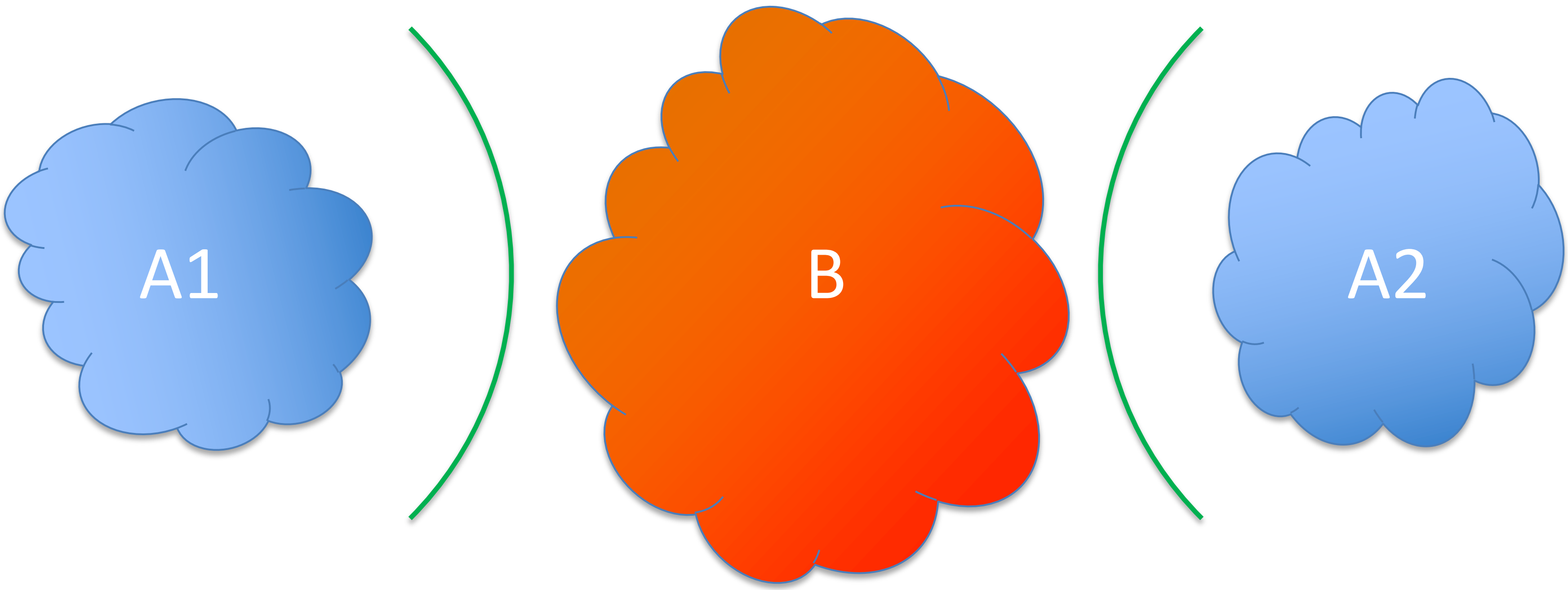
Consensus

Robbert van Renesse
Cornell University

Two Generals' Problem

a thought experiment



- ”A” can only win if A1 and A2 both attack. If one attacks, it will be decimated
- Generals of armies A1 and A2 can only communicate through messengers
- Messengers can get intercepted and killed when trying to pass through army B

This is an “agreement” problem

- Suppose there is a deterministic protocol that solves the problem
- Let n be the minimal number of messages required
- Since messages may or may not arrive, omitting the last message should also work
- Therefore, n = 0
- So only possible if the generals had decided ahead of time (“Global Knowledge”)

2 Generals in practice

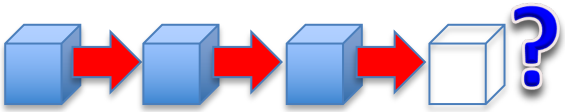
- TCP
  - How do endpoints agree on state?
  - When is it safe to garbage collect an endpoint?
    - They have to agree on the fact that the connection has terminated
      - A1 → A2: let’s terminate
      - A2 → A1: ok, let’s (unfortunately, gets lost)
        - » A2 cannot decide to garbage collect because it may leave A1 hanging
      - A1 → A2: let’s terminate (retransmission)
      - A2 → A1: ok, let’s
        - » A2 still cannot terminate for same reason as before
        - » A1 receives the message, but needs to inform A2 so
        - » …
  - In practice, time-outs are used

Keeping Replicas Synchronized

- The replicas agree on the transitions (operations) and the order in which to apply them
- The problem of a set of processes agreeing on something is called "consensus"
- Think of the sequence of transitions as a list of "slots"
- For each slot, State Machine Replication (SMR) has to solve consensus on a set of candidate transitions ("proposals")

What is Consensus?

- A way for multiple participants to agree on
  - the next update to perform in a replicated service
  - a leader
  - whether to abort or commit a transaction
  - a recovery action after a failure
  - *the next block in a block chain*



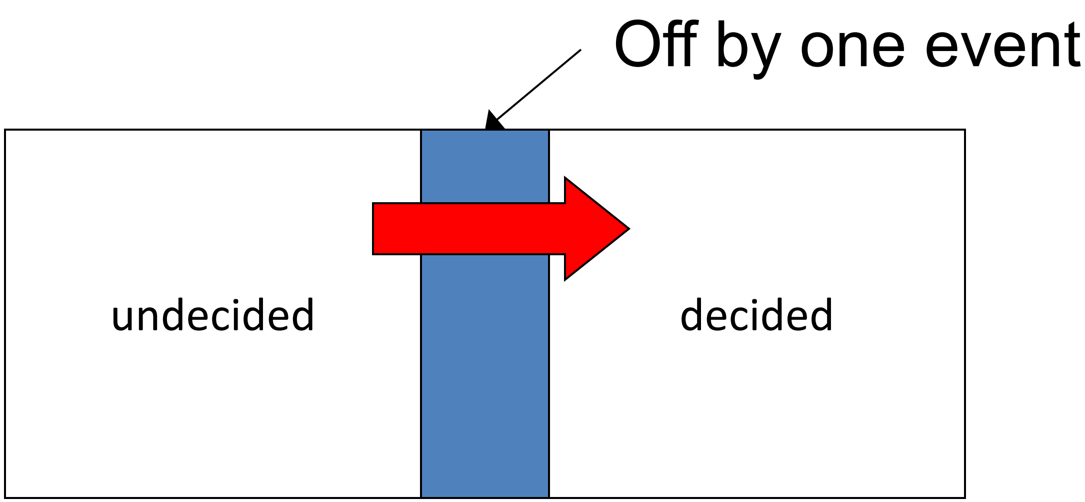
- Surprisingly hard with participant and network failures
  - whether accidental or malicious
- Even harder in the face of *asynchrony*
  - complete lack of bounds on latency

