

YSC4231: Parallel, Concurrent and Distributed Programming

Distributed Consensus

Consensus

- *Common meaning:*
a way for a **set of parties** to come to a **shared** agreement.
- *In distributed computing:* ensuring that among the values proposed by a collection of processes, a **single one** is chosen.
- **Uniformity**: Only a *single* value is chosen
- **Non-triviality**: *Only* a value that has been proposed may be chosen
- **Irrevocability**: Once *agreed* on a value, the processes do not change their decision.

Why Consensus?



ethereum

Why Distributed Consensus is difficult?

- Arbitrary message delays (asynchronous network)
- Independent parties (nodes) can go offline (and also back online)
- Network partitions
- Message reorderings
- Malicious (Byzantine) parties

Why Distributed Consensus is difficult?

- Arbitrary message delays (asynchronous network)
 - Independent parties (nodes) can go offline (and also back online)
 - Network partitions
 - Message reorderings
- Malicious (Byzantine) parties

Reaching a Consensus

(and constructing a protocol for this)



Waa Cow!

Sapore

Hwang's

Reaching a Consensus on where to have a dinner

Waa Cow!

Sapore

Hwang's

Waa Cow!

Sapore

Hwang's

??



??

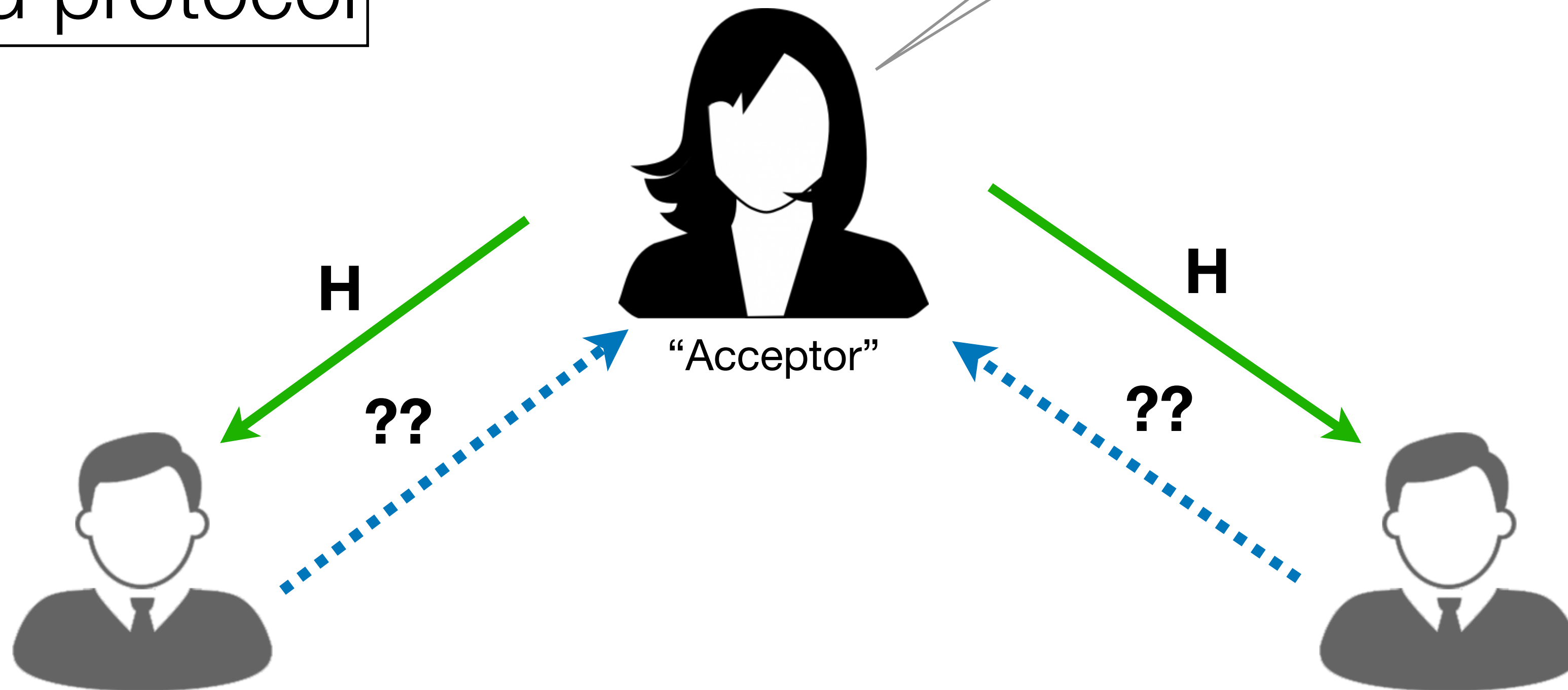


Waa Cow!

Sapore

Hwang's

Centralised protocol



Problem 1

A **single** acceptor can go *offline* or *take forever* to answer.

Waa Cow!

Sapore

Hwang's



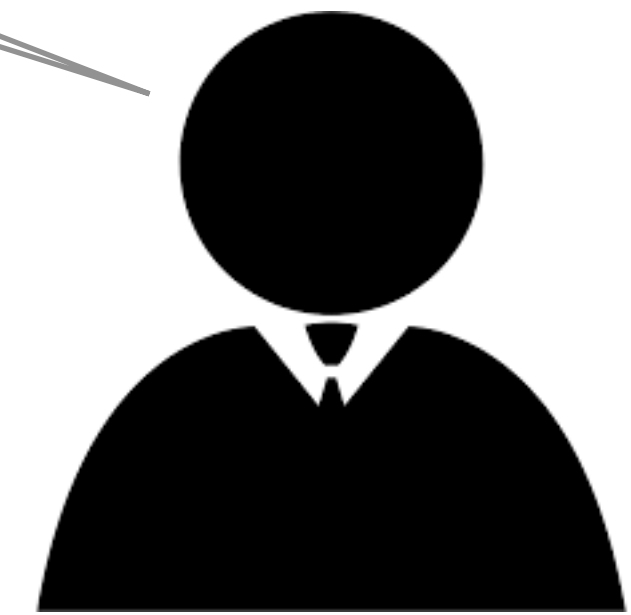
??

??

Waa Cow!

Sapore

Hwang's



??



??



Problem 2

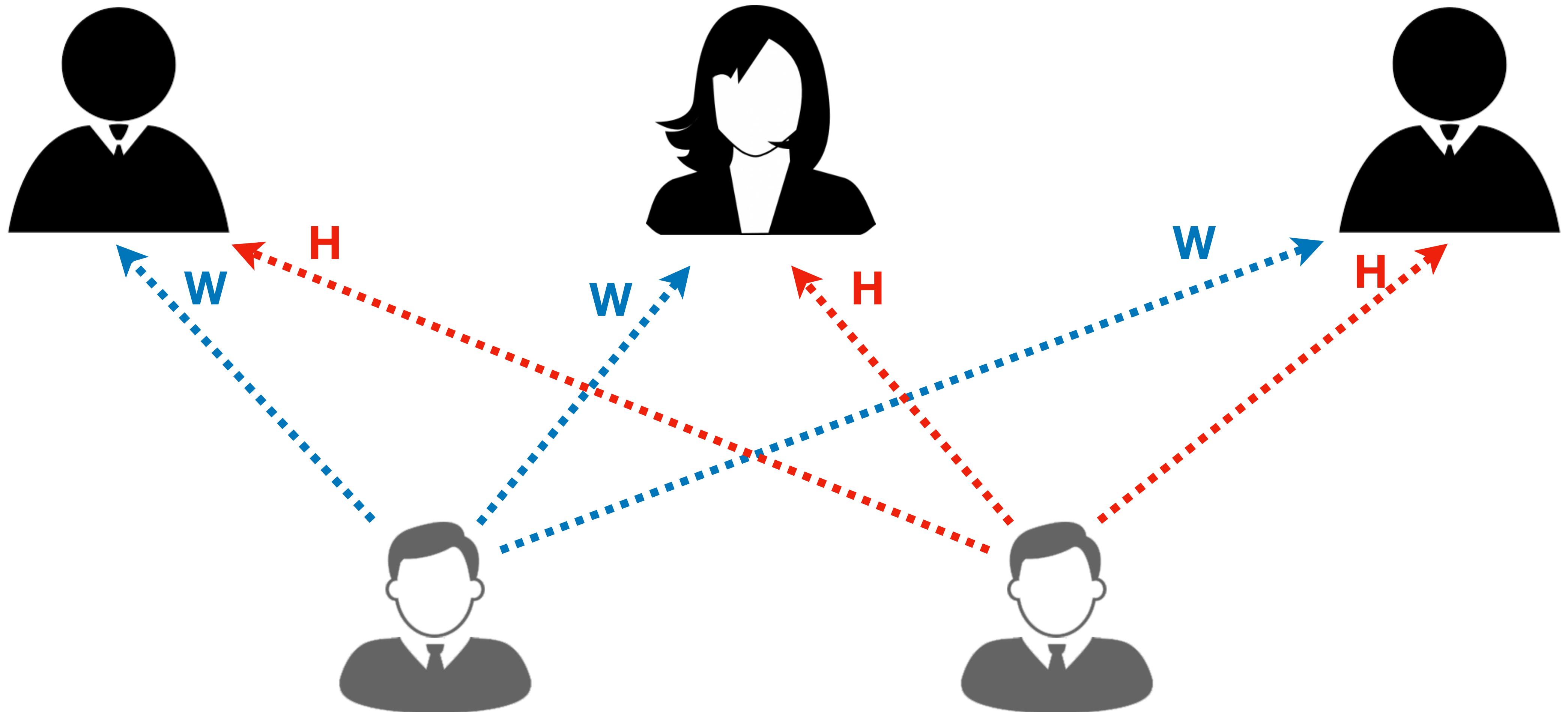
Multiple *acceptors* might disagree on the outcomes:
now they need to *reach a consensus* themselves.

Separation of Concerns

- **Proposers**: suggest a value (a restaurant to go);
- **Acceptors**: support some proposal;
- The proposer with a *majority of acceptors* supporting its proposal wins.

Others learn the outcome by querying all the acceptors.

Acceptors



Proposers

Acceptors



Proposers

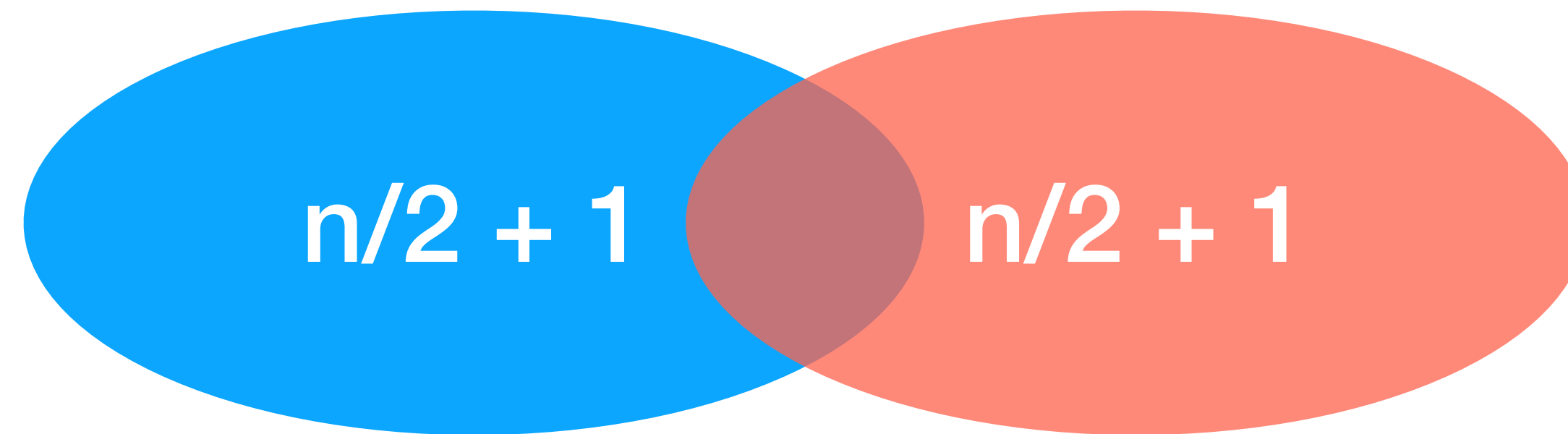
Key Idea 1

Rely on **majority quorums** for agreement to prevent the “split brain” problem.

- *Common meaning:* **Quorum** is the minimum number of members to conduct the business on behalf of the entire group they represent;
- *In computing:* quorum is a **necessary** number of processes to agree on the decision in the presence of potentially faulty ones.

Key Properties of Quorums

- *Property 1:* any two quorums must have non-empty intersection



- *Property 2:* no need for the *global* agreement: can tolerate some faults

$$n = 3$$



Quorum of $n/2 + 1$ acceptors

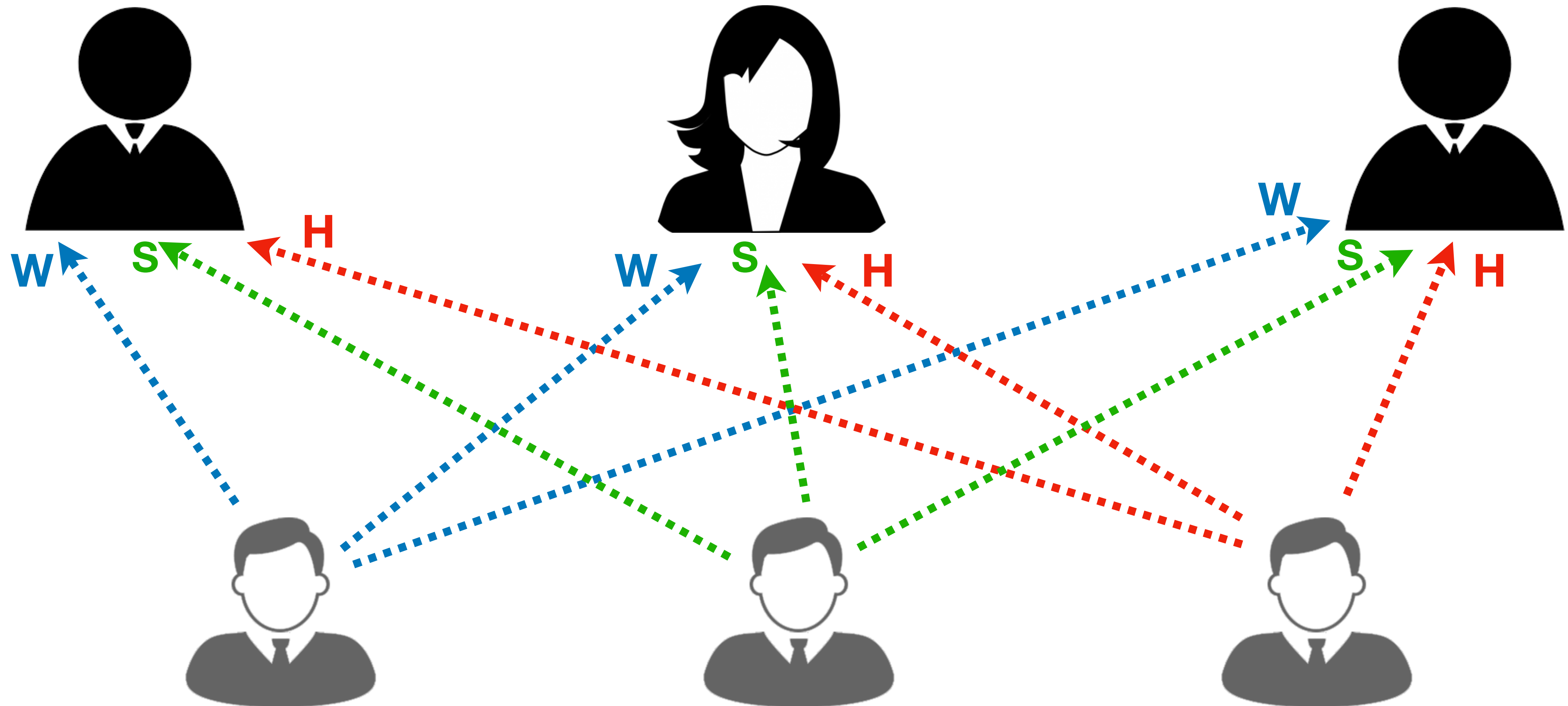


Problem

A quorum is difficult to obtain in a *single* interaction.

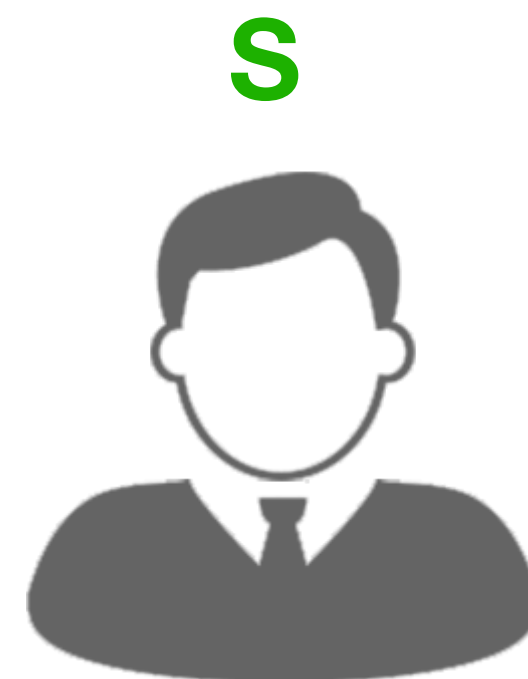
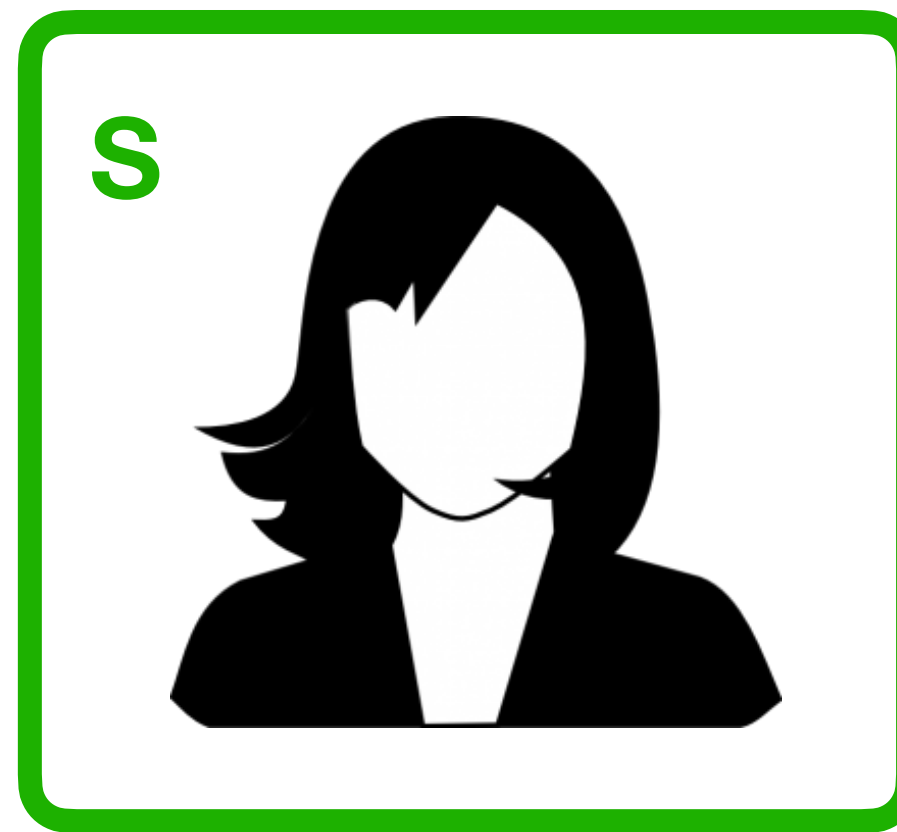
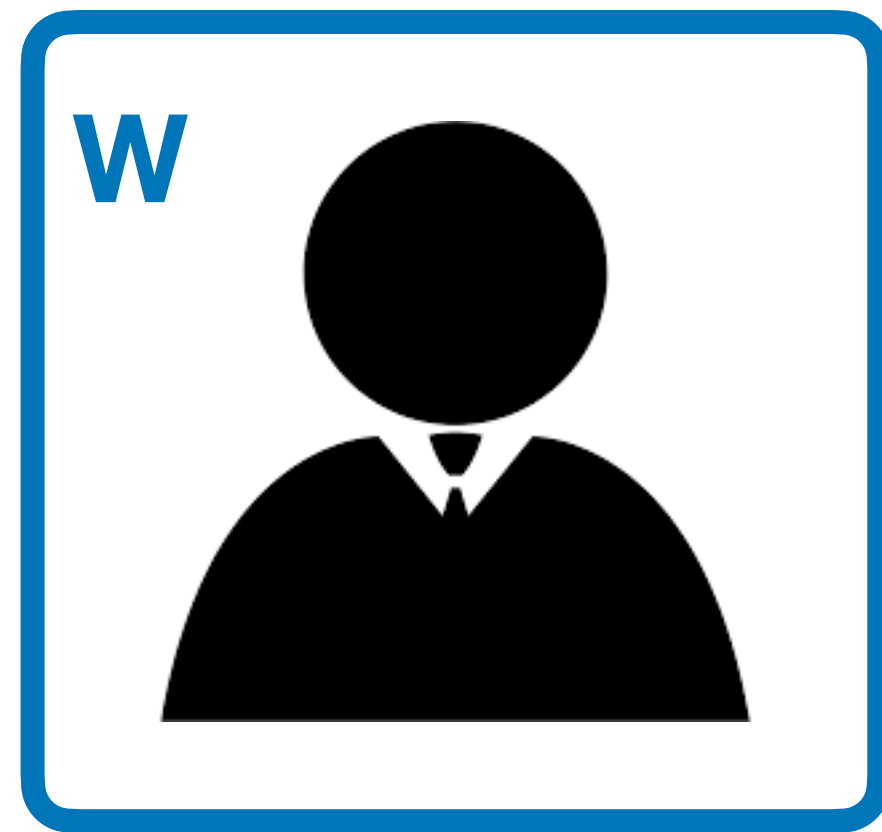
As the result, such a system will often *get stuck*.

