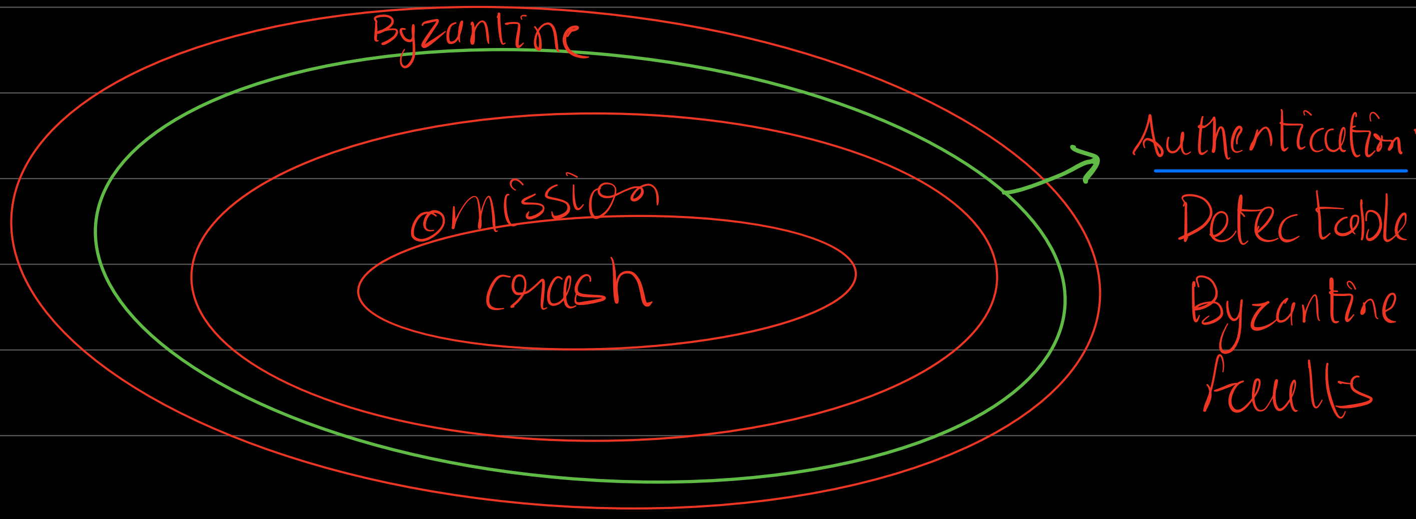
Agenda

- recap of fault classification
- forms of fault tolerance
- implementing reliable delivery
- idempotence
- at-least-once | at-most-once | exactly-once delivery
- reliable broadcast
- implementing reliable broadcast

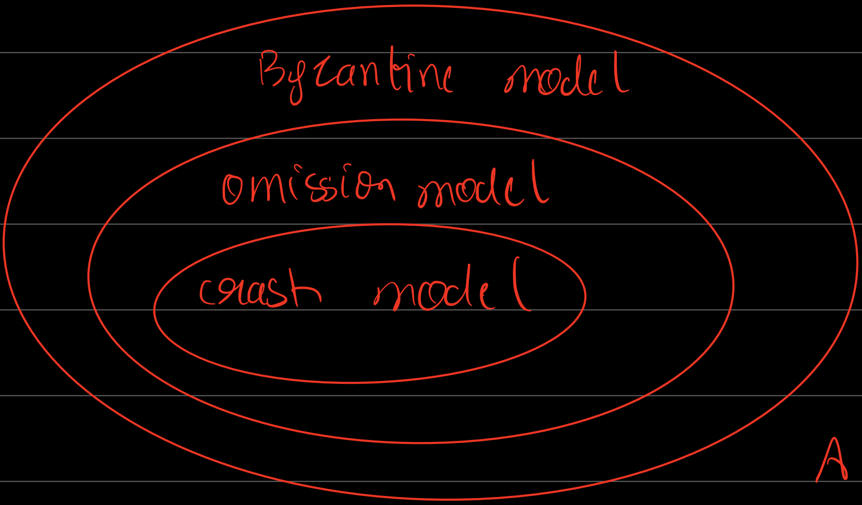
Fault models

Easy byzantine faults: - One that acts like a crash fault.