Consensus Formalized

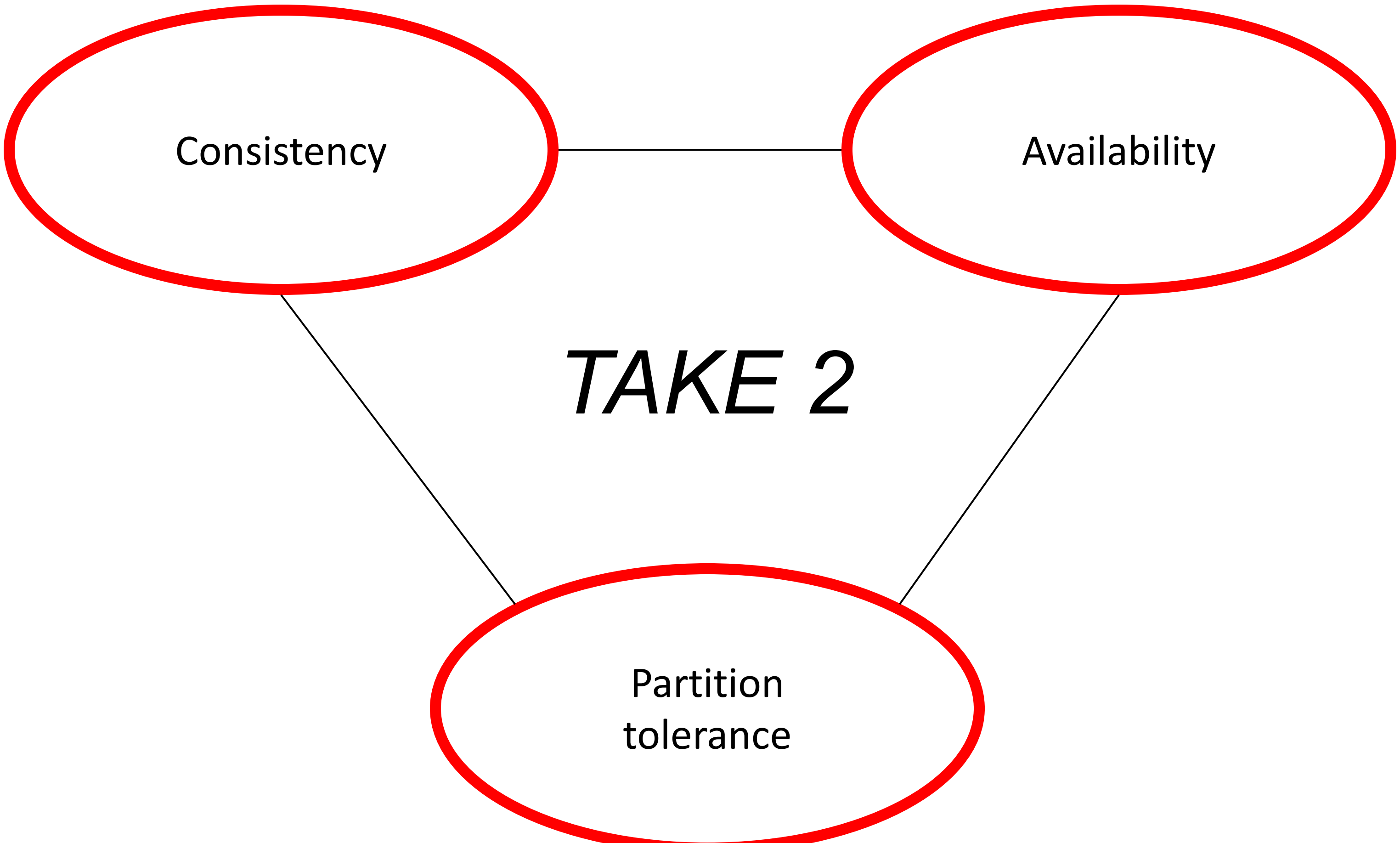
- Agreement:
  - if two replicas decide, they must decide the same proposed operation
- Validity:
  - a replica can only decide an operation that was proposed by some replica
    - without this requirement, replicas could just decide “no-op” each time
- Termination:
  - a correct (non-crashing) replica must eventually decide (assuming at least one operation was proposed)

Solving consensus is hard…

Crash failures + no assumptions about timing ⇒ *solving consensus is impossible* (FLP’ 83, FLP’ 85)



Add Network Failures…



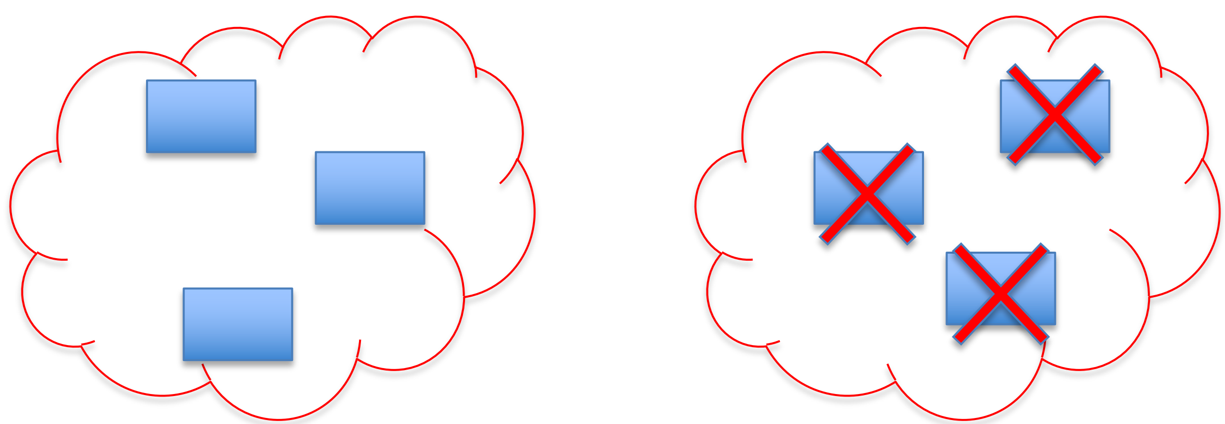
Lower Bound on number of participants

In an asynchronous environment with crash failures, you need at least $2f + 1$ replicas to tolerate f crash failures

- $2f$ is not enough: consider the difference between two groups of f processes being separated by a network partition and one group of those processes crashing: can the other group see the difference?

indistinguishability argument

(f = 3)