Acceptors



Proposers

Acceptors



Proposers

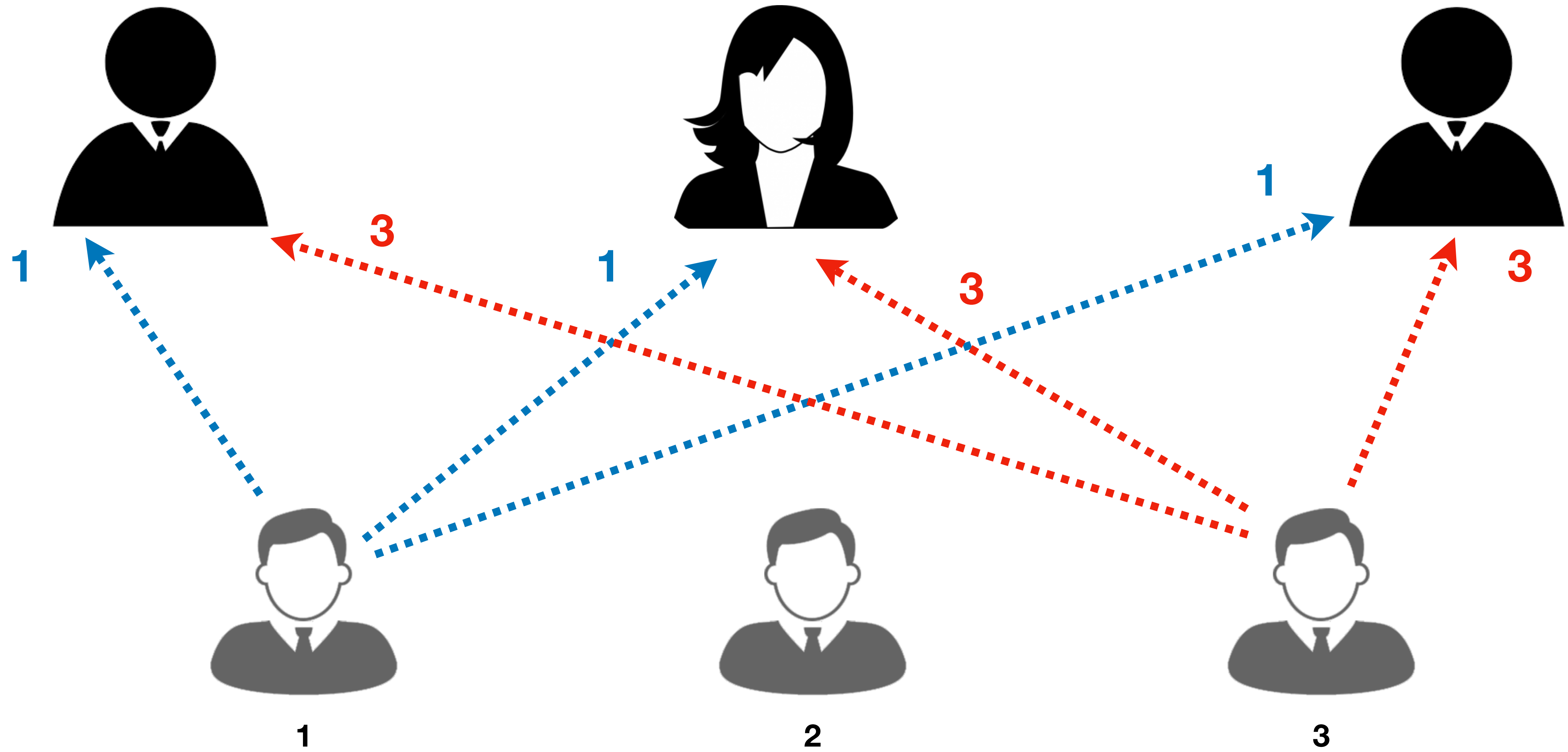
Key Ideas 2 and 3

- Proceed in rounds:
 - A proposer first “**secures**” itself a quorum, willing to support its proposal (i.e., becomes a “**leader**”);
 - *Only if a quorum is secured*, it goes on to “**propose**” a value.
- Introduce fixed globally known **priorities** between proposers to “break ties” when securing quorums.
- Acceptors only “choose to support” proposers with *higher* priorities than they have already seen.

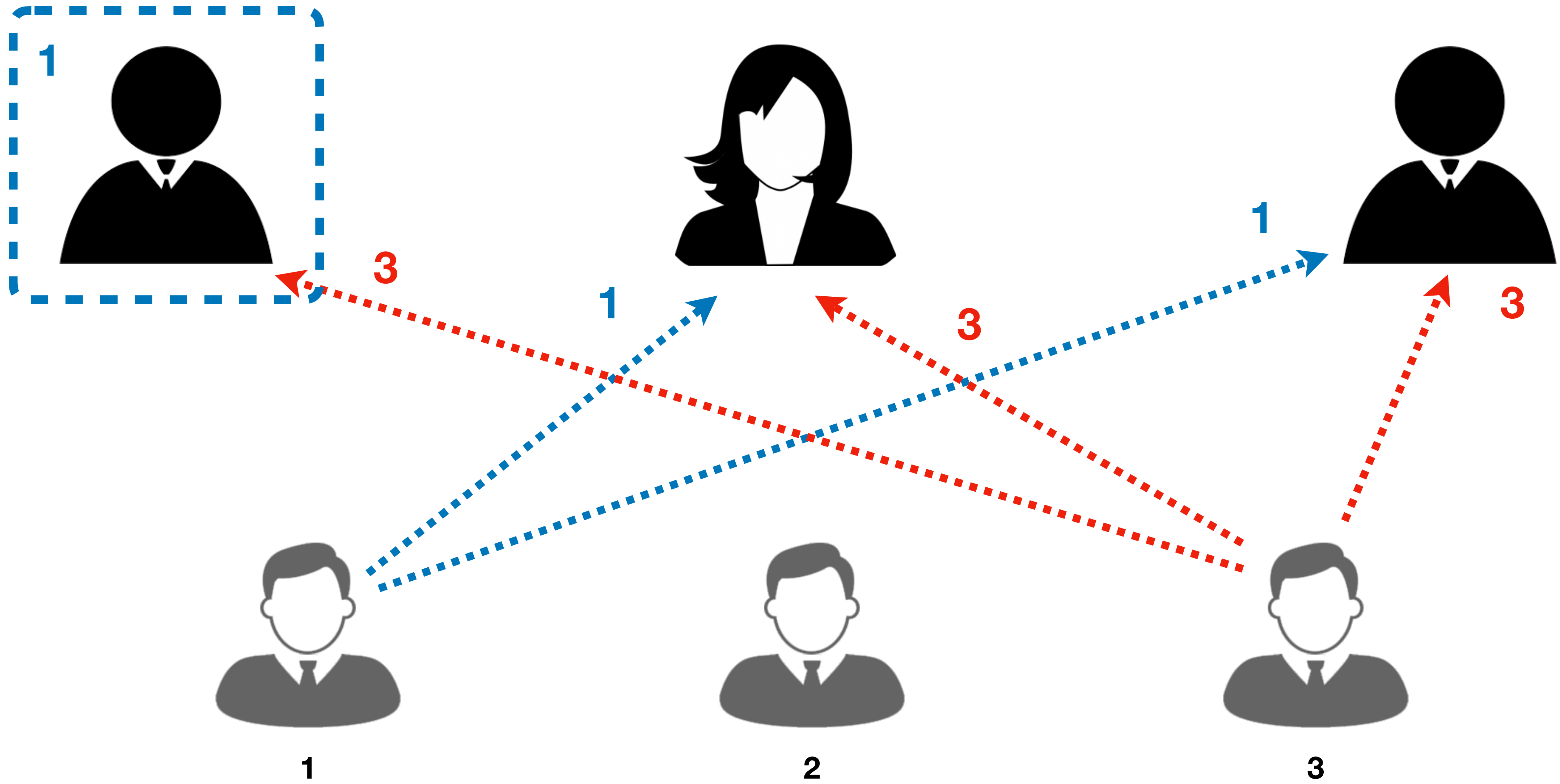
Some Terminology

- Rounds — **Phases**
 - Phase 1 — “prepare”, **securing** quorums to propose
 - Phase 2 — “accept”, **sending** values to accept
- Fixed priorities — **Ballots**

