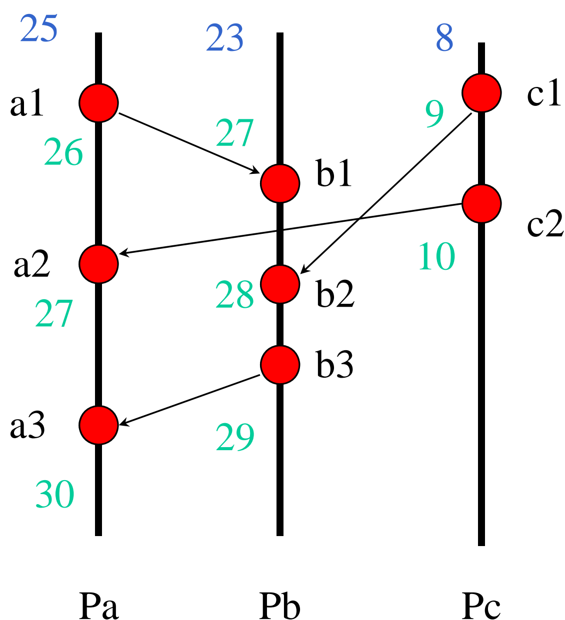
I2. For ∀a, sending of a message in process P_i, the message contains a clock as timestamp $TS = LC_i(a)$

I3. For ∀b, reception of a message in process P_j, the process put the logical clock at the greater value between current clock and timestamp $LC_j = \max(TS_{received}, LC_{icurrent}) + 1$

These rules introduce **a partial order relationship**

Many events concurrent a ⇈ b with equal timestamp

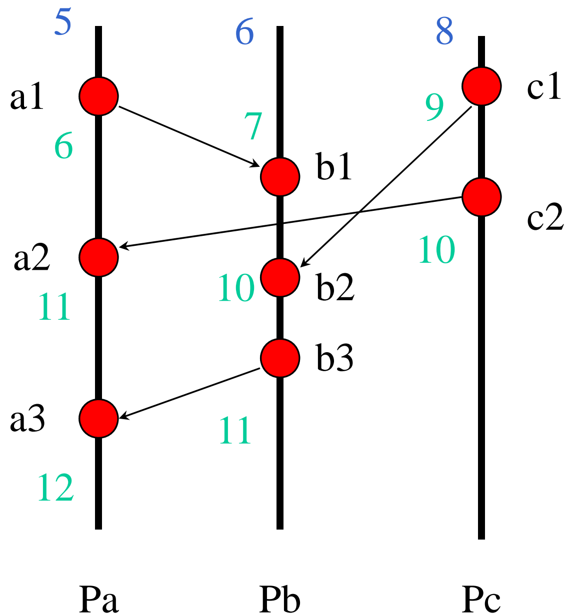
WHO DOESN'T RECIVE, DON'T UPDATE



The → relationship allows to order events according with a logical cause-effect relation

but the sender **has initiative and forces the update the logical clock of the receiver**, but not **its own**… (*it is the receiver that has to update clock to sender, with a transmission eventually*)

HAPPENED-BEFORE → PARTIAL



The → relationship **allows to catch cause-effect event ordering**

But … it also make you assume an ordering of events even without the → relationship

- Concurrent events in real world – such as c1 and b1- are considered one after the other … so in sequence

What is the relationship between the **b2 and c2** events (same timestamp)?

TOTAL ORDERING and ⇨

Sometimes it is necessary to introduce some **total order relationship** between all process events in the system

These cases are dealt with by a **global order relationship** ⇨ between all system events that is based on logical clock and on the partial ordering of →

total order relationship ⇨

If a is an event in process Pi and b an event in process Pj, than **a** ⇨ **b** if and only if

R1) $LC_i(a) < LC_j(b)$ or

R2) $LC_i(a) = LC_j(b)$ and $P_i < P_j$

The total ordering assume that in case of events of the same clock, there is an order between all processes

It is possible to use ⇨ to define an univocal and simple ordering to create synchronization upon

TOTAL ORDERING and REALITY

The Lamport relationship → is a **logical** one and it is **loosely connected** with the real world; it cannot be considered a physical world relationship (it does not respect 'reasonable' human behavior)

In general **who receive messages update its time**

Those who do not receive messages may maintain a very low timestamps and are not forced to sync logical clocks (so their timestamps can be very favorable)

Hidden channel problem

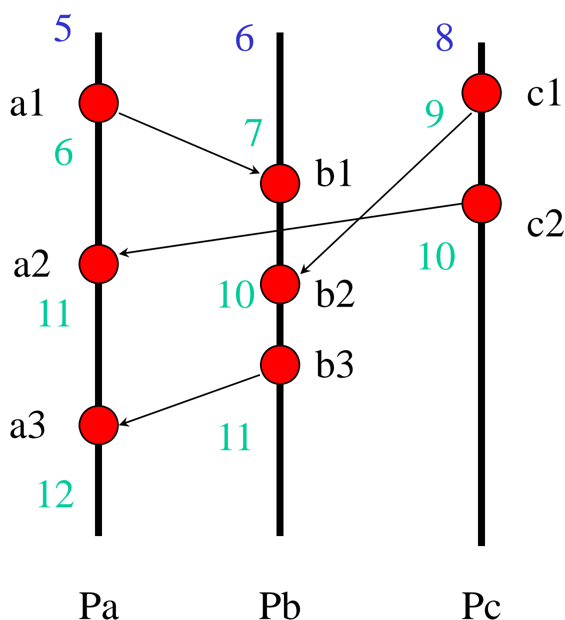
If a process can use a external and non mapped channel to communicate (**hidden channel**), that can lead to a situation that do not respect cause / effect relationship

