Communication et Routage

IF309

c travers

Trivia

Two parts

- Algorithmique pour les gros volumes de données. Olivier Beaumont olivier.beaumont@inria.fr

- Algorithmique pour la coordination dans les systèmes distribués. Corentin Travers ctravers@enseirb-matmeca.fr

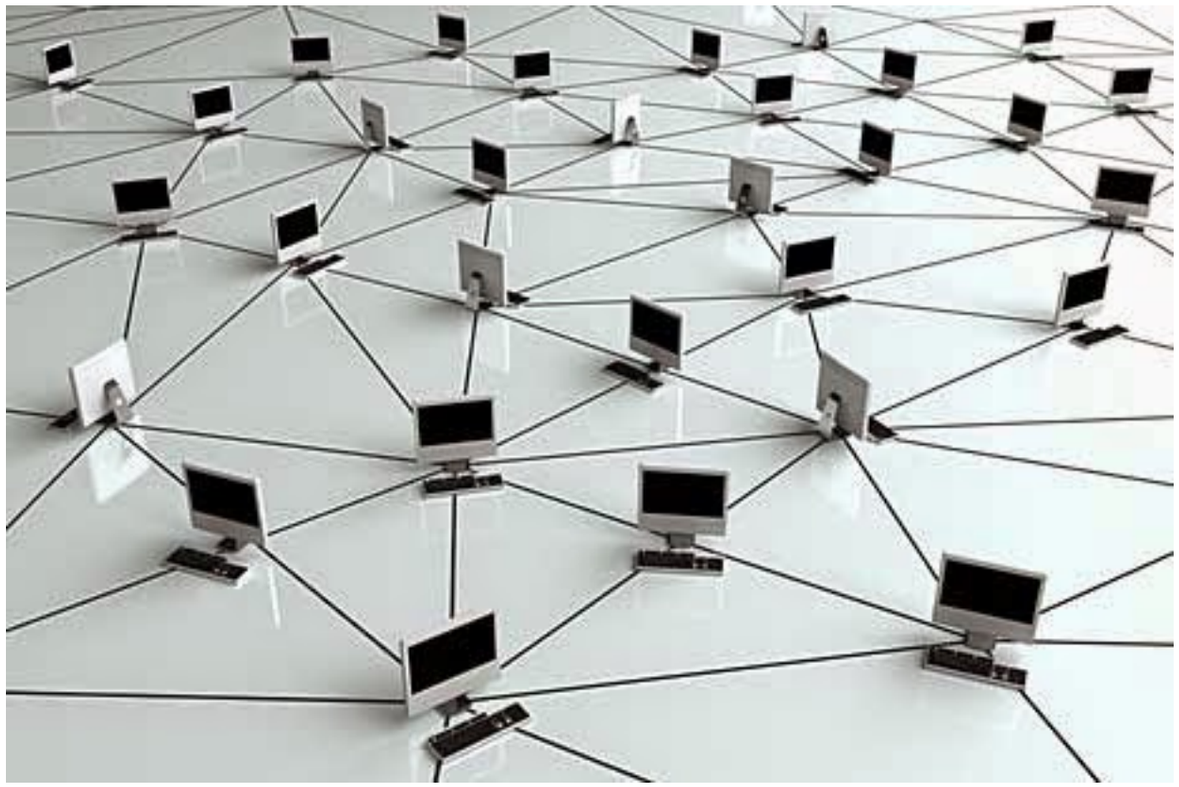
webpage:

- http://ctravers.vvv.enseirb-matmeca.fr/IF306/

Student seminar

- ~15/20mins presentation by pair of students
- mini-course from a research paper, book chapter, etc.
- audience is the class
- December 14, everyone must be there
- Suggestion of topics will be available soon
- do not hesitate to contact Olivier or me while preparing your talk

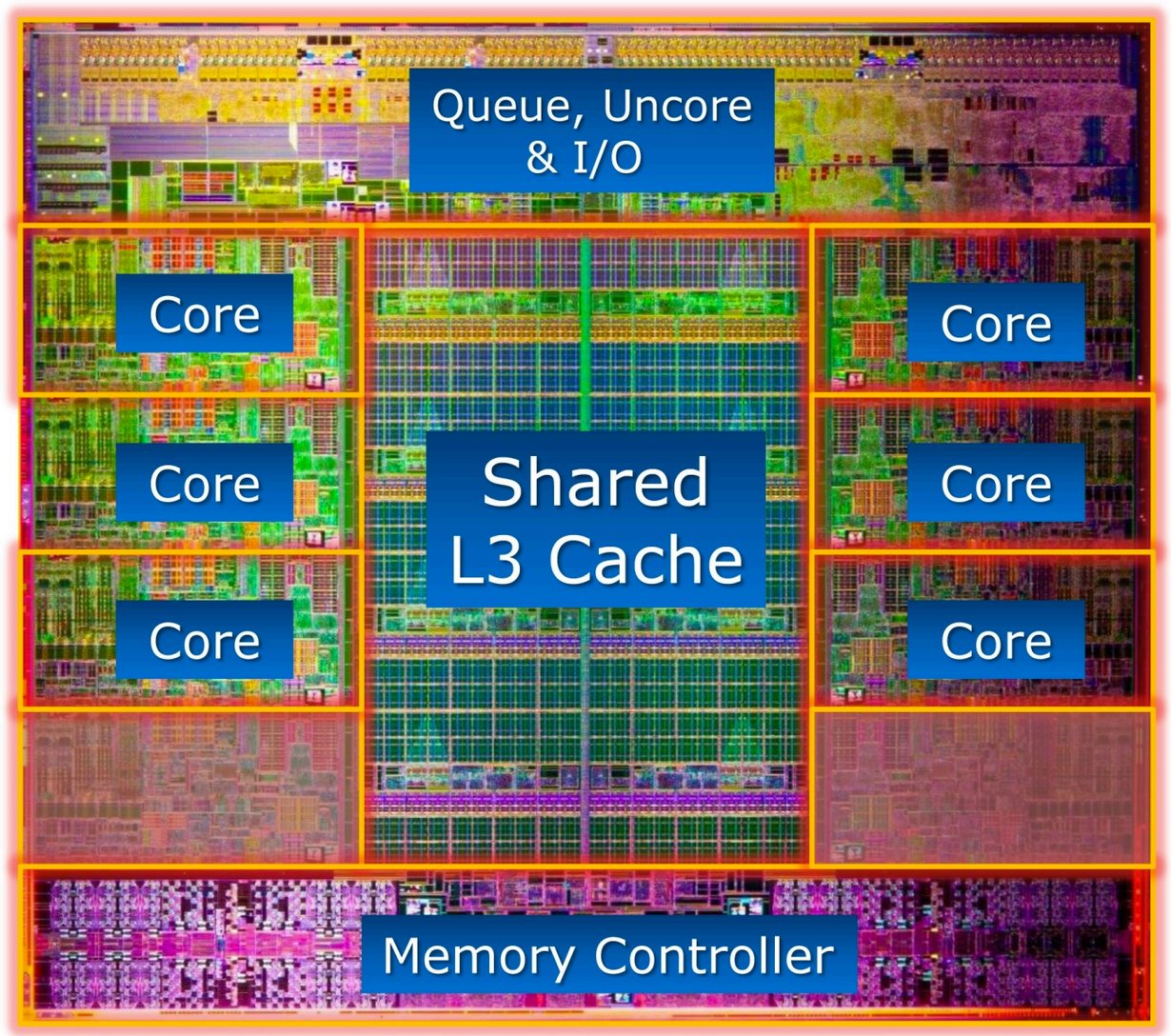
Distributed Systems



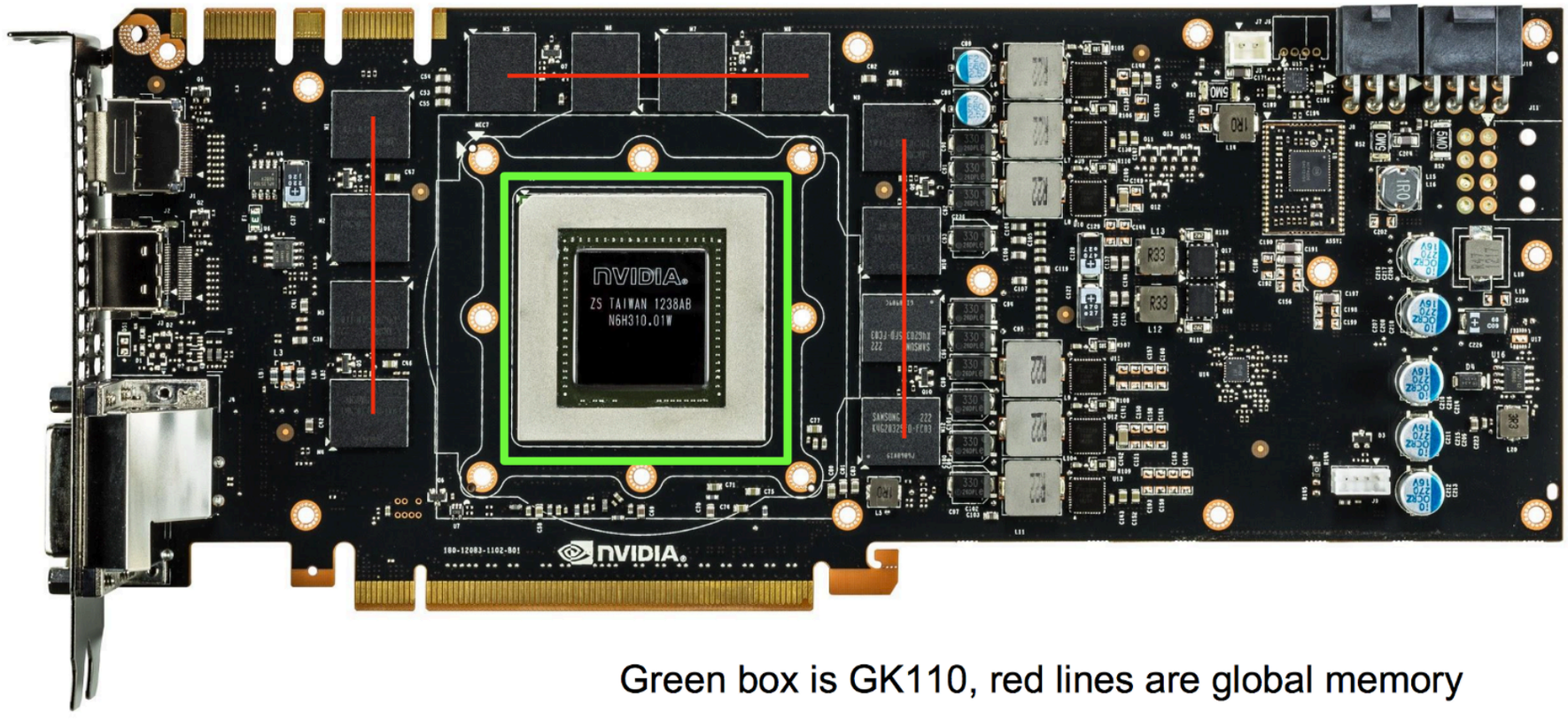
Distributed Algorithms

- E. Dijkstra (mutual exclusion) 60’s
- L. Lamport: “a distributed system is one that stops your application because a machine you have never heard from crashed” ~70’s
- J. Gray (transactions) ~70’s
- N. Lynch (consensus) ~80’s
- Birman, Schneider, Toueg (group membership) ~90’s
- P2P networks ~00’s
- Satoshi Nakamoto ~10’s

