06.Fault Tolerance

Introduction

In centralized applications we cares about bounds as faults are handled by the program and are really clear to the users. In a distributed system the faults are partial and can or cannot be seen to the users.
Building dependable systems:

- Availability: We want the system to be ready for use as soon as we need it.
- Reliability: The system should be able to run continuously for long time
- Maintainability: It should be easy to fix. This don't have a direct impact with the others property but helps to improve them.
- Safety: It should cause nothing bad to happen. This can also include the fact that you don't loose data if the system crashes.
  Reliability and Availability are related but they aren't the same thing.
  A system **fails** when it is not able provide its services. A **failure** is the result of an error. An error is **caused** by a fault. Some faults can be avoided, others cannot.
  Building a dependable system demands the ability to deal with the presence of faults
  A system is said to be fault tolerant if it can provide its services even in the presence of faults
  Faults can be classified according to the frequency at which they occur
- Transient faults occur once and disappear
- Intermittent faults appear and vanish with no apparent reason
- Permanent faults continue to exist until the failed components are repaired

Faults/Failures in DS

Different types of failures (failure model):

- Omission failures:Processes: fail-safe (sometimes wrong, but easily detectable output), fail-stop (detectable crash), fail-silent (undetectable crash). Channels: send omission, channel omission, receive omission
- Timing failures (apply to synchronous systems, only): Occur when one of the time limits defined for the system is violated

- Byzantine (or arbitrary) failures: Processes: may omit intended processing steps or add more. Channels: message content may be corrupted, non-existent messages may be delivered, or real messages may be delivered more than once

General techniques

- Information redundancy: we replicate information to avoid the damage caused by a fault. Add information to cope with the failure
- Time redundancy: redo an operation if you don't see its results
- Physical redundancy: replicate the system to cope with failure of the system
  Typical biologic system use physical redundancy(EX: a physical component that generates data and you have more copy of this component doing the same things.Then the data are passed to a voter that calculate which data is more probable and discard the other data. This operation is repeated multiple time)

Clients crash too

A computation started by a dead client is called an orphan. Orphans still consume resources on the server. The server must get rid of them:

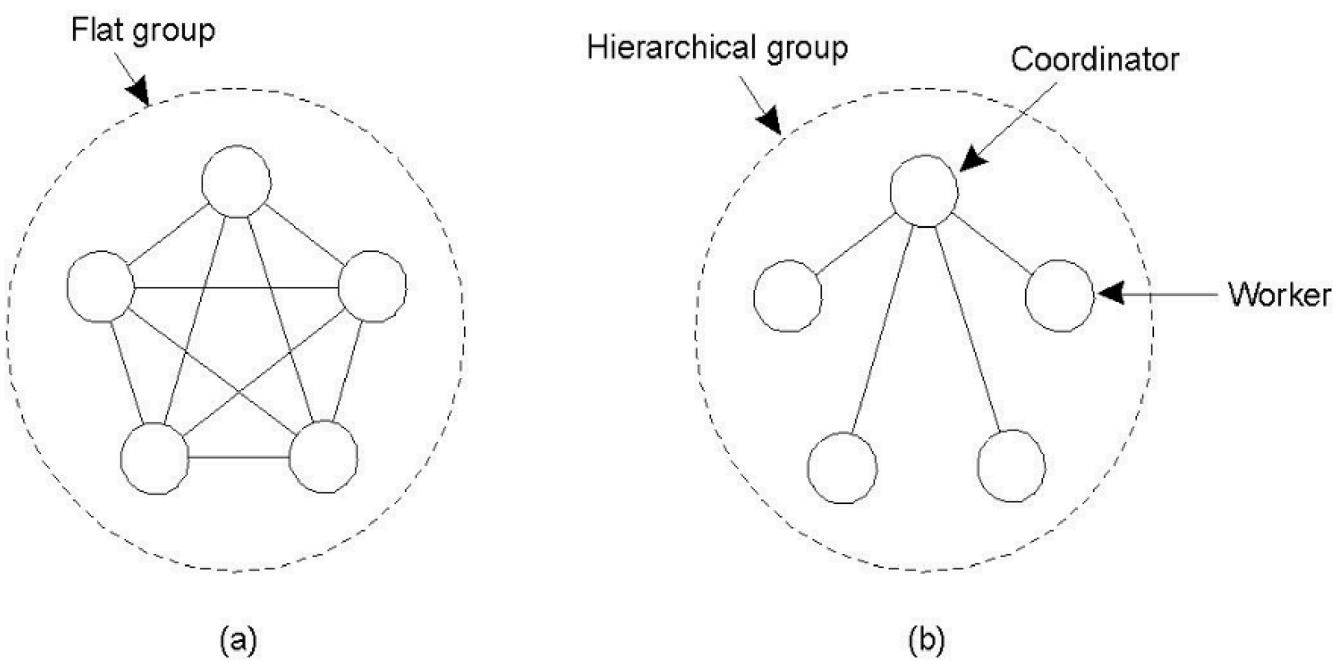
- Extermination: RPC’s are logged by the client and orphans killed (on client request) after a reboot -> costly (logging each call) and problem with grand orphans and network partitions
- Reincaration: when a clients reboots it starts a new epoch and sends a broadcast message to servers, who kill old computations started on behalf of that client (including grand orphans). Replies sent by orphans bring an obsolete epoch and can be easily discarded
- Gentle reincarnation: as before but servers kill old computations only if owner cannot be located
- Expiration: Remote computation expire after some time. Clients wait to reboot to let remote computations to expire
  Reincarnation may seem to solve all problems, but what if the killed orphans had open files? There’s a lot of bookeeping involved, and not all issues can be solved