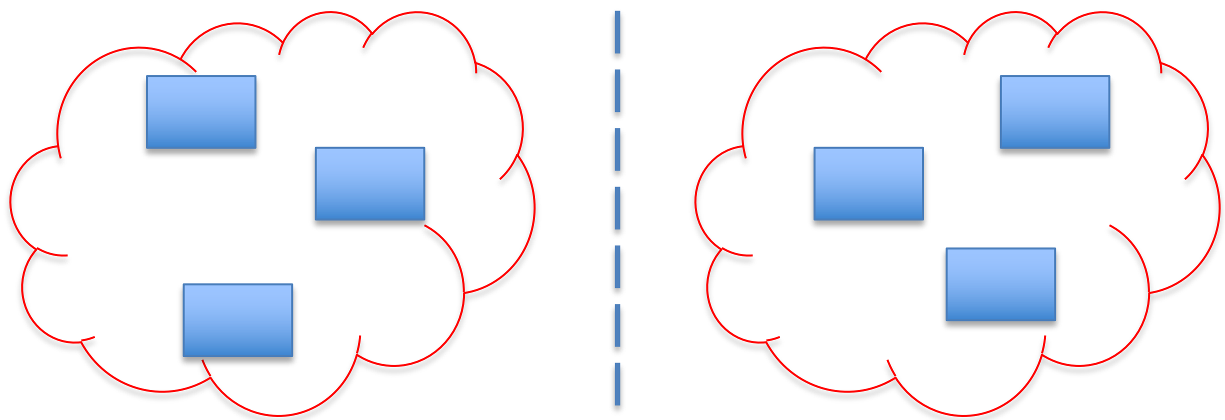
Lower Bound on number of participants

In an asynchronous environment with crash failures, you need at least $2f + 1$ replicas to tolerate f crash failures

- $2f$ is not enough: consider the difference between two groups of f processes being separated by a network partition and one group of those processes crashing: can the other group see the difference?

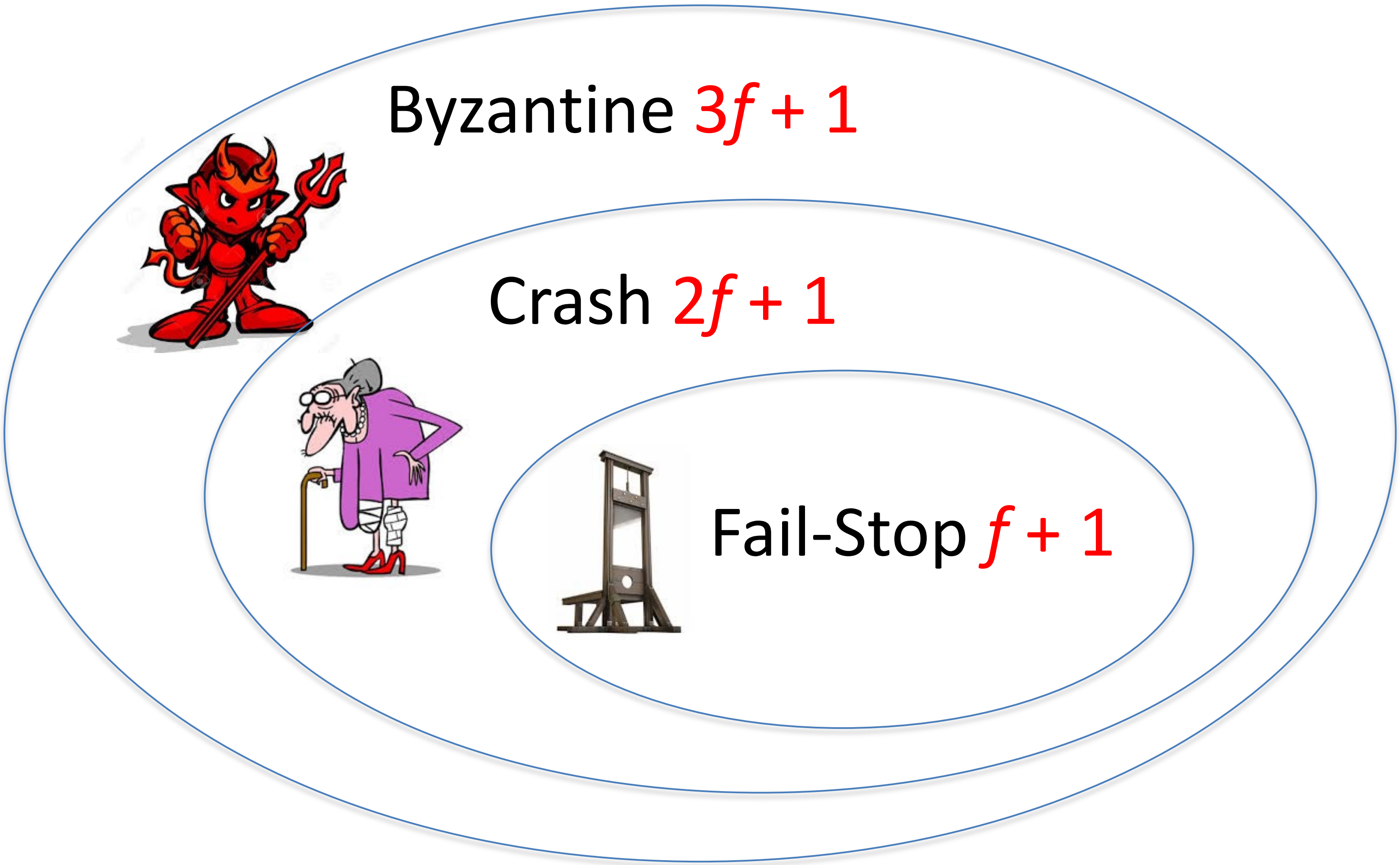
indistinguishability argument

($f = 3$)