Distributed System



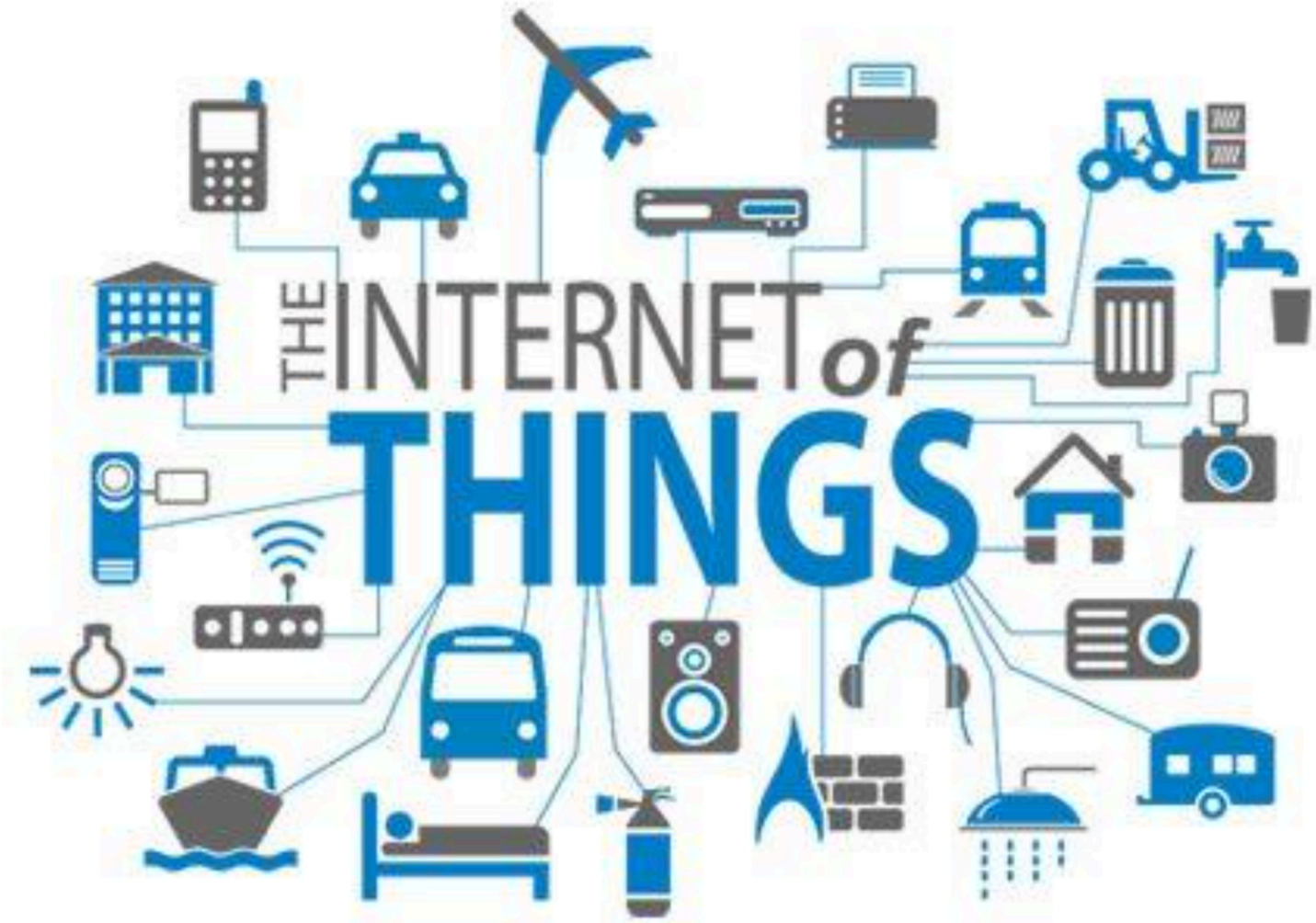
Distributed System



Green box is GK110, red lines are global memory

Distributed System

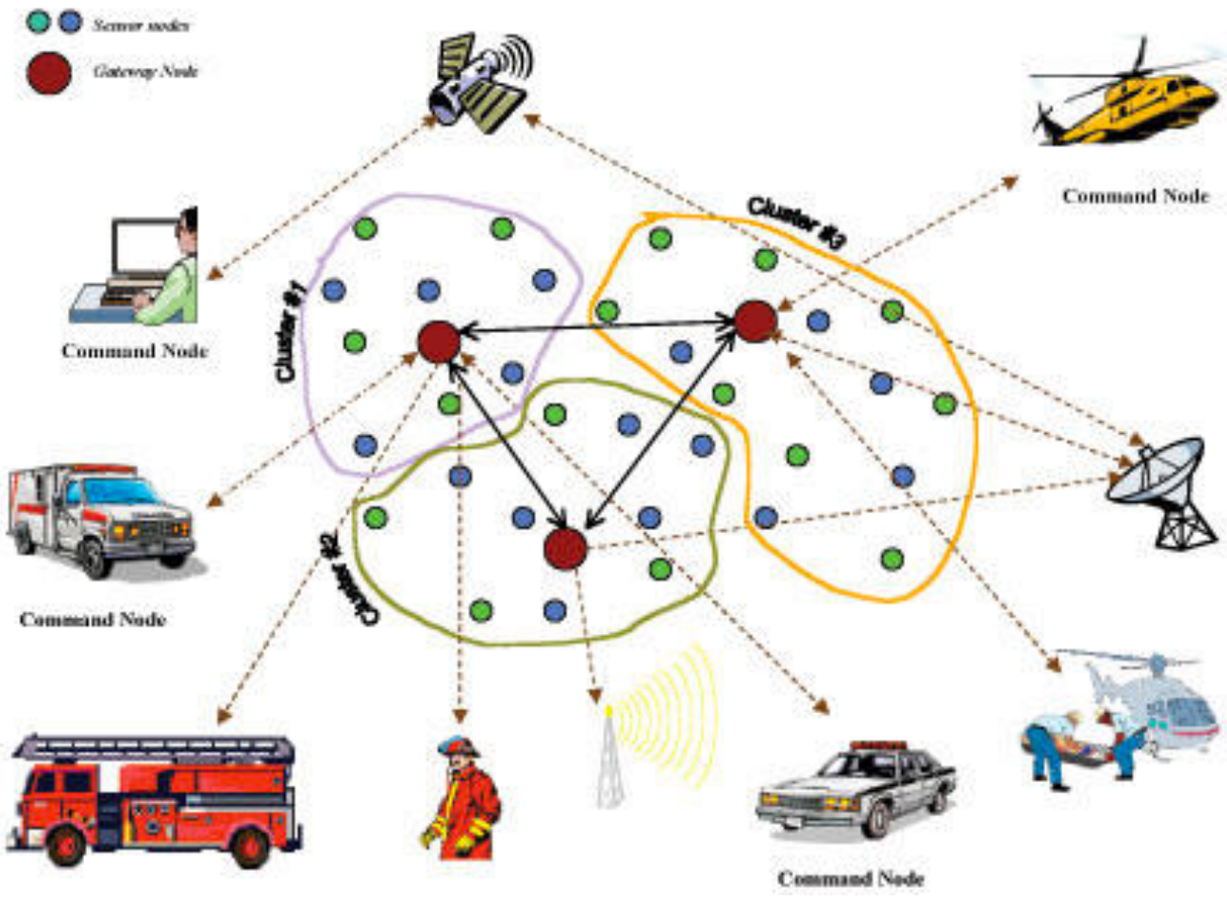
Distributed System



Kuva 1. Internet of Things. Lähde: Huffington Post

Distributed Systems

Distributed Systems



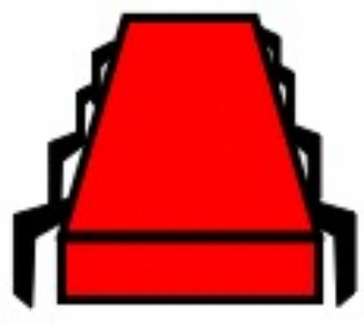
Sensor network

Distributed Computing

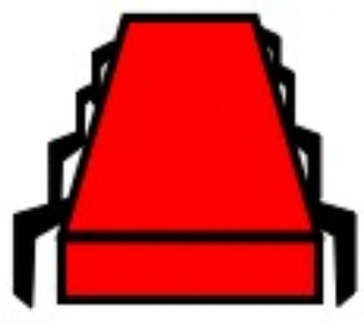
Processes

Communication medium

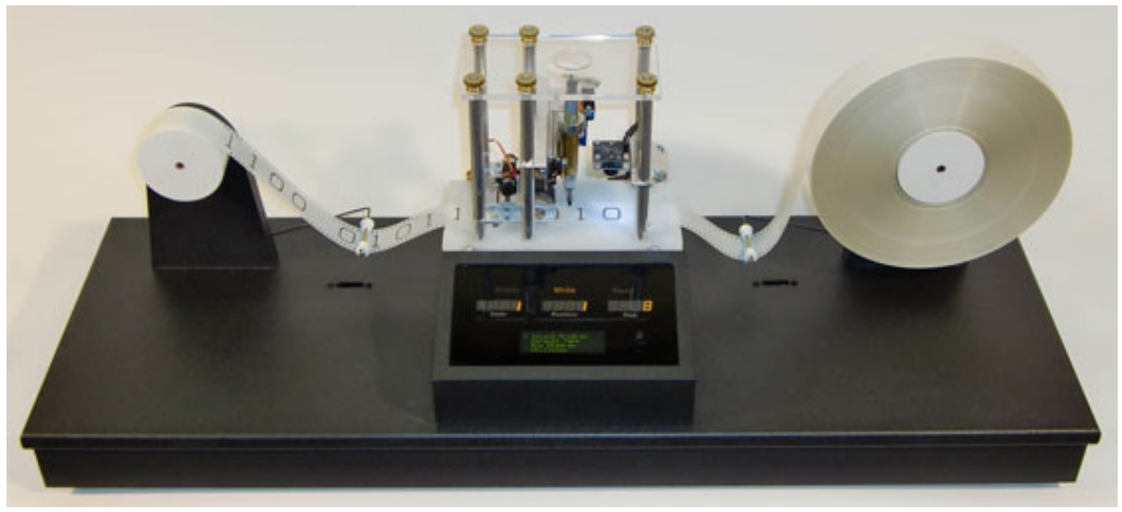
Processus, thread



Processus, thread



=



Communication

Message passing

Communication

read/write shared memory

Communication

TEST-AND-SET

CAS

LL/SC

shared object

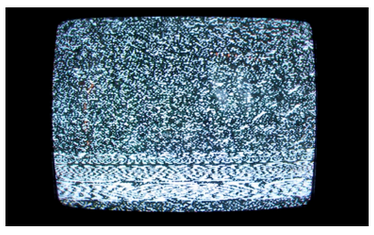
Failures

« A distributed system is one that stops your application because a machine you have never heard from crashed »

Leslie Lamport

Failures

Process Crash: unexpectedly stop and subsequently do nothing



Communication failures : faulty channel/shared object

Byzantine failure : faulty processes execute arbitrary code

Time

Processes share a clock and run at the same speed

Synchronous

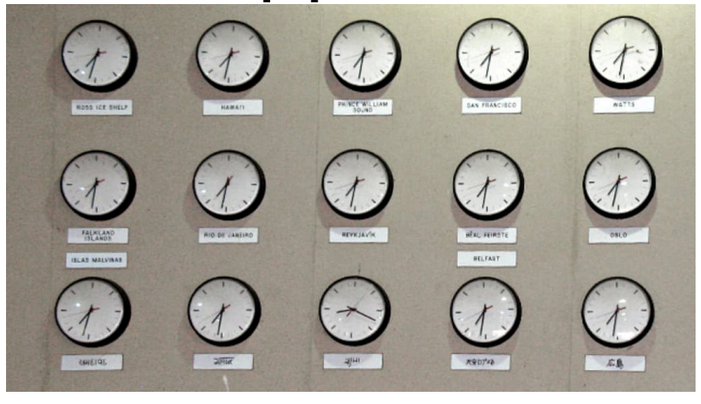
Time

Processes do not share a clock and run at their own speed

Asynchronous

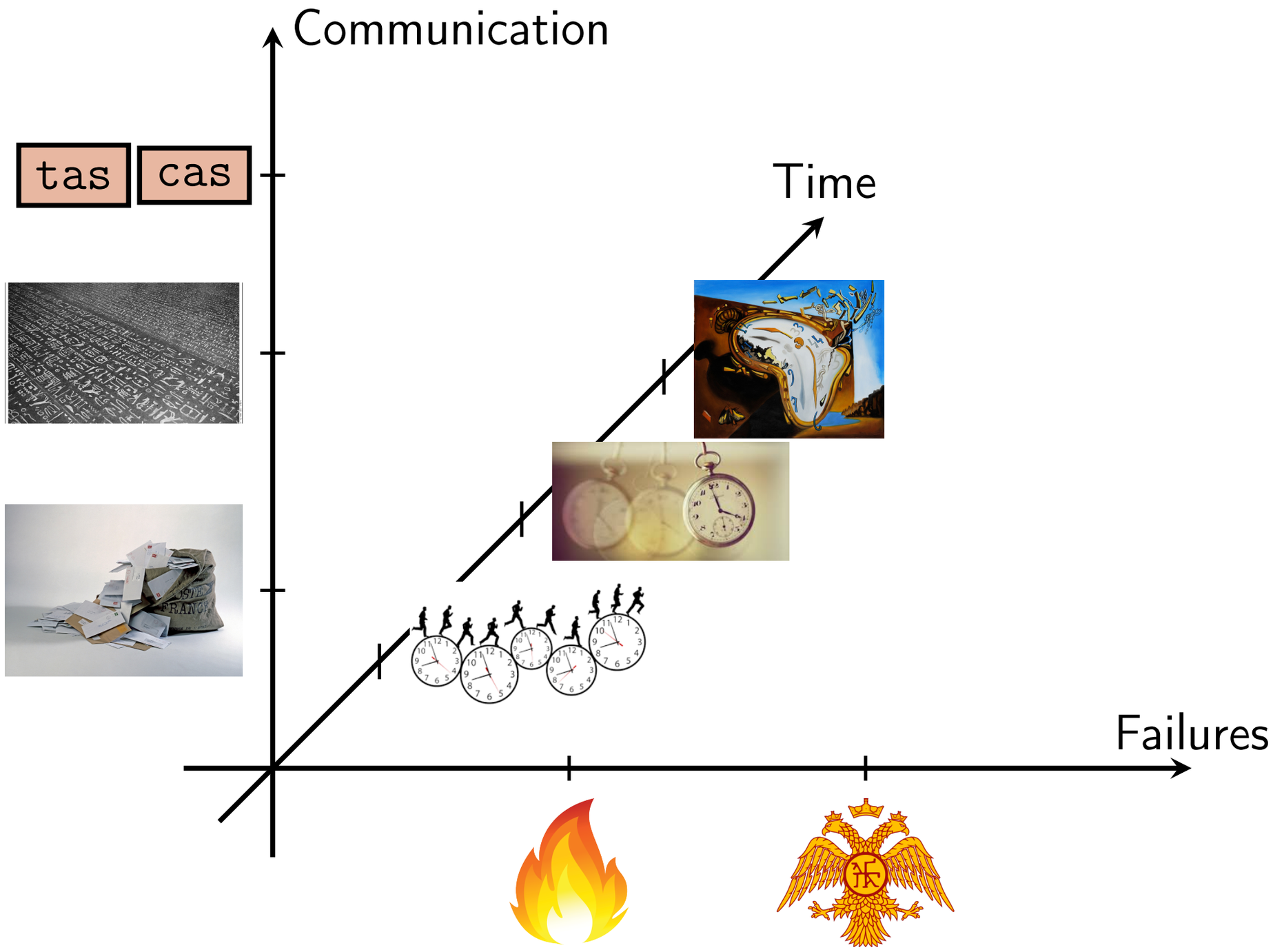
Time

Processes share an approximately synchronized clock and run at approximatively the same speed