Protection against process failures: Agreement in process groups

Process Failures

The easiest approach is to introduce redundancy. Redundancy can be used to mask the presence of faulty processes, with redundant process groups: the work that should be done by a process is taken care by a group of processes and the healthy processes can continue to work when some of the others fail

- Two possible organizations:



There are details that we didn't consider:

- We need to keep track of which processes constitute each group.
- We need to have join and leave announcements to be reliably multicast
  How large should a group be? In general of the type of failures that you want to tolerate are omission failures in processes to be k-fault-tolerant you need k+1 processes. If you have Byzantine failures things become worse as you need 2k+1 processes to have a k-fault-tolerance(to have a working voting mechanism)

Agreement in process groups

We want the components of a group to agree on something as who is the leader, the client request is satisfied, which value to send to the client...
This is different from before as the decision come from the components of the system.
The consensus problem

A set of processes must agree on some value: the non-faulty ones must agree
The value is subject to a validity condition: It cannot be any value
More precisely, we have the following conditions

- Each process starts with some initial value
- All non-faulty processes have to reach a decision based on these initial values
- The following properties must hold
    - Agreement: No two processes decide on different values
    - Validity: If all processes start with the same value v, then v is the only possible decision value
    - Termination: All non-faulty processes eventually decide

Consensus with process failures

The coordinated attack problem shows that consensus is not possible in the presence of arbitrary communication failures
Assumptions for crashing processes
Let’s review our assumptions:

- We consider a synchronous system, in which all processes evolve in synchronous rounds
    - A message sent by a process is received within the same round by the recipient (or within a bounded number of rounds)
    - Processes may fail at any point, by stopping taking steps (fail-silent)
- We want our processes to reach agreement according to the previous definition of consensus
  The problem we just stated is, luckily, easier than the previous one: it turns out that the problem can be solved provided that the processes take at least f+1 rounds with f being a bound on the number of failures. Moreover, it is sufficient that a single process be non-faulty

FloodSet algorithm

Let us now consider a simple algorithm to solve the problem:
Let v0 be a pre-specified default value
Each process maintains a variable W (subset of V) initialized with its start value
The following steps are repeated for f+1 rounds:

- Each process sends W to all other processes

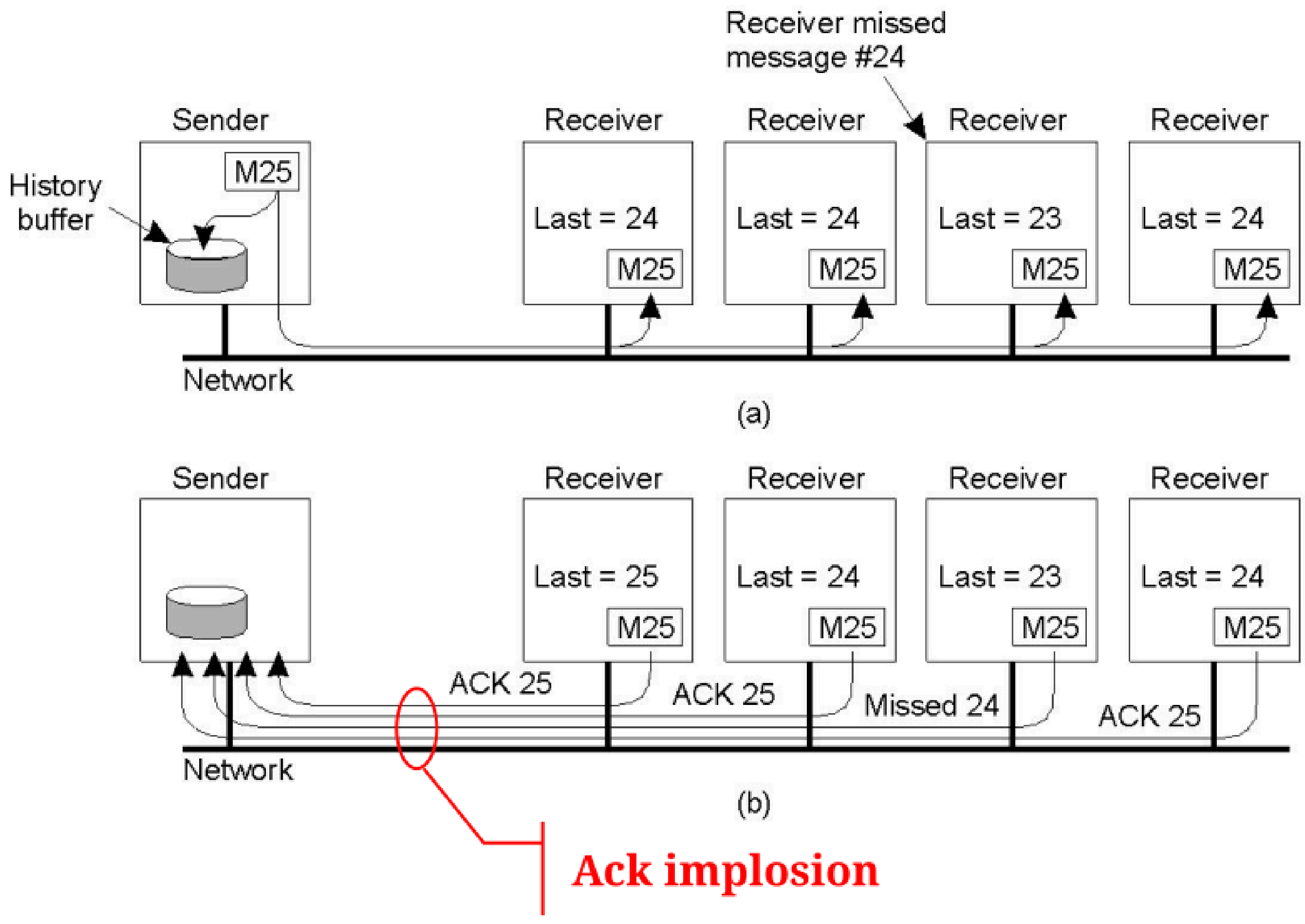
- It adds the received sets to W
  The decision is made after f+1 rounds
- if |W| = 1: decide on W's element
- if |W| > 1: decide on v0 (or use the same function to decide, e.g. max(W))
  Remember: each process may stop in the middle of the send operation
  Prooving FloodSet correcteness
  Lemma
  If no process fails during a particular round r : 1 ≤ r ≤ f+1, then Wi(r) = Wj(r) for all i and j that are active after r rounds
  Suppose that Wi(r) = Wj(r) for all i and j that are active after r rounds. Then for any round r1 : r ≤ r1 ≤ f+1, the same holds, that is, Wi(r1) = Wj(r1) for all i and j that are active after r1 rounds
- If processes i and j are both active after f+1 rounds, then Wi = Wj at the end of round f+1
  When we have Byzantine failures things complicates(In realities Byzantine failures don't happens often and to avoid them and usually there are crashes the system, usually it is during an attack that you have a Byzantine failures. They happen usually in channels and they are easy to detect and correct).
  Our problem is now defined as follows:
- Agreement: No two non-faulty processes decide on different values
- Validity: If all non-faulty processes start with the same value v, then v is the only possible decision value for all non-faulty processes
- Termination: All non-faulty processes eventually decide
  The idea is that we cannot guarantee anything about faulty processes
  So far we have considered synchronous systems. A real distributed system is inherently asynchronous. Let us consider the problem of reaching an agreement between n processes with 1 faulty process in an asynchronous system. Let’s start with the “easy case”: Fail-silent failures:
  Fischer, Lynch, and Paterson proved that no solution exists: “Impossibility of Distributed Consensus with One Faulty Process” (FLP Theorem)
- The result was proved in the case of crash failures
- What happens in the case of Byzantine failures?A byzantine general can simulate a crashed general so if a solution existed for byzantine failures that solution would also work for crash failures