if $2f$ were enough, each group could make a decision independently of the other

Other Lower Bounds

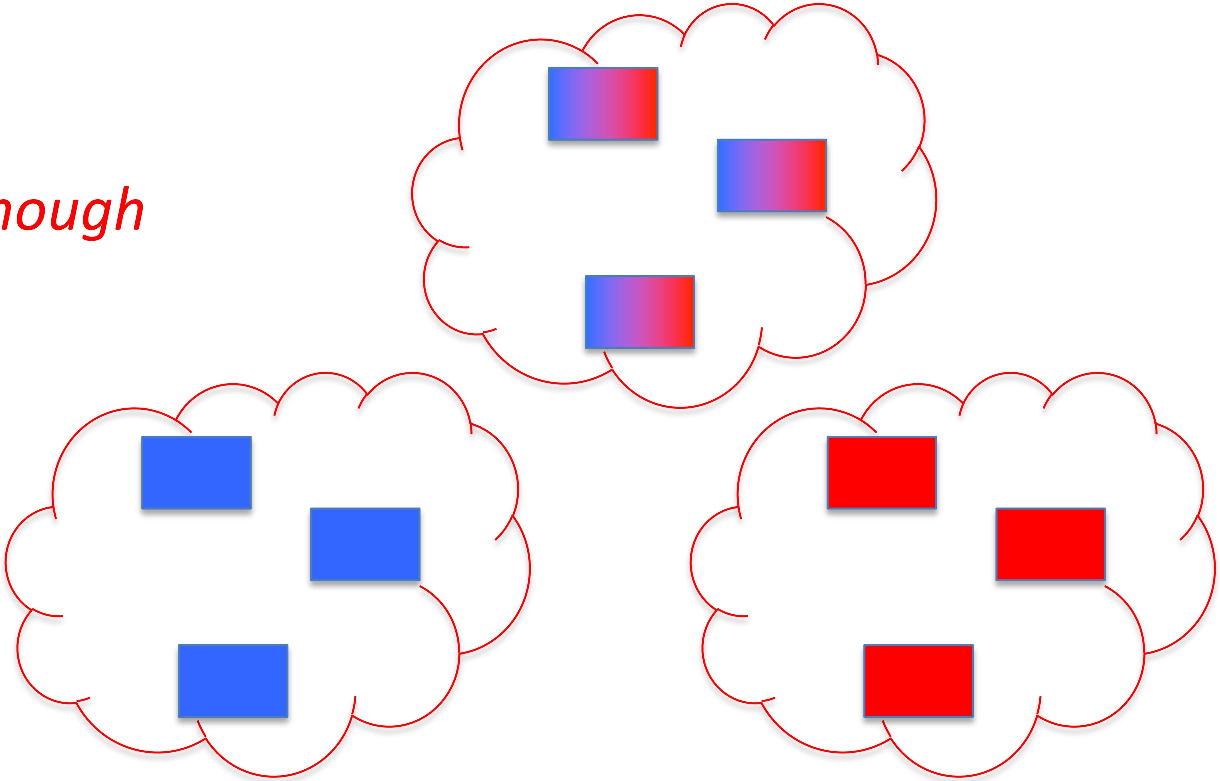


Lower Bound with Byzantine Failures

In an asynchronous environment, you need at least $3f + 1$ participants to tolerate f Byzantine failures

indistinguishability argument: 3f is not enough

(f = 3)



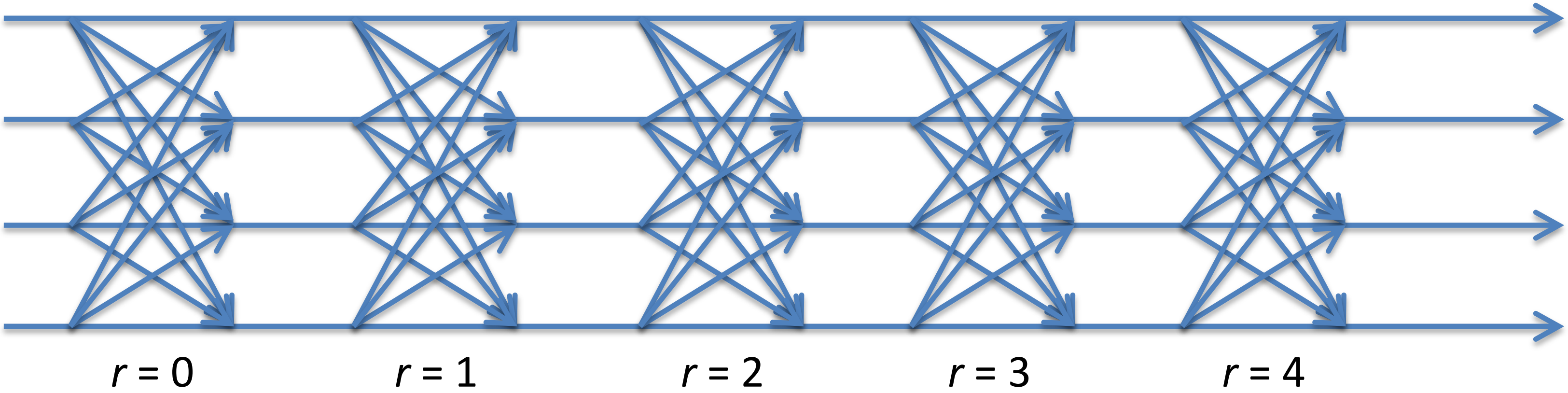
Example consensus protocol with $3f + 1$ processes: setup

- Asynchronous environment
- $3f + 1$ processes, at most f of which may experience a crash failure
  - note: $3f + 1$ is more than the lower bound $2f + 1$
    - thus this protocol will not be optimal in the number of processes
- The processes run *rounds* of communication
- Each process maintains a round number r and an estimate e
- Initially $r = 0$ and e is the proposal of the process.

Protocol with $3f + 1$ processes

1. Broadcast $< r, e >$ "vote" (including to self)
2. Wait for $2f + 1$ votes (out of $3f + 1$)
   - *Note*: because as many as f may fail, this is the maximum a process can safely wait for
3. If a majority of the $2f + 1$ votes contains the same proposal, change e to that proposal
   - *Note*: because $2f + 1$ is odd, there cannot be a tie
4. If not, set e to a proposal in any of the votes received
5. If all votes contain the same proposal (unanimity), decide that proposal
6. $r := r + 1$
7. Repeat (go to Step 1, starting next round)

Generic Asynchronous Consensus



Example Run with $f = 1$

	Process 1	Process 2	Process 3	Process 4
Vote 0	RED	RED	BLUE	BLUE
Receive	RRB	BRB	RRB	RBB
Vote 1	RED	BLUE	RED	BLUE
Receive	BRB	BBR	RRB	RBR
Vote 2	BLUE	BLUE	RED	RED
Receive	BRB	RBB	RRB	BBR
Vote 3	BLUE	BLUE	RED	BLUE
Receive	BBR	**BBB**	RBB	**BBB**
Vote 4	BLUE	BLUE	BLUE	BLUE
Receive	**BBB**	BBB	**BBB**	BBB

univalence

Validity?

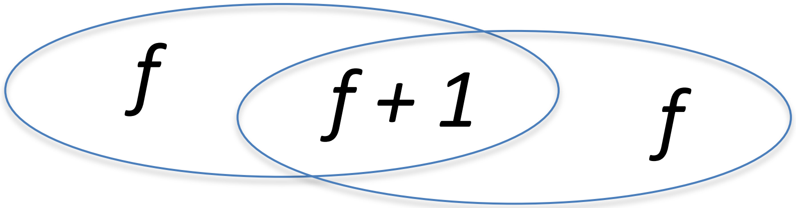
Obvious:

- no proposals invented by the protocol
- processes always vote for one of the original proposals

Agreement?