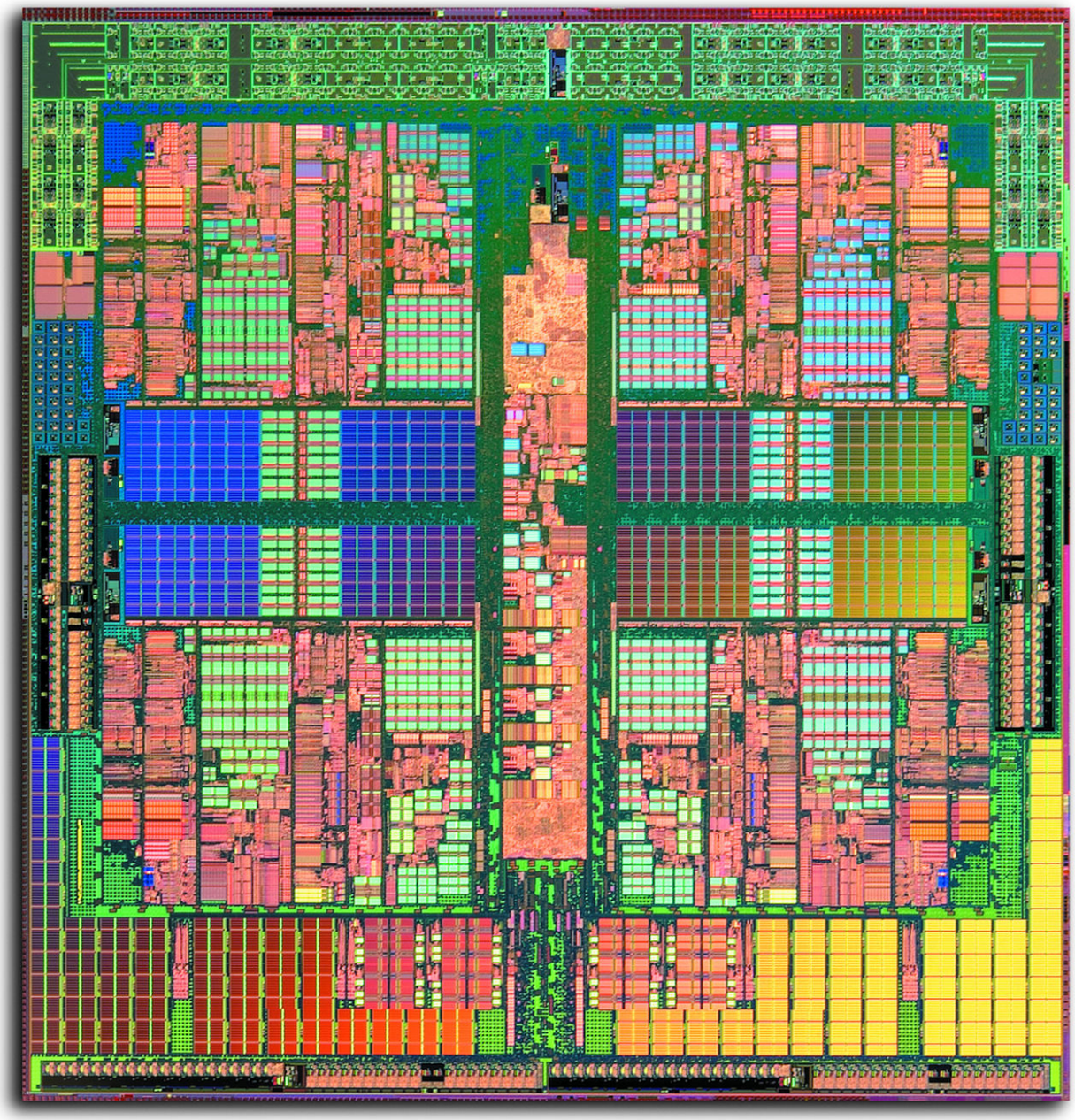
Semi-synchronous

Many Models



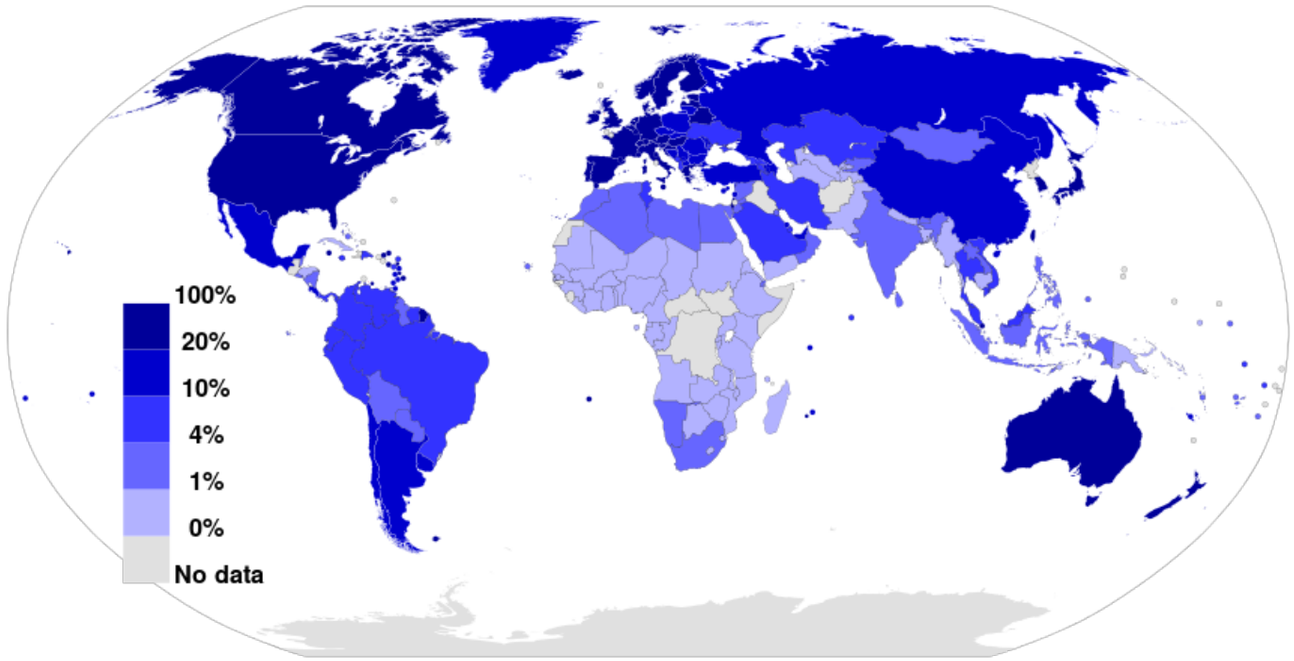
Multicore Processor

Asynchronous
Wait-free
Shared Memory



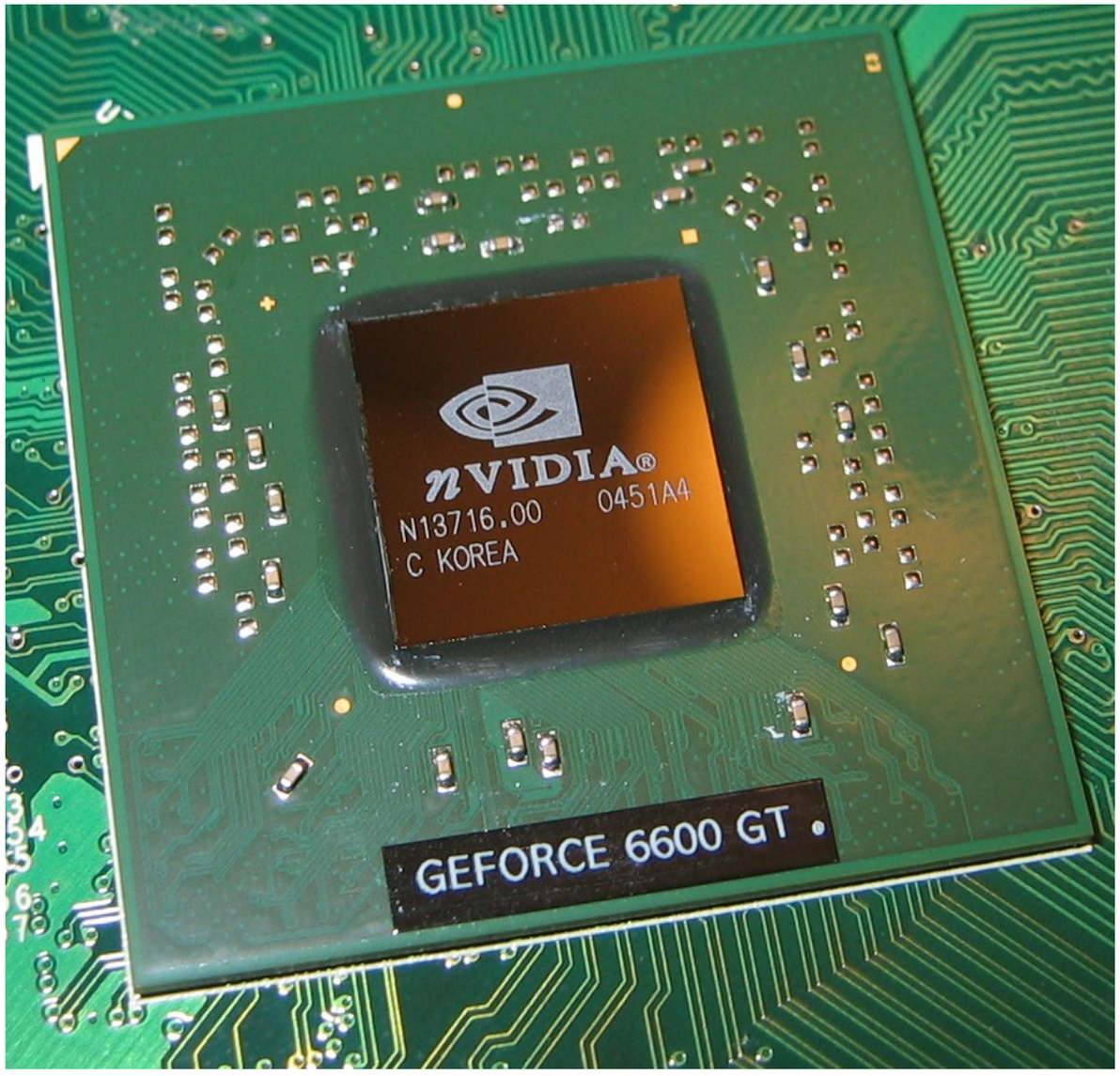
Internet

Asynchronous Message-Passing



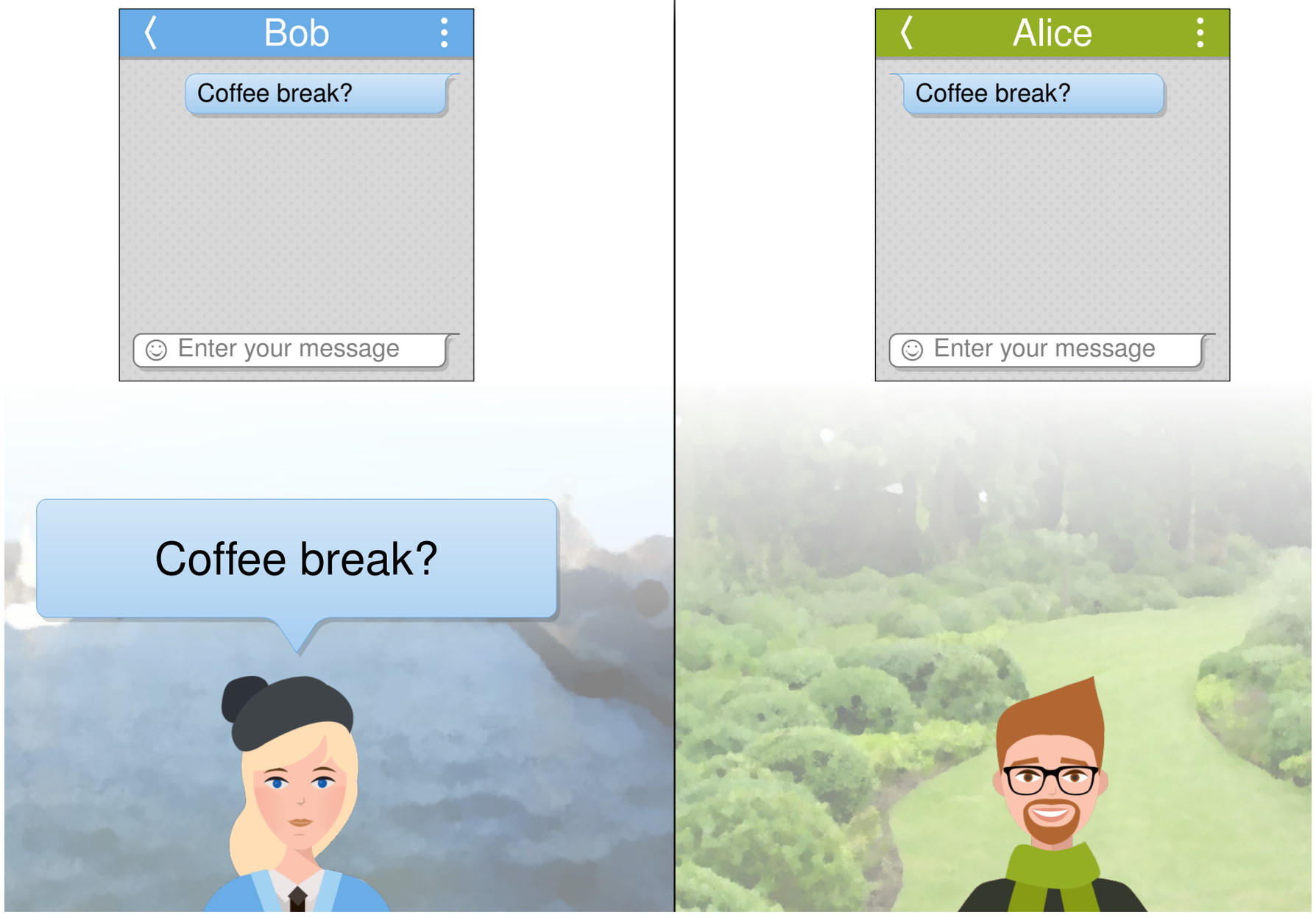
Parallel Computing

Synchronous Message-Passing/Shared Memory

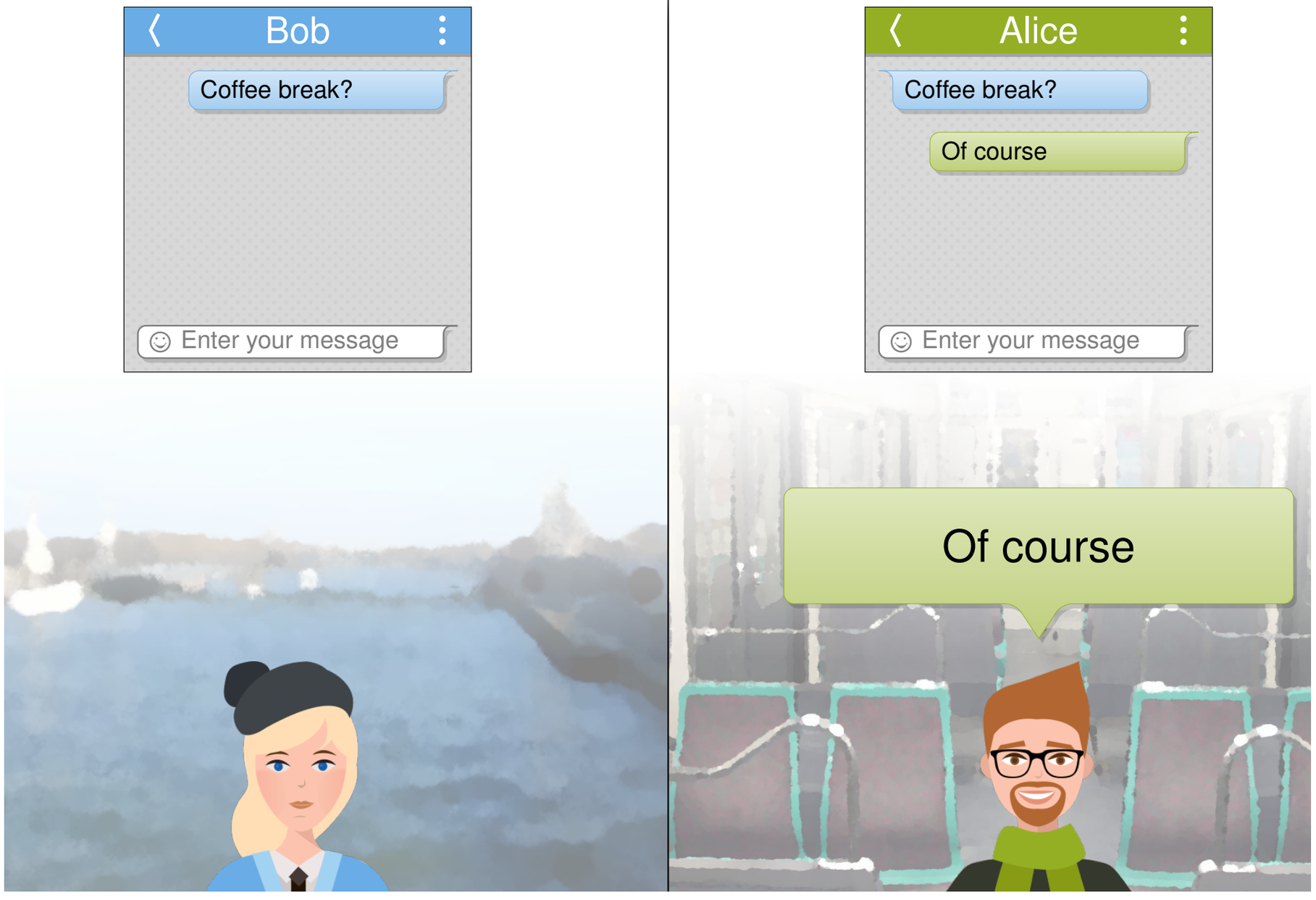


Consistency

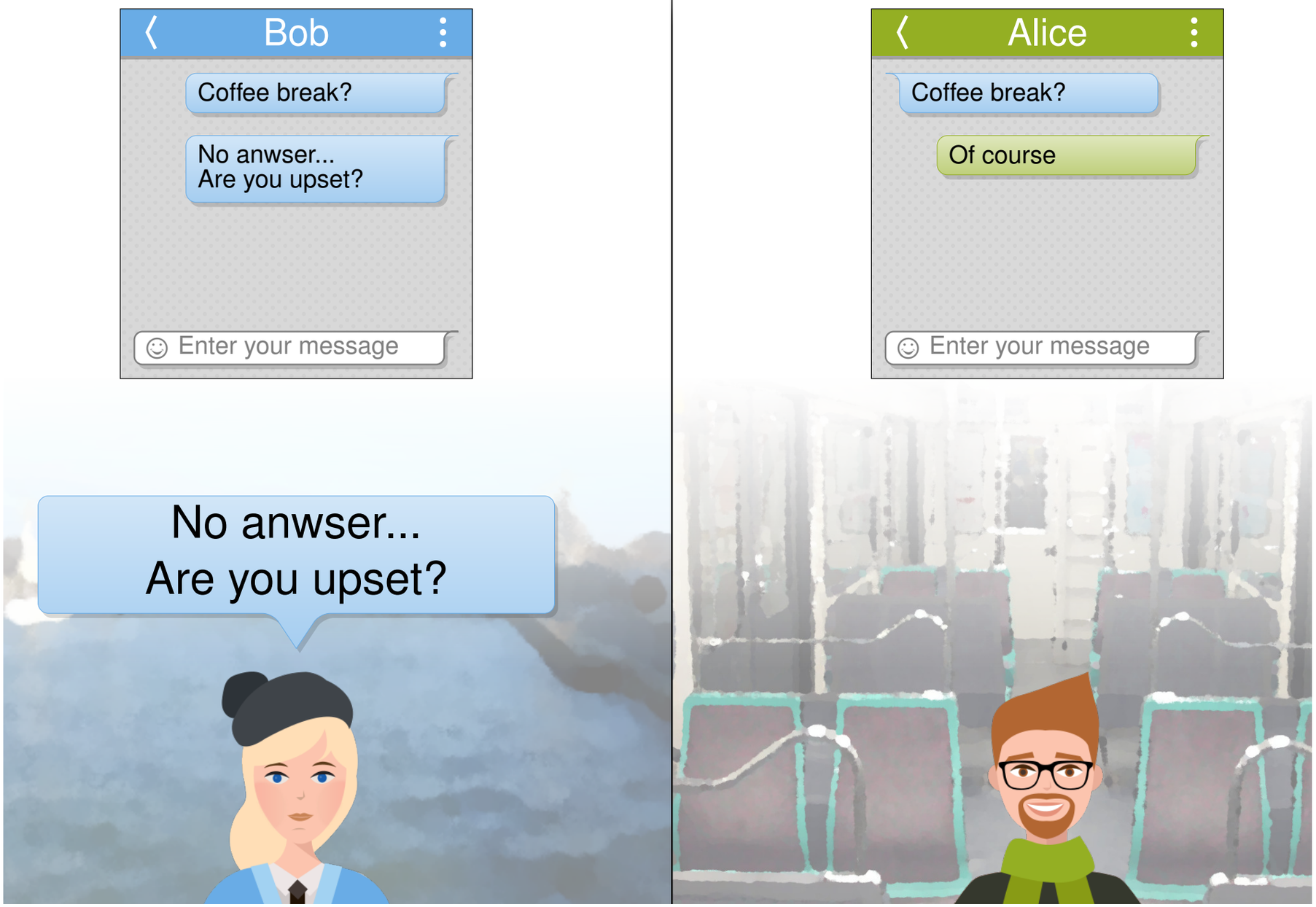
Consistency



Consistency

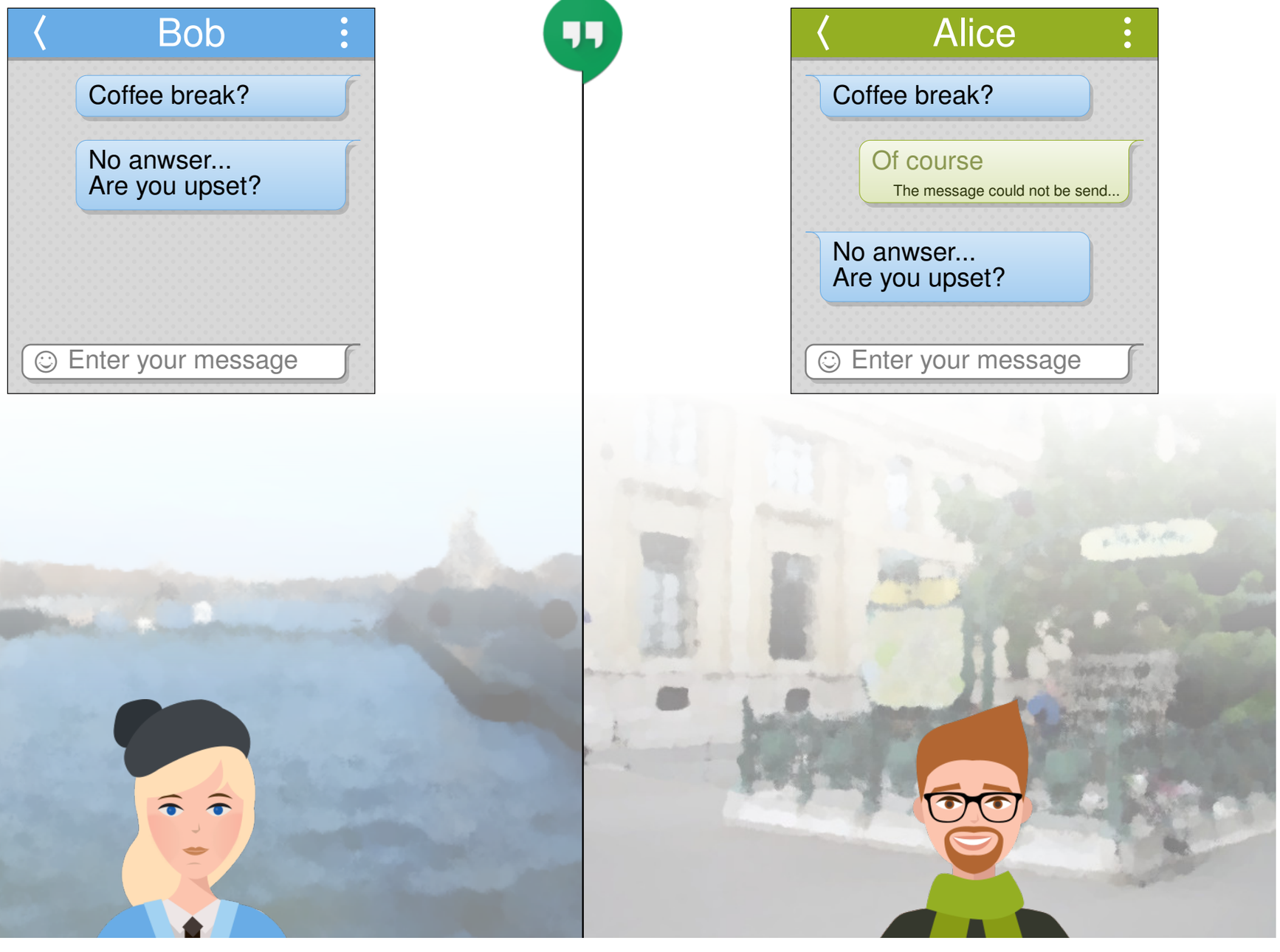


Consistency

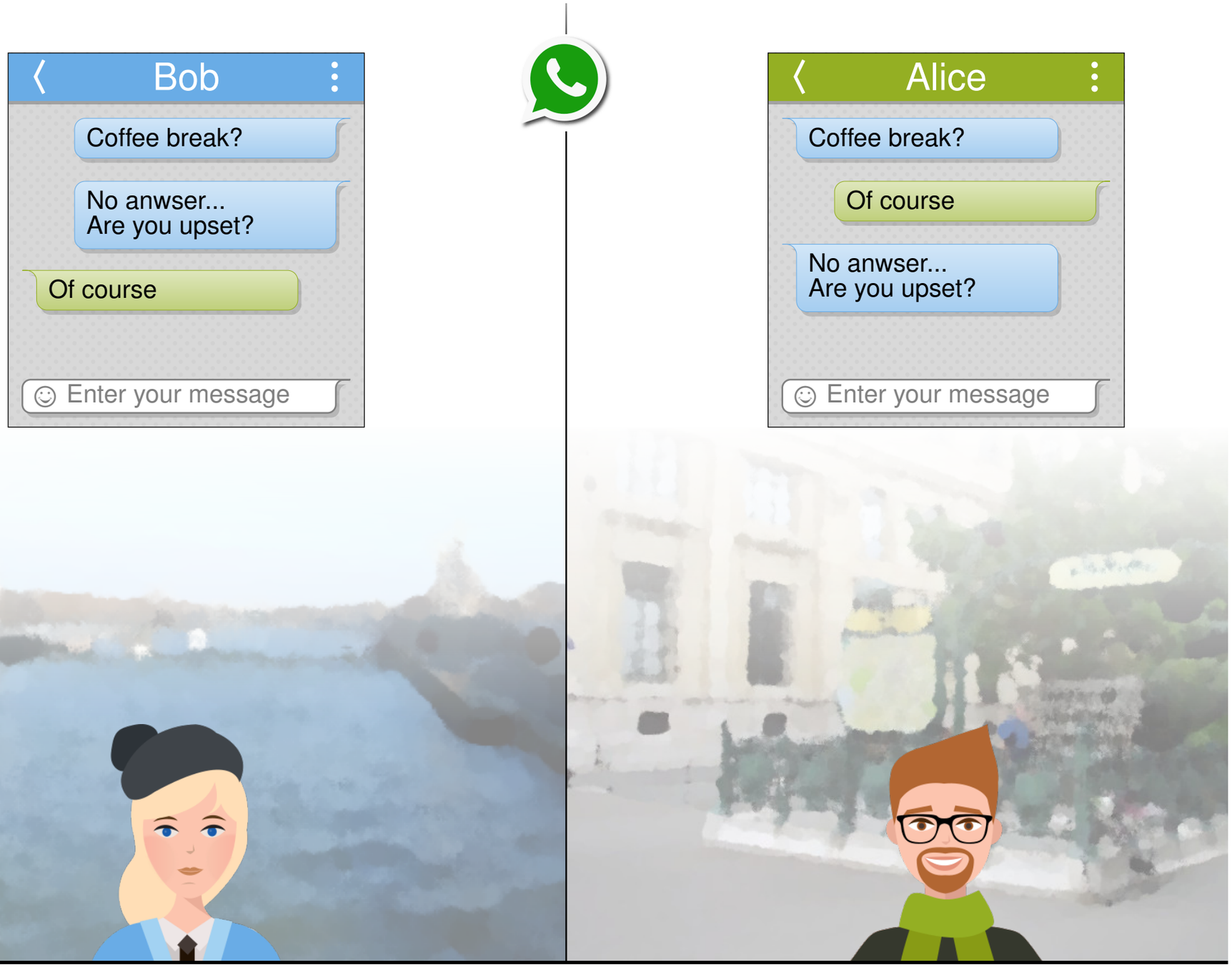