The **effect** in **real world** can have a timestamp more low than **cause**

Causality problem

Two events considered by Lamport in a causal relationship can instead be not related

HAPPENED-BEFORE ⇨ TOTAL



The ⇨ relationship orders any pair of events

it makes possible to consider in sequence two events that are instead concurrent in real world

c2 and b2
are managed as in sequence, by considering first process Pb, then Pc

VECTOR CLOCK ORDERING

There are other strategies

it is possible also to consider **vector logical clocks** or **Vector Clocks** to order events in a process set

Processes must maintain a vector of all known clocks of all the processes and use that in communication

Every process keep its **timestamp** and a **vector** $V_i[k]$ of integers of a **dimension of the number of processes**

A **vector clock** element $V_i[k]$ contains information on what a process knows about the clocks of the others processes

The process P_i in the vector keeps:

1) $V_i[i]$ its timestamp (index i)

2) $V_i[k]$ the timestamp of any other process P_k at its knowledge

So the *data structure* is more complex for processes and also for the protocol to communicate and update the vector

A VECTOR CLOCK PROTOCOL

The **Vector clock** update protocol is based on the steps:

1. Every process P_i increments $V_i[i]$ between two events
2. For ∀ sending of a message to process P_i, the message contains the whole vector clock at best knowledge of P_i after incrementing its own $V_i[i] = V_i[i] + 1$
3. For ∀ reception of a message, the process P_j increments its own $V_j[j] = V_j[j] + 1$ and updates its vector according to

$$V_j[k] = \max (V_j[k], V_i[k])$$

The receiver obtains information on the logical time of the sender process and also on time that it knows of all others

Vectors clocks allow a better information propagation and permits a wider information exchange and diffusion (sometimes matrices are used)

LOGICAL CLOCK vs. VECTOR CLOCK

The **logical clock** protocol produces updating of clock values at message reception

In case of communication ➔ the receiver clock adapts
then all successive events are ordered with →

The main cons is that events not in the → relationship can be taken as if they were

The **vector clock** protocol instead **pays the cost** of the propagation of the entire vector and **requires to adjustment** of the entire vector at the receiver

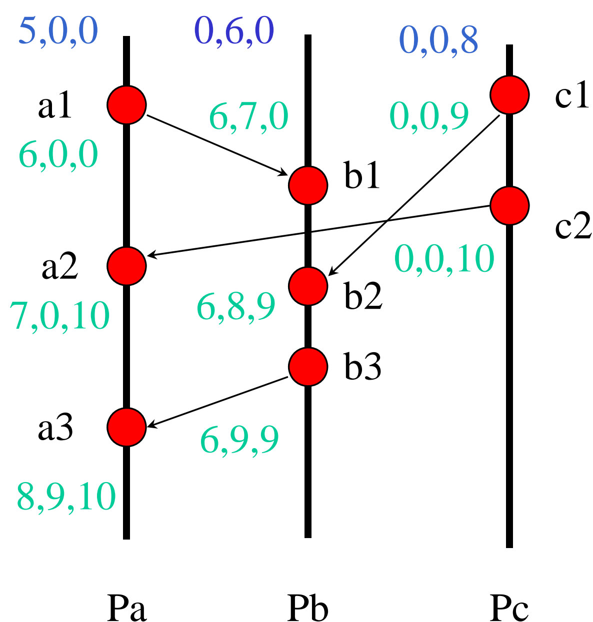
The **vector clock** ➔ requires to apply to all dimension of the clock, … so it becomes more significant and the relationship if bi-univocal

With vector clock algorithms

The events in → are recognized to be in cause effect relationship and

the events not in that relationship, i.e., concurrent events ⇈, are recognized not to be in the → cause-effect sequence

VECTOR CLOCK



With **vector clocks** we can identify if two events are **in a real cause-effect relationship**

Not only **events in relationship are tagged and ordered**, but others **events that are not cause effects are recognized as such**

Concurrent events in real world - c1 and b1, a1 and c2,… - are **not considered** in the **cause effect relationship**

SYNCHRONIZATION

The **simplest synchronization** case is the scenario of a set of processes that have to access a **resource** in a **mutually exclusive way**

We have to assume that *every process must access to the resource for a limited time and release it after usage*

OBJECTIVES:

Safety - only one process at a time can have access to the resource

Liveness - every process that has done a request receives the access after a limited delay

Fairness - different requests must be managed by a fair policy

Obviously we have many ways to realize it

We **exclude fixed priorities** that are **unfair** and can **cause starvation**

RESOURCE SYNCHRONIZATION

We can follow an approach based on a

Coordinator process

An **approach completely centralized** considers a unique **coordinator process** known to all others processes (all participants must not know each others – C/S model, but they know the cooordinator)

Every process that intends to access the resource sends the request to the **coordinator** and after usage, **notifies it**

The coordinator process decides the scheduling of the resource accesses by using its policy to grant mutual exclusion

Obviously the coordinator can decide different policies (**FIFO management** or others)

We assume that the **coordinator receives all requests sent and queued in a reliable way (but with any delay)**

RESOURCE COORDINATOR

Protocol of the coordinator

1) a process when it intends to access to the resource **sends a request message** (***request***) to the coordinator
2) The coordinator serves its request queue and it is free of deciding the request to reply to (***reply***). Obviously, it must send **only one *reply*** to one request at a time (typically FIFO).
3) when receiving the *reply*, the process can use the resource and at the end, must send a **release message** to the coordinator (***release***), that can decide to reply to another request, etc., etc.