By contradiction:

- two processes deciding *e* and *e*’ in the same round?
  - can’t happen because they each need $2f + 1$ votes for their proposal, and there are only $3f + 1$ processes
- two proc’s deciding *e* in round *r* and *e*’ in round *r*’?
  - can’t happen: if a process decides *e* in round *r*, then $2f+1$ process must have voted for *e*. Thus any correct process must have received at least $f + 1$ votes for *e* in the same round, and change its estimate to *e*. Hence starting in round $r + 1$, all votes will be for *e* and no other value can be decided.



Termination?

This protocol doesn't guarantee it

- Suppose $f = 1$, and thus there are four processes
- In round 0, two processes propose RED and two processes propose BLUE.
- In round 1
  - two processes receive two RED and one BLUE vote and set their estimate to RED
  - the other two processes receive one RED and two BLUE votes and set their estimate to BLUE
- *Status quo maintained*…
  - this scenario can be repeated indefinitely

FLP Impossibility Result

Fisher, Lynch, and Patterson 1985:

– There does not exist a deterministic consensus protocol that can guarantee all of Validity, Agreement, and Termination in an asynchronous environment that admits one or more crash failures

Proof Sketch

- Consider a correct binary determistic consensus protocol
  - Validity, Agreement, *and* Termination
- Call a state of the protocol x-valent if all executions from that state can only decide x (x = 0 or 1)
  - For example, the state in which all processes propose x is x-valent because of Validity
  - A state in which x is already decided is also x-valent
- Call a state bivalent if it can decide either 0 or 1

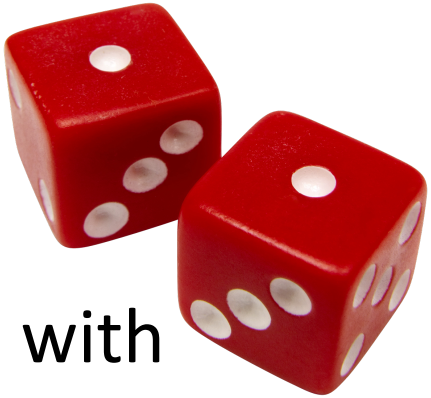
Proof Sketch, cont'd

- Lemma: the protocol has an initial bivalent state
- By contradiction
  - consider two initial states S0 and S1, one 0-valent and one 1-valent, that only differ in the proposal of some process p (clearly must exist)
  - since protocol can tolerate one failure, there must exist a deciding execution from S0 in which p takes no steps. Now run same execution from S1 (changing p's proposal). It'll still decide 0, but S1 is 1-valent…

Proof Sketch, cont'd

- Consider a bivalent state and a process *p* such that if *p* takes a step the state becomes 0-valent
- There cannot be a step by another process to a state that is 1-valent
  - What would happen if both processes took a step?
  - Depends on the order, but resulting state is the same
- But since the state is bivalent, there must exists an execution to a 1-valent state
  - So, let's follow that path (except for the last step) instead of having p take a step
- Hence, we can create an infinite execution that never decides, contradicting Termination

Is all hope lost?



- No, protocols exist that reach termination with probability 1
  - that is not quite as good as a guarantee
    - similar to tossing a coin repeatedly: in theory it may never happen that heads comes up
    - but it's extremely unlikely (probability 0)
- Most consensus protocols are likely to terminate in one or two rounds
- Even with very weak additional assumptions, termination can be guaranteed
  - e.g., the existence of a bound on latency, even if that bound is unknown

Meeting the 2*f*+1 lower bound

- The trick is to create a protocol that guarantees that *if* two processes vote in the same round, they vote for the same proposal
- One instantiation of this trick is to assign to each round a "leader"
  - for example, the leader role could rotate among the processes from round to round
- Processes are allowed to abstain from voting, for example if they don't hear from the leader within a reasonable amount of time

$2f + 1$ consensus protocols

- Again, round-based
- Each round consists of two *phases*:
  1. Determine a single proposal to vote on
     - For example, by leader or majority
     - This may fail and is no substitute for consensus in its own right
  2. Vote on the proposal if there is one
     - Protocol decides if majority votes (for the proposal)
     - Processes may abstain, so again there is no guarantee that a decision is made

What is Paxos?

- Paxos is a state machine replication protocol for asynchronous environments with crash failures [Leslie Lamport, 1989].
- It uses a consensus protocol called "Synod" that meets the lower bound
  - you need $2f + 1$ "acceptors" to tolerate f failures
  - rounds are called "ballots"
  - *each ballot has a leader*
  - the leader determines the proposal for a ballot
    - based on input from a majority of acceptors
    - each acceptor reports its highest vote by ballot number, or NULL if it never voted
  - the leader selects the proposal with the highest ballot number, or its own proposal if all acceptors report NULL
  - the leader broadcasts the selected proposal and ballot number
  - the acceptors vote if they have not heard from a leader of a ballot with a higher ballot number
  - a replica decides if it learns a majority of acceptors voted on the same ballot

Why so popular?

Paxos is *pragmatic*:

- it meets the lower bound for number of processes needed ($2f + 1$)
- leader-based protocols deal well with contention (multiple concurrent proposals from different clients)
- Synod has an important optimization when running multiple instantiations so that most slots require only the second phase
  - the leader can be reused from slot to slot for the first ballot
  - most decision involve only three message latencies:
    1. a leader broadcasting a proposal, requesting acceptors to vote
    2. the acceptors voting and responding (the leader is waiting)
    3. the leader learning the decision and notifying the replicas
- Synod is guaranteed to terminate *if* there exists a bound on message latencies and processing times
  - by doubling the timeout on waiting in each ballot

Comparison to Primary-Backup

Paxos	Primary-Backup
• aka Active Replication	• aka Passive Replication
• needs $2f + 1$ participating processes (although f of those only need to be voting *witnesses*)	• needs $f+1$ participating processes (1 primary and f backups)
• each replica applies all operations	• only the primary applies operations, backups maintain only state
• does not require accurate failure detection	• requires accurate failure detection (unrealistic?)
• *masks* failures	• failures require complicated recovery
• requires three message latencies in the normal case	• requires two message latencies in the normal case

Glossary (by way of conclusion)