- Check message integrity using checksums, etc. This downgrades to a omission fault



Can always be sub-divided in more tolerable faults.



What does it mean to tolerate a class of faults?

A correct program satisfies both its safety & liveness properties!!

How wrong does a program go in a given class of faults?

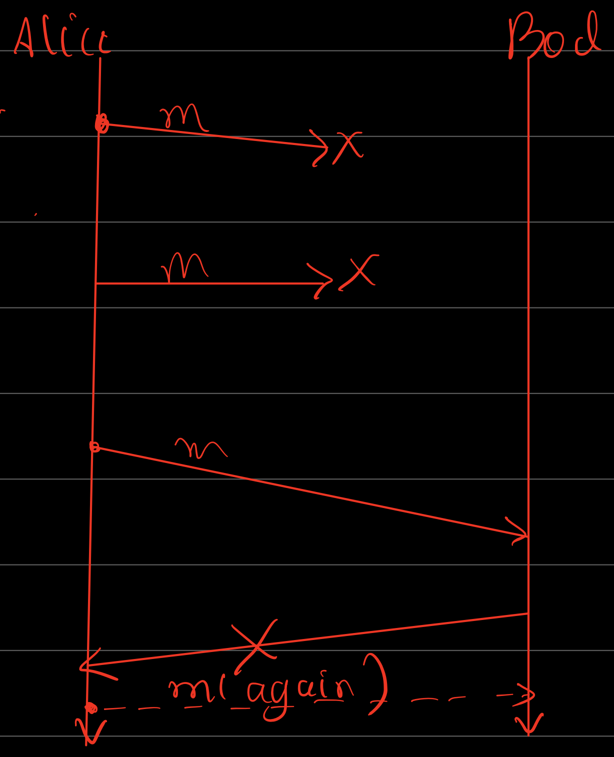
	live	not live
safe	masking	fail-safe
not safe	non-masking	BAD!

→ Even if msgs are delayed, DONT deliver them out of order

Reliable delivery (a liveness property)

Let P_1 be a process that sends message 'm' to P_2. If neither P_1 nor P_2 crashes (and not all msgs are lost) P_2 eventually delivers 'm'.