3 messages for every access to the critical section

There are several disadvantages stemming from the centralized and unique role of the coordinator

The case of coordinator fault and of its potential unfairness

Differentiated delays in reaching the coordinator

LAMPORT SYNCHRONIZATION

Lamport proposes a decentralized solution without single failure points

A set of **N processes** that must access to a single **single resource** in **mutual exclusion**, without assuming any **centralized** role and trying to grant that requests are served in order (in a **fair** way)

Participant processes must only examine their request queue

Processes **exchange messages** between each other to obtain synchronization and must use Lamport clock relationship (up to ⇨ relationship)

Assumptions:

- *messages between processes must arrive in FIFO order*
- *messages can be delayed but not lost*
- *the connection between processes is complete and direct*

LAMPORT PROTOCOL

Use of logical clocks and Lamport relationship

Every process has a **local queue** of received messages, in which **messages are queued in order of timestamps**

For every process, the local queue initially contains the message $T_0:P_0$, lesser than every clock in the system

Clock is considered a logical time, specified by an integer and with the process identity that owns it

Every message has a timestamp that depends on both components (process and logical clock) and allows fair ordering

A process that decides to access a resource must execute a global coordination protocol

Every process must know any other and faults are not expected

(N processes in order of index compose a static group)

Mutual Exclusion PROTOCOL

Protocol

1) The process P_i sends the request message $T_m:P_i$ to every process (even in its queue) to signal its intention to access to the resource
2) At message $T_m:P_i$ reception, the process P_j (and already in its queue) sends a reply with its updated timestamp
3) *The process P_i can use the resource if in its local queue*:

- It has the request $T_m:P_i$ ordered **before** any other request of other processes (⇨ relationship)
- It has at least **one** message coming from **any other process** with a **timestamp successive** to T_m

4) At the release, P_i removes the message from its queue and sends a release message with its timestamp to every process
5) Every process P_j receives the release request and removes the request message from its queue

RESOURCE SYNCHRONIZATION

That solution grants that every process that executes the protocol can receive the resource with a limited time delay, if every process respects the constraints

Let us note that the process that requested to access, but waits and enforces a coordination with any other participant

Every request message sent requires a response from all others

The wait for messages form any other process in the system, allows for the arrival of other possible requests from other processes that may precede the current one and, once arrived, are queued ordered by timestamp

Every process queue is ordered, and so a process can pass only when 'previous' requests have been served already

At least **(N-1)** messages sent and the same number received before entering

SYNCHRONIZATION COST

The synchronization worst case is when all processes want to access the resource at the 'same' time

In case two processes make a request, they separately agree on the fact that first to enter is the one with the lower timestamp

So there cannot be conflicts

The algorithm occurs without *any centralization*, but in a *completely distributed way*

For every **action** on the **critical section** the **number of exchanged messages** is (considering a probable broadcast as N-1 messages, unless you can obtain lower cost)

Number of messages **3 * (N-1)** or N-1 and 2 broadcast

We have a **high cost** due to decentralization

Heavy assumptions on the **static group** and **no faults**

OTHER M.E. PROTOCOL

Ricart & Agrawala protocol

1) Process P_i sends the request message T_m:P_i to any process (even it its queue) to signal its intention to access to the resource
2) At message T_m:P_i reception the process P_j sends

- an immediate approval reply if does not need the resource or the requester has a higher priority
- Delay its reply if is using the resource or it has already asked to enter and its has a higher priority

3) Process P_i access the resource only if receive **N-1 approval** messages
4) At release, Process P_i must send approval to all arrived requests
5) *The requests are deleted after approval*

Only one process can have **N-1** approval responses and
only a process can access the resource at a time

RA - SYNCHRONIZATION

For every action in the critical section, the number of exchanged messages is (a possible broadcast costs as N-1 messages)

Number of messages **2 * (N-1)**

So, there are N-1 messages from requester and N-1 from everyone else

difficult to foresee a coordination at lower cost

These algorithms are based on **variations of Lamport relationship**
are **completely distributed** (no unique manager)
fair and *free from deadlock* and *starvation*
But they may
have **high costs** in terms of exchanged messages for coordination
have **high costs** due to decentralization

Heavy assumptions of messages not lost and static group without faults

ATOMIC MULTICAST

Distributed implementation of **atomic multicast** can be less centralized than the obvious one with a unique coordinator

CATOCS

CAusal & **T**otally **O**rdered **C**ommunication operations **S**upport

based on a by-need dynamic *coordination of a set of managers* that decide internally the request order

The group *does not* have a *unique central manager*, but coordinates on need and create a unique vision: it is possible to have a manager selected for every request that negotiate with others and obtains all the requests to synch with others

Realization not **scalable** and **implementation** of different **efficiency (?)** or at least efficient only in specific cases

*Availability of a **broadcast at a low level can** solve many implementation problems and **enhance efficiency** (we also need a support that grant the assumption of not losing messages, connecting all processes, … etc.)*

ATOMIC MULTICAST - ISIS

ISIS appeared in the 1990… for CATOCS

ISIS is system based on **groups** with **active replication** and with necessity of a vision with **different degrees** of **coordination** of group components

The system obtains coordination with many different forms of group multicast (called broacast)

Many different multicast forms are available (**BCast**)

FBCast (**F**ifo **BCast**)

CBCast (**C**ausal **BCast**)

ABCast (**A**tomic **BCast**)

GBCast (**G**roup **BCast**)

Providing also support to the case of no copy coordination

In general, there are no assumptions on group central roles, but any **operation need a manager**, typically chosen dynamically according to any kind of policy (closeness, …)