Term	Meaning
Acceptor	voting participant in Paxos
Agreement	no two processes decide differently
Asynchrony	no bounds on timing
Ballot	essentially the same as a round
Consensus	a protocol for agreeing on a proposal
Crash	process stops making transitions
Leader	proposes a value in the first phase of a round
Phase	part of a round
Replica	a copy of a state machine
Round	an exchange of messages between participants
Termination	correct processes eventually decide
Validity	a process can only decide a proposal

Crash Tolerant

Protocol with $3f + 1$ processes

1. Broadcast $< r, e >$ "vote" (including to self)
2. Wait for $2f + 1$ votes (out of $3f + 1$)
   - *Note*: because as many as f may fail, this is the maximum a process can safely wait for
3. If a majority of the $2f + 1$ votes contains the same proposal, change e to that proposal
   - *Note*: because $2f + 1$ is odd, there cannot be a tie
4. If not, set e to a proposal in any of the votes received
5. If all votes contain the same proposal (unanimity), decide that proposal
6. $r := r + 1$
7. Repeat (go to Step 1, starting next round)

Byzantine Tolerant

Protocol with $5f + 1$ processes

1. Broadcast < *r, e* > "vote" (including to self)
2. Wait for $4f + 1$ votes (out of $5f + 1$)
   – *Note*: because as many as f may fail, this is the maximum a process can safely wait for
3. If a majority of the $4f + 1$ votes contains the same proposal, change *e* to that proposal
   – *Note*: because $4f + 1$ is odd, there cannot be a tie
4. If not, set *e* to a proposal in any of the votes received
5. If all votes contain the same proposal (unanimity), decide that proposal
6. $r := r + 1$
7. Repeat (go to Step 1, starting next round)

Example Run with $f = 1$

	Process 1	Process 2	Process 3	Process 4	Process 5	Process 6
Vote 0	RED	RED	BLUE	BLUE	BLUE	RED/BLUE
Receive	RRRBB	BRBBB	RRRBB	RRBBB	RRRBB	
Vote 1	RED	BLUE	RED	BLUE	RED	RED/BLUE
Receive	BRRBB	BBRRB	RRRRB	RBRRR	RRRBB	
Vote 2	BLUE	BLUE	RED	RED	RED	RED/BLUE
Receive	BBRRB	RRBBB	BRRBB	BBBRR	RRRBB	
Vote 3	BLUE	BLUE	BLUE	BLUE	BLUE	RED/BLUE
Receive	BBBBR	**BBBBB**	RBBBB	**BBBBB**	BBRBB	
Vote 4	BLUE	BLUE	BLUE	BLUE	BLUE	RED/BLUE
Receive	**BBBBB**	BBBRB	**BBBBB**	BRBBB	BBRBB	

TRANSLATING CRASH TOLERANT PROTOCOLS INTO BYZANTINE TOLERANT PROTOCOLS

Plan

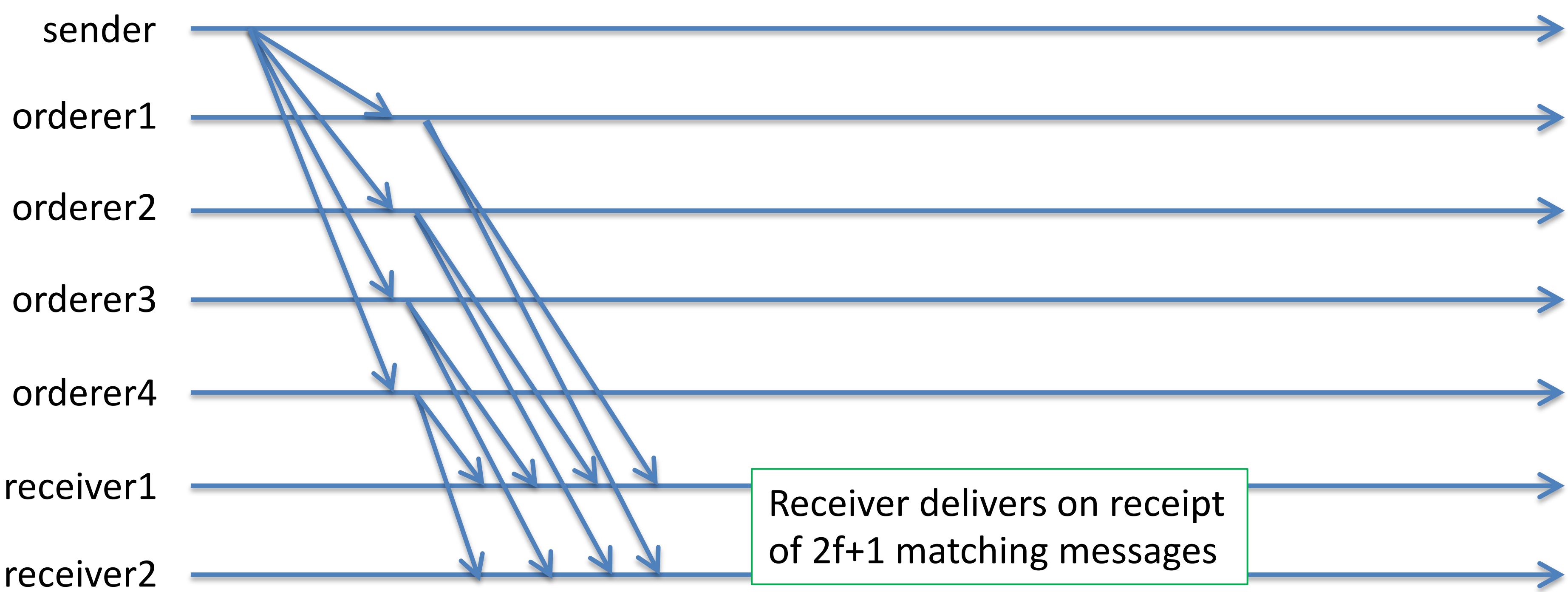
- Introduce OARCAST
- Show how OARCAST can be used to translate any crash tolerant protocol into a Byzantine tolerant one

OARCAST

- Ordered Authenticated Reliable Broadcast
- 1 sender, *N* receivers
- Properties:
  1. Persistence: if sender is correct, all correct receivers will receive all the sender's messages
  2. Relay: if one correct receiver delivers a message, all correct receivers will deliver the same message
  3. Authenticity: if sender is correct and does not send m, no correct receiver will deliver m
  4. FIFO: if sender is correct, correct receivers deliver its messages in the order sent
  5. Order: if two correct receivers deliver m1 and m2, then they deliver m1 and m2 in the same order (even if the sender is Byzantine)