Reliable group communication

It is of critical importance if we want to exploit process resilience by replication
Achieving reliable multicast through multiple reliable point-to-point channels may not be efficient
or enough. What happens if the sender crashes while sending? What if the group changes while a message is being sent?
Groups are fixed, and processes non-faulty

- All group members should receive the multicast (not necessarily in the same order)
- Easy to implement on top of unreliable multicast
    - Positive acknowledgements: large number of ack messages
- Negative acknowledgments: fewer messages but sender has to cache messages
  Case of faulty processes (we will see it later):
- All non-faulty group members should receive the message
- There must be agreement about who is in the group

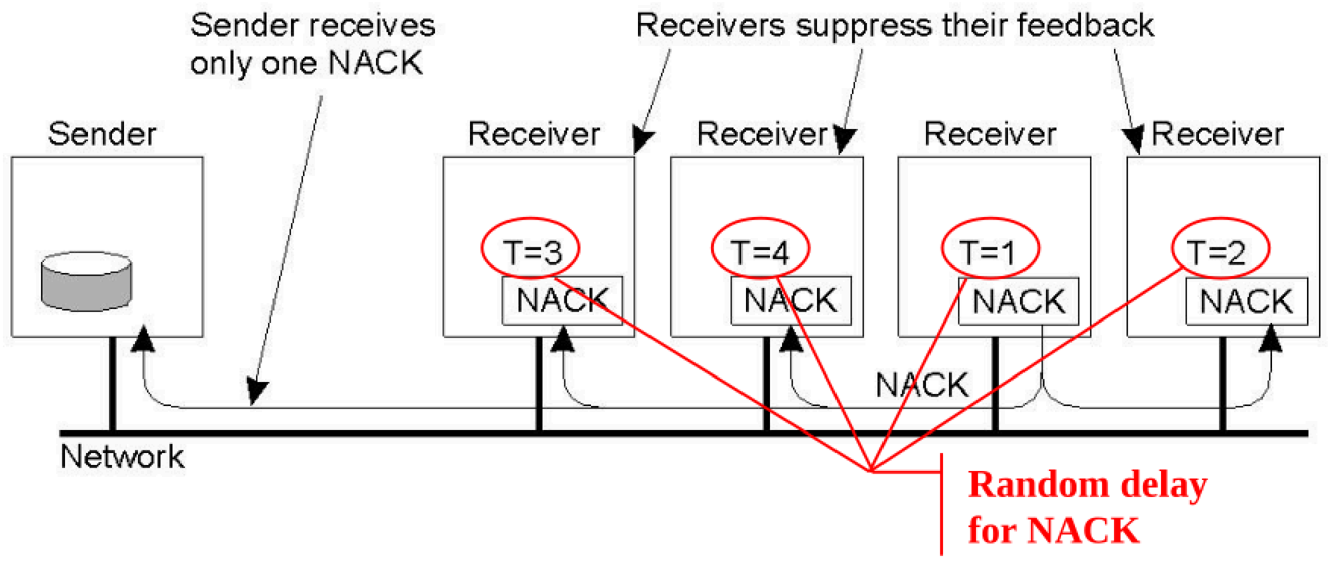
Non-faulty processes: Basic approach



Scalable Reliable Multicast

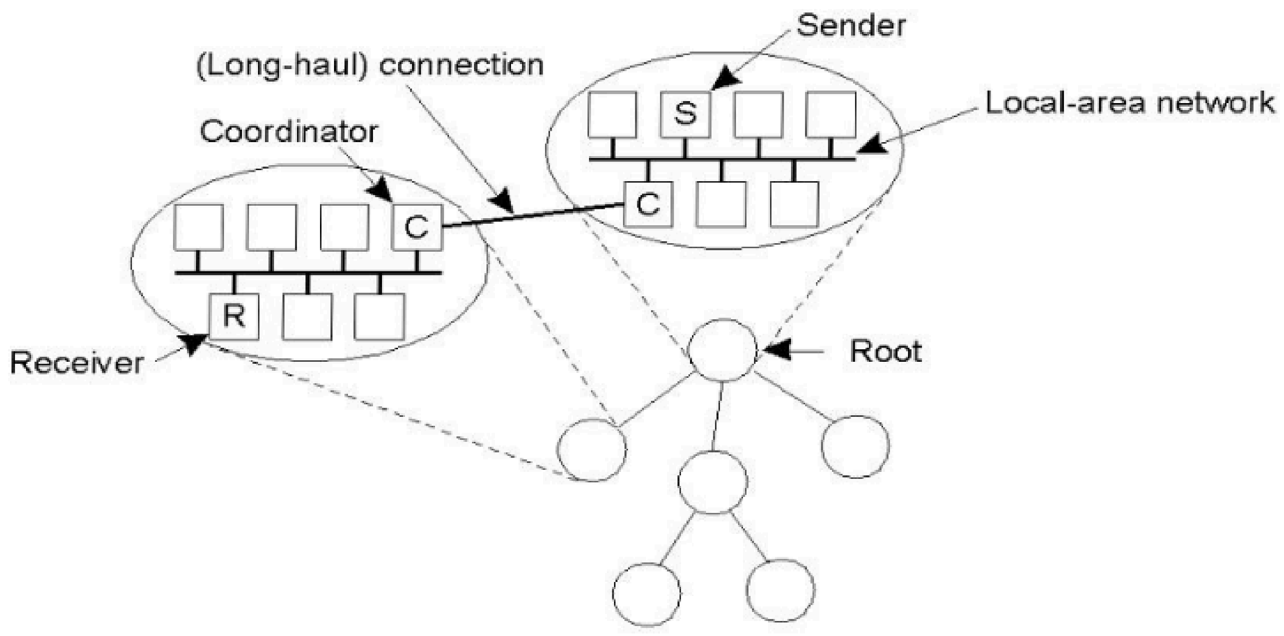
A first solution is non-hierarchical feedback control:

- Negative acknowledgements are multicast, good to suppress replicated NACK but also bad since everyone must process NACKs



To use a NACK we need a way to know we lost a packet. We give the packets a number and if we don't have all the ID we can send a NACK. The advantages is that the network is reliable and we don't need to send NACKs often BUT the problem is that if something goes wrong instead of immediatly sending a NACK you wait for a random time and look around the channel. When the timer end you send in broadcast the NACK message.
Works when the group is flat

Hierarchical Feedback COntrol



With Hierarchical Feedback control, receivers are organized in groups headed by a coordinator and groups are organized in a tree routed at the sender. You use Scalable Multicast in the group and then elect a leader that forward the messages between groups.