ATOMIC MULTICAST - ISIS

ISIS ABCast (Atomic BCast)

That CATOCS uses a queue for every corresponding component of the group and Lamport relationship

The messages that arrives to any group element are tagged with an ***initial arriving timestamp*** and only considered if labeled as ***final*** in the right order for Lamport relationship

Every **arrived message** requests a coordination phase of the manager **(and hold-back)** to **determine the final timestamp**

The coordinator receives the message:

- **labels it**, and **sends it to all others** (with *its timestamp*)
- anyone else labels the answer with its **timestamp** based on its time (clock) and sends the answer back with its timestamp
- **labels it as final** with the received **highest** timestamp (is it *necessary?*)
- **resends the message with the final timestamp** to all others to communicate the **final decision**

Any in the group has all messages in the same order in its queue

Problems: delay and **overhead - message cost of 3 * (N-1)**

ATOMIC MULTICAST

ISIS ABCast (**A**tomic **BCast**) achieves the **total ordering of messages** for a group toward a **coherent group vision**

The group must reach an **internal agreement** and that can also not be a **compliant vision** to **external timestamping** (not to be respected)

Group members cannot operate on one request until it is sure that the message:

- has been **seen also by everyone else** (**arrived to anyone**)
- has been **ordered with respect to any other message** for the group (**arrival order**)

The group is achieving consistency in operation ordering and, so, **atomicity and global order is guaranteed**

- *other messages not yet arrived, maybe sent before these, will be considered only afterwards by the entire group*

And if we have to guarantee **causal multicast?**

How do we do that? It is more or less complex?

CAUSAL MULTICAST - ISIS

ISIS CBCast (**C**ausal **BCast**) also **partial ordering**

That multicast tends to consider only **some external events** that are considered to **be ordered respectively**; all other events can be ordered differently by any group components (limiting costs and coordination)

ABCast tends to impose an order based on timestamping decided inside the receiver group (internal event ordering strategy)

CBcast requests a behavior decided outside the receiver group that have to detect a **cause-effect** relationship by inferring it via timestamps arriving from outside (internal event ordering strategy)

The Causal Broadcast assumes a ***coordination between senders*** *that must update their "logical clock" and send information to receivers* (requests are queued in the group with the senders timestamps)

Group members must only respect that ordering

If a ***cause*** *would not reach the group before processing the* ***effect****?*

Necessity of undo or error

OTHER MULTICAST - ISIS

ISIS GBCast (**Gr**oup **BCast**)

In the real world, the **group of processes can dynamically change in cardinality**, so it is possible to join or to leave the group for different reasons (possible group inconsistencies and problems)

For every ***concurrent multicast****, the message arrive in two states:*

- to every member before group changing

- to every member after a gropu changing

consistent ordering of any BCast events, either before or after

GBCast makes possible to **order all Bcasts**, so either the **message is received only after receiving every previous BCasts still in process** (or **before, in a consistent way**)

GBCast requires a monitoring support for **group variation events** (insert and extraction, because of fault and reinsertion, triggers one GBCast)

Every group member uses of a table for other member memberships to the group, and that table is updated by a GBCast (so all other Bcasts can be aware of it and consistently ordered)

JGROUPS – RELIABLE MULTICAST

JGROUPS

Support for reliable multicast and for group concept

Designed in Java and with user defined proprieties

JGROUPS starts with a **transport** level, either not connected or connected, and it is also possible to work with JMS (Java Message Service) for message specifications

The goal of JGROUPS is **group and message delivery ordering**: it proposes a **reliable** implementation, intended as delivery with **message retransmission**, with most common different ordering: atomic, FIFO, causal, etc.

For the **group property**, groups are **dynamic** and managed in membership: every group element benefits from group messages, both from outside that from inside the group

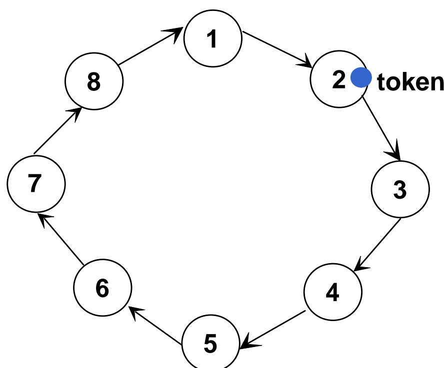
Possibility of **security**, like encryption and other secure support protocols

SYNCHRONIZATION by using TOKEN

To overcome the problem of one **central coordinator**, the synchronization can be deployed by changing the coordinator

without a **fixed role**, but varying the responsibility

The synchronization is **associated** with a **token** dynamically passed between N different participants



The nodes are organized in a **logical ring (ON)**, where every node knows the next one (successor and predecessor)

Every node acts as the group manager when it owns the **token** that it must keep for a while, then must pass to the next one

The token **circulates among** the N different participants

RING SYNCHRONIZATION

A **logical RING** connects all N participants and the token current owner is the manager of ME

Protocol to access to the resource: who has the token,

- verifies that it is the expected recipient and
- uses the token for a time period with a maximum detention (it **manages ME to access resources** for all **N nodes**)
- addresses it to the following

Only one process at a time can access ME resources, only one process has the token at a time and no conflicts can arise Starvation is not possible, if the token moves in the ring in one direction only

Number of messages **N** for complete token passage

The working scheme is typically proactive: the token must circulate even without requests

Problem if the token is lost (*failure of the node that has it*)