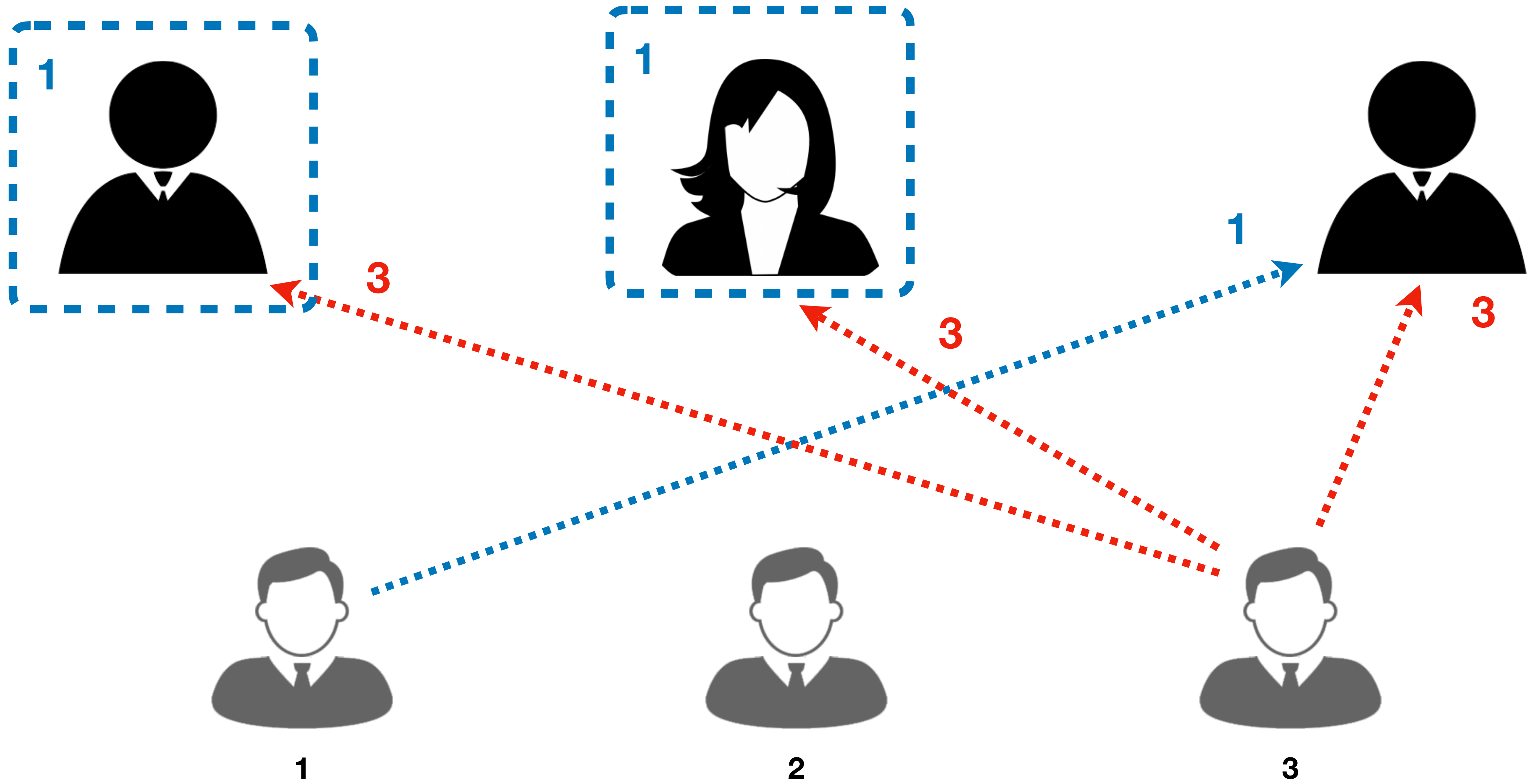
Phase 1



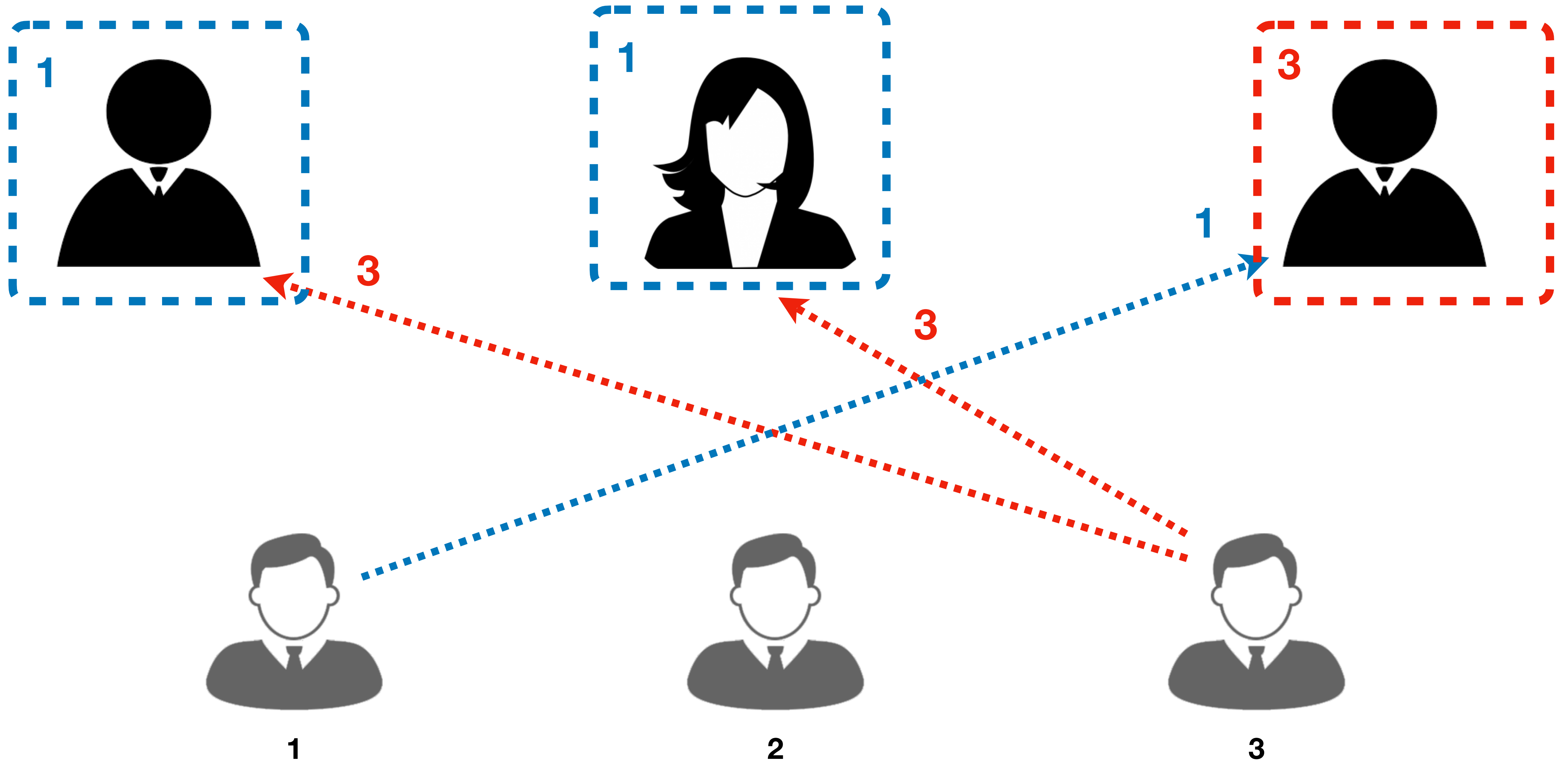
Phase 1



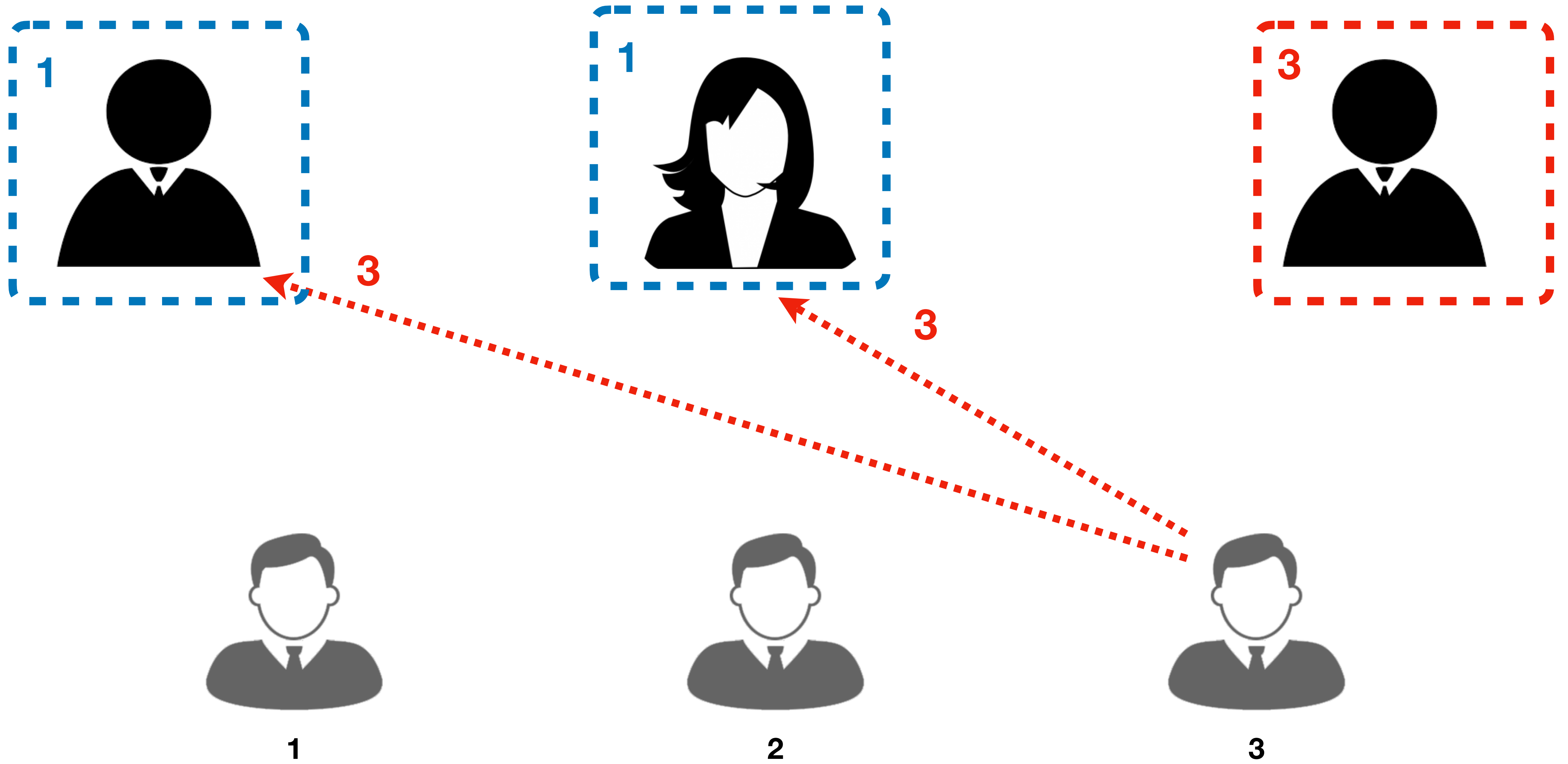
Phase 1



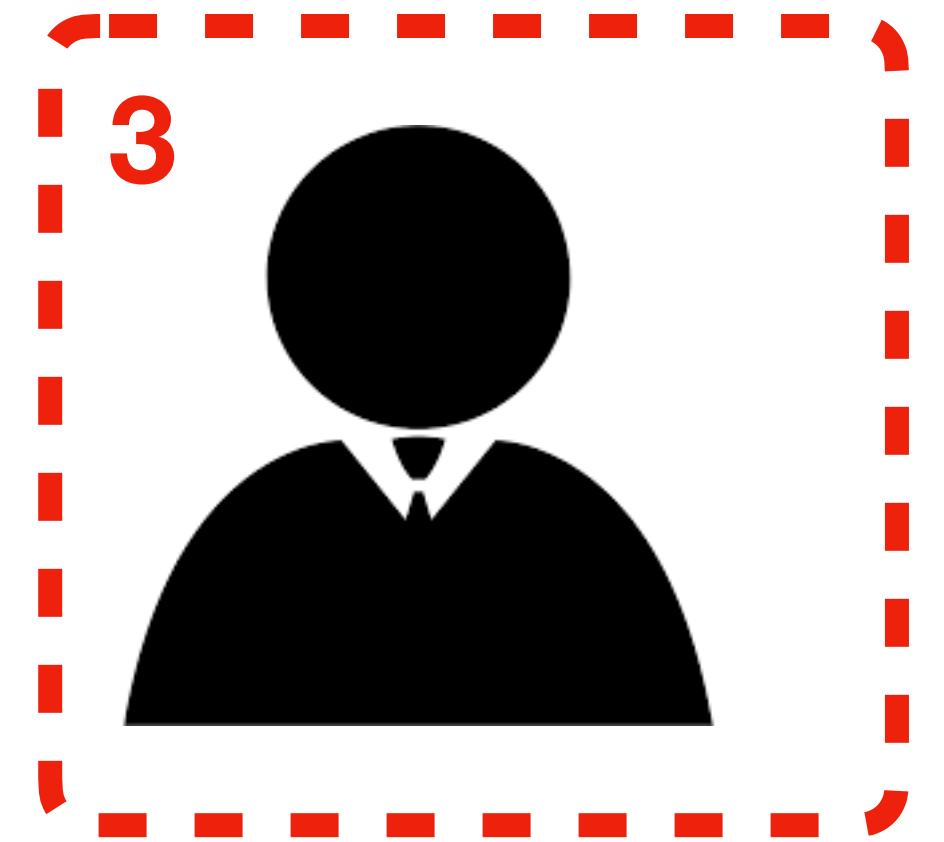
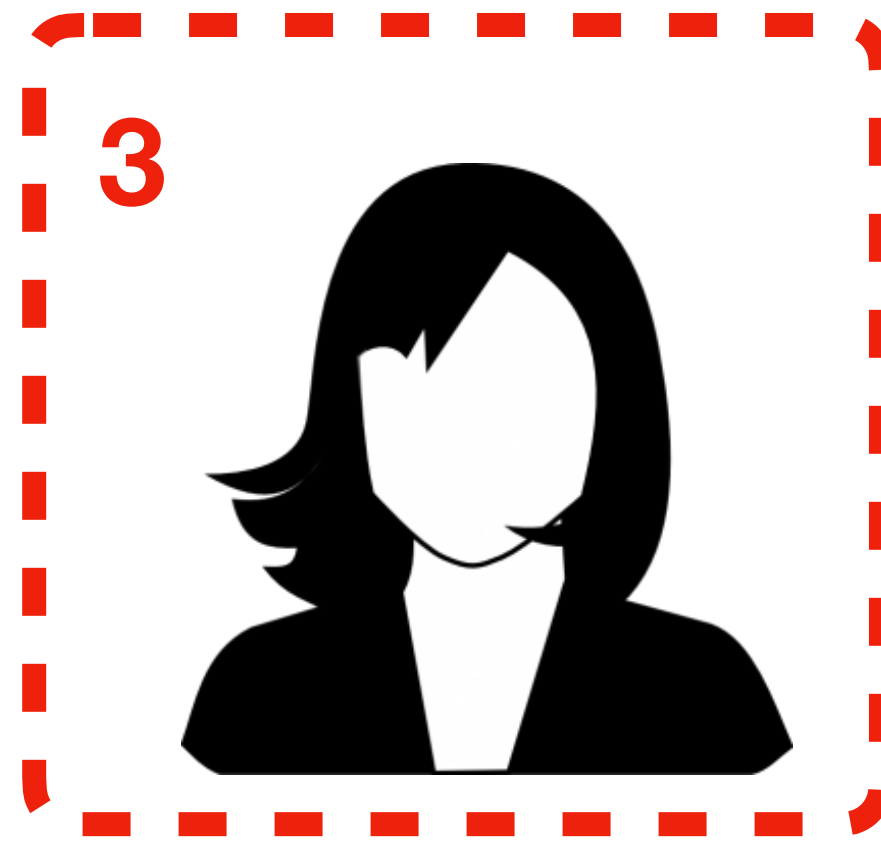
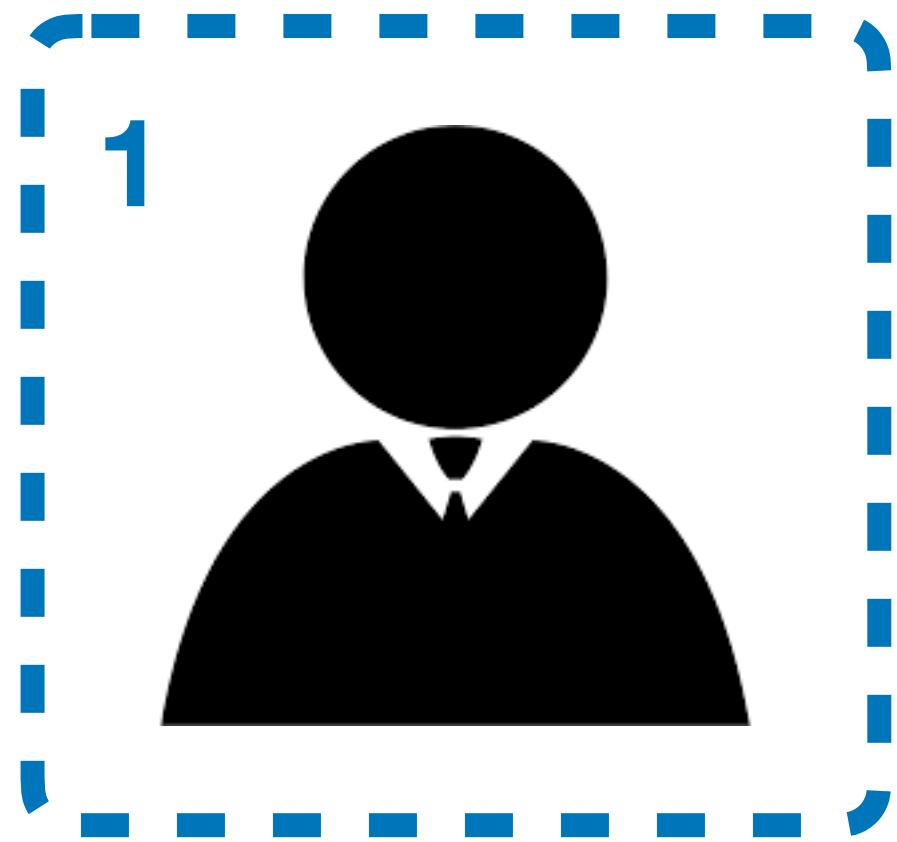
Phase 1



Phase 1



Phase 1



1

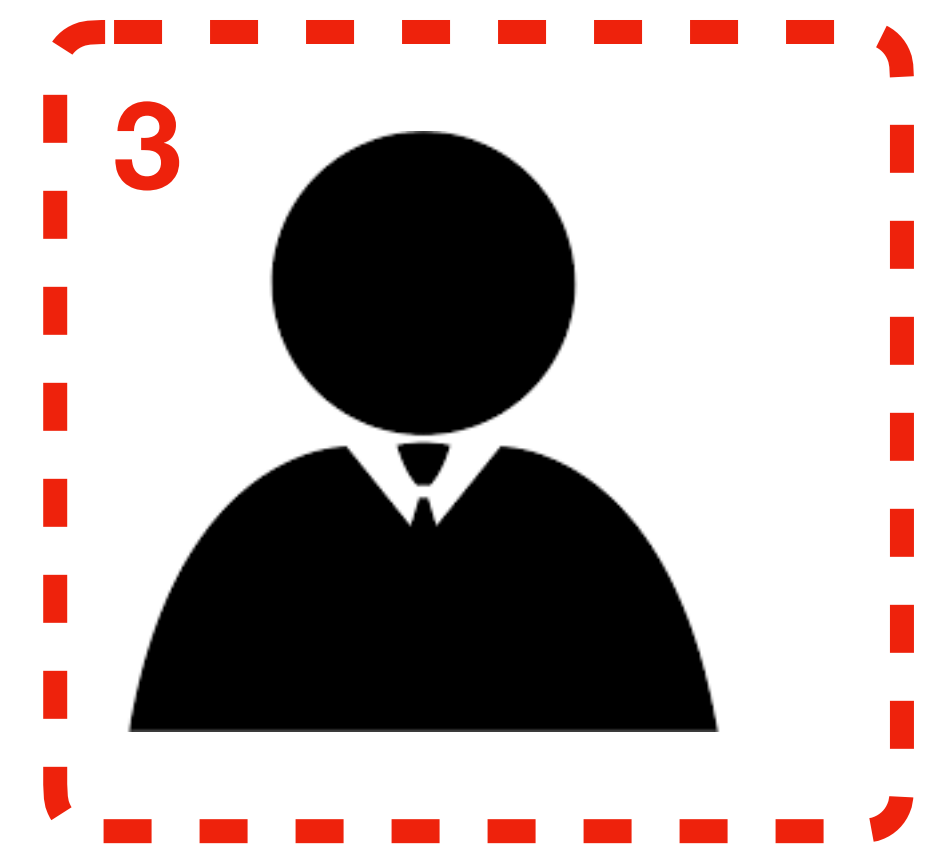
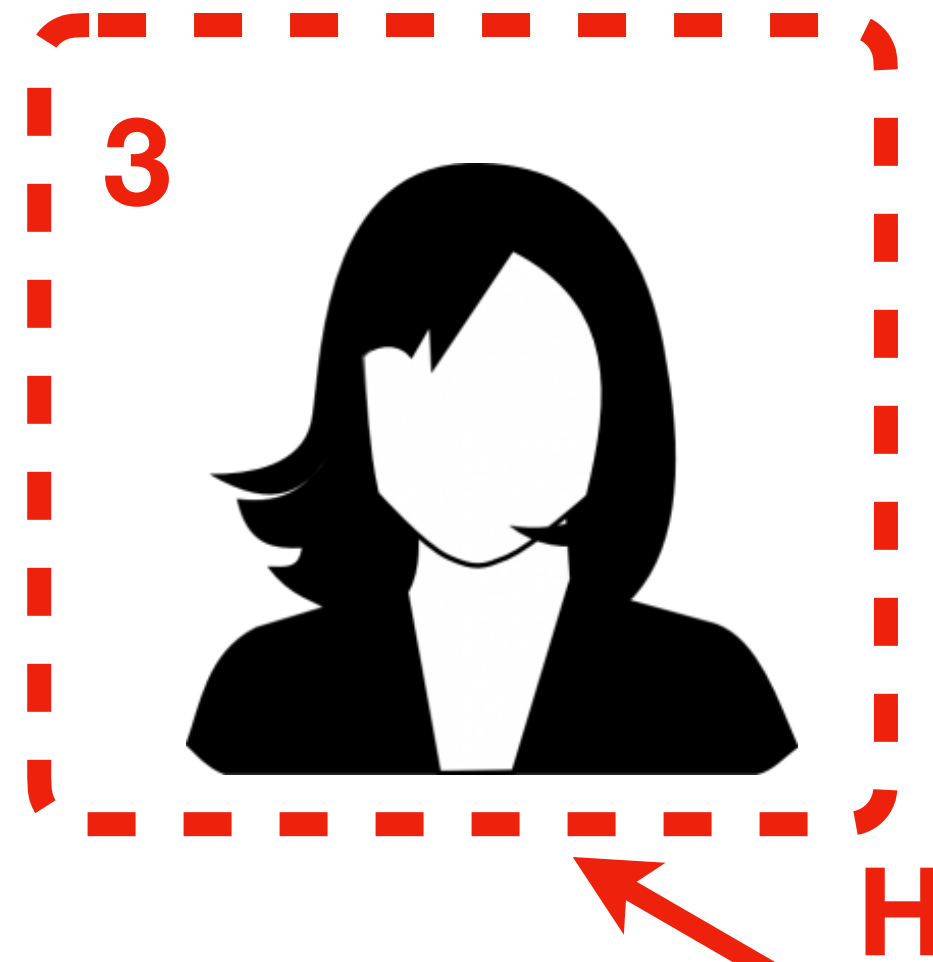
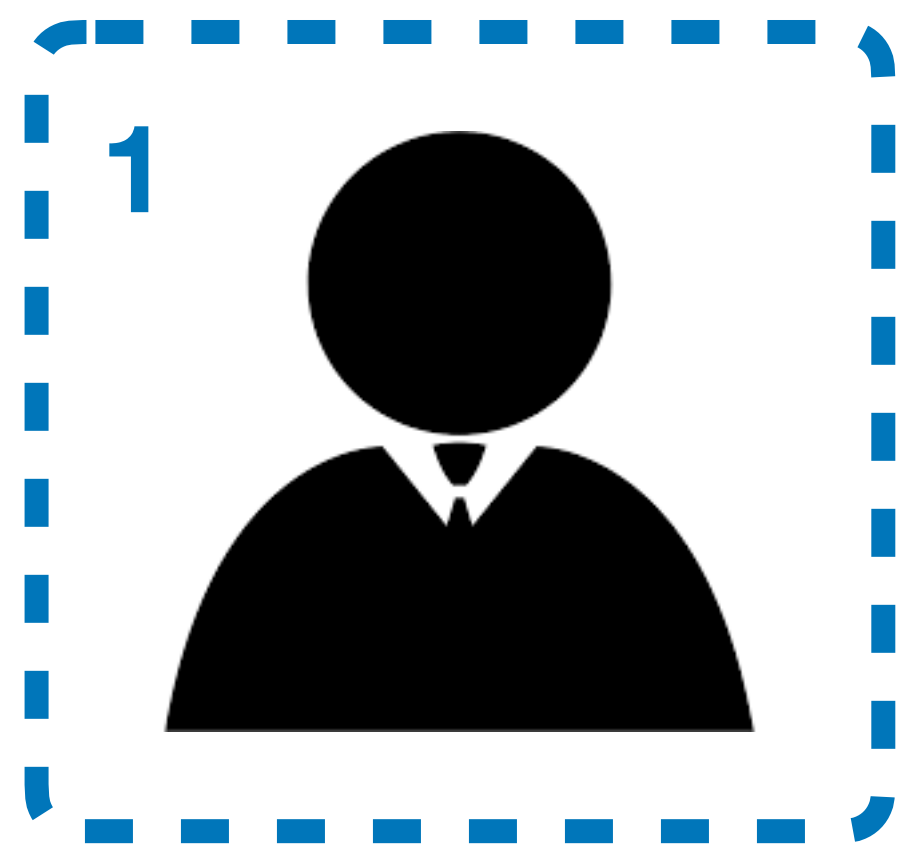


2



3

Phase 2



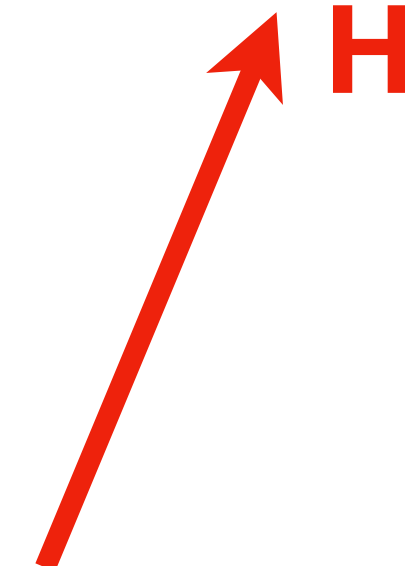
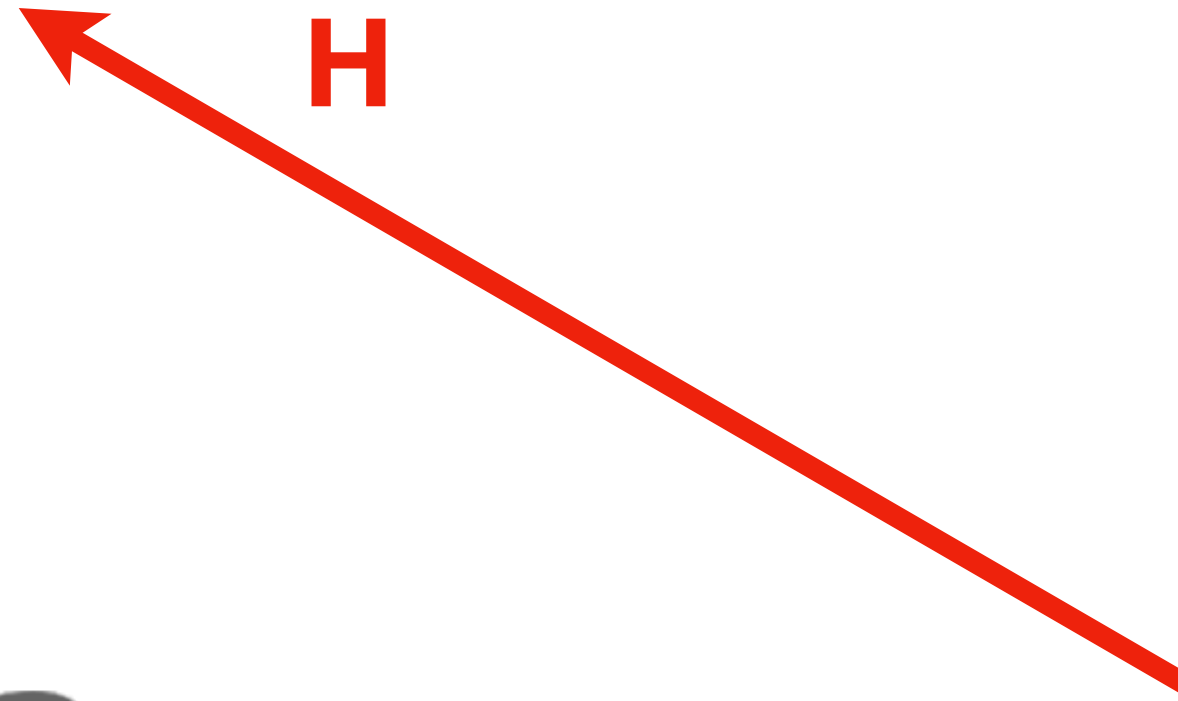
1



