consensus - 1

get a group of processes **all agree** on the same value,

even when **network may fail, processes may fail**

replicated state machines

the key idea

- state machine:

  f(state, input) -> (updated_state, output)

- replicate the state machine on N different processes (must on **different physical machines** to tolerate fault)
- all processes should produce exactly the same outputs

to tolerate failures

quorum: any majority of processes can keep the service available

N	Quorum	Tolerated failures *f*
1	1	0
2	2	0
3	**2**	**1**
4	3	1
5	**3**	**2**
6	4	2

Common group size

insight:

1. group size in practice should be an odd number, 因为同样的可tolerate数下能够少用一台机器
2. at least N = 3 could tolerate 1 failure

replicated state machines example: Google Chubby

Chubby replicas keep their log (input) consistent using a consensus algorithm.

- use case 1: lock service

  clients can acquire locks on files

- use case 2: leader election

  - All candidates attempt to lock a file, only 1 succeeds
  - The winner records its identity in the locked file and releases the lock
  - Other candidates can identify the leader by reading the contents of the file

defining consensus

- examples of values to agree on:
  - distributed mutual exlusion: acquiring the lock
  - total-order broadcast: next message to deliver
  - replicated state machines: the next state update
- desirable properties
  - agreement (safety - 坏事不会发生)

    all **correct** processes must output the same value

    if failed, the process outputs nothing

  - validity (safety - 坏事不会发生)

    output must have been provided as input

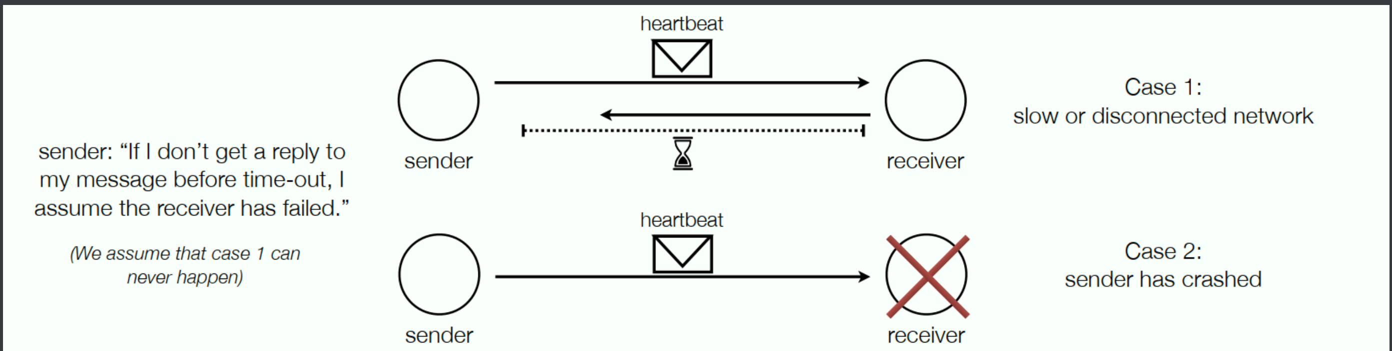
  - termination (liveness - 好事总会发生)

    every correct process *eventually* outputs some value

system models

synchronous system model

- synchronized clocks,
- an upper bound on message delivery across network links
- can distinguish network failure or a process failure by **time-outs**



asynchronous system model

- no synchronized clock,
- no upper bound (unbounded) message delivery across network links
- can't accurately distinguish between a node / network failure

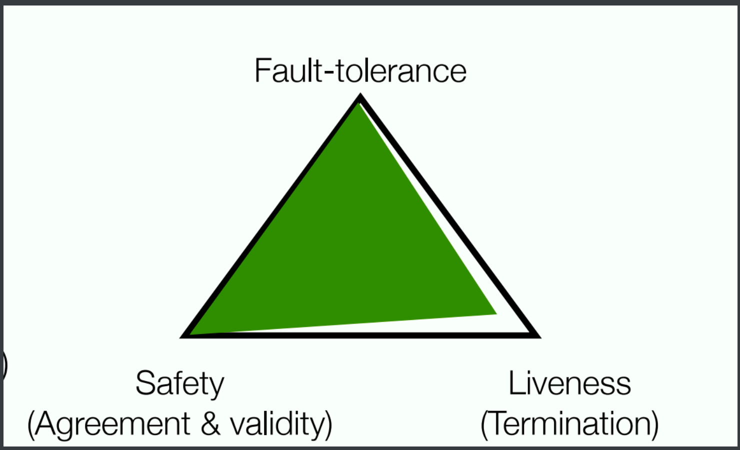
partially synchronous system model

- better approximates real-world distributed systems

  (synchronous model is overly optimistic, asynchronous modelis overly pessimistic)
- systems behaves synchronously most of the time
- but there may be bounded periods where the system behaves asynchronously

FLP impossibility result

- **"agreement, termination & fault-tolerance: choose two"**
- doesn't mean "always impossible"! just "can't be guaranteed"



- by assuming a **partially synchronous system model**, we can give up on liveness but still achieve it with high probability (asynchronous periods are bounded)

- solution in practice for fault detection & repair
  - detect failures using time-outs
  - checkpoint / restore crashed processes
  - apply randomness to avoid electing the same failing process as a leader over and over again

consensus & replicated state machines

- consensus <=> reliable total-order broadcast problem,

  ```
  on request to perform update u do
      send u via FIFO-total order broadcast
  end on

  on delivering u through FIFO-total order broadcast do
      update state using arbitrary deterministic logic!
  end on
  ```

  if we can implement reliable total-order braodcast, we can implement consensus
- 2 algorithms for total-order braodcast,:
  1. single-leader

     but if leader crashes
  2. lamport timestamps

     but if any progress fails to send an ack

  -> neither of these tolerates failures!

weaker broadcast protocols for replication?

if we assume that the state updates might be **commutative**

- Two state updates *f* and *g* are commutative if $f(g(x)) = g(f(x))$ for all states x

we can use:

- causal-order broadcast:

  只要有严格causal顺序的按顺序deliver即可

assuming **all concurrent updates** are commutative

- reliable-order broadcast:

  只要能保证reliable即可

  assuming **all updates** are commutative