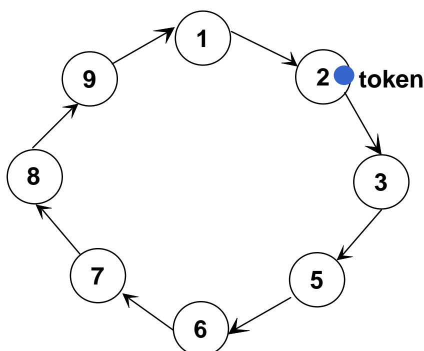
SYNCHRONIZATION in a RING

The case of losing **the token** or **having more than one** must be avoided (since they are unsafe for ME)

In case of ***failure of the node that has*** the token, it is necessary to regenerate it

Token loss must be prevented (due to fault on manager node)

Every node, sending the token in the ring, activates a timeout interval (depending on N and on maximum permanence of token in a node) that is reset at token return



In case the **timeout** is triggered, the node starts a **recovery** procedure to regenerate the token

Note that more than **one node** can start this procedure

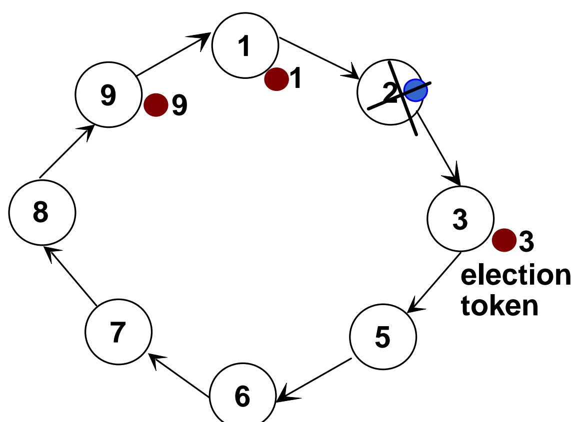
ELECTION REQUIREMENT in a RING

Election protocol to decide who must become the manager (by generating a unique new token) based on **static priority**

- At timeout, the process creates an **election token** (*ET*) with his name and enter an **election state** until the token returns
- If the process receives the normal token before the generated *ET* is back, the election is considered useless and terminated (***ET destroyed at return***)
- If the process receive an *ET* from another process, it is registered on an *election list* together with *identity of process* that generated it, and it is passed inside the ring
  if it has already generated an *ET* token, verifies the *static priority* and decides who has ***highest priority*** in the election
- If the process receive its *ET*, removes it and verify registration list
  The process generate a new token, if the it is the node with ***minimum index*** (***top priority***) inside the registration list

ELECTION in a RING

The RING architecture allows to execute **very simple recovery** algorithms in case of **single fault with limited duration**
Obviously, any node must execute some local neighbor correctness checks to re-create the ring and overcome failure in case of neighbor failure
Any node must also know the further following node



In this case, the token can be regenerated from the node with higher priority among the considered ones (here **the number 1**)

The election token become the new token

ELECTION PROTOCOLS

The election protocols are used **any time an agreement among participants** must be found **without a predefined policy**

They are typically necessary in case of **fault** and **recovery** in a **group** to obtain distributed and easy agreement on a decision

In many cases, it is based on a potential **static order of participants**

BULLY algorithm

Every participant P_i that detects necessity of an election (event local to everyone) or a recovery for a management role can do it

Three types of messages are considered

- message **Election**
- answer **Answer**
- announcement **IAmCoordinator**

How many phases there are in election protocols?

BULLY ELECTION

Every **participant** can start the **election at any time** triggered by some timeout events

1) sends an **election** message to **processes with higher priority** (**Election**)

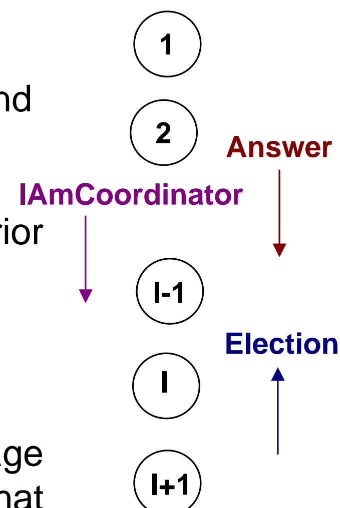
1'- in case of election message from a lower priority process, sends an **answer** to block and **a new election is started**

2) *after some time*, …

Answer coordination messages from superior nodes can arrive

- if they arrive, the low priority process stops
- if no message arrives from higher priority processes, it becomes a coordinator and

3) signals its presence with the message **IAmCoordinator** to lower priority nodes that are advised



TWO PERSPECTIVES

Optimist: A distributed system is a collection of independent computers that appears to its users as a single coherent system

Pessimist: “You know you have one when the crash of a computer you have never heard of stops you from getting any work done.” (Lamport)

Academics like point of view:

Clean abstractions, Strong semantics, Things that are formally provable and that are smart

Users like point of view:

Systems that work (most of the time), Systems that scale well, Consistency not important per se

ACID PROPERTIES

The idea of granting the **maximum of consistency** is embodied by the **ACID properties** typically considered in

- Concurrent execution of multiple transactions
- Recovery from failure

- **Atomicity**: Either all operations of the transaction are properly reflected in the database (commit) or none of them are (abort)
- **Consistency**: If the database is in a *consistent* state before starting a transaction, it must be in a *consistent* state at the end of the transaction
- **Isolation**: Effects of ongoing transactions are not visible to transactions that executed concurrently
  Basically “we’ll hide any concurrency”
- **Durability**: Once a transaction commits, updates can not be lost or their effects rolled back