The coordinator can adopt any strategy within its group. In addition it can request retransmissions to its parent coordinator
A coordinator can remove a message from its buffer if it has received an ack from:

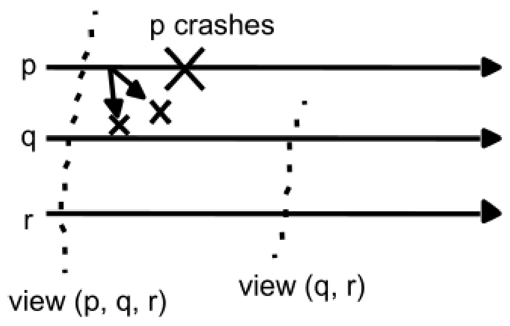
- All the receivers in its group
- All of its child coordinators
  The problem is that the hierarchy (tree) has to be constructed and maintained.

The case of faulty processes

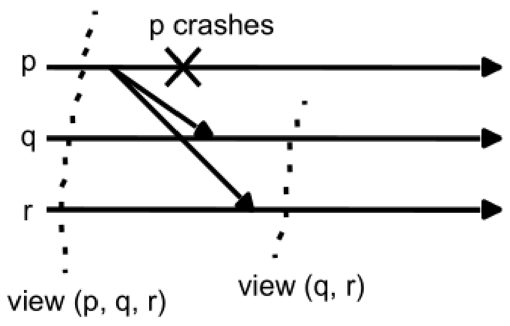
If the sender process crashes, and in particular during transmission, there is the problem that the message could be sent to a portion of the receivers. If this issue don't interleave with the fact that the crash occurs.
Things get harder if processes can fail or join and leave the groups during communication. What is usually needed is that a message be delivered either to all the members of a group or to none, and that the order of messages be the same at all receivers. This is known as the atomic multicast problem.
Ideally we would like:

- Any two processes that receive the same multicast messages or observe the same group membership changes to see the corresponding events in the same order
- A multicast to a process group to be delivered to its full membership. The send and delivery events should be considered to occur as a single, instantaneous event
  Unfortunately, close synchrony cannot be achieved in the presence of failures
  We can obtain virtual synchrony: the message is delivered either to all of the member in the group or to none and it will be delivered before telling that the sender has crashed(We cannot receive messages after we know that the sender crashed).