Consistency

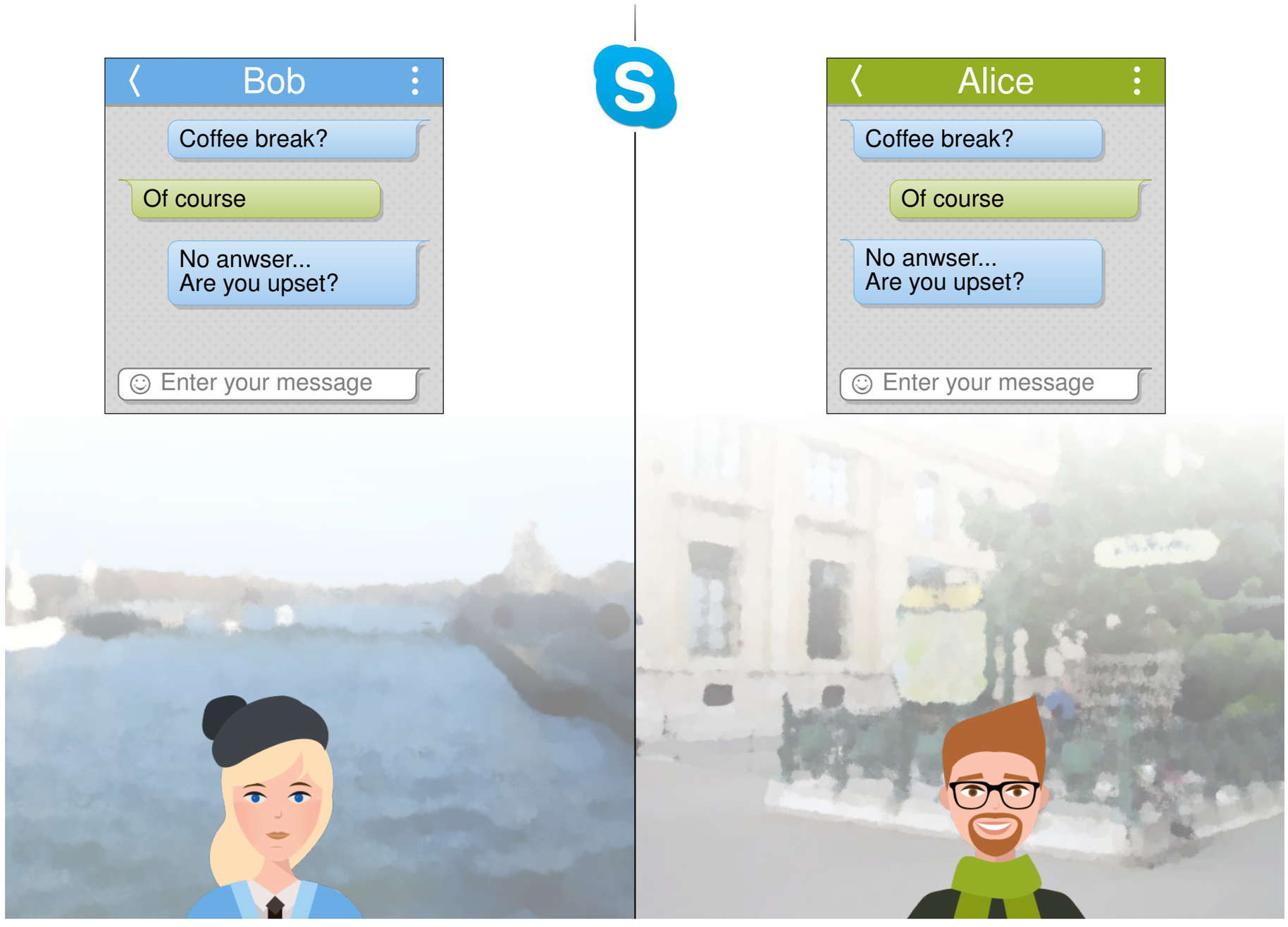
Consistency



Consistency



Consistency



Coordinated Attack

Attack at dawn

Attack at noon

- Alice and Bob must agree on when to attack
- Message-passing
- Messages may be lost (intercepted by the ennemy)

Coordinated Attack

- Alice and Bob must agree on when to attack
- Message-passing
- Messages may be lost (intercepted by the ennemy)

Coordinated Attack

Theorem

There is no protocol that ensures that Alice and Bob Attack simultaneously

Proof (Operational)

Bob receives "attack at dawn"