ACID EXECUTION and COSTS

A **“serial” ACID execution** is one where is at most one transaction runs at a time, and it completes via commit or abort before another starts: “serializability” is the “illusion” of a serial execution but with heavy costs
The costs of transactional ACID model on replicated data can be surprisingly high in some settings.
Let us think to two cases:

- **Embarrassingly easy ones**: transactions that do not conflict at all (like Facebook updates by a single owner to a page that others only read and never change)
- **Conflict-prone ones**: transactions sometimes interfere and replicas could be left in conflicting states, if no attention is paid to order the updates. Scalability for this case is terrible

Solutions must involve ad hoc solutions, such as sharding and coding ad-hoc transactions

BASE MOTIVATIONS

eBay researchers

- Found that many eBay employees came from transactional database backgrounds and were used to the transactional style of “thinking”
- But the resulting applications **did not scale well** and **performed poorly on their cloud**

Goal was to guide that kind of programmer to a cloud solution that performs much better and to give new guideline sin designing internal applications

- **BASE is the solution that reflects experience with real cloud applications** and provide a new workflow
- “Opposite” of ACID

CAP STRATEGY

Brewer's **CAP** theorem:

"you can't use transactions at large scale in the cloud"

Or in large dimension systems

- We saw that the **real issue is mostly in the highly scalable and elastic outer tier** ("stateless tier") close to the users and does **not impact on the second inner layer**
- In reality, cloud systems use transactions all the time, but they do so in the "back end", and they shield that layer as much as they can from users, **to avoid overload and not to create bottlenecks**

BASE PROPERTIES

Basically Available: the goal is to provide fest responses
BASE start from the fact that in data centers **partitioning faults** are very rare and are mapped **to crash failures** by forcing **the isolated machines to reboot**
But we may need rapid responses even when **some replicas can't be contacted on the critical path**
Basically Available: Fast response even if some replicas are slow or crashed
Soft State Service: Runs in first tier

- cannot store any permanent data
- restarts in a "clean" state after a crash
- to maintain data, either **replicate it in memory in enough copies to never lose all** in any crash (active copies in memory) or pass it to some other service that keeps "hard state"

MORE BASE PROPERTIES

- **Basically Available**: Fast response even if some replicas are slow or crashed
- **Soft State Service**: No durable memory

- **Eventual Consistency**: abbreviate return path by send "optimistic" answers to the external client
  - Could use **cached data** (without checking for staleness)
  - Could **guess** at what the **outcome** of an update will be
  - Might **skip locks**, hoping that no conflicts will happen (optimistic approach)
  - Later, if eventually needed, correct any inconsistencies in an offline cleanup activity

SOME IMPLEMENTATION

- Use transactions, but removing **Begin/Commit points**
  - Now **fragment it into "steps" that can be done in parallel**, as much as possible
  - Ideally each step can be associated with a single event that triggers that step: usually, **delivery of a multicast**
- the transaction Leader stores these events in a **MOM "message queuing middleware"** system
  - ✓ Like an email service for programs
  - ✓ Events are delivered by the message queuing system
  - ✓ That provides a kind of 'all-or-nothing' behavior
- Consider **sending the reply to the user before finishing the operation**
- Modify the end-user application to mask any **asynchronous side-effects that might be noticeable**, by "weaken" the semantics of the operation and coding the application to work properly anyhow

BASE EFFECTS

Before **BASE**, the code was often too slow and scaled poorly, so end-user waited a long time for responses

With **BASE**

- Code itself is **more concurrent**, hence faster
- **Eliminate locking, with early responses**, all make end-user experience snappy and positive
- But we do **sometimes notice oddities when we look hard**
  - Suppose an eBay auction running fast and furious
  - Does every single bidder necessarily see every bid? And do they see them in the identical order?
  - Clearly, everyone needs to see the winning bid, but slightly different bidding histories shouldn't hurt much, and if this makes eBay 10x faster, the speed may be worth the slight change in behavior!

ACID vs. BASE

ACID

- Strong consistency for transactions highest priority
- Availability less important
- Pessimistic
- Rigorous analysis
- Complex mechanisms

BASE

- Availability and scaling highest priorities
- Weak consistency
- Optimistic
- Best effort
- Simple and fast

ACID + BASE = CAP

What goals you might want from a large organization support system for sharing data globally

Consistency, Availability, Partition tolerance

- **Strong Consistency**: all clients see the same view, even in presence of updates
- **High Availability**: all clients can find some replicas of the data, even in presence of failures
- **Partition-tolerance**: the system properties hold even when the system is partitioned and the work can go on without interruption

You can respect only **two out of these three properties**

The choice of which feature to discard determines the **nature of your system**

Consistency and Availability

Providing transactional semantics requires all functioning nodes to be in contact with each other (and no partition is allowed)

When a **partition occurs**, **no work can go on and the reconnection must be awaited**

- **Examples**:
  - Single-site and clustered databases
  - Other cluster-based designs
- **Typical Features:**
  - **Two-phase commit**
  - **Cache invalidation** protocols
  - **Classic DB** style

Partition-Tolerance and Availability

If you neglect consistency, life is much better and easy….