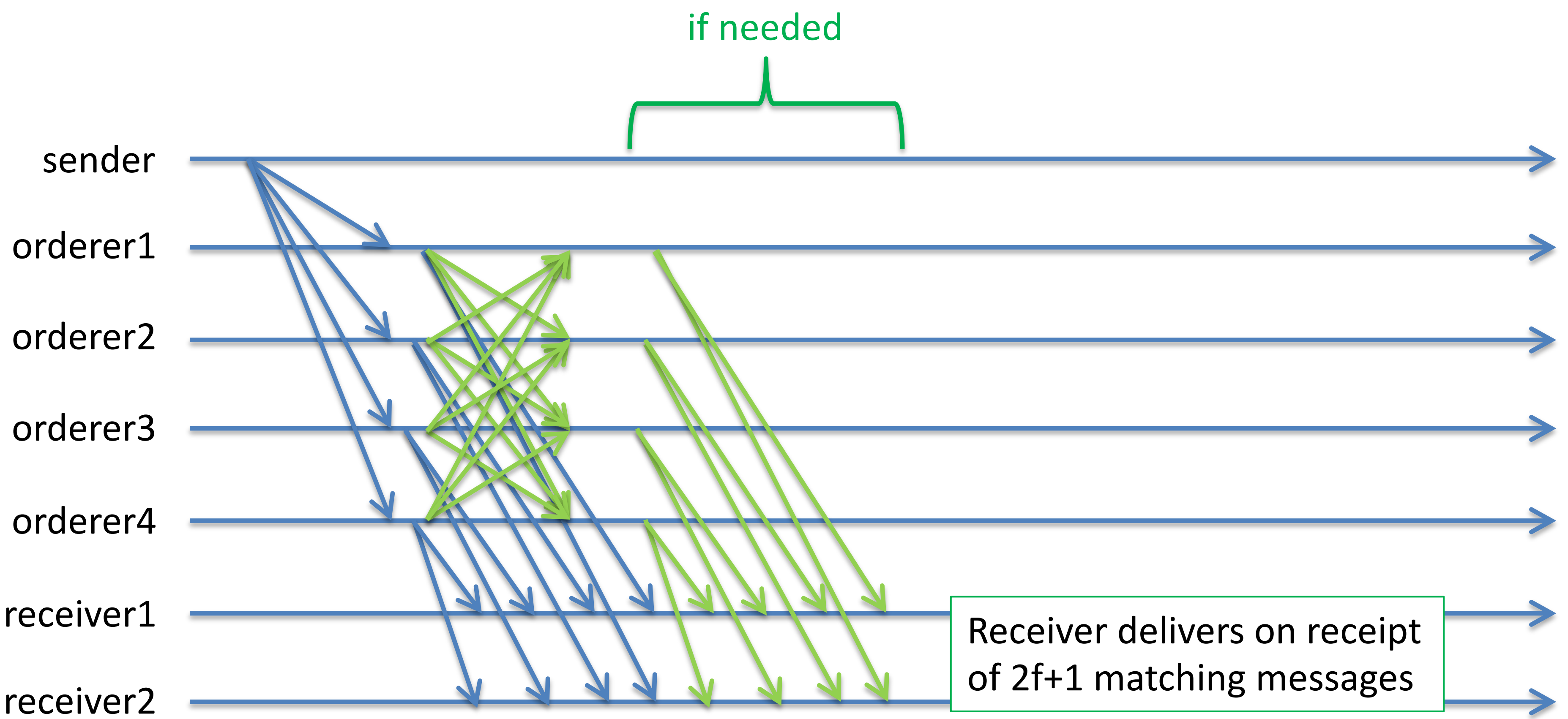
OARCAST Protocol

- All messages signed and contain sequence number
- 3*f*+1 *orderers,* check seq numbers and echo



OARCAST Protocol

- All messages signed and contain sequence number
- 3*f*+1 *orderers,* check seq numbers and echo



OARCAST Persistence

If sender is correct, all correct receivers will receive all its messages

- All correct orderers will receive the sender's messages in the correct order
- As there are at least 2f+1 correct orderers, all receivers will receive at least 2f+1 matching echoes for each of the sender's messages

OARCAST Relay

If one correct receiver delivers a message, all correct receivers will deliver the same message

- All correct orderers echo each other’s messages to one another, and then onto receivers
- If one correct receiver receivers 2f+1 matching echoes, all correct receivers receive 2f+1 matching echoes

OARCAST Authenticity

If sender and receiver are correct, and sender delivers a message, then the sender sent it

- All messages are signed, so receivers can reject any message not signed by sender

OARCAST FIFO

If sender is correct, correct receivers deliver messages in the order sent

- All messages contain a sequence number and are signed by the sender

OARCAST Order

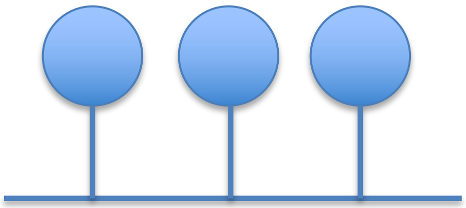
Correct receivers deliver messages in the same order

- By contradiction: suppose R1 delivers x before y, and R2 delivers y before x
- Then 2f+1 orderers must have echoed x, and 2f+1 orderers must have echoed y
- Since there are only 3f+1 orderers, f+1 orderers must have echoed both x and y
- At least one of these orderers must be correct
- Correct orderers check sequence numbers and don’t echo messages twice