Proof (Operational)

Bob receives "attack at dawn"

Alice doesn't know if Bob has received "attack at dawn"

Proof (Operational)

Bob receives "attack at dawn"

Alice doesn't know if Bob has received "attack at dawn"

Bob sends an acknowledgment

Proof (Operational)

Bob receives "attack at dawn"

Alice doesn't know if Bob has received "attack at dawn"

Bob sends an acknowledgment

Bob doesn't know if Alice got that message

Proof (Operational)

Bob receives "attack at dawn"

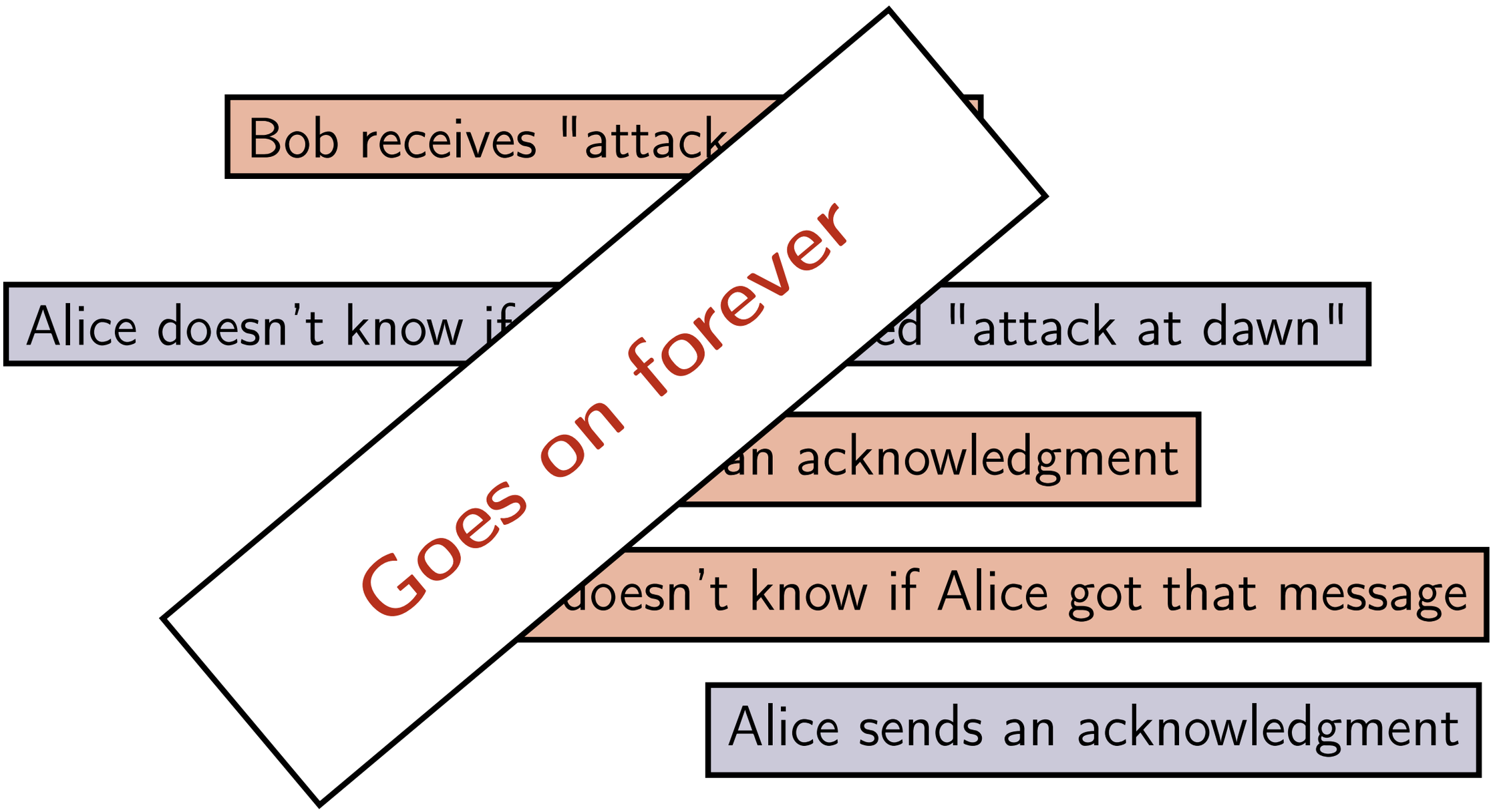
Alice doesn't know if Bob has received "attack at dawn"