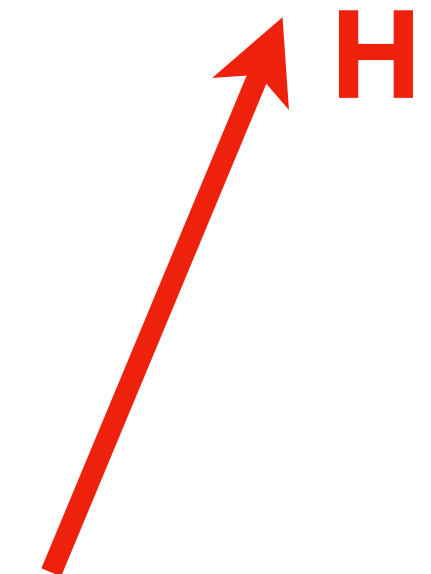
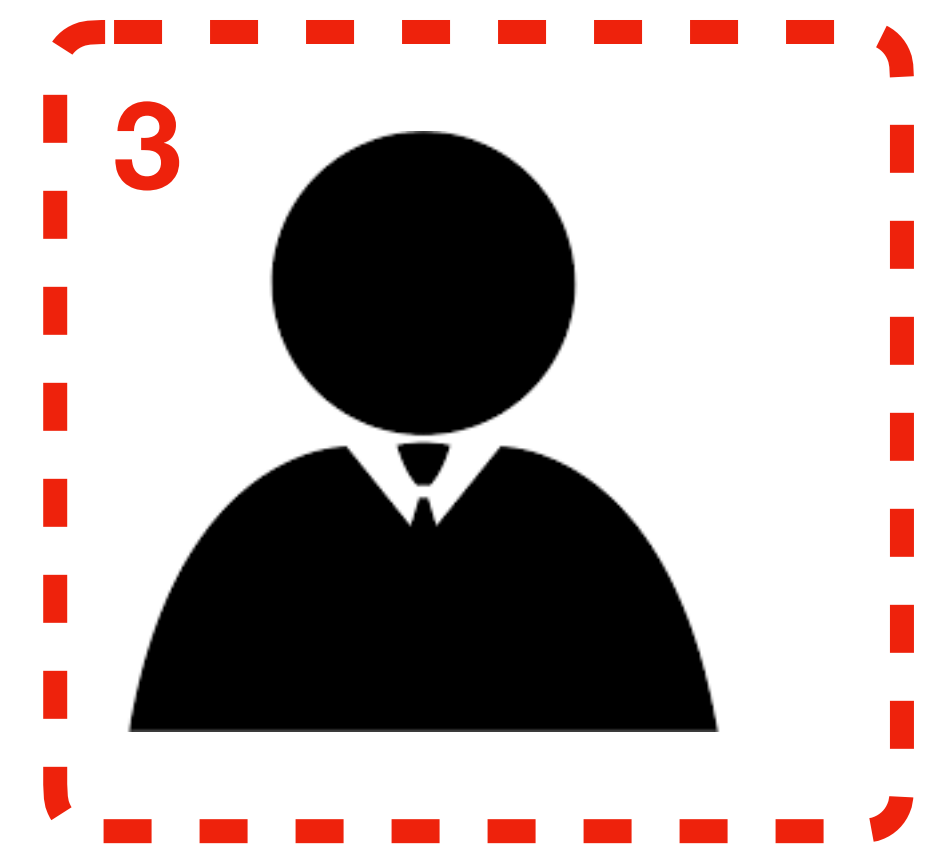
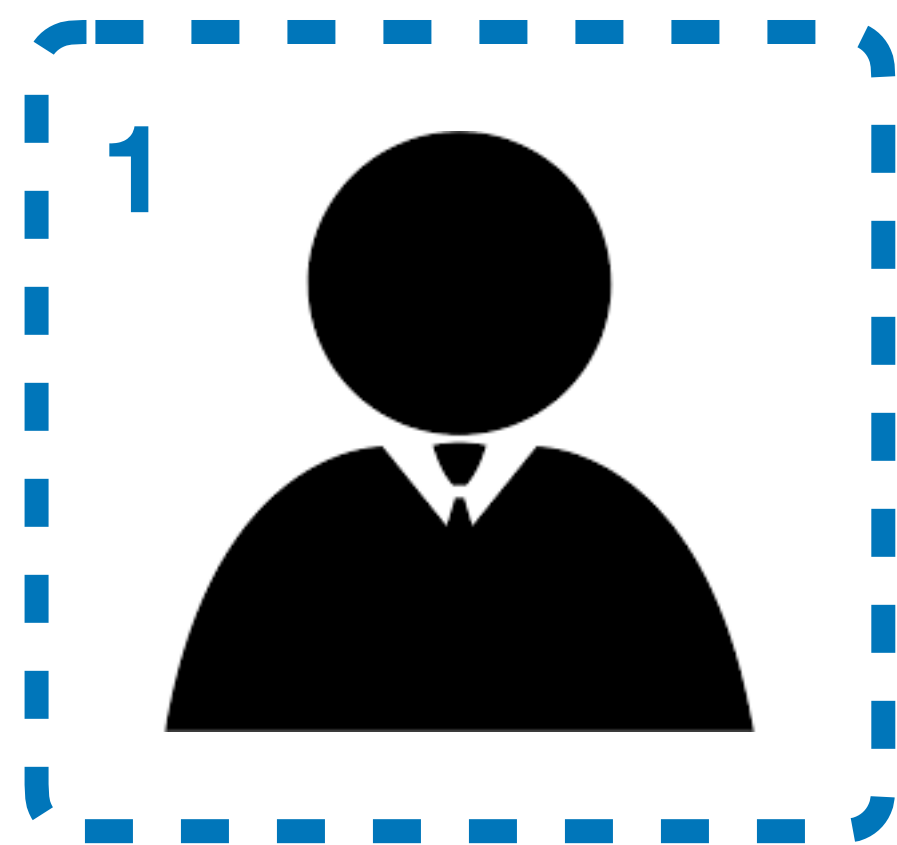
2



3



Phase 2



1

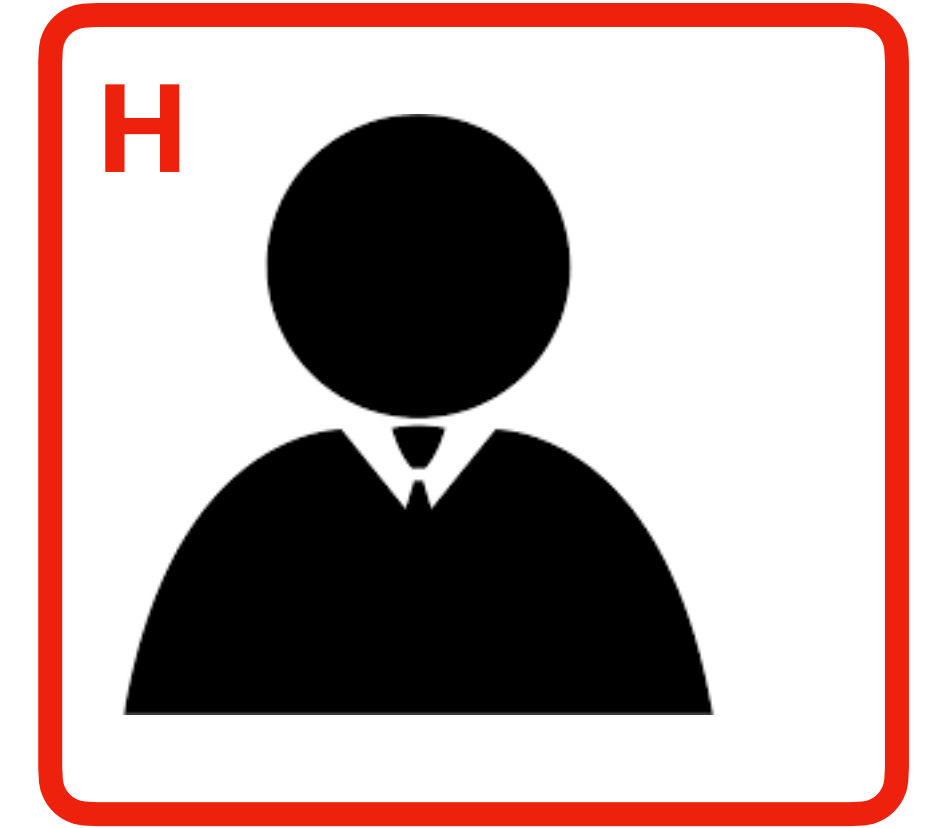
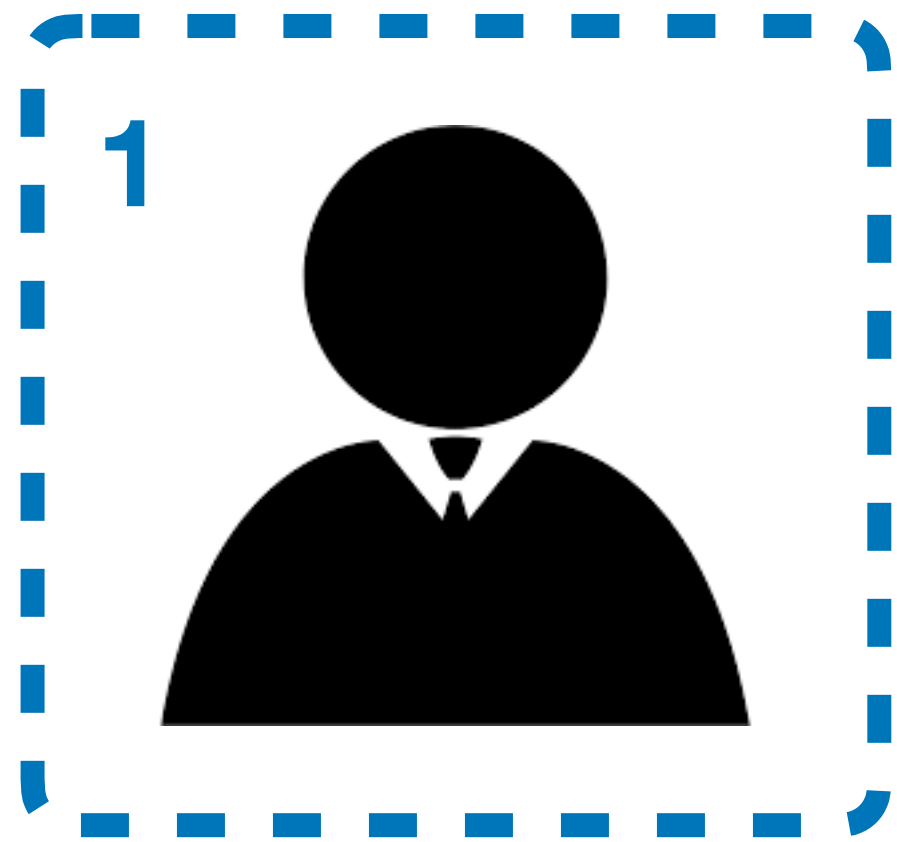


2



3

Phase 2



1



2



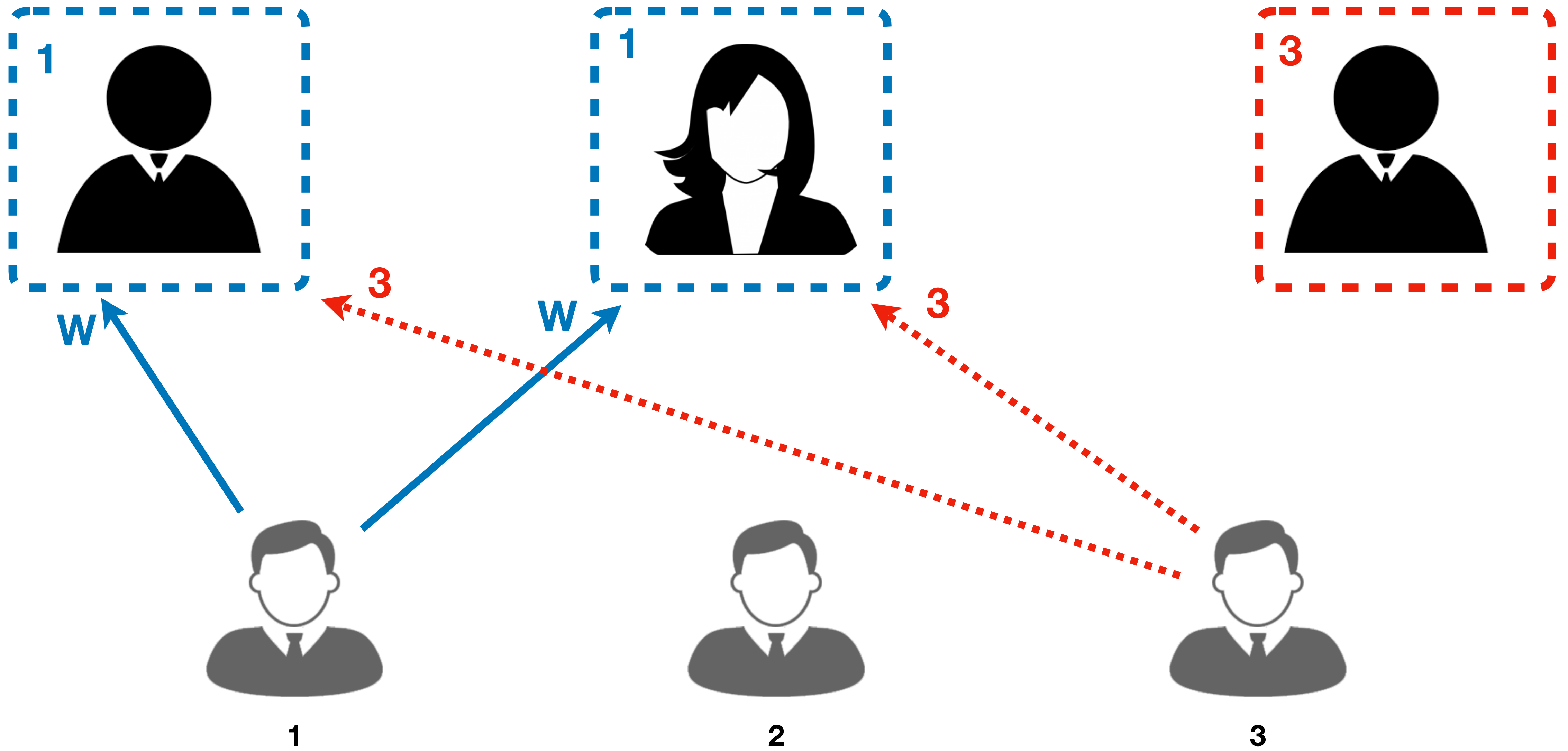
3

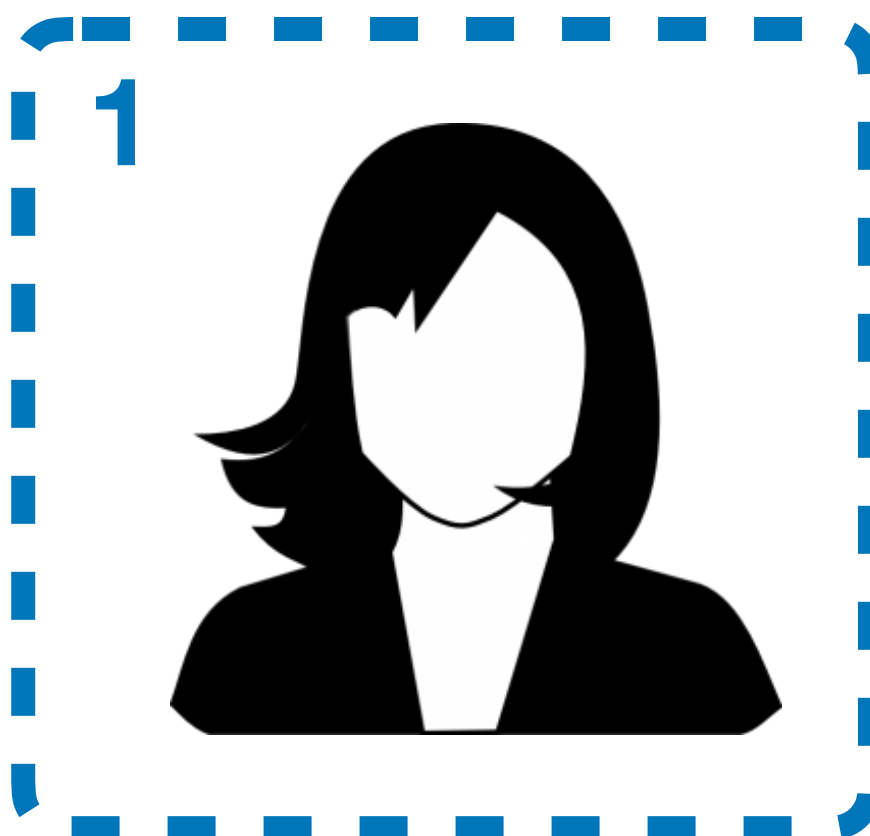
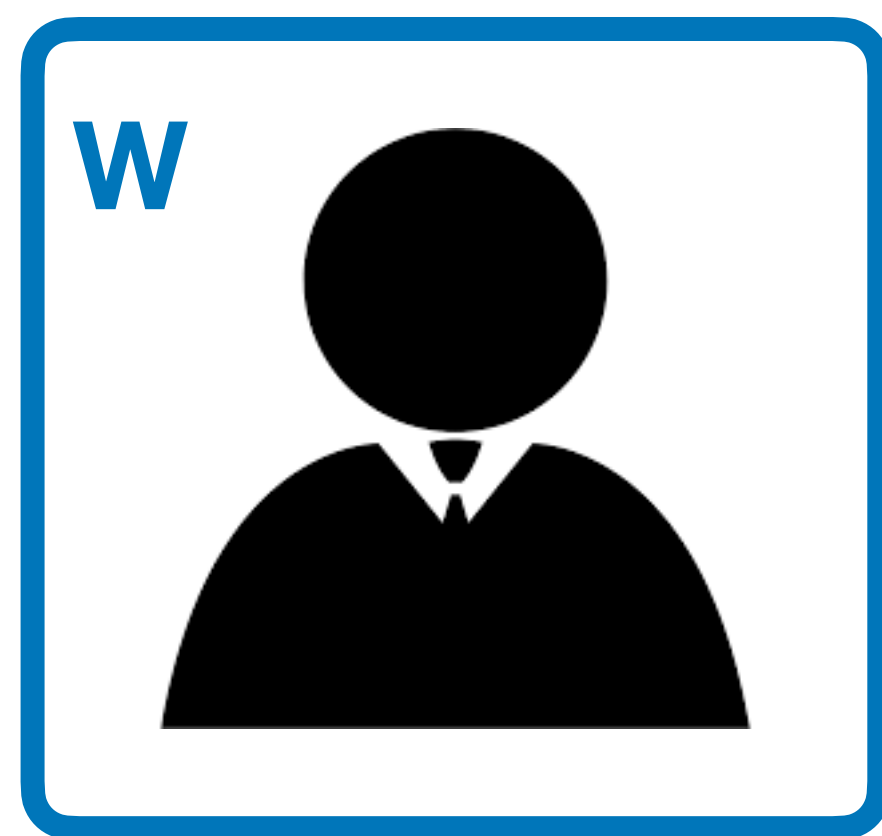
Problem 3

Because of asynchrony, **low-priority** Phase 2 can be *interrupted* by a **high-priority** Phase 1