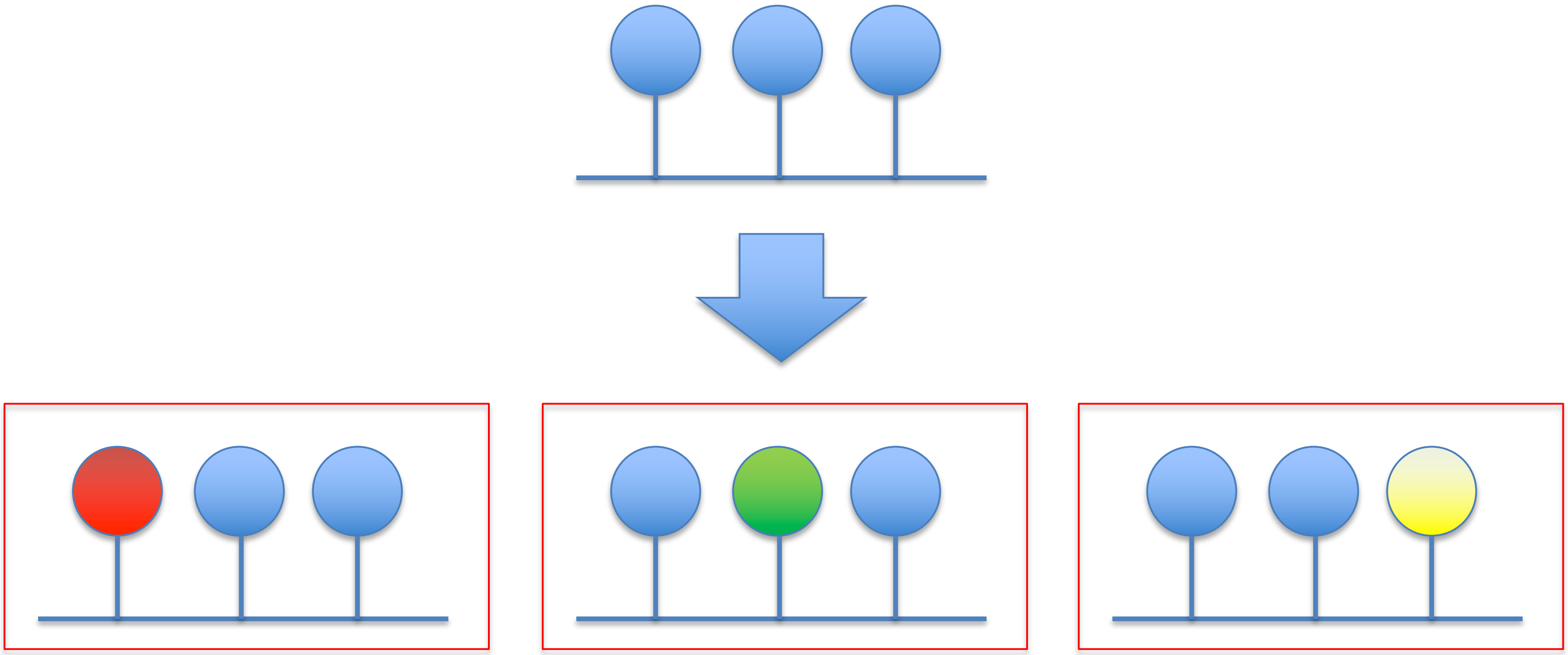
Translation

- Start with a crash tolerant protocol
  - *N* participants
- Create *N* copies of the protocol
- Run each copy on a single machine using a simulated network on the machine
- Keep the various copies in synch with one another
  - use *N* instantiations of OARCAST
  - each is used to order incoming messages to a participant
    - only payload needed is the source identifier of the message as message content is generated by the machine itself

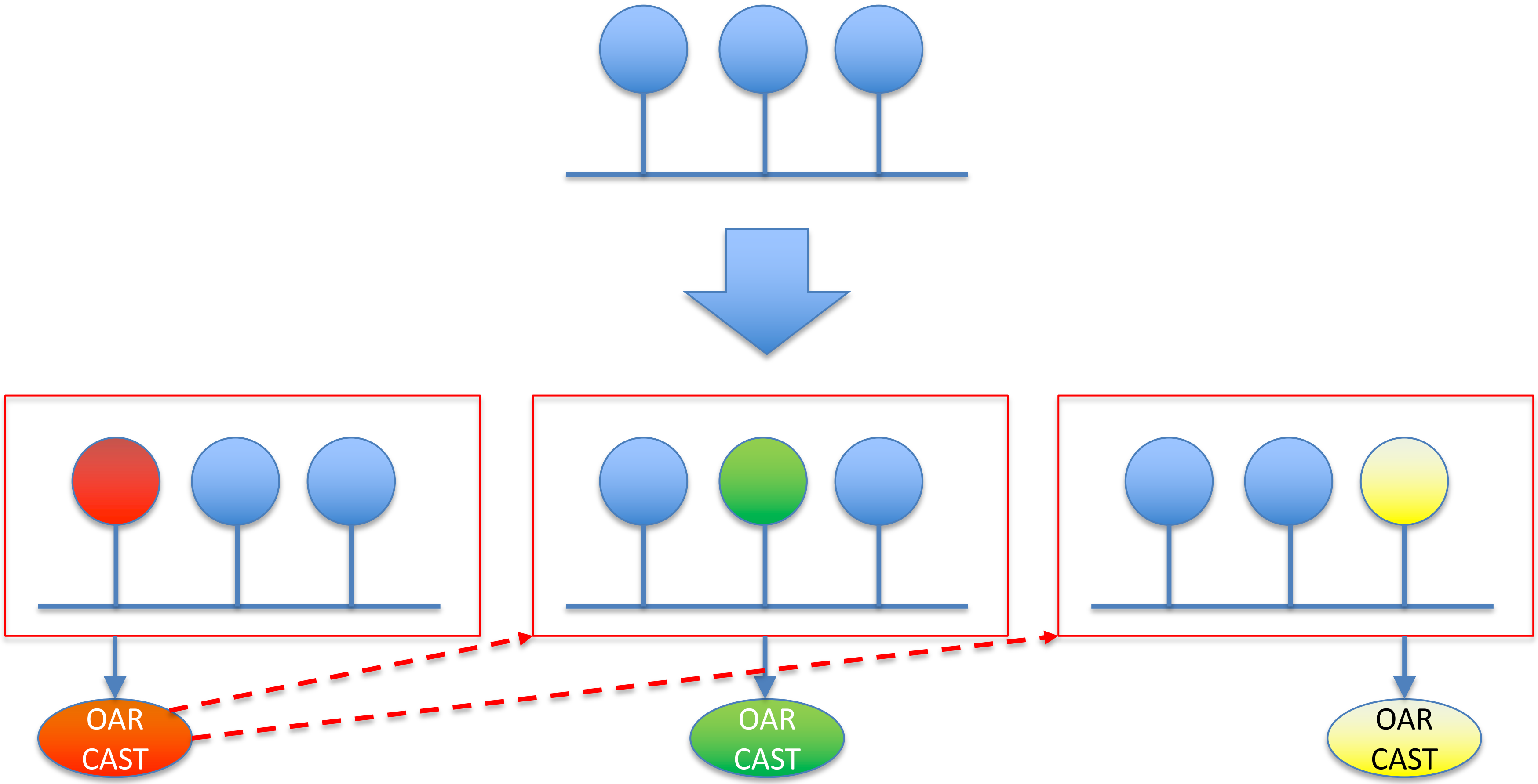
Example



Example



Example



Simulation within a machine

- Each machine simulates all participants
- One is the “coordinator” participant
- When the coordinator participant receives a simulated message from some peer p, the machine OARCASTs p to the other machines
  - other non-deterministic events must be OARCAST also
- Each machine delivers messages to each participant in the order it receives OARCASTs to that participant

Net Result

- Each correct machine delivers the same messages to the same (simulated) participants
- A Byzantine machine that is ”caught” acts like a crashed machine in the simulation

→ All correct machines run the same simulation

Dealing with output

- Byzantine machines can still generate bad output
- Output can be trusted if at least f+1 machines generate the same output