Need green if we are working in omission model

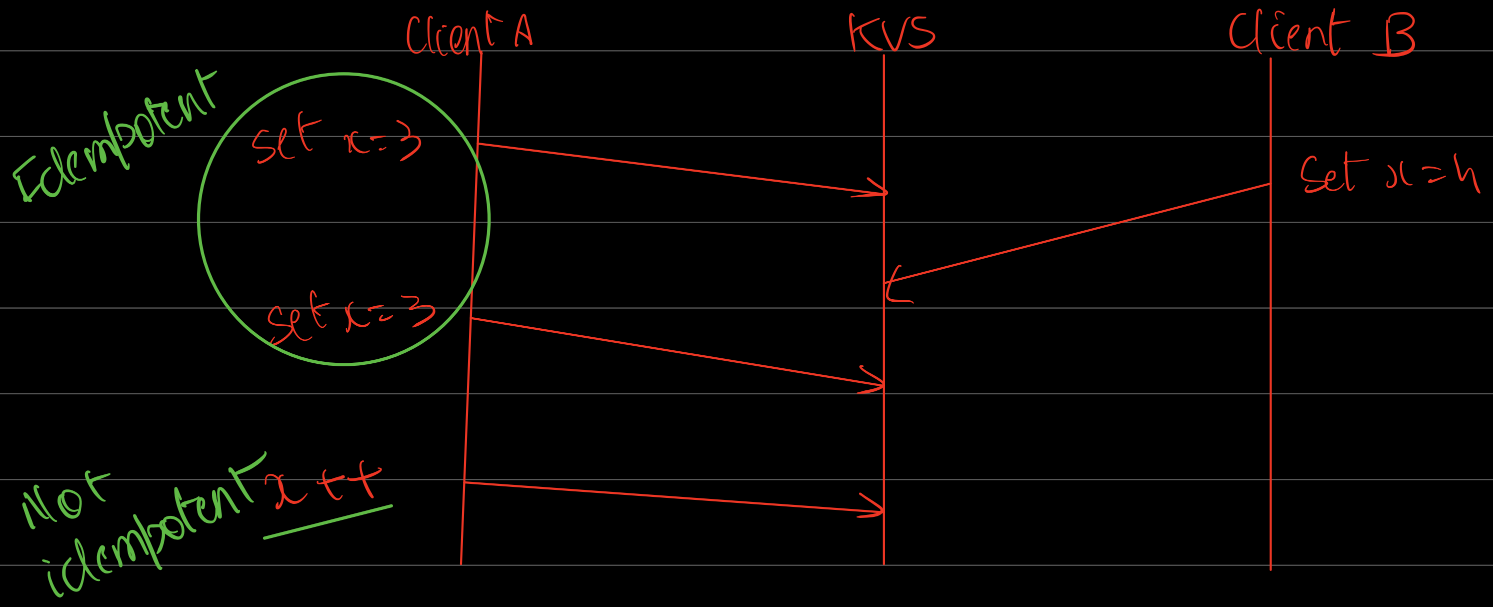


Alice :

- put message in send buffer
- on timeout, send what's in the buffer
- when an ack is received, delete msg from send buffer

If Bob's Ack is dropped, then he will receive 'm' again.

- Receiving duplicate messages can be OK depending on the usecase.



Idempotent: A message is idempotent if its okay to receive it more than once

$f(x) = f(f(x)) = f(f(f(x)))$ . . .

$\boxed{\text{Reliable delivery is at-least-once delivery}}$

At-most-once:

- Send message. If it arrives, OK.
  If not, OK.

Exactly-once delivery:

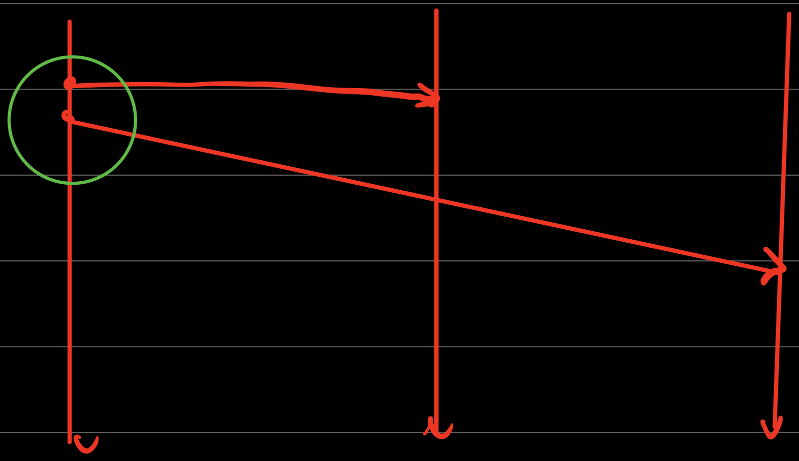
- Systems that claim exactly-once delivery
  - the messages were idempotent anyway
  - they are making an effort to deduplicate messages.

Reliable broadcast

- Every process sends a message to every other process
- one sends, everyone receives.