Bob sends an acknowledgment

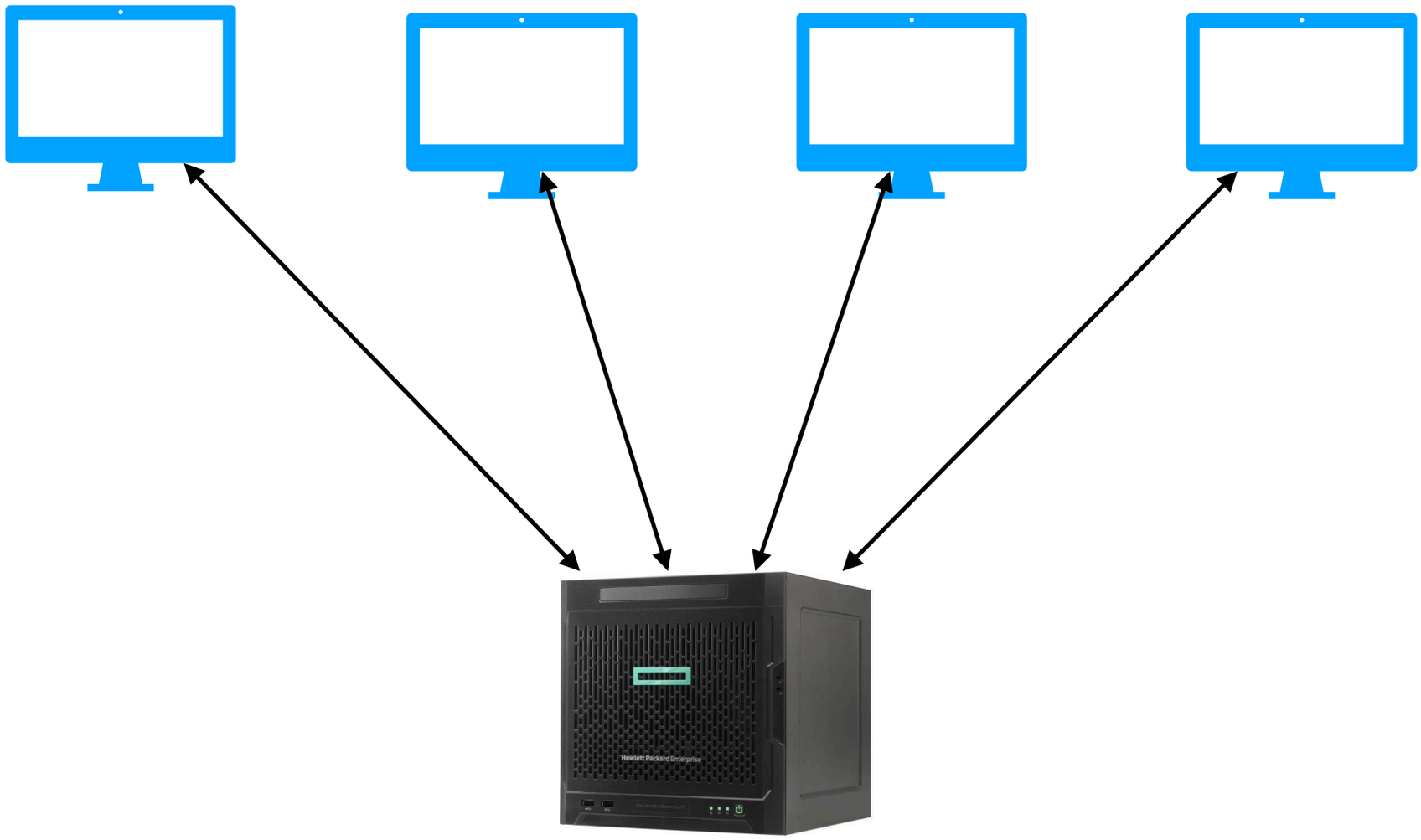
Bob doesn't know if Alice got that message

Alice sends an acknowledgment

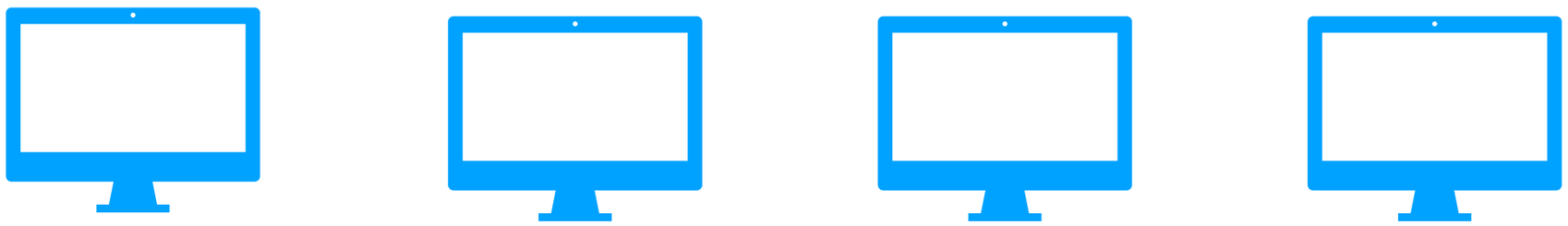
Proof (Operational)



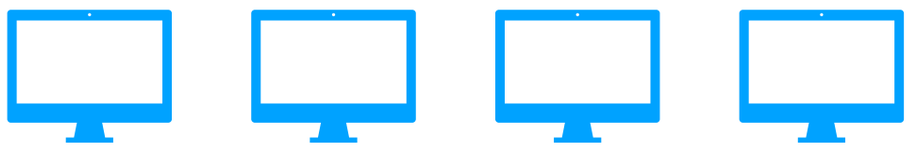
Client-Server



Client-Server

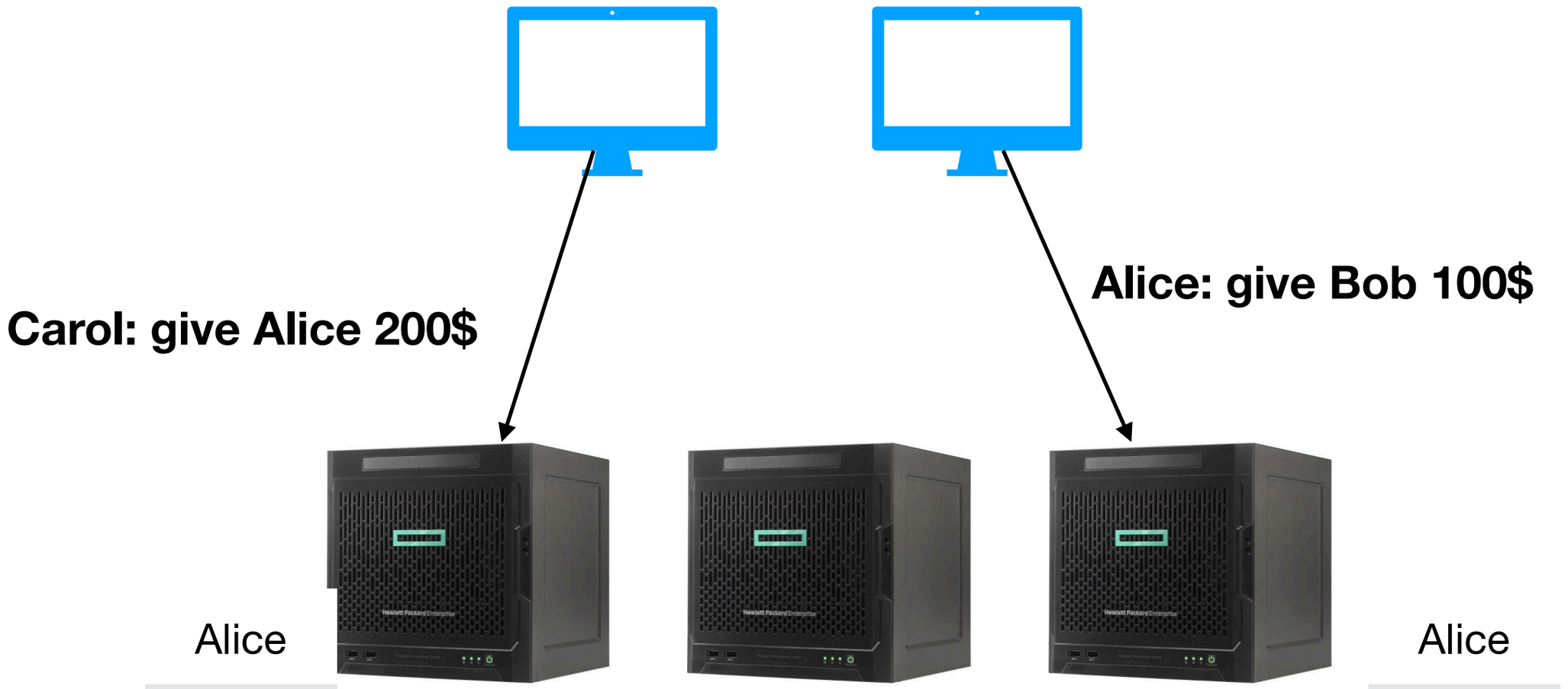


Client-Server

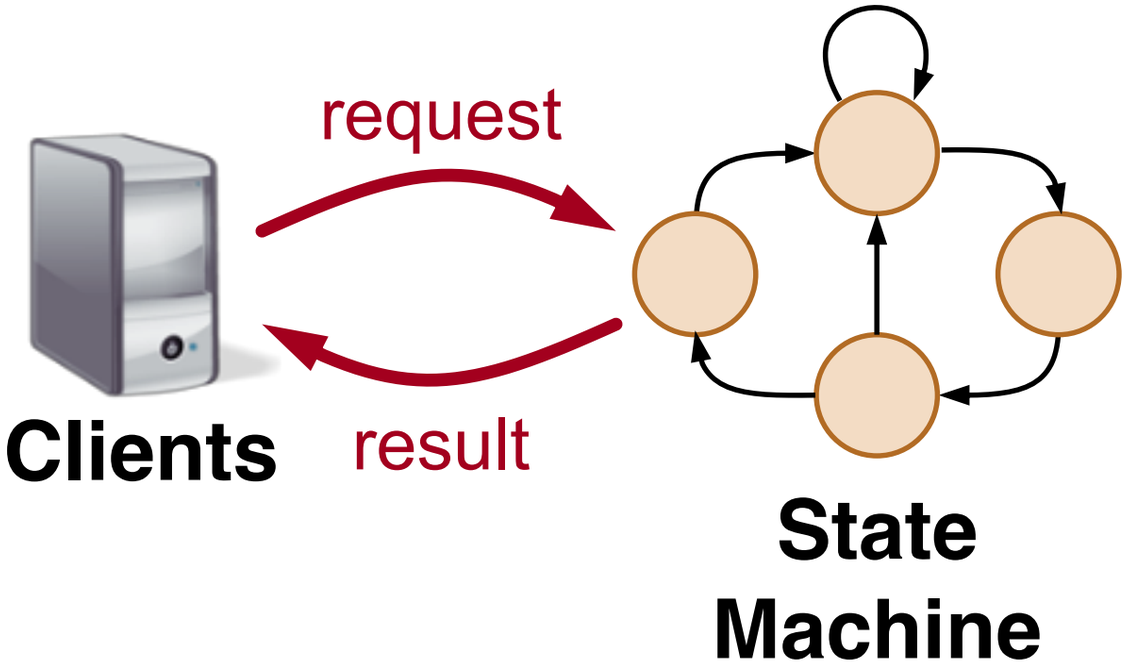


- Availability
- Fault-Tolerance
- Load-Balancing

Consistency?



Server as a State Machine

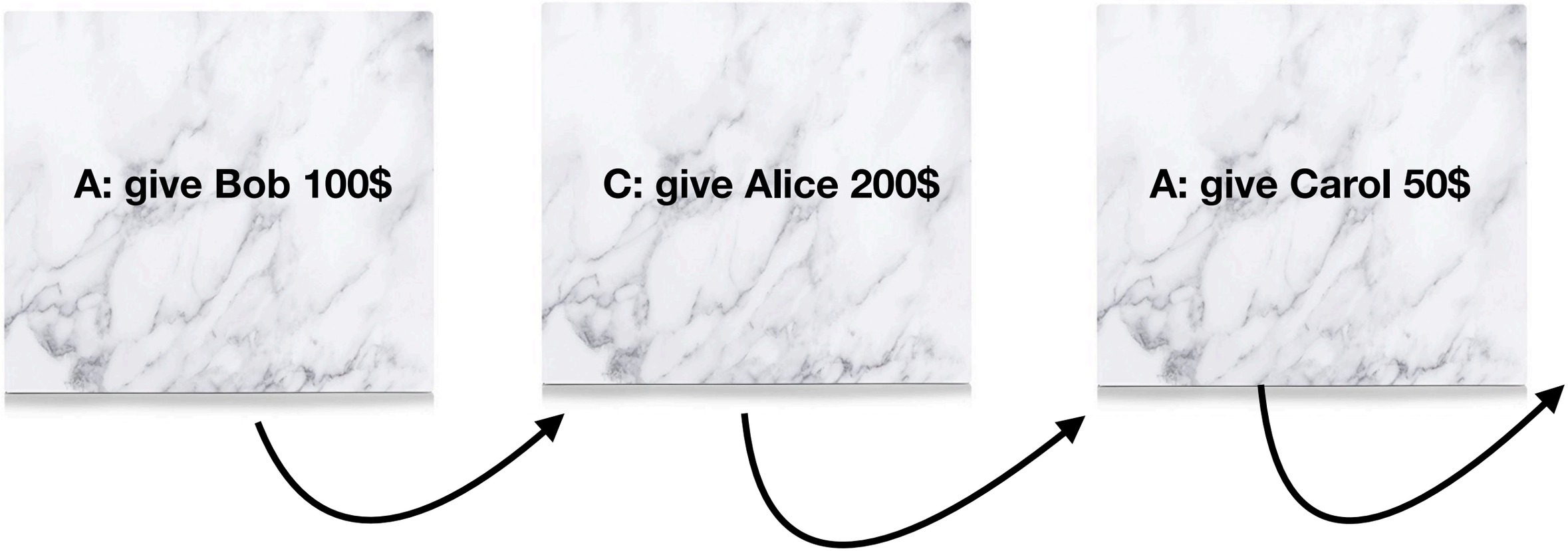


Client request -> state transition, output

State Machine Replication

Consistency: Process client requests in the same order

Blockchain



Log of requests

- **Total order**
- **Immutable**
- **Current state : replay every request in order**
- **Verifiable**

Agreement

- Fundamental problem
- Agree on the order of client request
- Which algorithms ?