Phase 2

Phase 1





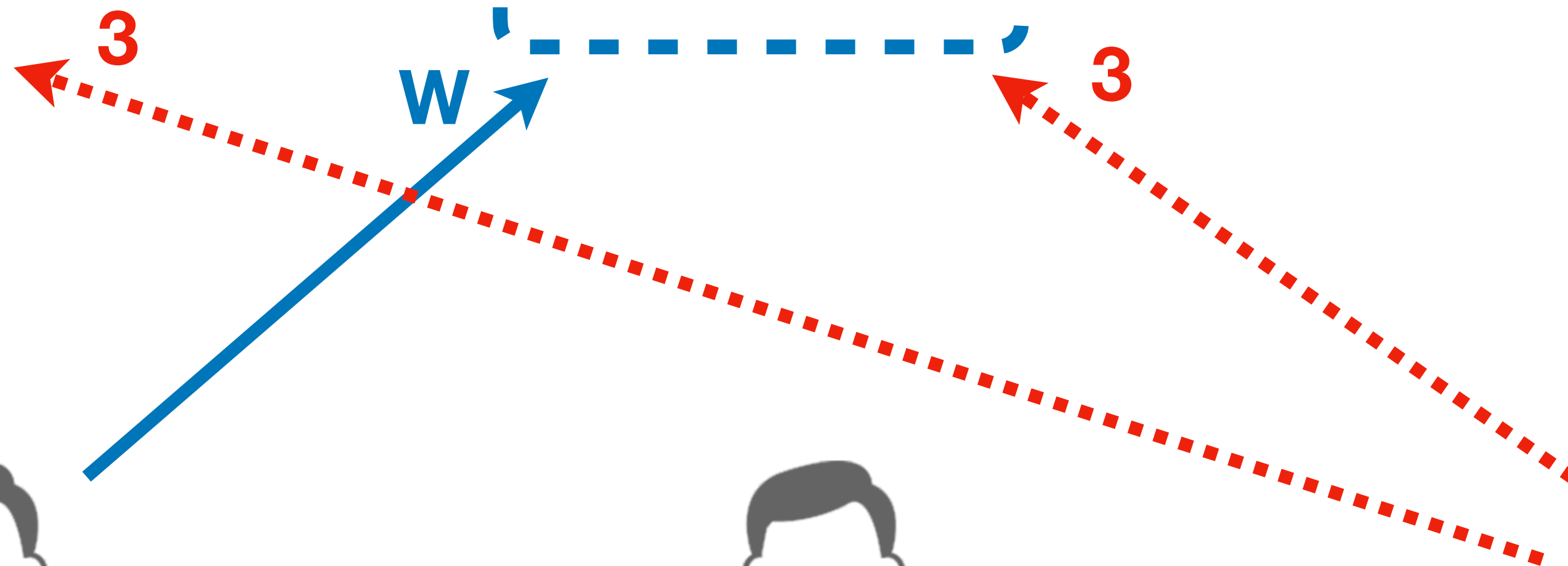
1

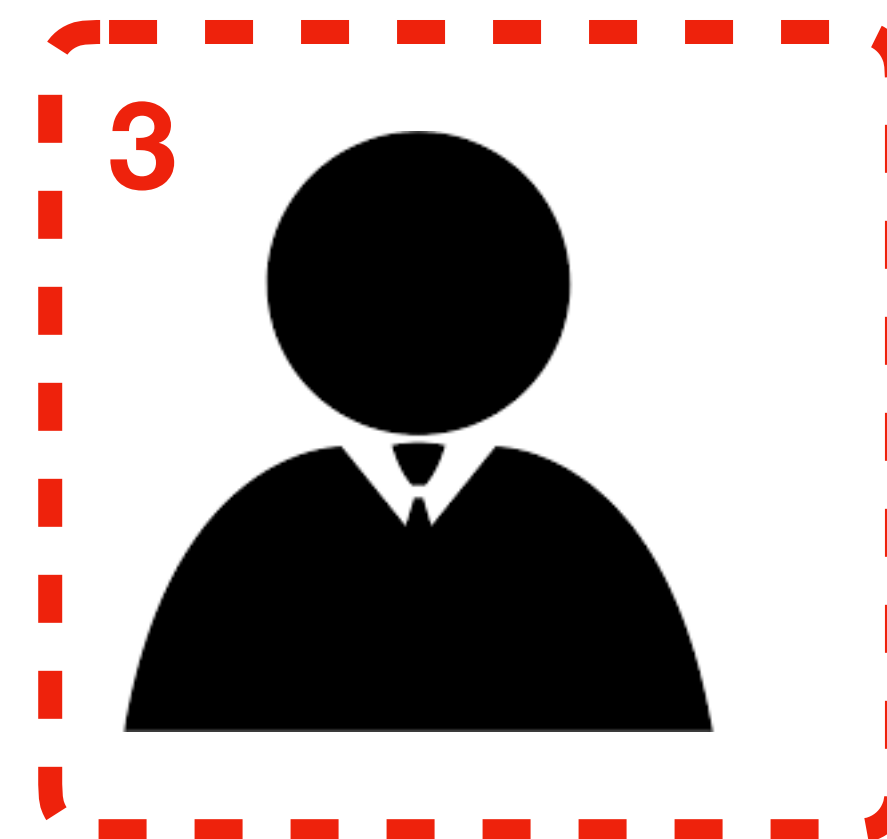


2



3





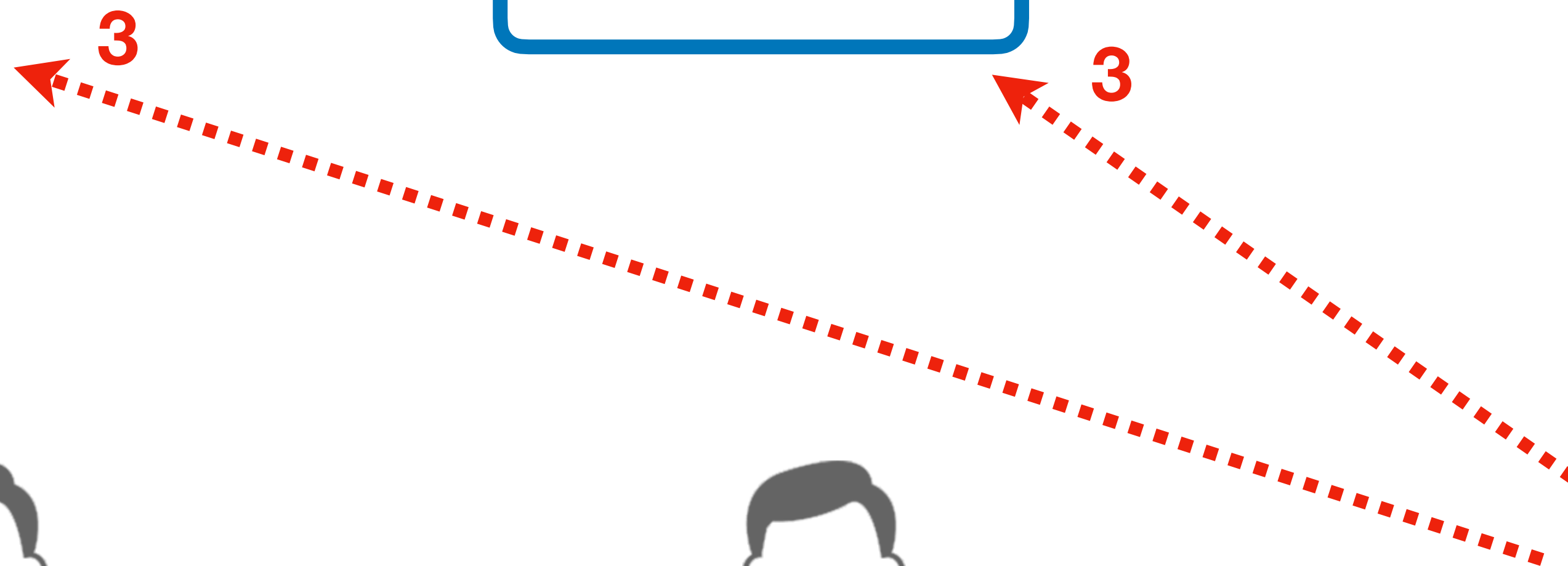
1

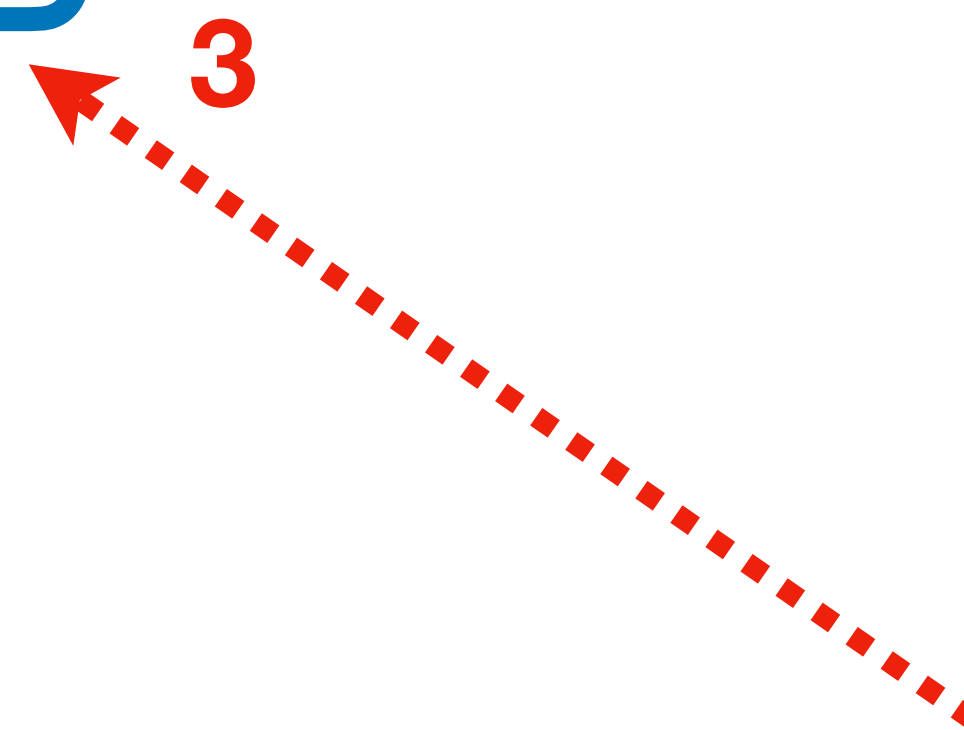
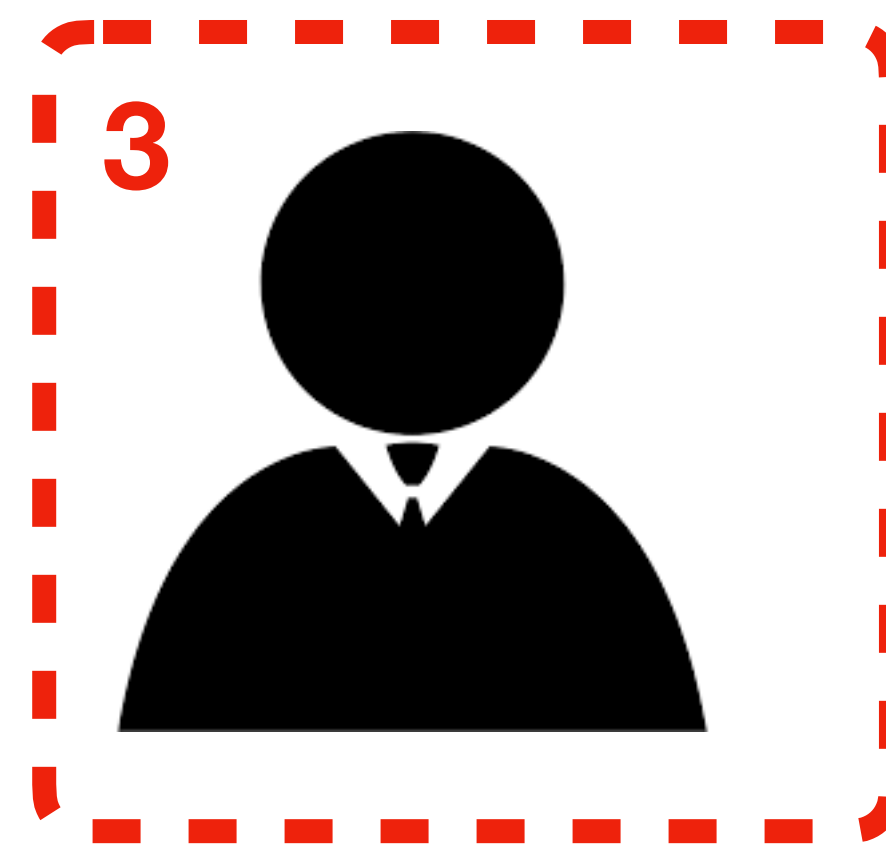
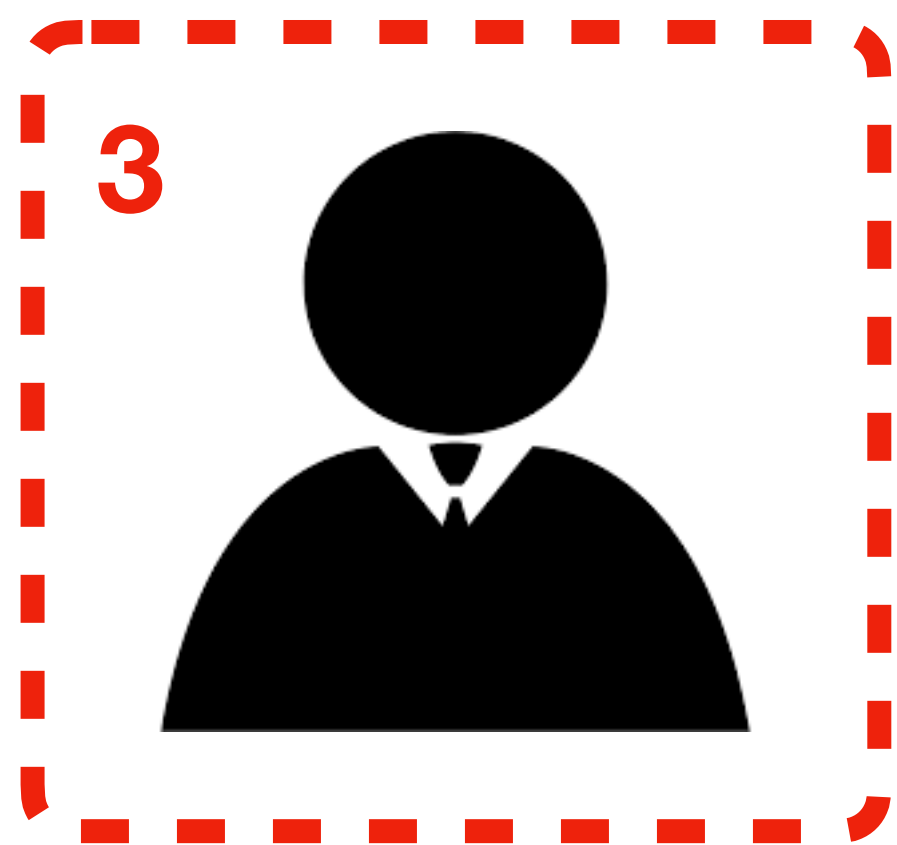


2



3





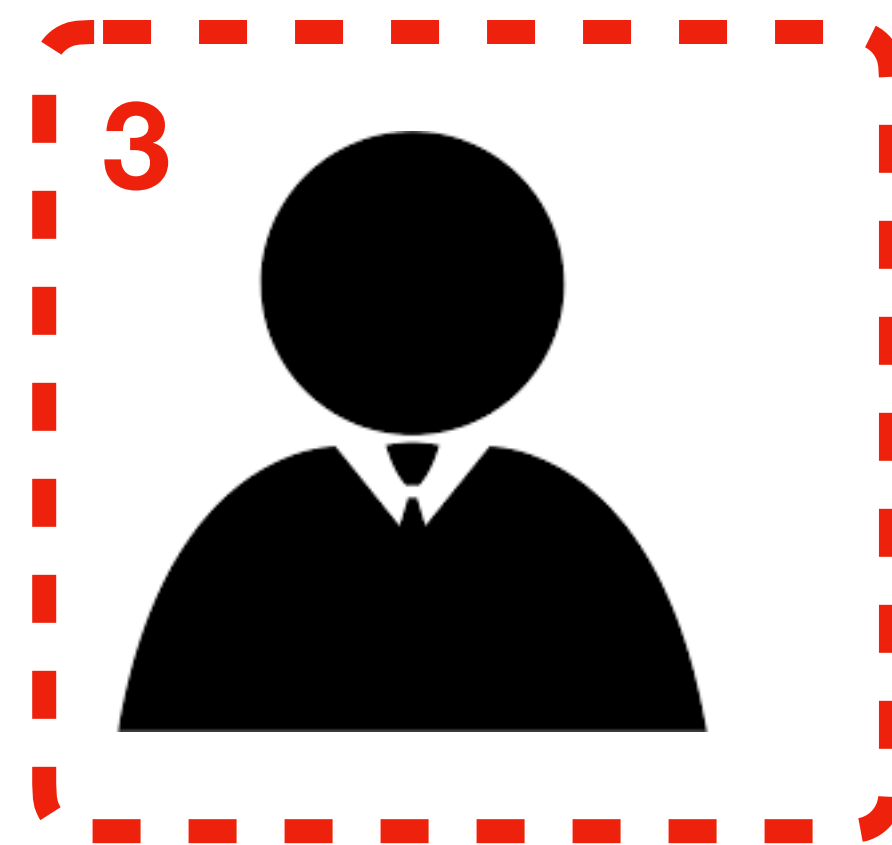
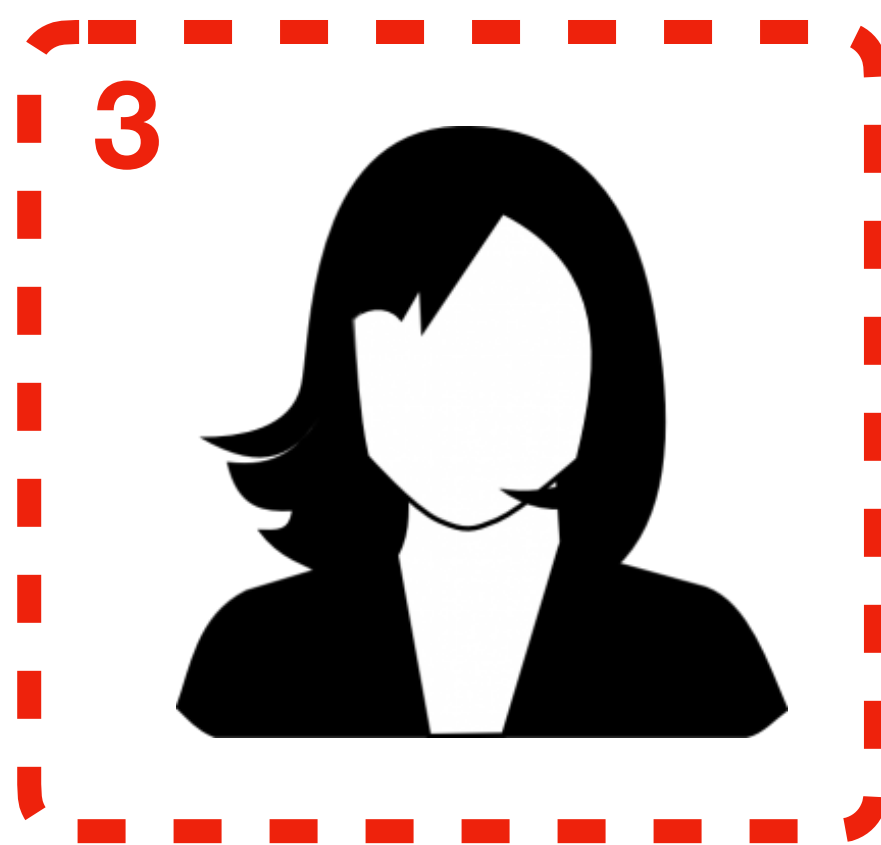
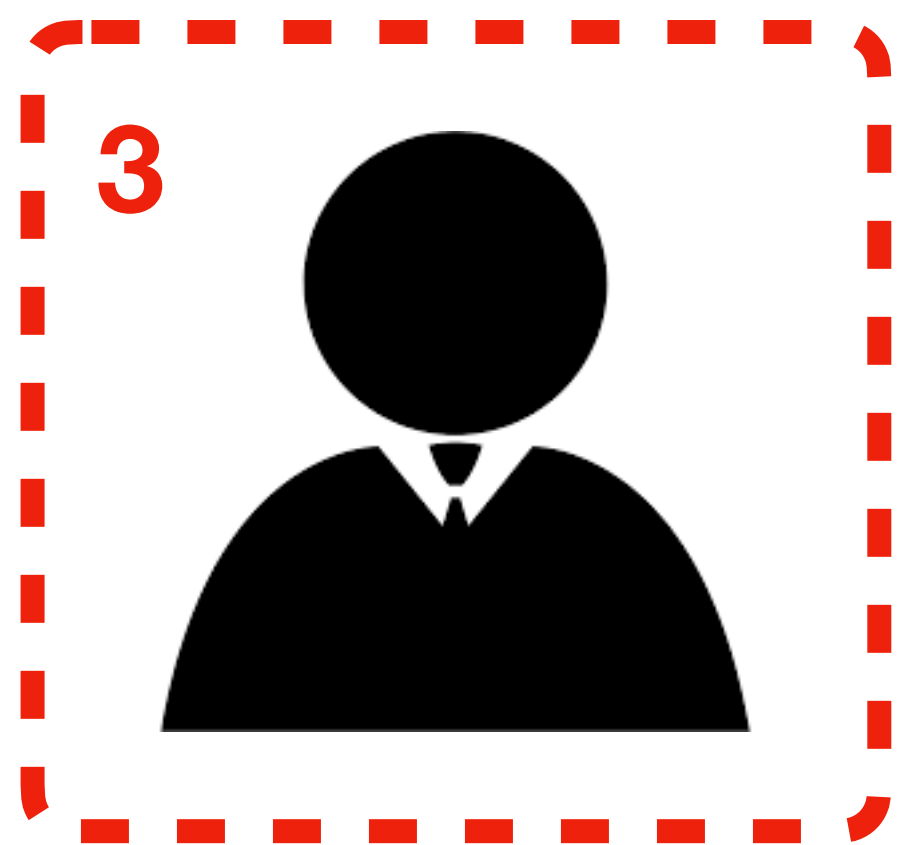
1