- Special case of multicast

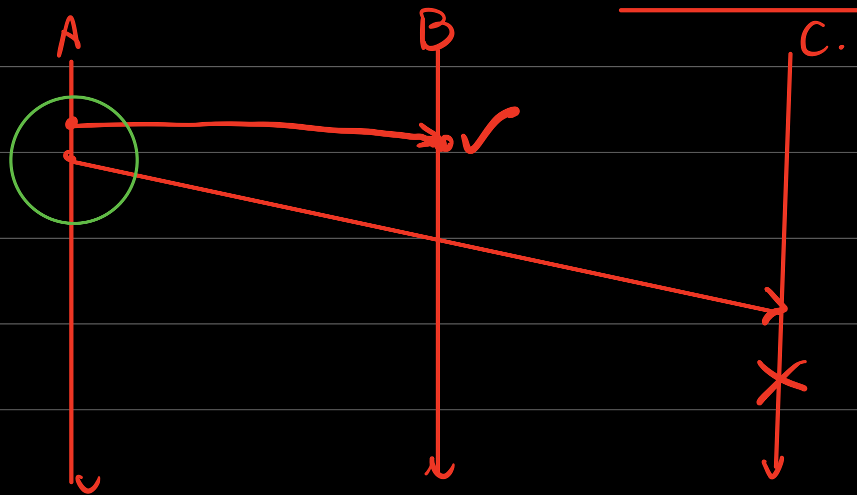
If you have a <u>unicast</u> primitive, you can implement broadcast by :
sending series of unicast messages



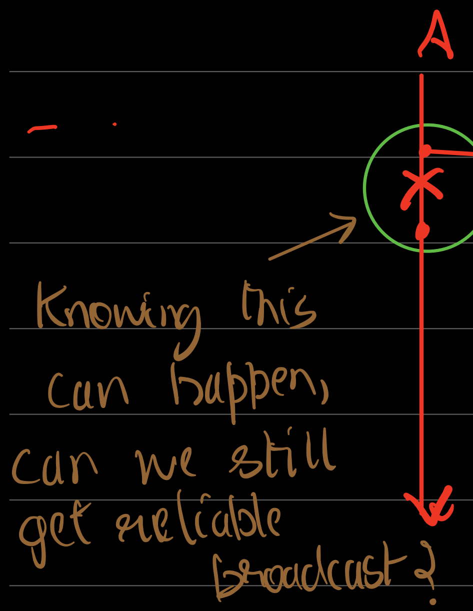
Reliable broadcast : If a correct process delivers m, then all correct processes deliver m

Correct process : Depends on your fault model !

* For now, fault model : <u>crash model</u>



- If C crashes, it is NOT a violation of reliable delivery since 'C' is not a correct process (under crash model)

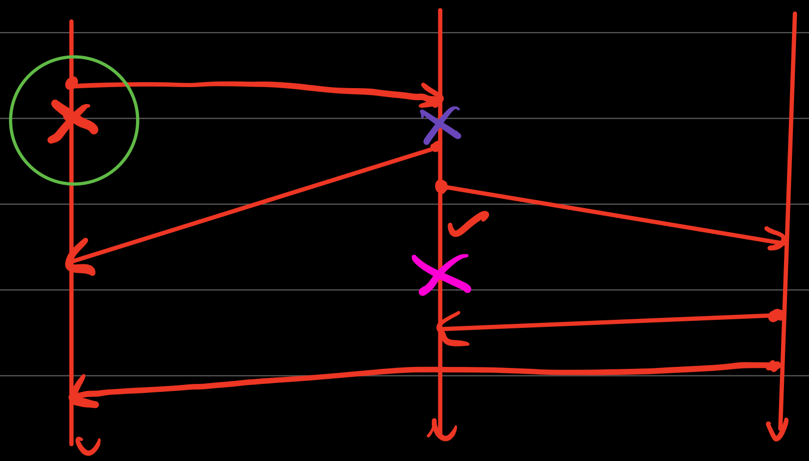
- 
A
B
C
I'm a broadcast
Knowing this can happen, can we still get reliable broadcast?

A crashed before sending to C

- NOT reliable broadcast

For brown,

- Before delivering (v), broadcast to everyone else first & then deliver.



In purple, if bob crashed, then:
- Bob would not deliver to itself
- This is OK.

In pink, bob crashes after sending to carol
- Only carol is the correct process.
- whether or not she delivers, it will still NOT be a violation i.e its OK.

Does this protocol result in duplicate msgs?
- Absolutely!
- Can be solved by keeping track of delivered (✓) messages
  - Once delivered, ignore any duplicates

- Fault tolerance often involves making multiple copies!!
- Can also be implemented for the omission model.

Replication: save the state in multiple processes to avoid data loss.