This course

- **Algorithms** for **Distributed agreement** (aka consensus)
- Message passing
- From synchronous, simple failures …
- … to byzantine, open system

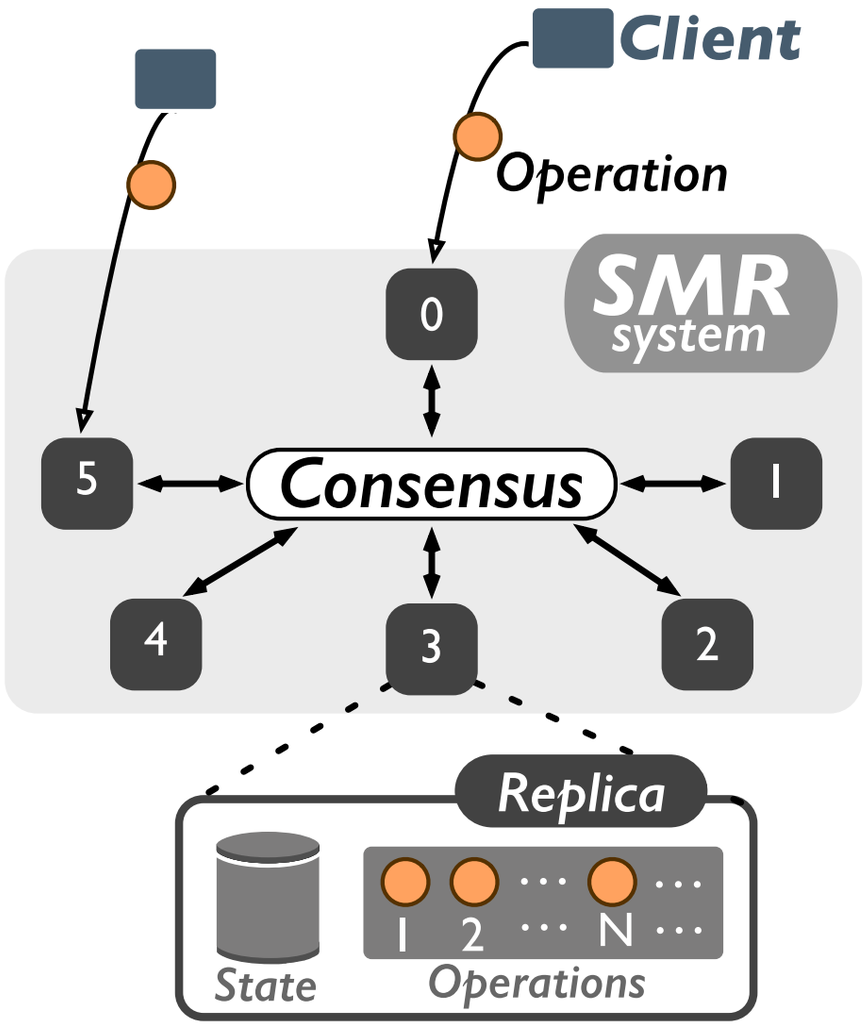
Consensus

Each process starts with an input value

Goal : **agree on one of the initial value**

- **Validity:** every decided value is an initial value
- **Agreement:** all decided values are the same
- **Termination:** every non-faulty process decides

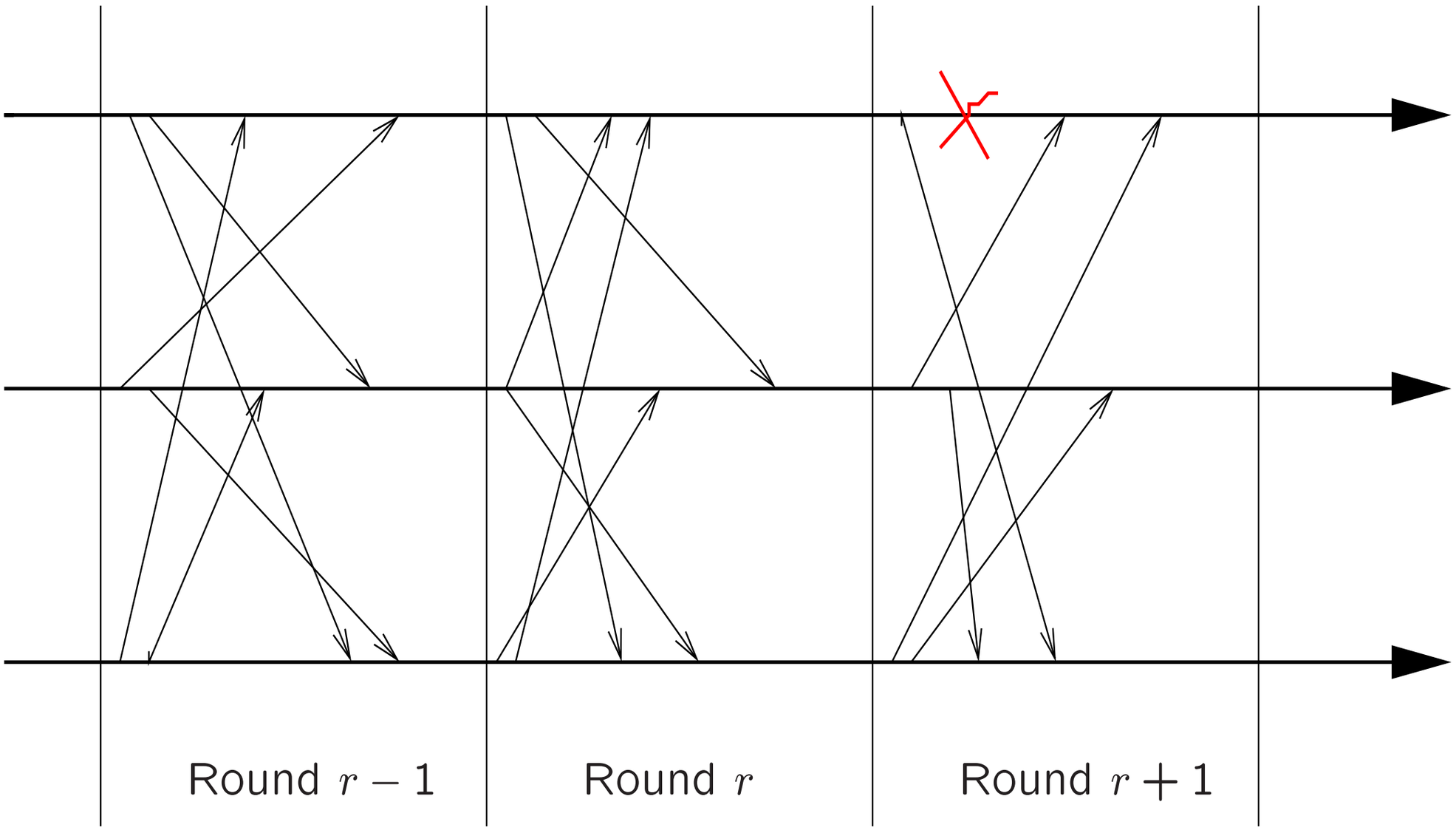
State Machine Replication



©Guerraoui et al.

Consensus in Synchronous Systems

Synchronous Model with Crash Failures



Exercise

Design a synchronous, crash tolerant consensus algorithm

- Start with 3 processes
- Initial values are integers
- Must tolerant at most t <n failures, (n is the number of processes)

Crash-tolerant Synchronous Protocol

Protocol for **n processes** **p1,p2,…,pn**

Tolerate up to **t < n** failures

Decide in **t+1 rounds**

Code for process pi

propose(v) :

est <- v

for r = **1,…,t+1** *do*

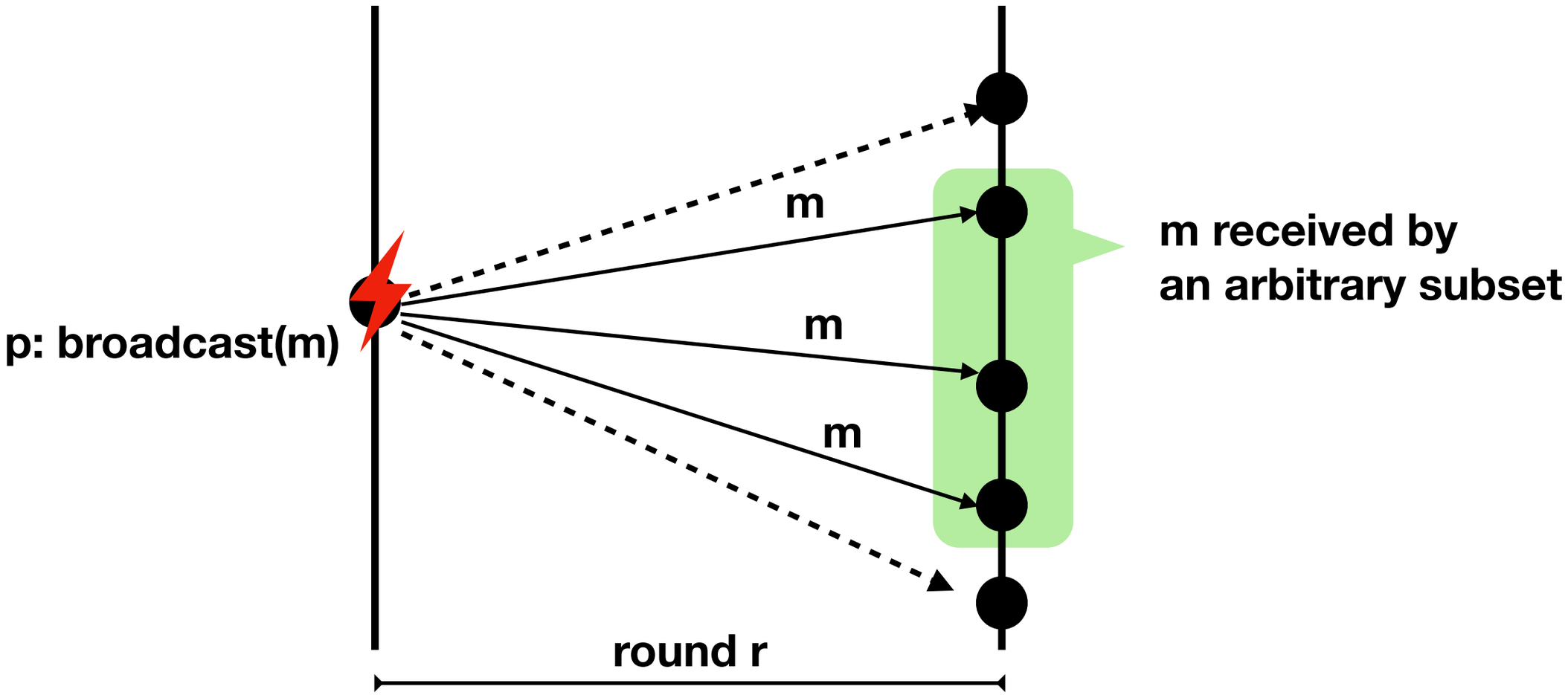
if **i = r** *then* broadcast(est) *endif*

if est’ is received *then* est <- est’ *endif*

endfor

return est

Broadcast by Faulty Process



Correctness

- **Termination**: t+1 rounds
- **Validity**: trivial

- **Agreement** : At most t failures => at least one round R coordinated by a correct process.
  At the end of round R, every non-crashed process has the same estimate

Complexity (1)

- **t+1** rounds

- **n(t+1) messages,** each message carries a value

Complexity (2)

- Protocol always costs t+1 rounds
  - even if there is no failures
  - can decision be reached faster ?

actual number of failures

Theorem : every synchronous crash-tolerant consensus protocol requires **min(f+2,t+1)** rounds

Byzantine Failures

- Processes may be corrupted ：under the control of an adversary
- Corrupted processes execute arbitrary codes
- Corrupted processes may coordinate to defeat the protocol

Byzantine Agreement

- **Termination**: every correct process decides
- **Agreement**: no two correct processes decide differently
- **Validity**: if every correct process proposes the same value v, then v is decided

Berman-Garay Protocol

- requires **t < n/4**
- **t+1** phases, rotating coordinator
- A phase : 2 rounds
  - round 1 : estimate exchange
  - round 2 : commit to the value most frequently raved in round 1 or adopt coordinator’s value

Berman Garay

operation propose(v_i)
(1) $est_i \leftarrow v_i$;
(2) **when** $r = 1, 3, \ldots, 2t-1, 2t+1$ **do**
begin synchronous round
(3) broadcast EST1(est_i);
(4) **let** rec_i = multiset of values received during round r;
(5) $most\_freq_i \leftarrow$ most frequent value in rec_i;
(6) $occ\_nb_i \leftarrow$ occurrence number of $most\_freq_i$
end synchronous round;
(7) **when** $r = 2, 4, \ldots, 2t, 2(t+1)$ **do**
begin synchronous round
(8) **if** $(i = r/2)$ **then** broadcast EST2($most\_freq_i$) **end if**;
(9) **if** (a value v is received from $p_{r/2}$) **then** $coord\_val_i \leftarrow v$ **else** $coord\_val_i \leftarrow v_i$ **end if**;
(10) **if** $(occ\_nb_i > n/2 + t)$ **then** $est_i \leftarrow most\_freq_i$ **else** $est_i \leftarrow coord\_val_i$ **end if**
(11) **if** $(r = 2(t+1))$ **then** return(est_i) **end if**
end synchronous round.