2



3



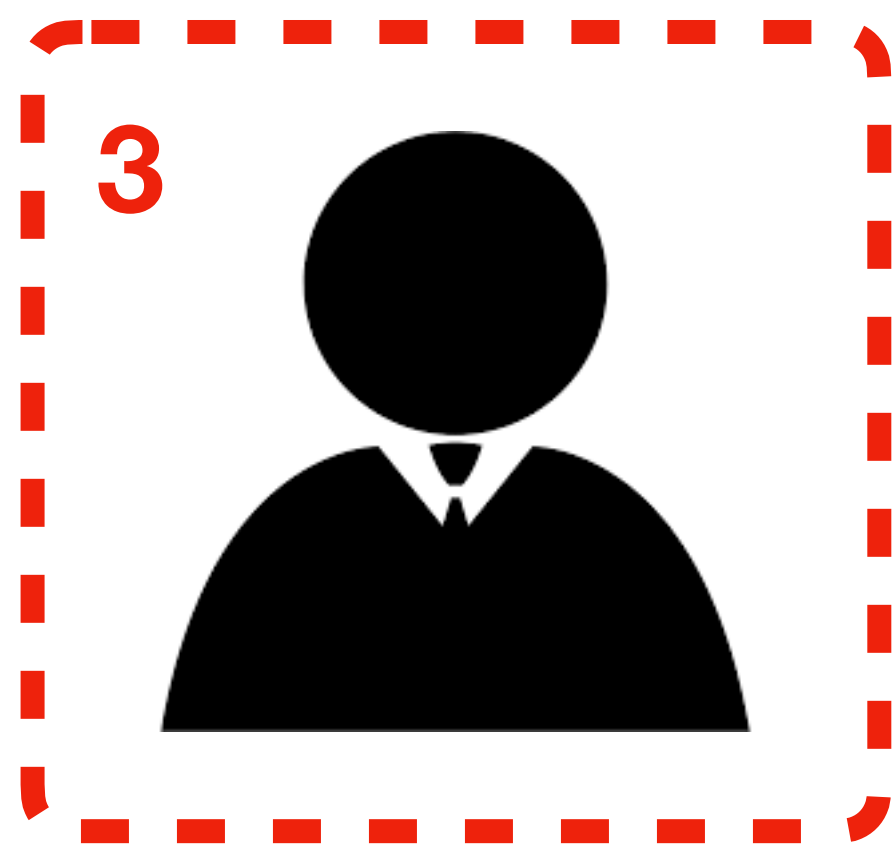
1



2



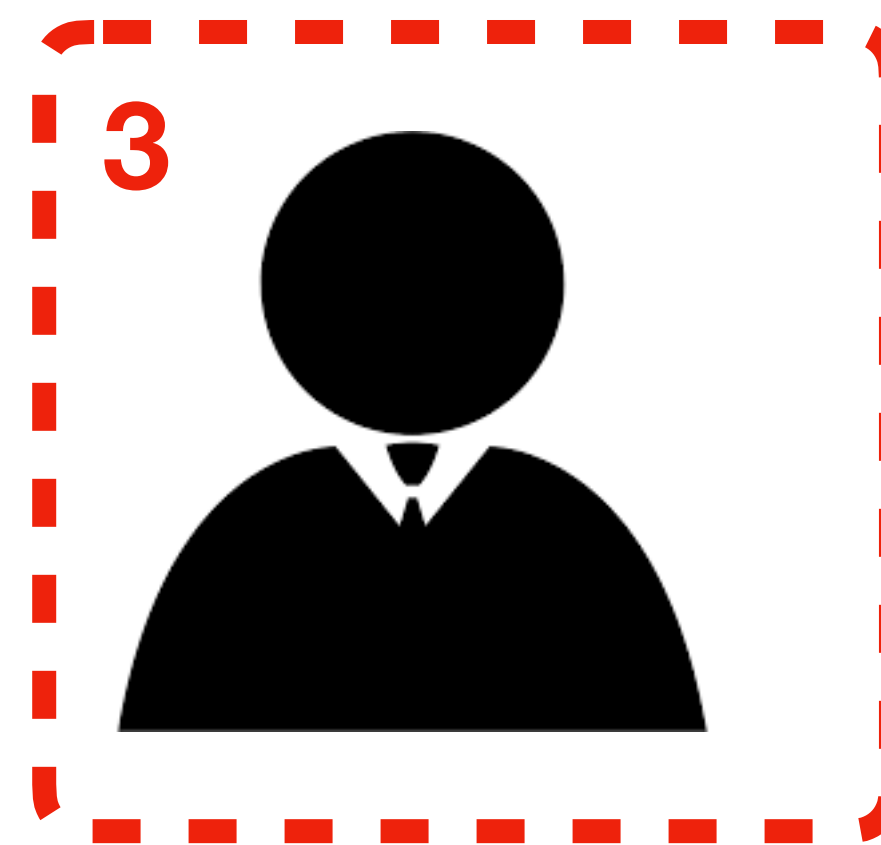
3



H



H



H



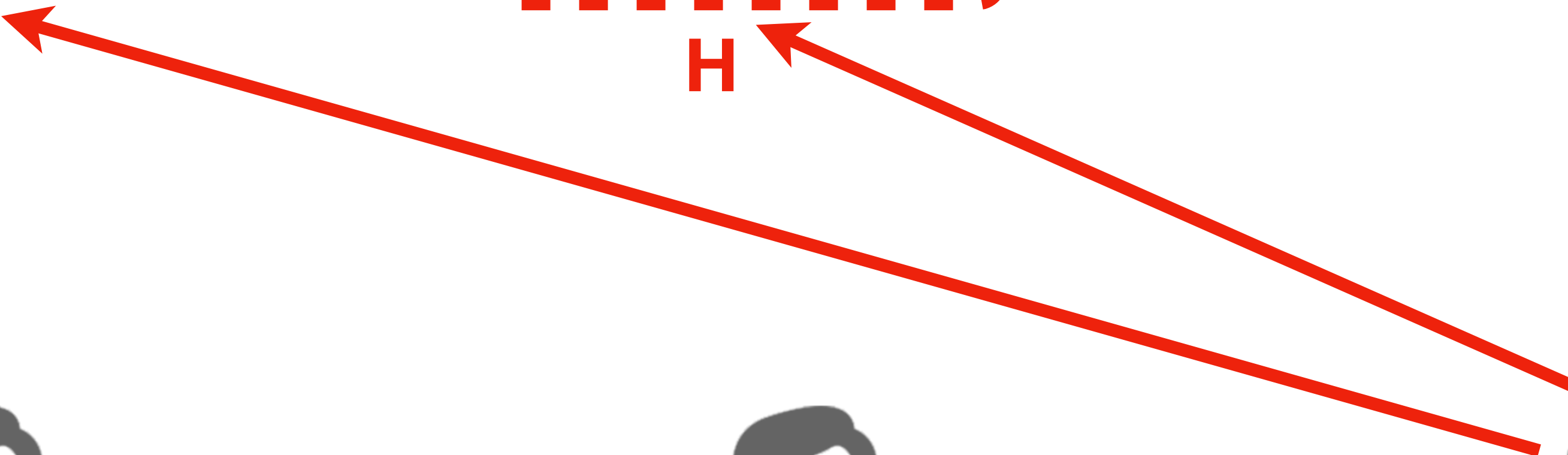
1

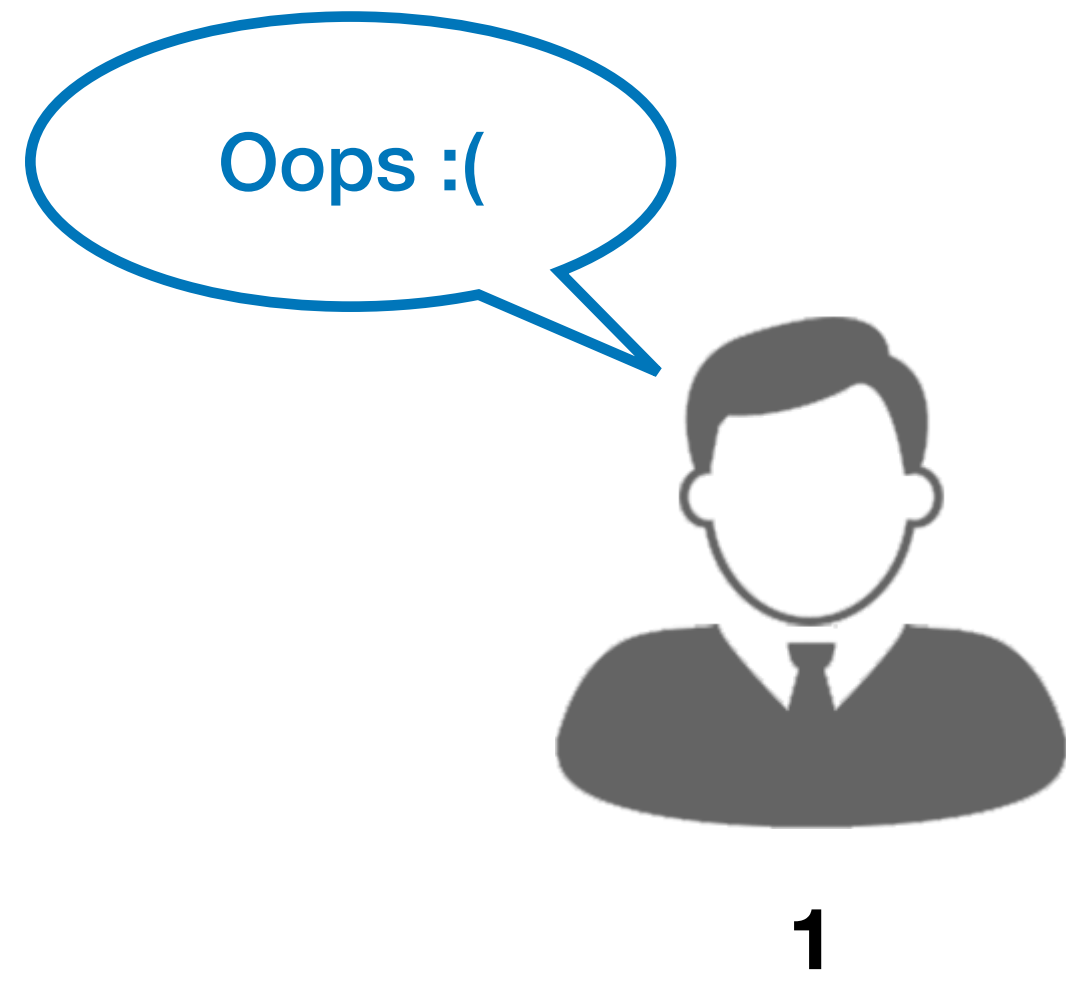
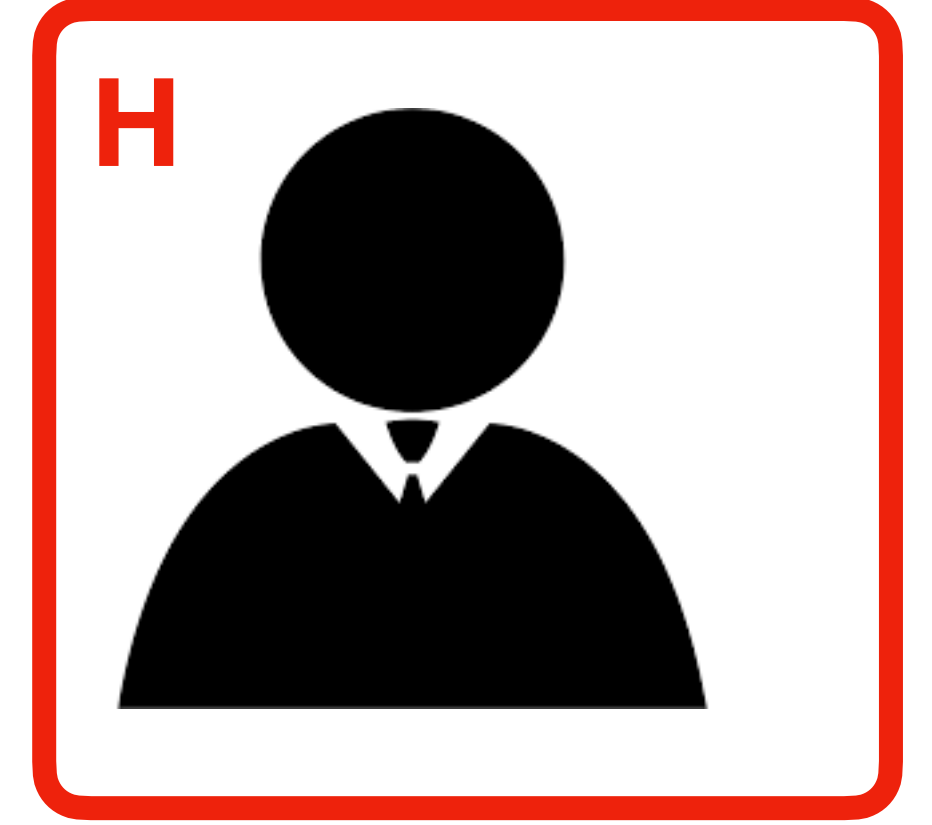
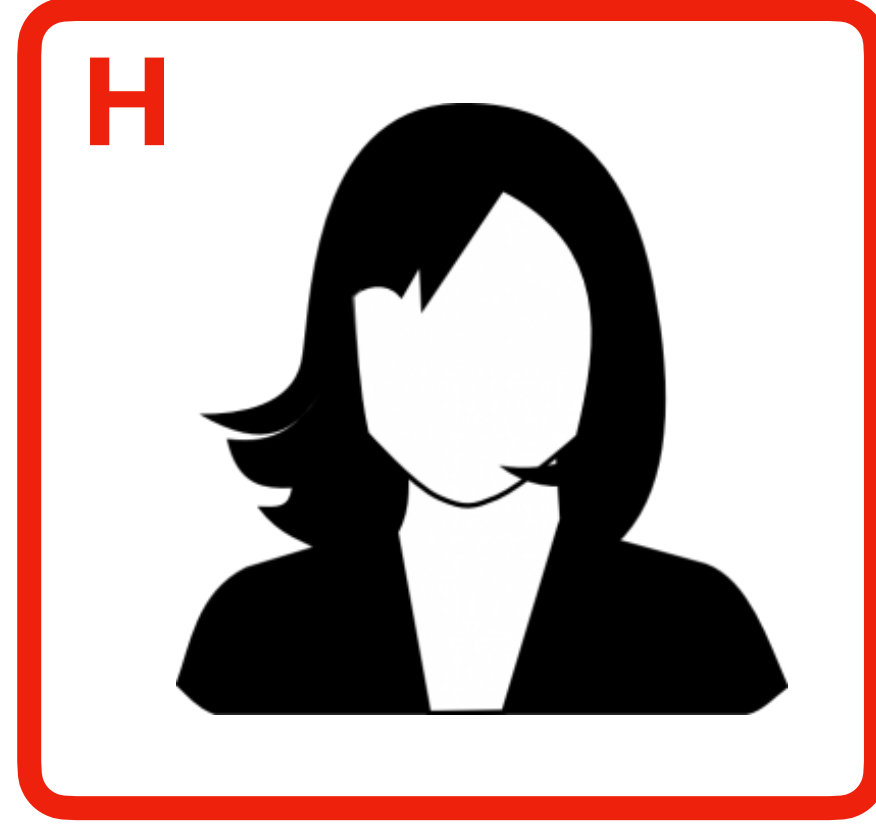
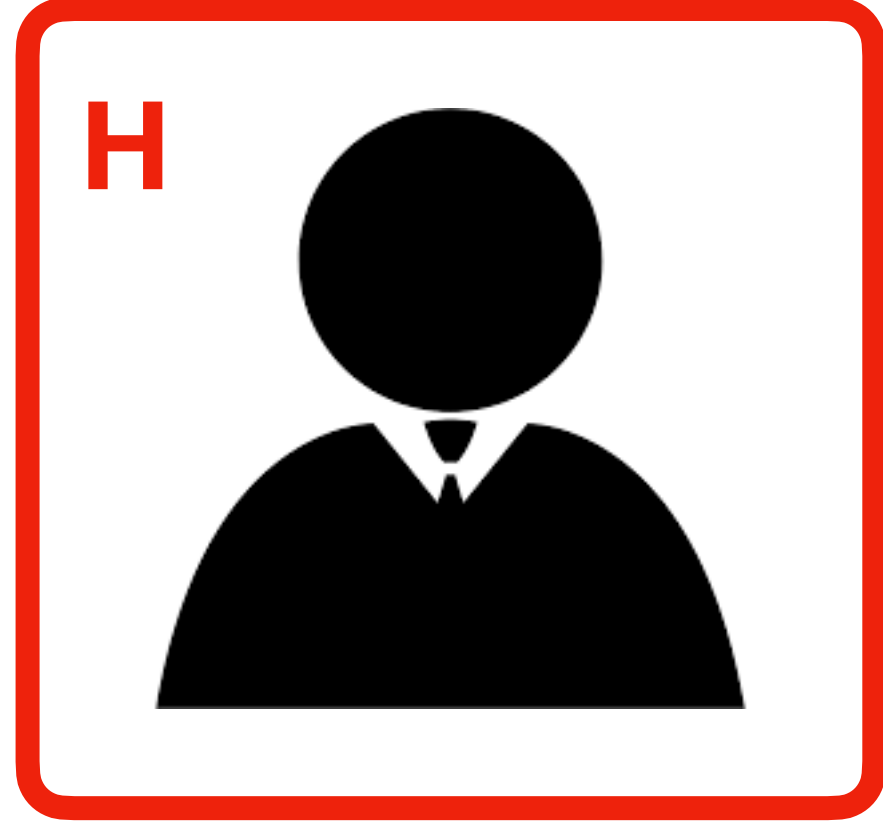


2



3



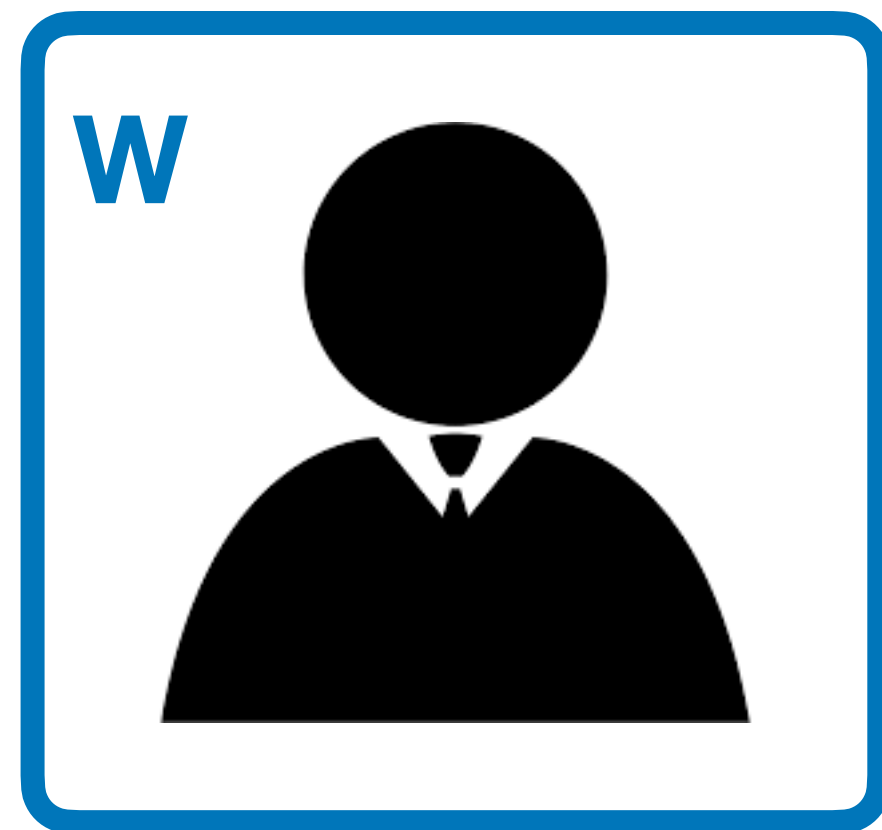


Problem 3

How to ensure irrevocability of consensus
in the presence of *priorities* and *asynchrony*?