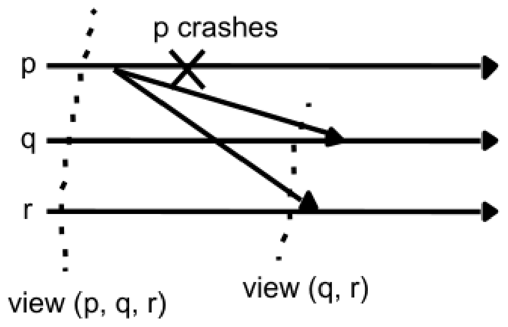
a (allowed).



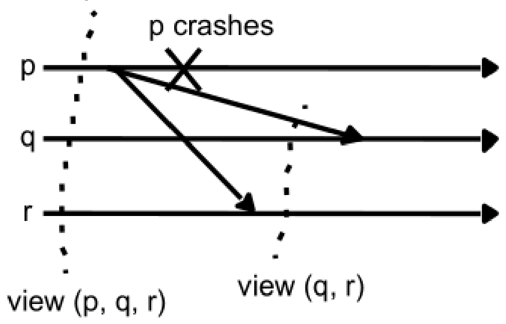
b (allowed).



c (disallowed).

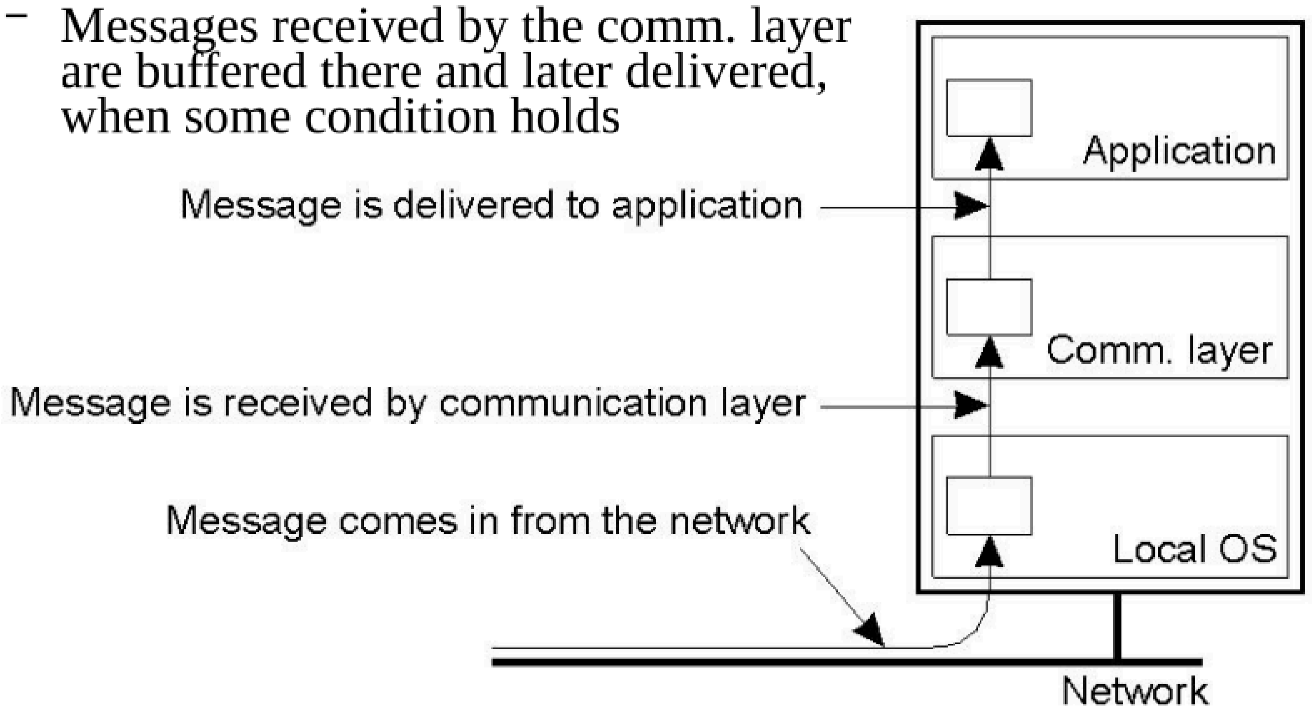


d (disallowed).



You must order and synchronize messaging with the information of crashed processes. You can do it at the middle layer

– Messages received by the comm. layer are buffered there and later delivered, when some condition holds


Message is delivered to application
Message is received by communication layer
Message comes in from the network
Application
Comm. layer
Local OS
Network

We say that a view change occurs when a process joins or leaves the group, possibly crashing
All multicast must take place between view changes:

- We can see view changes as another form of multicast messages (those announcing changes in the group membership)

- We must guarantee that messages are always delivered before or after a view change
- If the view change is the result of the sender of a message m leaving the message is either delivered to all group members before the view change is announced or it is dropped
  Multicasts take place in epochs separated by group membership changes
  A multicast system can be reliable and unordered.
  Retaining the virtual synchrony property, we can identify different orderings for the multicast messages
- Unordered multicasts
- FIFO-ordered multicasts
- Causally-ordered multicasts
  In addition, the above orderings can be combined with a total ordering requirement(whatever ordering is chosen, messages must be delivered to every group member in the same order)
  Virtual synchrony includes this property in its own definition