You **can work in case of a partition and give answers**, then you will grant reconciliation afterwards

- **Examples:**
  - DNS
  - Web caches
  - Practical distributed systems for mobile environments are choosing like that (eBay as the pioneer)
- **Typical Features:**
  - **Optimistic updating** with conflict resolution
  - That is the **Internet philosophy**
  - **TTLs** and **lease cache** management

SEVERAL CONSISTENCIES

- **Strict**: updates must happen instantly everywhere
  - A read must return the result of the latest write on that data: instantaneous propagation are not so realistic
- **Linearizable**: updates appear to happen instantaneously at some point in time
  - Like "Sequential" but operations ordered by a global clock
  - Primarily used for formal verification of concurrent programs
- **Sequential**: all updates occur in the same order everywhere
  - – Every client sees the writes in the same order
    - Order of writes from the same client is preserved
    - Order of writes from different clients may not be preserved
  - – Equivalent to Atomicity + Consistency + Isolation
- **Eventual consistency**: if all updating stops then eventually all replicas will converge to the identical values
  - – **Equivalent to CAP** Atomicity + Consistency + Isolation

EVENTUAL CONSISTENCY

When all updating stops, then eventually all replicas will converge to the identical values

Write propagation can be implemented with two steps:

- All writes **eventually** propagate to all replicas
- Writes, when they arrive, are written **to a log and applied in the same order at all replicas** (timestamps and “undo-ing”)

Update propagation in two phases

- **Epidemic stage**: Attempt to spread an update quickly willing to tolerate incomplete coverage for reduced traffic overhead
- **Correcting omissions**: Making sure that all replicas that weren’t updated during the first stage get the update

GLOBAL STATE

In a distributed system it is sometimes necessary to coordinate and support a **global state associated with the current situation**

The state can be successively used to replay the system from a previous point and restart execution in a safe situation

The main point is to locally coordinate the event of the single component parts to compose a consistent view, without paying too much for coordinating

checkpoint for recovery, distributed garbage collector

Let’s assume an **asynchronous model** with **processes** on different nodes that can send messages reciprocally (there are *channels with only one-way communication between processes*)
Processes can execute locally and also exchange messages via **channels** that must grant that all nodes are reachable by any other one (no partitioning)

GLOBAL STATES

The **global state** stems from the private states of any **participant process**, but also should keep into account exchanged **messages** (currently exchanged) between different processes

The main point is to record the whole needed information to avoid a situation in which you are losing content

The snapshot must be taken while processes are running, so it must **minimally intrude in the normal execution and be safe**

Distributed snapshot

Compose the needed information in a unique meaningful state, but acquiring it in a distributed scenario with a minimal coordination

Recall that we have to grant a safe global vision in a consistent way

how to compose every single state?

compose meaningful state and potentially consistent

excluding meaningless states (also not consistent)

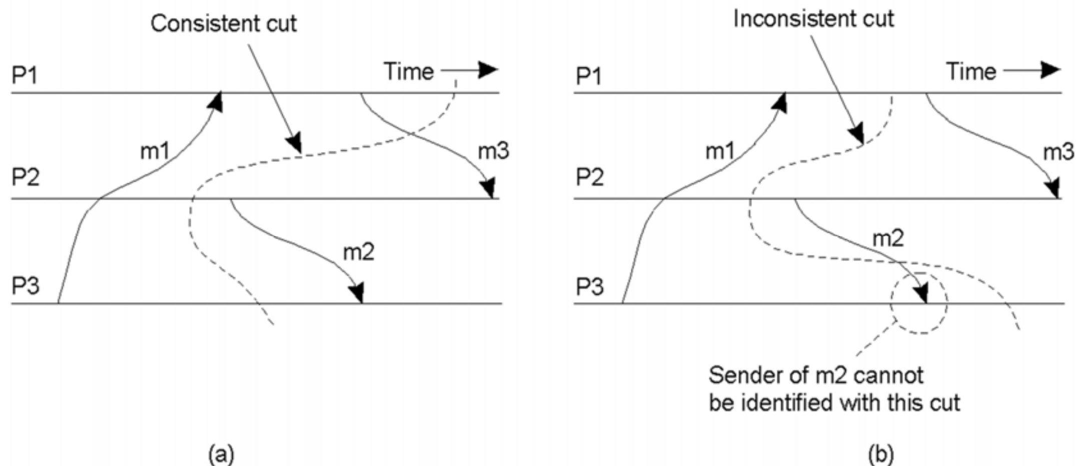
GLOBAL STATES CONSISTENCY

Consistent cuts in a distributed system

Non all states are admissible and safe for snapping the shot

Consistent cuts (a) represent a safe global state and

Inconsistent cuts (b) produce an unreasonable global states



SAFE GLOBAL STATE

Consistent cuts in distributed system exclude **unreasonable situations from the operation point of view**

Consistent Cut or Message (a) message m3 from P1 to P2

In case of the m3 message, where we record the sending state in the snapping of P1, and we cannot not record the arrival within the state of the receiving node P3 – the message must be recorded

We have to keep track of that **message inside the global state** in case of replay (**the message is part of the snapshot process, and it is must be recorded in the input of the receiver**)

Inconsistent Cut or Message (b) message m2 from P2 to P3

In case of messages where we record the arrival in the state of the receiver node, but not the sending in the sender node

This type of recording or cut is **inconsistent**, because it embodies the message in the receiver state, but the message has not recorded in the sender state: in case of replay, the sender will forcedly resend the message that causes the effect of a double reception in the receiver and an unsafe behavior (this event must be avoided)

GLOBAL STATES VIA SNAPSHOT

Distributed Global Snapshot (one at a time)

Local algorithms plaid by nodes to put together a **single organization** starting from **all participant states** (checkpoint) and the exchanging **compatible messages** (**channels state**)

OBJECTIVE: propagate a state **snapshot wave** from processes that individually record the local state; the wave expands to cover the entire system (assumption of **complete reachability**)

Every process is characterized

- by IN and OUT channels in FIFO mode and enough connections
  (*every bidirectional channel→ is separated into two channels*)
- by two states and two **colors** and **marker** management messages
  - **white** - initial state (*before snapshot*)
  - **red** - successive state (*doing snapshot or completed*)