Key Idea 4

- Cooperation between Proposers and Acceptors:
 - **Acceptors**, when agreeing to support a proposer, *must “tell” what was the highest-ballot value they have accepted*;
 - *Higher-ballot* **proposers** *re-propose* already (partially) accepted values from the *lower-ballot* proposers, who secured the quorum before.
- This way, a proposer “knows” that, once it secured its quorum, either
 - **its own proposal**, or some higher-ballot one will be accepted
 - **if its proposal got accepted, it will not be revoked**
(thanks to quorum intersection)



3

3



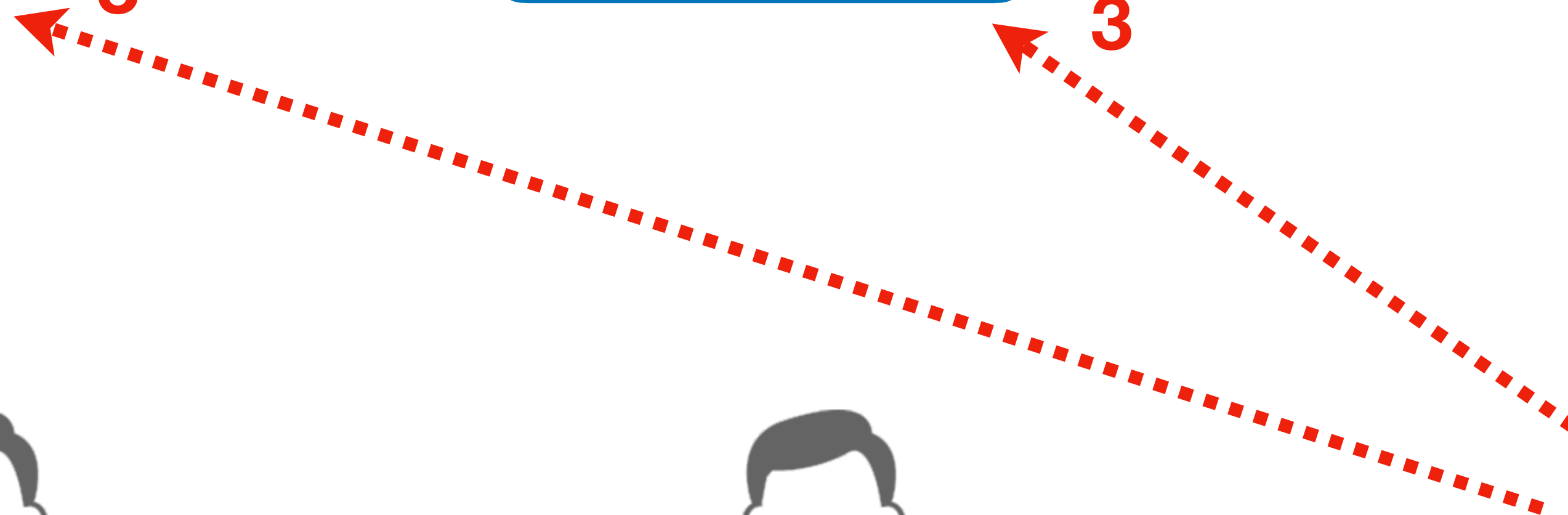
1

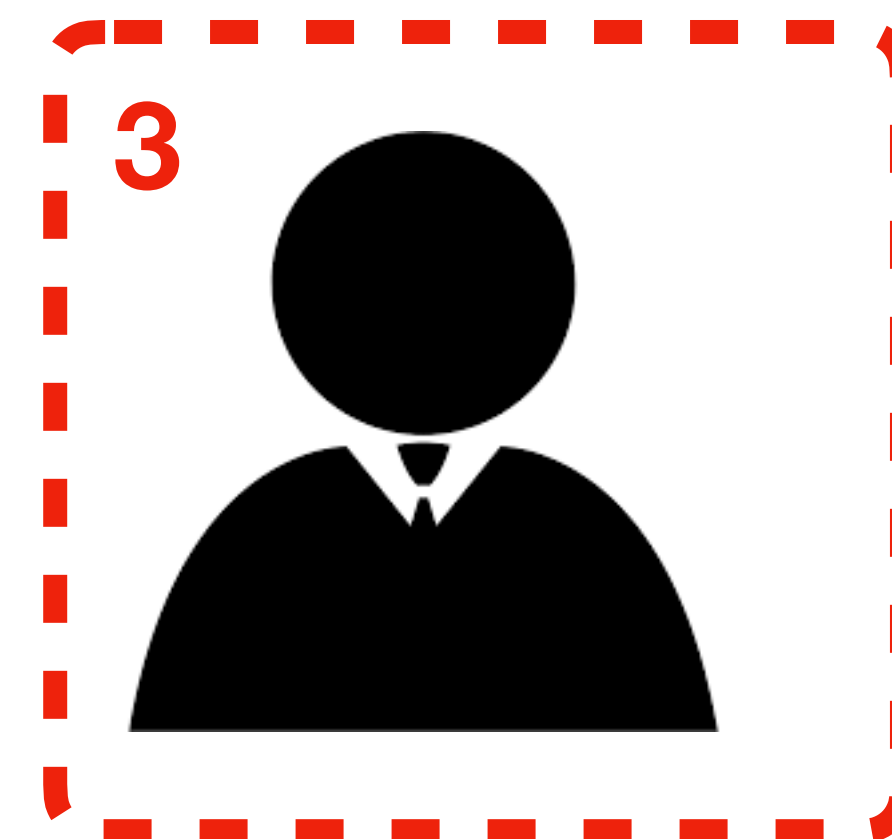
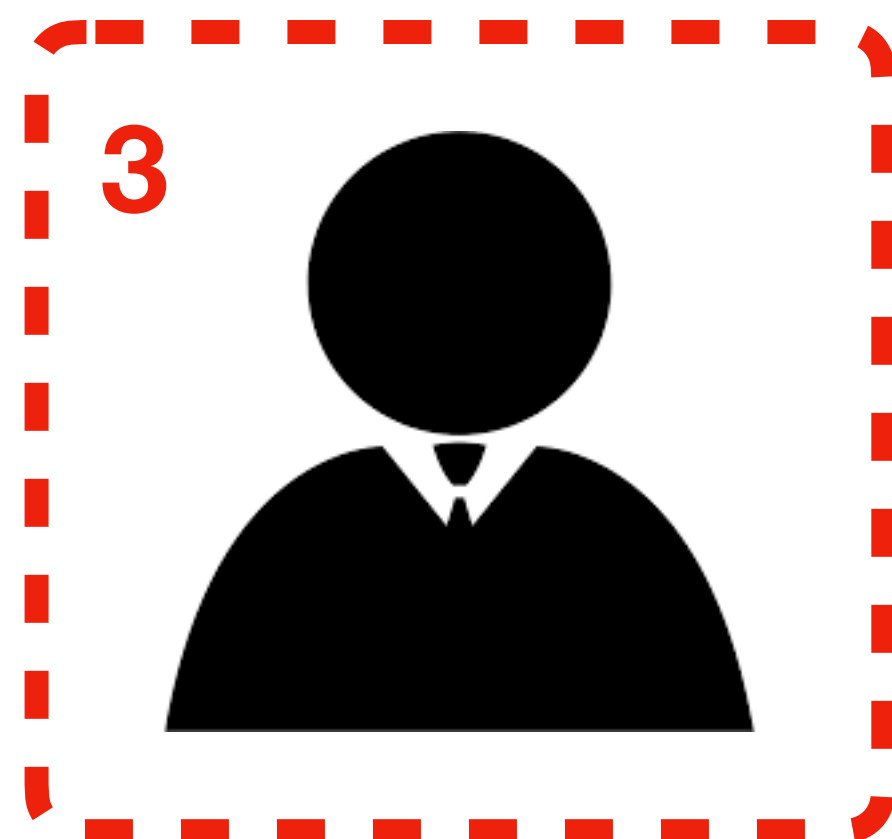


2



3



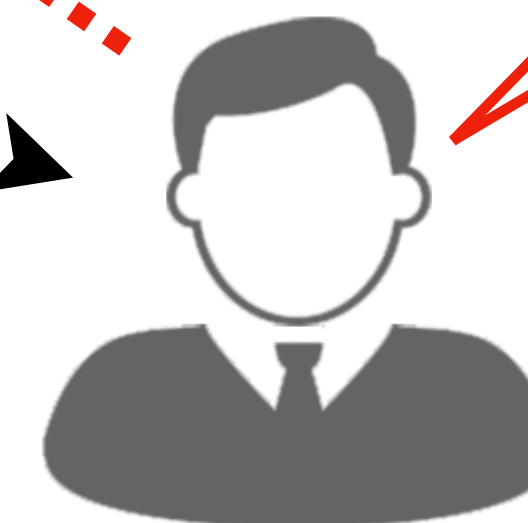
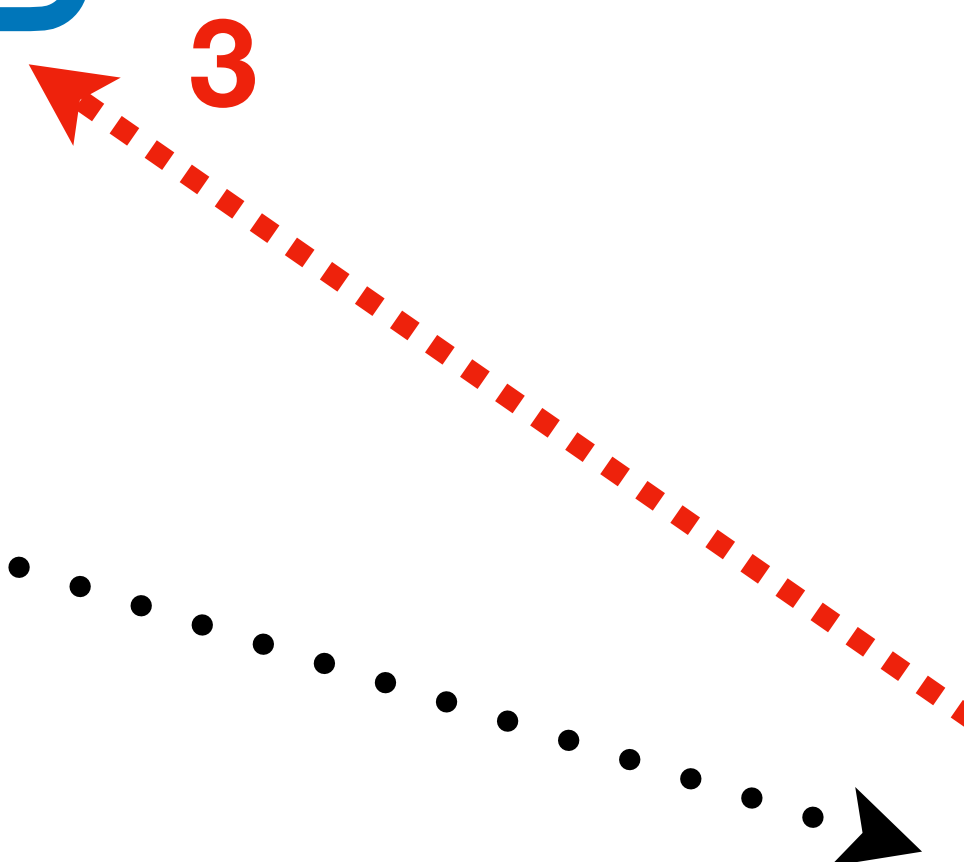


1

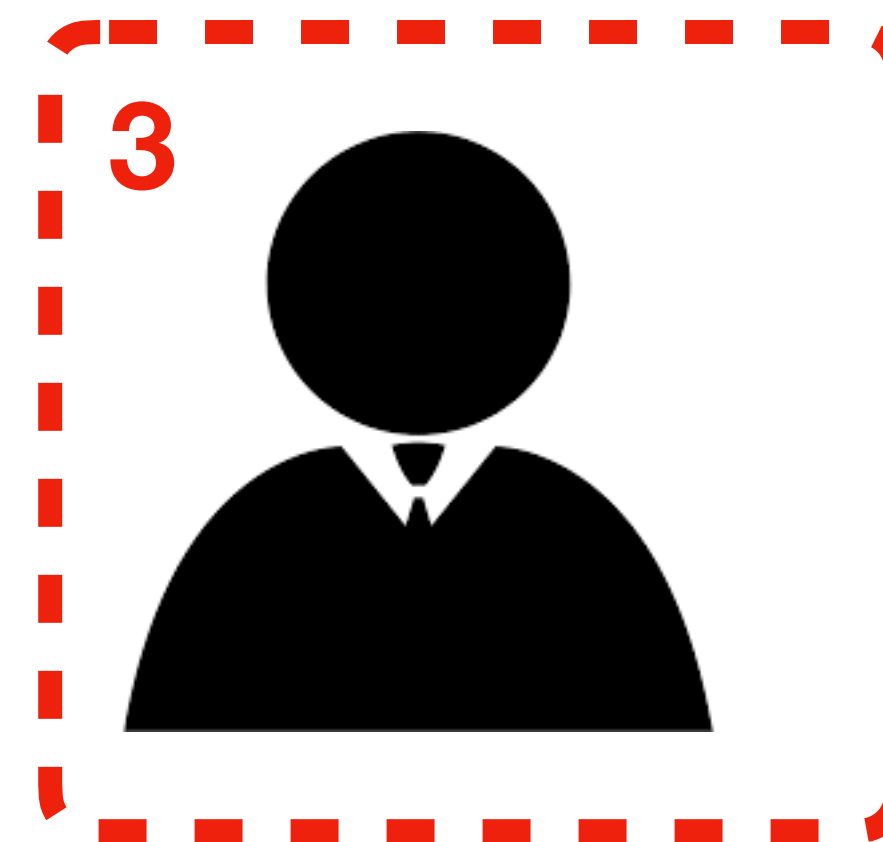
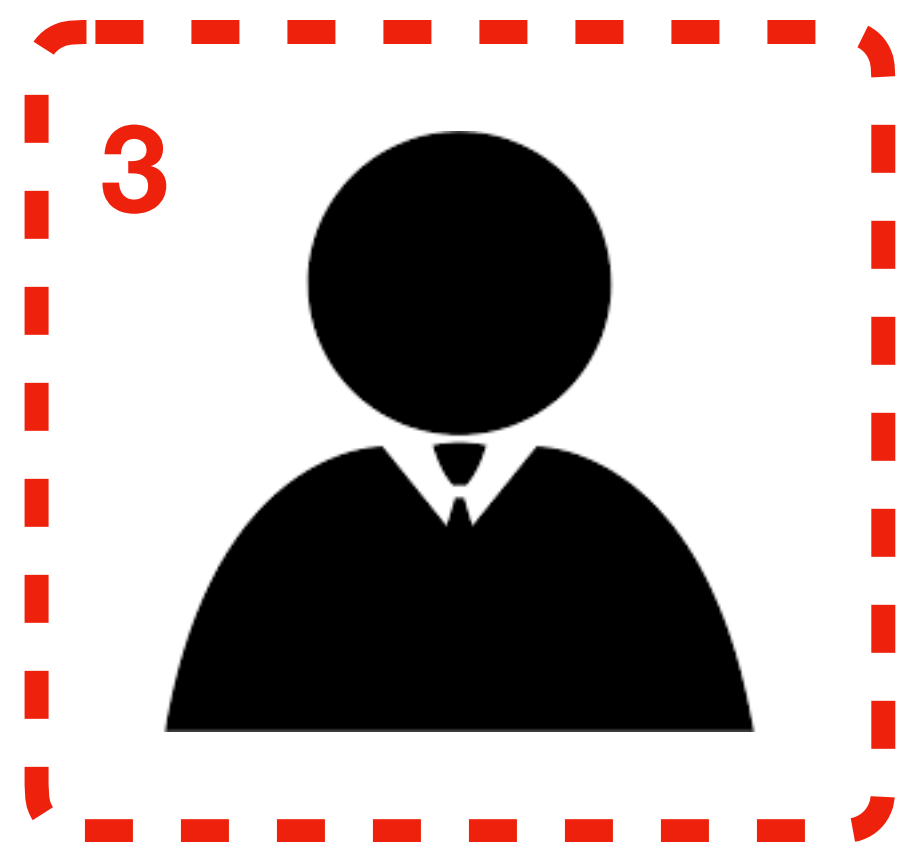
accepted W from 1



2



3



... accepted W from 1 ...