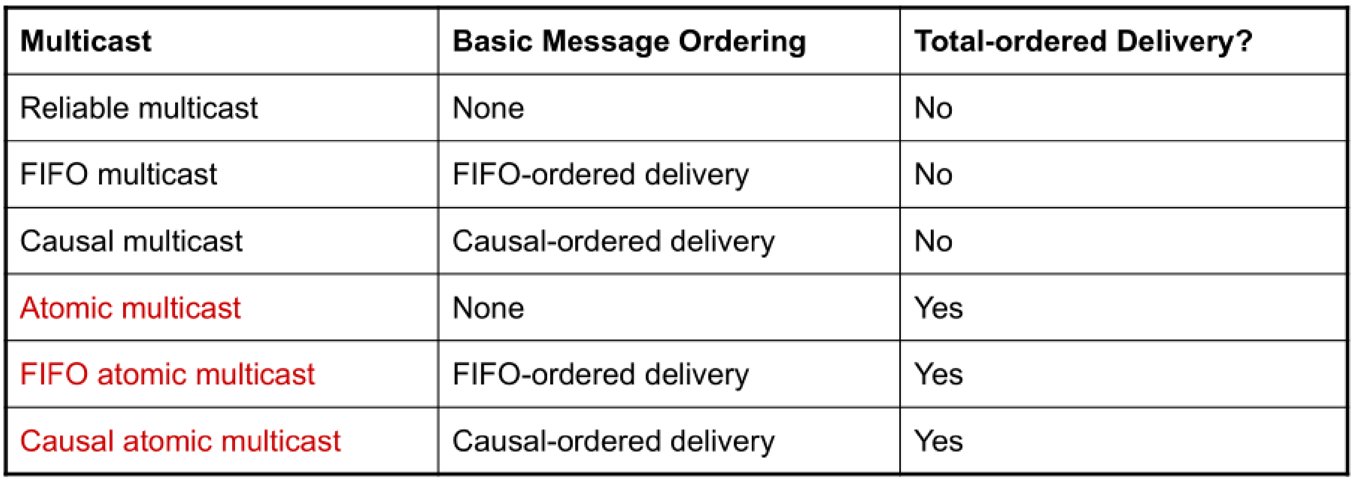
Virtual Synchrony

A group view is the set of processes to which a message should be delivered as seen by the sender at sending time. The minimal ordering requirement is that group view changes be delivered in a consistent order with respect to other multicasts and with respect to each other
This, together with the previous requirements, leads to a form of reliable multicast which is said to be virtually synchronous
We say that a view change occurs when a process joins or leaves the group, possibly crashing
All multicast must take place between view changes:

- We can see view changes as another form of multicast messages (those announcing changes in the group membership)
- We must guarantee that messages are always delivered before or after a view change
- If the view change is the result of the sender of a message m leaving the message is either delivered to all group members before the view change is announced or it is dropped
  Multicasts take place in epochs separated by group membership changes

Atomic multicast

Atomic is a synonimous of totally ordered. Atomic multicast is defined as a virtually synchronous reliable multicast offering totally-ordered delivery of messages



Multicast	**Basic Message Ordering**	**Total-ordered Delivery?**
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

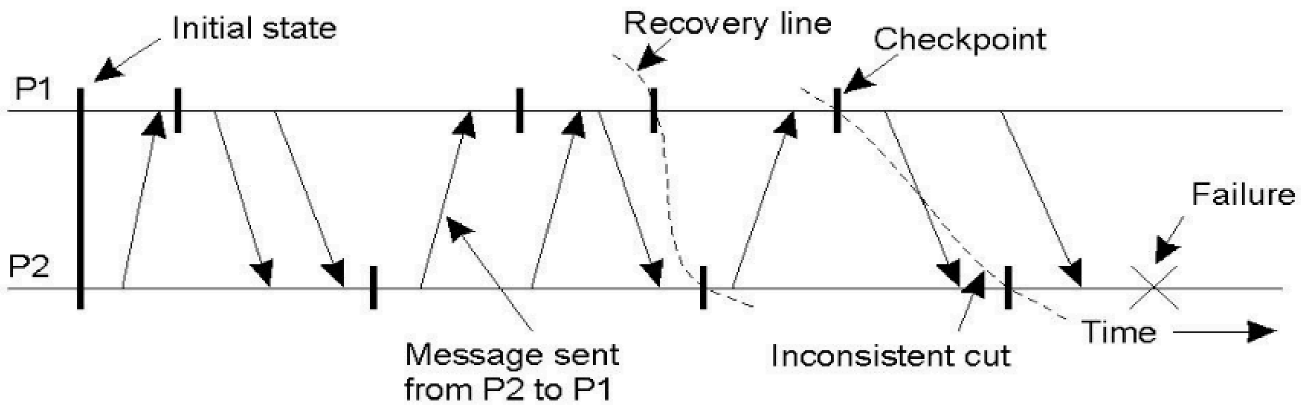
Virtual Synchrony is the best we can do.