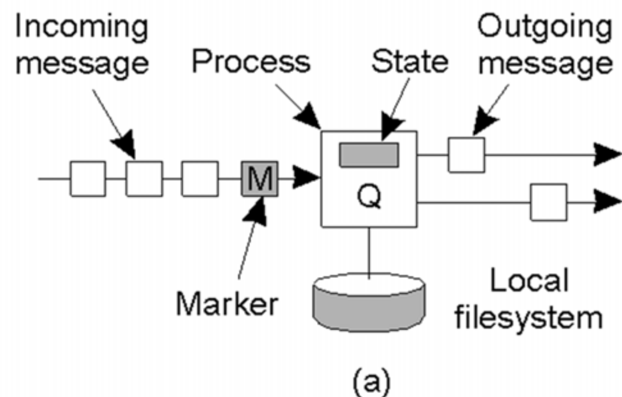
Every process that becomes red makes a *local snapshot* and *sends one marker message* via any OUT channels
Every process that receives the marker becomes red
The marker message passes through channels in order, together with all other application messages

Distributed Global Snapshot

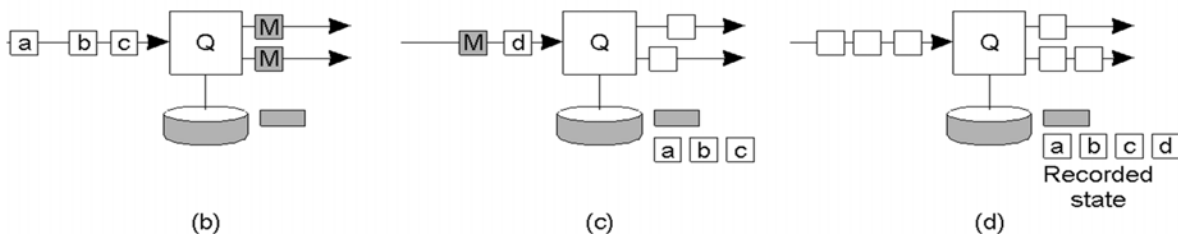
Every process organizes state to save in two parts:
- its **local environment** (state) to record as soon as they become red
- some sets of **possible messages** associated to one **input *channels***
  these messages are recorded until a marker message arrives in the queue

A process become **red** either at a **marker reception** on an input channel or if it has decided to make a snapshot; after it remains red ***(stability of the state)***

A **process completes its snapshot** after receiving a marker on **every input channels** (and completes the snapshot)



Distributed Global Snapshot Algorithm



a) The process Q receives a marker and registers its internal state (**checkpoint**)

b) The process Q sends out new markers to output queues and start **registering all incoming messages** from open **input channels**
The messages are meanwhile processed and consumed

c) The process Q receives a **marker** on a specific input channel (except the one where it arrived first that is already closed)

d) The process Q closes the **registration for that channel (but messages continue to be served)**

When a process ends the snapshot on every input channel it has completed the **node snapshot** (state an all **messages saved from input channels**)

STATE as union of LOCAL STATES

Distributed Global Snapshot
Every process can start a snapshot (checkpoint of local state) and send the marker on every out channel
The snapshot global state result composed by:
- **local states** of every process
- **state of** input connection **channels**
  (messages sent by senders and *recorded by receiver*)

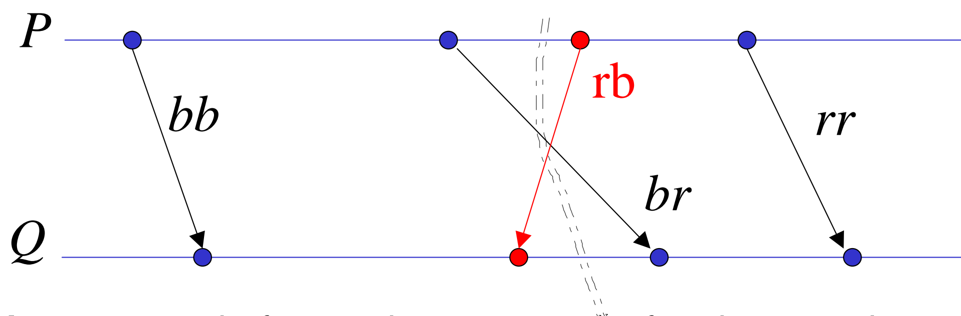
For the **process state**, it is created when a process starts the snapshot or receives a marker

> Every process that receives the marker makes the **checkpoint** of **its local state** e **sends a marker** message in any output queue

For the **channel state**, every incoming message **from incoming channels** is recorded until that channel gets a **marker that signals the end of the information** to be recorded for that channel
The registration in that channel can then be closed (*checkpoint*)

Distributed Global Snapshot

The global state is composed by:
- **local state** of every process
- **state of** connection **channels** (messages sent)



- **bb** messages before and **rr** messages after the snapshot
- **br** messages **to be recorded in the channel state**
- **rb messages not consistent** (avoided by the protocol since the marker will pass beforehand and makes the node red beforehand)

Messages as rb are avoided by protocol construction

SNAPSHOT MANAGEMENT

The process P can start a snapshot and request the collaboration of every other process that record their processor states and channel states

How it is all recorded and where?

Every process that ends can send the state to the process that started the snapshot or to a defined node P devoted to management collection

About snapshots management

At first snapshots are intended as rare events inside the system because of the cost

What happen if more snapshot are executed together?
How is it possible to execute more snapshots concurrently and distinguish them?
Are they compatible and how?