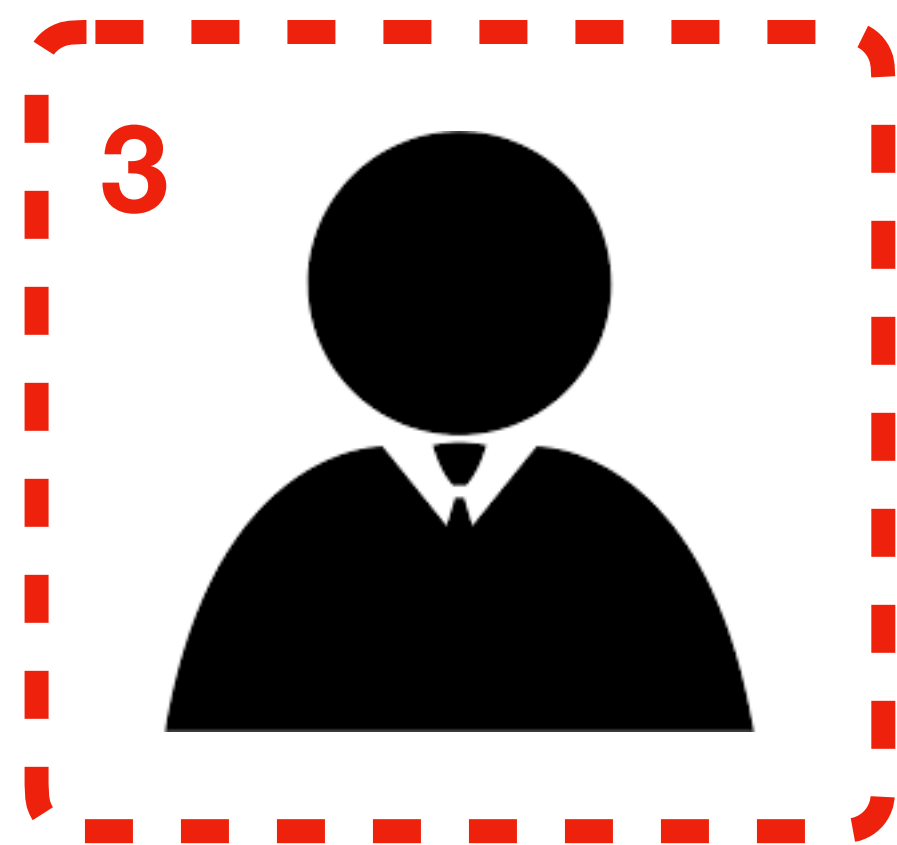
1



2



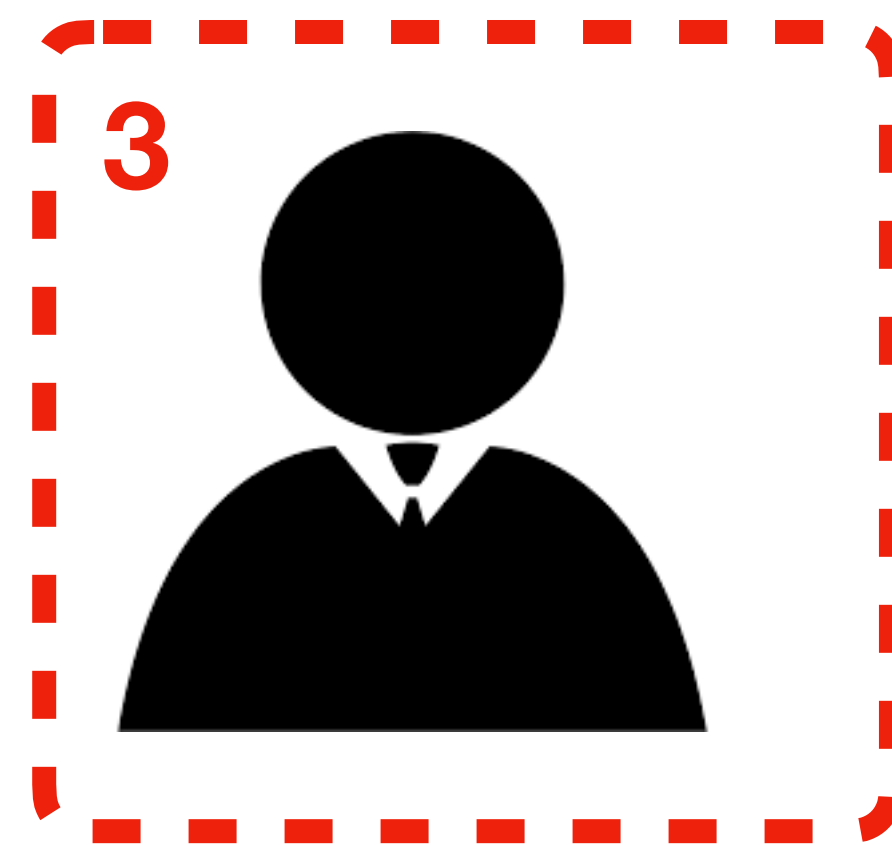
3



W



W



W



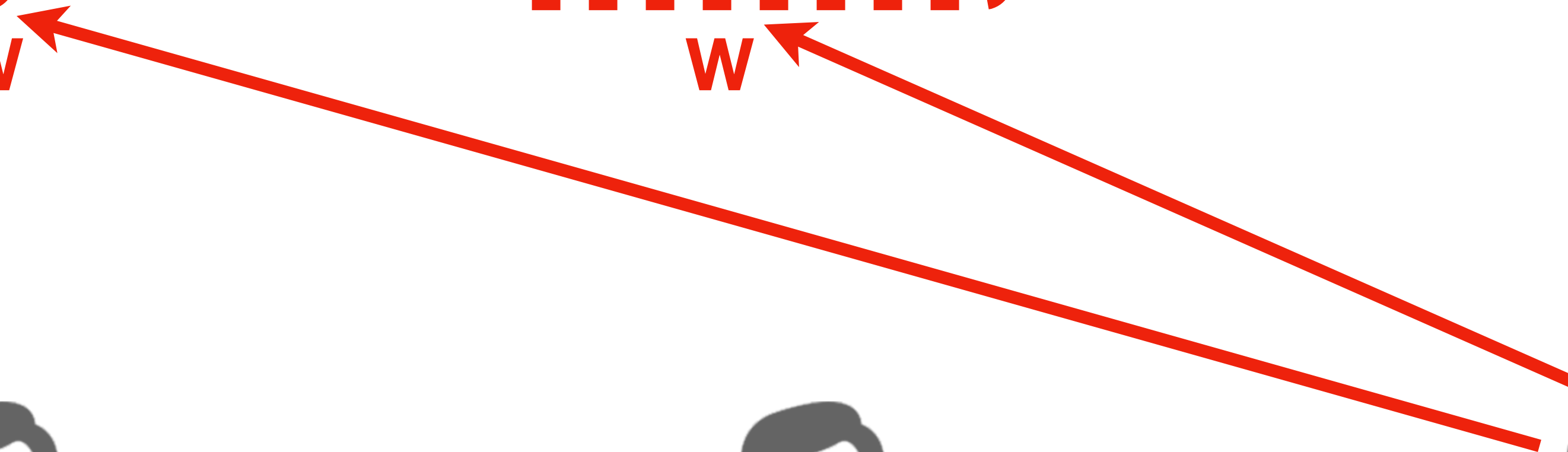
1

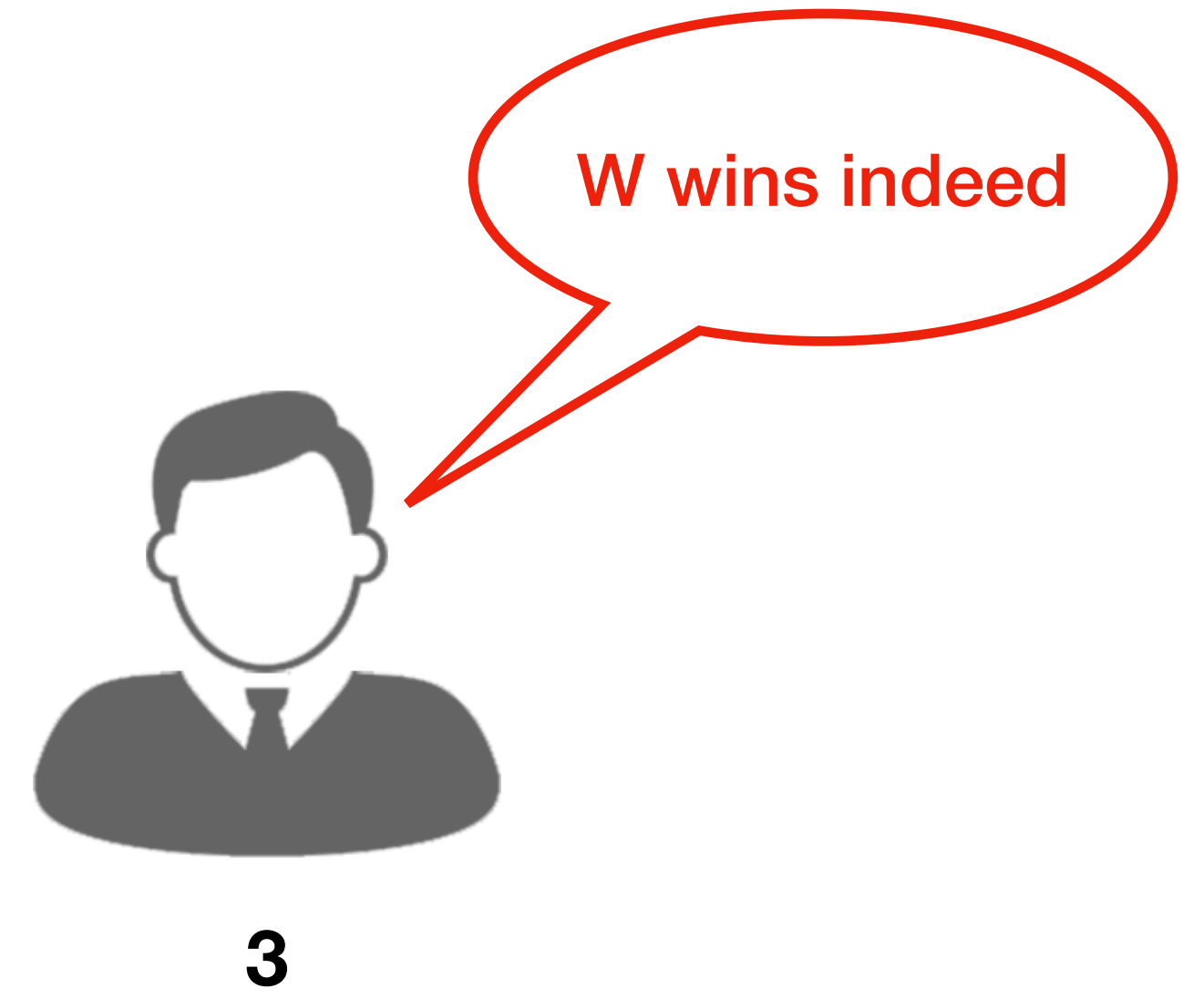
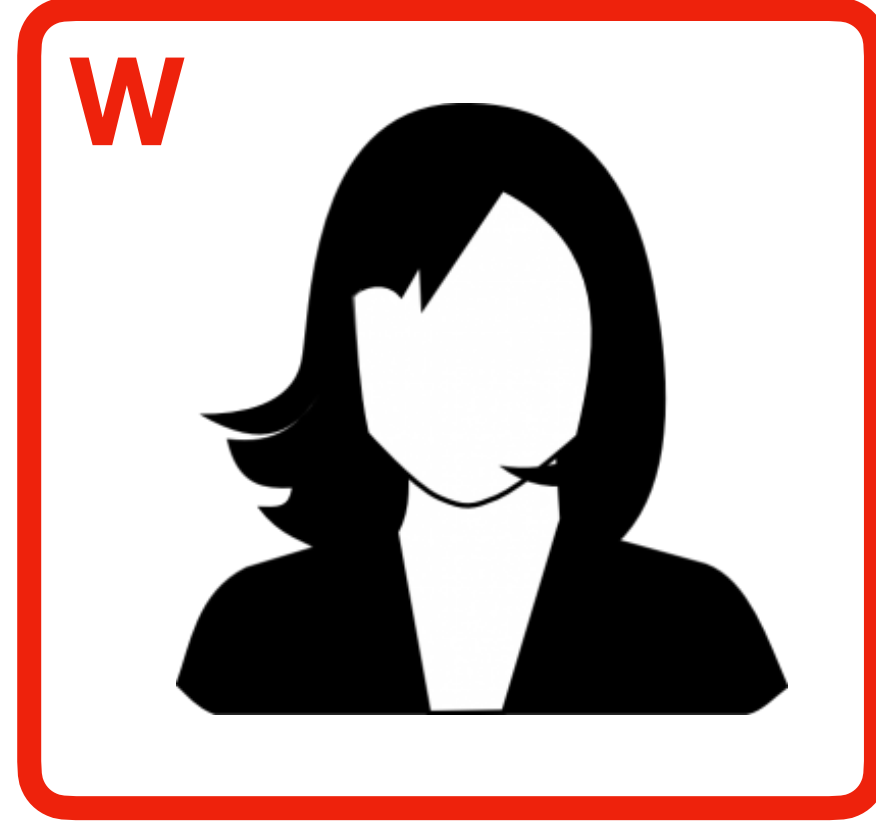


2



3





Two-Phase Ballot-based Consensus

- **Proposers** suggest values, **acceptors** decide upon acceptance;
- Each proposal goes in two rounds:
 - Phase 1: **securing** a quorum of acceptors for a proposal
 - Phase 2: **sending** out the proposal
- Acceptors agree only to support ballots *higher* than what they've seen;
- They inform proposers of *previously accepted values*, which those then re-propose.

The Algorithm in a Nutshell

Proposer

Acceptor

Phase 1

- Send my ballot **b** to all acceptors
- Wait for response of *at least* $n/2 + 1$ acceptors

- Upon receiving a ballot **b**
 - if it's the first one, remember it and send "ok" back.
 - if it's higher than **b'** we supported before, send back a previously accepted (b', v') , and remember **b** as what's currently supported.

Phase 2

- When heard back from $n/2 + 1$ acceptors, send them back (b, w) , where
 - **b** is my ballot
 - **w** is the value from the acceptors with the highest ballot, or *my own* value.
- Accept incoming value **w** if it comes with a ballot **b**, which we currently support; ignore otherwise.

Learning an Accepted Value

- A dedicated *learner* sends request to all *acceptors*;
- If at least $n/2 + 1$ acceptors respond back with *the same* value **v**, this is an accepted value.
- Correctness of this reasoning follows from *irrevocability*.

Learning an Accepted Value

- A dedicated *learner* sends request to all *acceptors*;
- If at least $n/2 + 1$ acceptors respond back with *the same* value **v**, this is an accepted value.
- Correctness of this reasoning follows from *irrevocability*.
- And what if $n/2 + 1$ have the value, but one of them *does not respond*?
- In this case we need to introduce a *time out* (*synchrony*).

Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

Yale University, New Haven, Connecticut

NANCY A. LYNCH

Massachusetts Institute of Technology

AND

MICHAEL S. PATERSON

University of Warwick, Coventry, England

If we expect failures,
we should rely on time-outs.

Abstract. The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the “Byzantine Generals” problem.

Paxos

- A practical fault-tolerant distributed consensus algorithm;
- Invented in 1990, published in 1998;
- Nowadays used everywhere: Google (Bigtable, Chubby), IBM, Microsoft;
- You have just seen it explained.

History of Paxos

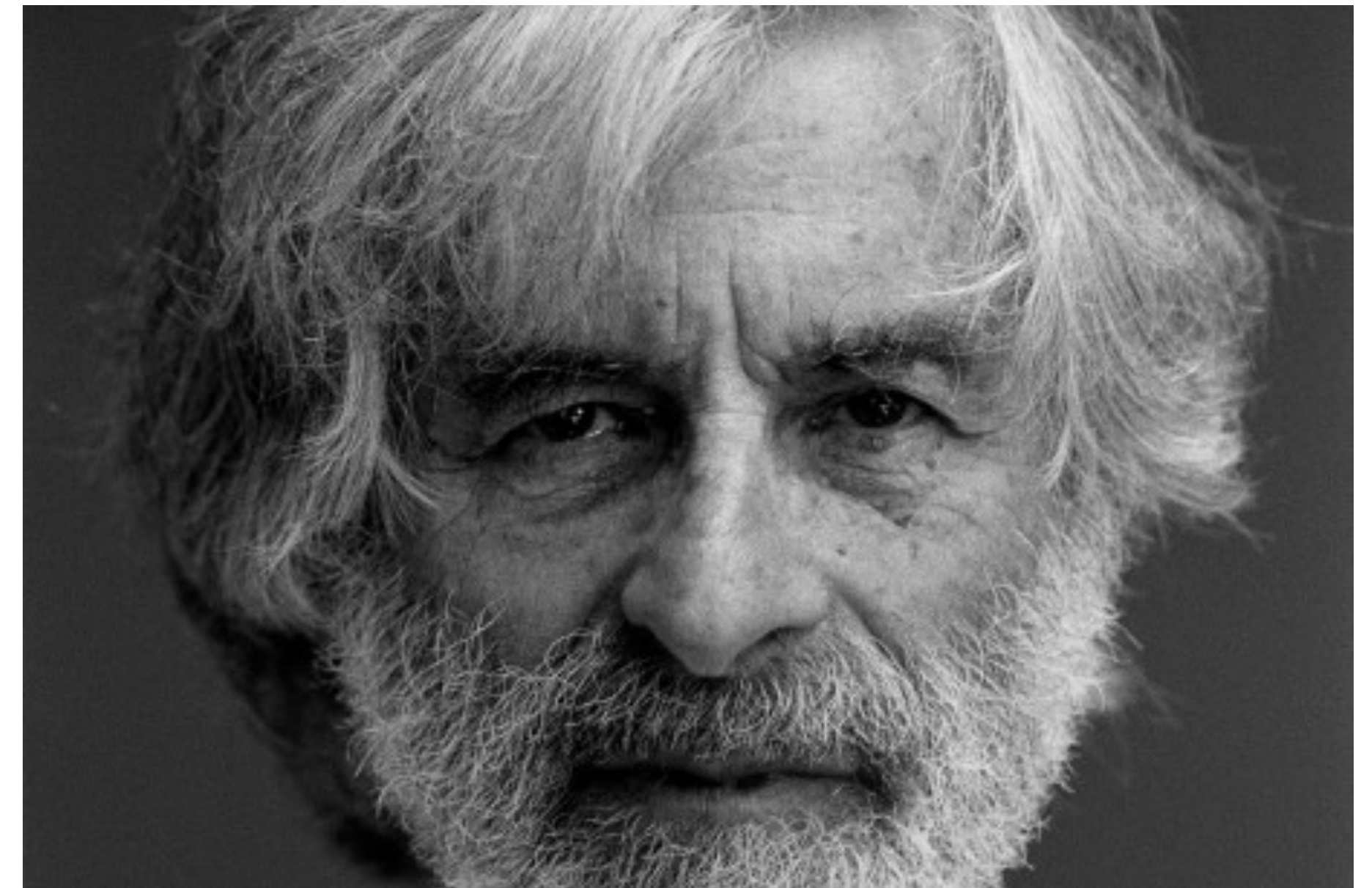
1990: Paxos first described

1998: Paxos paper published

2005: First practical deployments

2010: Widespread use!

2014: Lamport gets Turing Award



Leslie Lamport

(also known for LaTeX, Vector clocks, TLA)

Turing Award winner 2014

History of Paxos

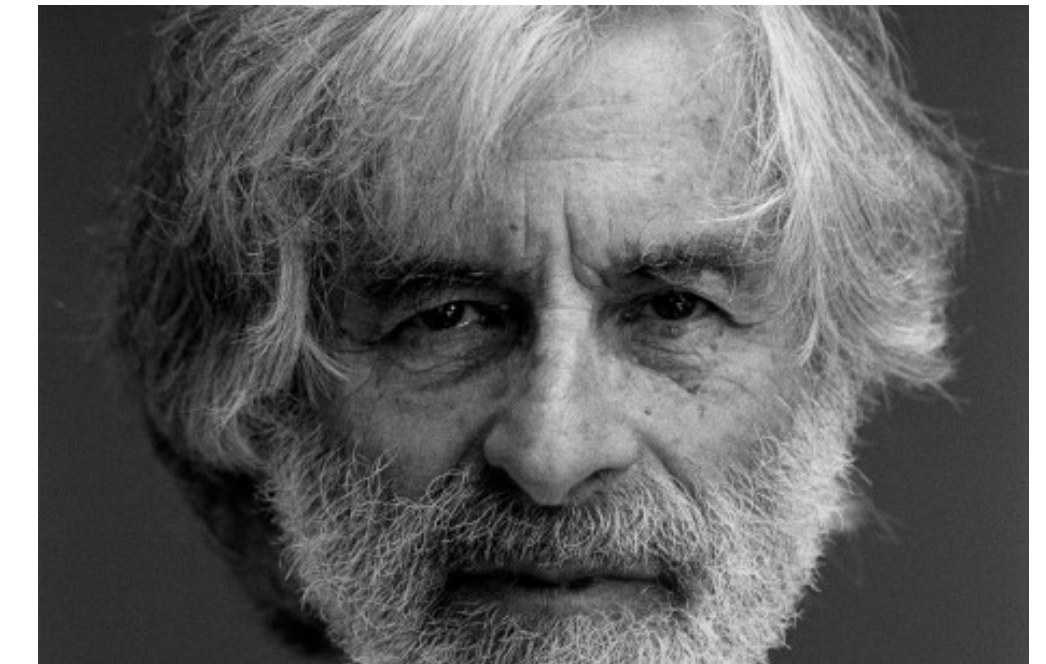
1990: Paxos first described

1998: Paxos paper published

2005: First practical dep

2010: Widespread use!

2014: Lamport gets Turing



Leslie Lamport
(also known for LaTeX, Vector clocks, TLA)
Turing Award winner 2014

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators.

The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers.

History of Paxos

1990: Paxos first described

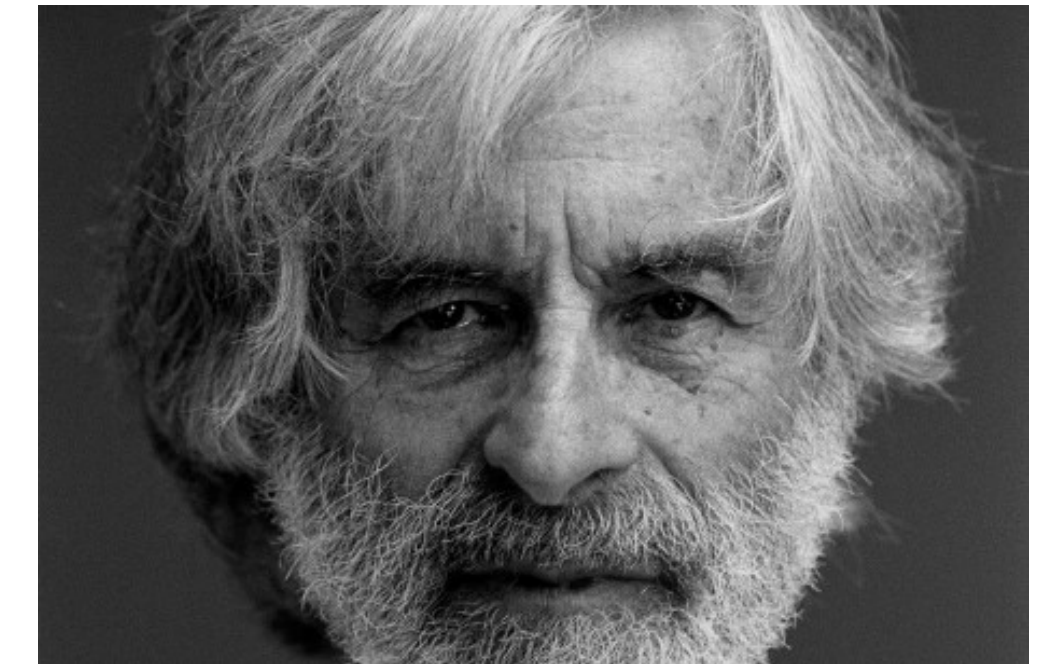
1998: Paxos paper published

2005: Fi

2010: W

2014: L&

- The ABCDs of Paxos [2001]
- Paxos Made Simple [2001]
- Paxos Made Practical [2007]
- Paxos Made Live [2007]
- Paxos Made Moderately Complex [2011]
- Paxos Consensus, Deconstructed and Abstracted [2018]



Leslie Lamport
(also known for LaTeX, Vector clocks, TLA)
Turing Award winner 2014

Multi-Paxos

- Presented in the original Lamport's 1998 paper.
- Uses the described idea for a *sequence* of “*slots*” (think *transactions*).
- Includes *reconfiguration* (changing set of acceptors on the fly).
- Naive implementation: run Simple Paxos for *each slot*.
- Better approach — secure a quorum for *several slots*.

Running Paxos with Scala Actors

```
class PaxosConfiguration(val proposers: Seq[ActorRef],  
                        val learners: Seq[ActorRef],  
                        val acceptors: Seq[ActorRef]) { ... }
```

```
def createPaxosInstance(system: ActorSystem, numProposers: Int,  
                       numAcceptors: Int, numLearners: Int): PaxosConfiguration = {  
  
    val acceptors = createAcceptors(system, numAcceptors)  
    val proposers = createProposers(system, numProposers, acceptors)  
    val learners = createLearners(system, numLearners, acceptors)  
    new PaxosConfiguration(proposers, learners, acceptors)  
}
```

```
def createProposers(system: ActorSystem, numProposers: Int, acceptors: Seq[ActorRef]) = {  
    for (i <- 0 until numProposers) yield {  
        system.actorOf(Props(ProposerClass, this, acceptors, i), name = s"Proposer-P$i")  
    }  
}
```

Live Demo

Alternative Consensus Protocols

- **View-Stamped Replication**
by Brian M. Oki and Barbara Liskov, 1989
- **Raft**
by Diego Ongaro and John K. Ousterhout, 2014

To Take Away

- Fault-Tolerant Consensus Protocols are a *critical component* of modern *distributed systems and applications*
- Consensus properties are *uniformity*, *non-triviality*, and *irrevocability*
- The key ideas of Lamport's Paxos protocol are:
 - Majority *quorums* (avoiding split brain and enabling fault-tolerance);
 - *Two-phase* structure (secure-then-commit);
 - Dichotomy and cooperation between *proposers* and *acceptors*.

To be continued...