Recovery techniques

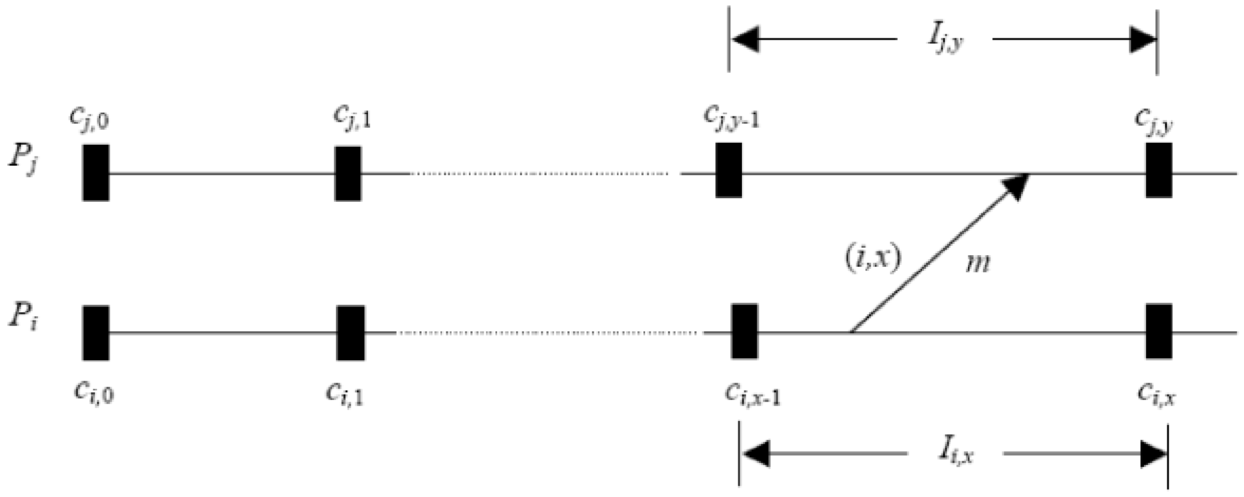
Recovery: backward vs. forward

When processes resume working after a failure, they have to be taken back to a correct state
Backward recovery: react from a state that we know is working. The system is brought back to a previously saved correct state
Forward recovery: try to resolve without going back in time. The system is brought into a new correct state from which execution can be resumed
Recovering a previous state is only possible if that state can be retrieved

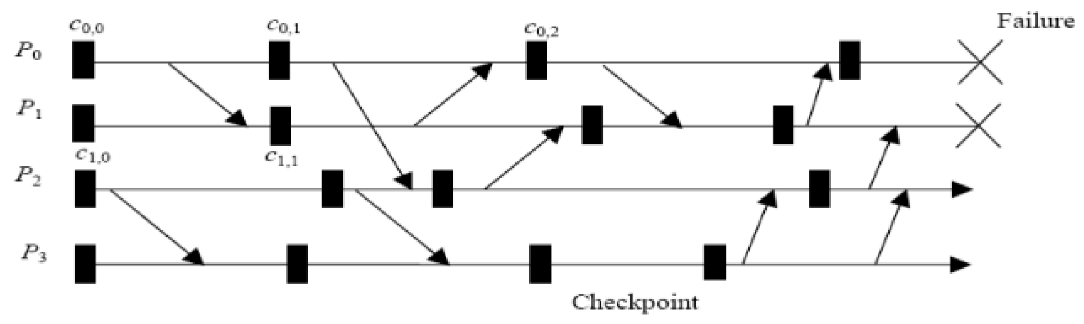
- Checkpointing consists in periodically saving the distributed state of the system to stable storage. It’s an expensive operation. It is also complex in a distributed system
- With Logging, events (messages) are recorded to stable storage so that they can be replayed when recovering from a failure. Periodically we take a checkpoint to not have to use a very long log.



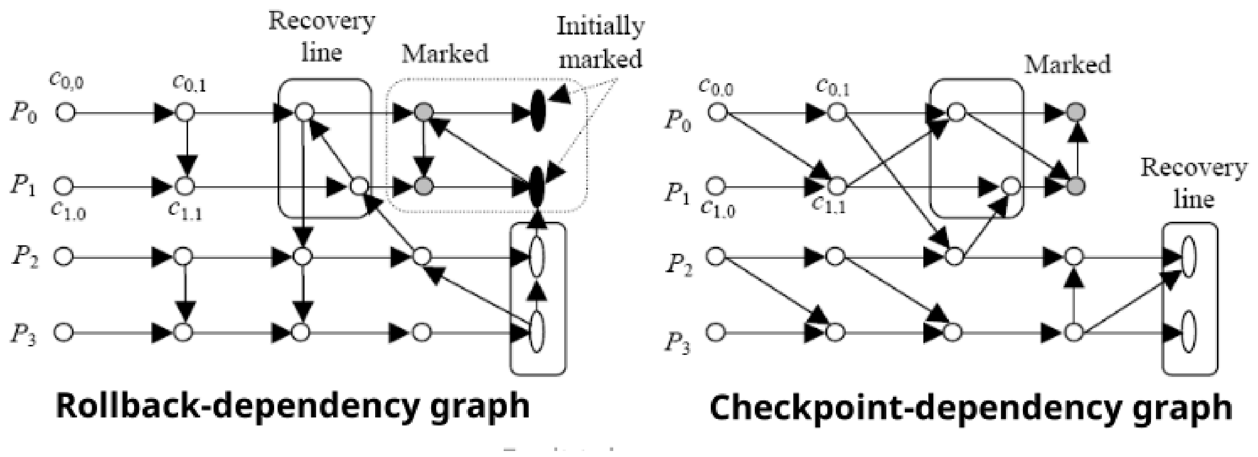
Checkpointing is something you do periodically. In a distributed system we need to checkpoint each process. This can be done locally for each process(Recovery Line). This can be problematic as the state taken by two different checkpoint and then restored, on different nodes, is not consistent, the cut is not consistent. We need to find a way to reconstruct a consistent system when a crash occur. In principle we cannot accept any checkpoint and need to go back to the start of the system(really expensive).
The idea is that each message need to be labelled with the interval between the last and the next checkpoint. So each process can discover if any interval depends on another interval of another process. The information is recorded in respect on the owns intervals.