Proof

Agreement persistance: if every correct has the same estimate v at the beginning of phase k, they will never change their estimate thereafter

Theorem: if $t < n/4$, the protocol solves byzantine agreement in t+1 rounds

Improving Failures Resilience

- Berman-Garay is simple, elegant and has constant size message
- But tolerate up to t < n/4 byzantine processes

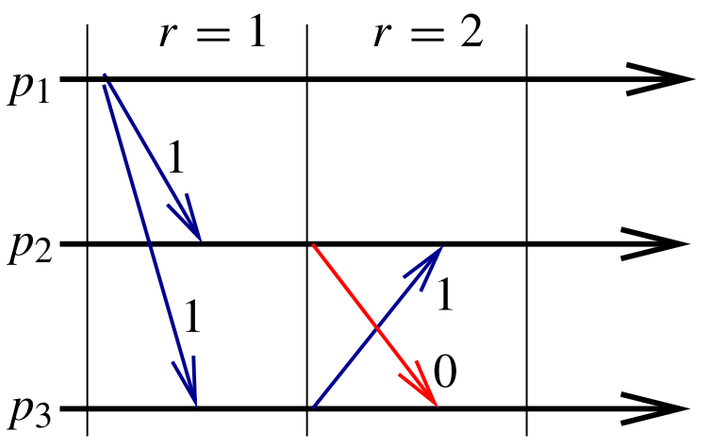
Can we do better ?

Theorem: there is no synchronous byzantine agreement protocol that tolerates **t >= n/3** failures

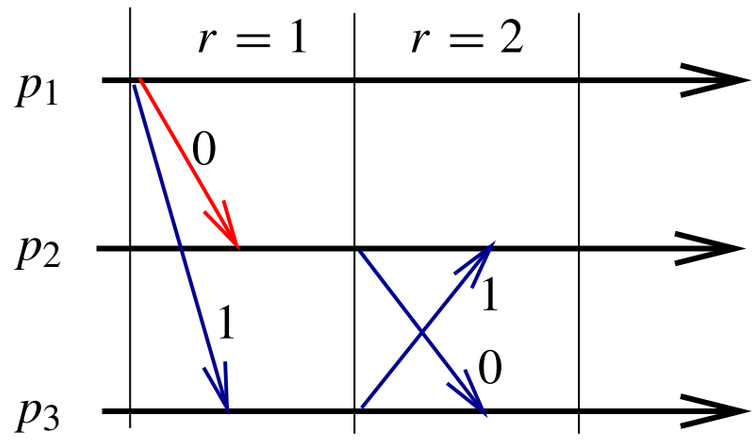
Impossibility n=3, t=1

Theorem: there is no synchronous consensus protocol for **3 processes** tolerating **1 byzantine process**

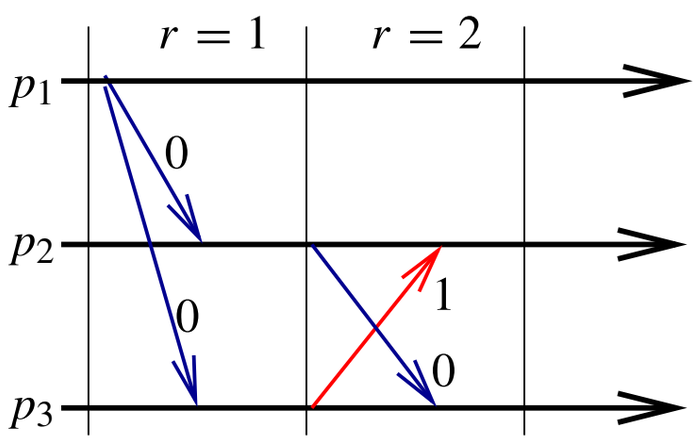
Impossibility n=3, t=1



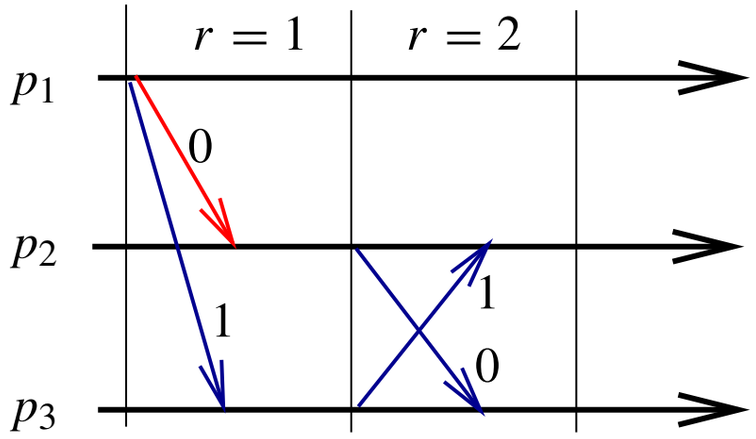
Execution $E1$



Execution $E2$



Execution $E3$



Execution $E4$

Impossibility $n \leq 3t$

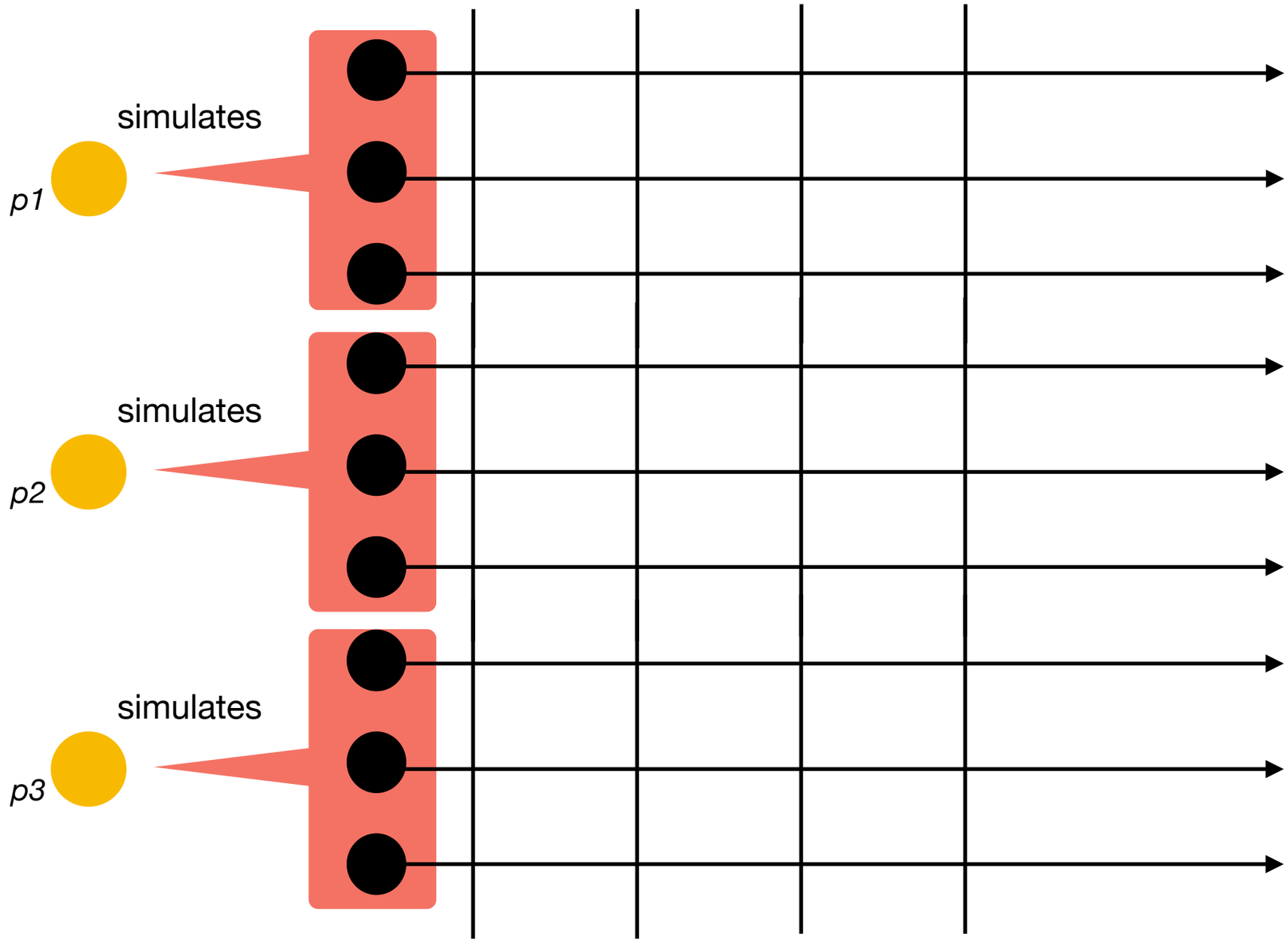
Theorem: there is no synchronous consensus protocol for n **processes** tolerating t >= n/3 **byzantine processes**

Proof

By contradiction and reduction

- Assume A solves consensus for n procs, t <= n/3 byz. procs
- Uses A to solve consensus among 3 process, 1 byz. procs

Impossibility $n \leq 3t$



Consensus in Asynchronous Systems

Asynchronous Model with Crash Failures

- Processes may **fail-stop**
- **Reliable but asynchronous** communication :
    - Any message is eventually received
    - Unpredictable time between send and receive

Exercise

Design a crash-tolerant asynchronous consensus algorithm

- For 2 processes
- Initial values are 0 or 1
- Tolerate 1 failure

Bad News

Theorem [FLP] There is no asynchronous binary consensus protocol for 2 processes that tolerates one crash failure

Consequences Asynchronous consensus requires

- Additional power, e.g., failure detection
- Relax the problem specification, e.g., liveness
- Randomization
- Any combination of the items above

(Unreliable) Leader

- **leader_i** current leader according to proc. pi
  - May change over time
  - Different procs may have different leaders for a while

Eventual leadership after some time, every process has the same non-faulty leader

Leader Based Consensus [MR01]

- Always safe, may not terminate while common leadership does not hold

- Requires $t \leq n/2$

- Asynchronous stages. In stage k:

    1. try to select a common value (each proc. picks its current leader’s value)

    2. try to commit to their current value v. If v is committed, no other value can be decided

Leader-based Consensus

```
upon propose(v):
  r ← 0 // current round
  u ← v // current estimate
  while not decided do
        r ← r + 1
        send(PHASE1, r, u) to all  // phase 1
        wait for (receive(PHASE1, r, v′) from p_l s.t. l=leader_i)
        u ← v′
        send(PHASE2, r, u) to all      // phase 2
        wait for (receive(PHASE2, r, u′) from majority of processes)
        U ← set of values u′ received in vote messages
        if U = {u′} for some u′ ≠ ⊥ then aux ← u′
        else aux ← ⊥
        send(PHASE3, r, aux) to all    // phase 3
        wait for (receive(PHASE3, r, aux′) from majority of processes)
        if (received (PHASE3, r, aux′) with aux′ = v′ ≠ ⊥) then u ← v′
        if (all (PHASE3, r, aux′) messages are such that aux′ ≠ ⊥) then
                broadcast(DECIDE, u); decided ← true
upon deliver(DECIDE, v):
  decided ← true
  decide(v)
```

Ben Or Byzantine Consensus $n > 9t$

1: $x_i \in \{0, 1\}$ ◁ input bit
2: r = 1 ◁ round
3: decided = false
4: Broadcast `propose`(x_i,r)
5: **repeat**
6: Wait until $n - f$ `propose` messages of current round r arrived
7: **if** at least $n - 2f$ `propose` messages contain the same value x **then**
8: $x_i = x$, decided = true
9: **else if** at least $n - 4f$ `propose` messages contain the same value x **then**
10: $x_i = x$
11: **else**
12: choose x_i randomly, with $Pr[x_i = 0] = Pr[x_i = 1] = 1/2$
13: **end if**
14: r = r + 1
15: Broadcast `propose`(x_i,r)
16: **until** decided (see Line 8)
17: decision = x_i