Computing the recovery line



Let’s consider the situation in the figure; we obtain the following



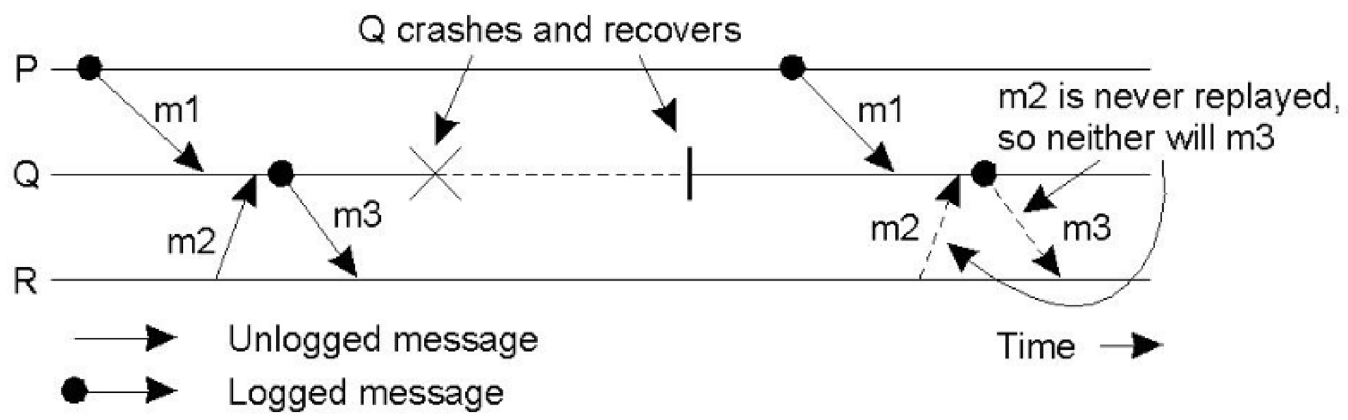
For each interval a process need to save at most the number of processes that refer to the interval(small number).
In the rollback-dependency graph we draw an arrow between two checkpoints $c_{i,x}$ and $c_{j,y}$ if either:

- i=j and y=x+1
- i≠j and a message is sent from $I_{i,x}$ to $I_{j,y}$
  The recovery line(consistent cut created by taking checkpoints) is computed by marking the nodes corresponding to the failure states and then marking all those which are reachable from one of them, These marked nodes are no good. Each process rolls back to the last unmarked checkpoint. An equivalent approach can be used on the checkpoint graph. Take the latest checkpoints that do not have dependencies among them.
  The first hypothesis is to take a node and the last consistent checkpoint between two nodes. If there are dependencies you discard the arrival state and redo the check on another state with the same consistent checkpoint.
  The good point is that there isn't coordination but the bad point is that as there

isn't coordination you could go back to the initial point.
The solution is to coordinate checkpoints(No useless checkpoints are ever taken). A simple solution is the following:

- Coordinator sends CHKP-REQ
- Receivers take checkpoint and queue other outgoing messages (delivered by applications)
- When done send ACK to coordinator
- When all done coordinator sends CHKP-DONE
An improvement: Incremental snapshot: only request checkpoints to processes that depend on recovery of the coordinator, i.e., those processes that have received a message causally dependent (directly and indirectly) from one sent by the coordinator after the last checkpoint
A further alternative which tries to share the benefits of independent and coordinated checkpointing is communication-induced checkpointing. Processes take checkpoints independently but piggyback information on messages to allow the other processes to determine whether they should also take a checkpoint

Logging schemes



The distributed system advance only when messages are received. When to log messages? We save the messages asynchronously after we have received it. In many case messages aren't saved before there are crashes. So there are orphans.
Each message’s header contains all necessary information to replay the messages
A message is stable if it can no longer be lost. It has been written to stable storage.
For each unstable message m define:

- DEP(m): Processes that depend on the delivery of m, i.e., those to which m has been delivered or to which m’ dependent on m has been delivered

- COPY(m): Processes that have a copy of m, but not yet in stable storage. They are those processes that hand over a copy of m that could be used to replay it
  We can then characterize an orphan process in the following way
- Let Q be one of the surviving processes after one or more crashes
- Q is orphan if there exists m such that Q is in DEP(m) and all processes in COPY(m) have crashed
  There is no way to replay m since it has been lost. If each process in COPY(m) has crashed then no
  surviving processes must be left in DEP(m). This can be obtained by having all processes in DEP(m) be also in COPY(m). When a process become dependent on a copy of m, it keeps m.

Pessimistic vs Optimistic Logging

Pessimistic: Ensure that any unstable message m is delivered to at most one process

- The receiver will always be in COPY(m), unless it discards the message
- The receiver cannot send any other message until it writes m to stable storage
- If communication is assumed to be reliable then logging should ideally be performed before message delivery
  Optimistic: Messages are logged asynchronously, with the assumption that they will be logged before any faults occur
- If all processes in COPY(m) crash, then all processes in DEP(m) are rolled back to a state where they are no longer in DEP(m)
- This is similar to the rollback technique used in uncoordinated checkpointing
  Pessimistic logging prevents orphans from being created. Instead optimistic logging create orphans BUT force them to roll back until they are no longer orphans.
  A variant of pessimistic logging requires messages to be logged by the sender.
  Logging was originally designed to allow the use of uncoordinated checkpointing. However, it has been shown that combining coordinated checkpointing with logging yields better performance.