Chapter 5

Consensus II

Distributed Systems

SS 2019

Fabian Kuhn

Overview

- Introduction
- Consensus #1: Shared Memory
- Consensus #2: Wait-free Shared Memory
- Consensus #3: Read-Modify-Write Shared Memory
- Consensus #4: Synchronous Systems
- Consensus #5: Byzantine Failures
- Consensus #6: A Simple Algorithm for Byzantine Agreement
- Consensus #7: The Queen Algorithm
- Consensus #8: The King Algorithm
- Consensus #9: Byzantine Agreement Using Authentication
- Consensus #10: A Randomized Algorithm
- Shared Coin

Consensus More Formally

Setting:

- n processes/threads/nodes $v_1, v_2, \dots, v_n$
- Each process has an input $x_1, x_2, \dots, x_n \in \mathcal{D}$
- Each (non-failing) process computes an output $y_1, y_2, \dots, y_n \in \mathcal{D}$

Agreement:

The outputs of all non-failing processes are equal.

Validity:

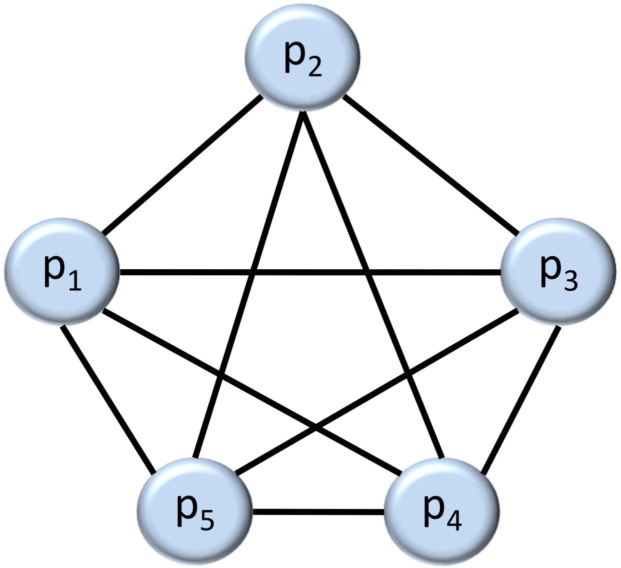
If all inputs are equal to x, all outputs are equal to x.

Termination:

All non-failing processes terminate after a finite number of steps.

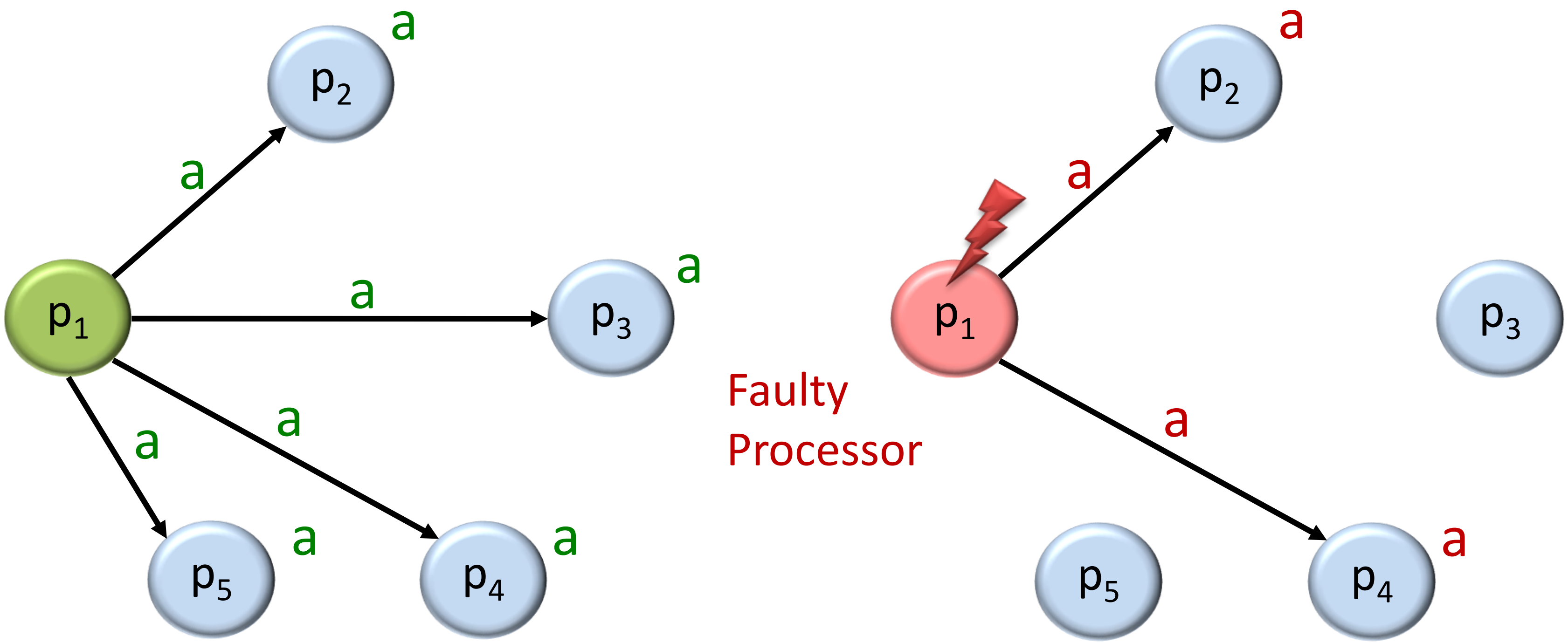
Consensus #4: Synchronous Systems

- One can sometimes tell if a processor had crashed
  - Timeouts
  - Broken TCP connections
- Can one solve consensus at least in synchronous systems?
- Model
  - All communication occurs in synchronous rounds
  - Complete communication graph

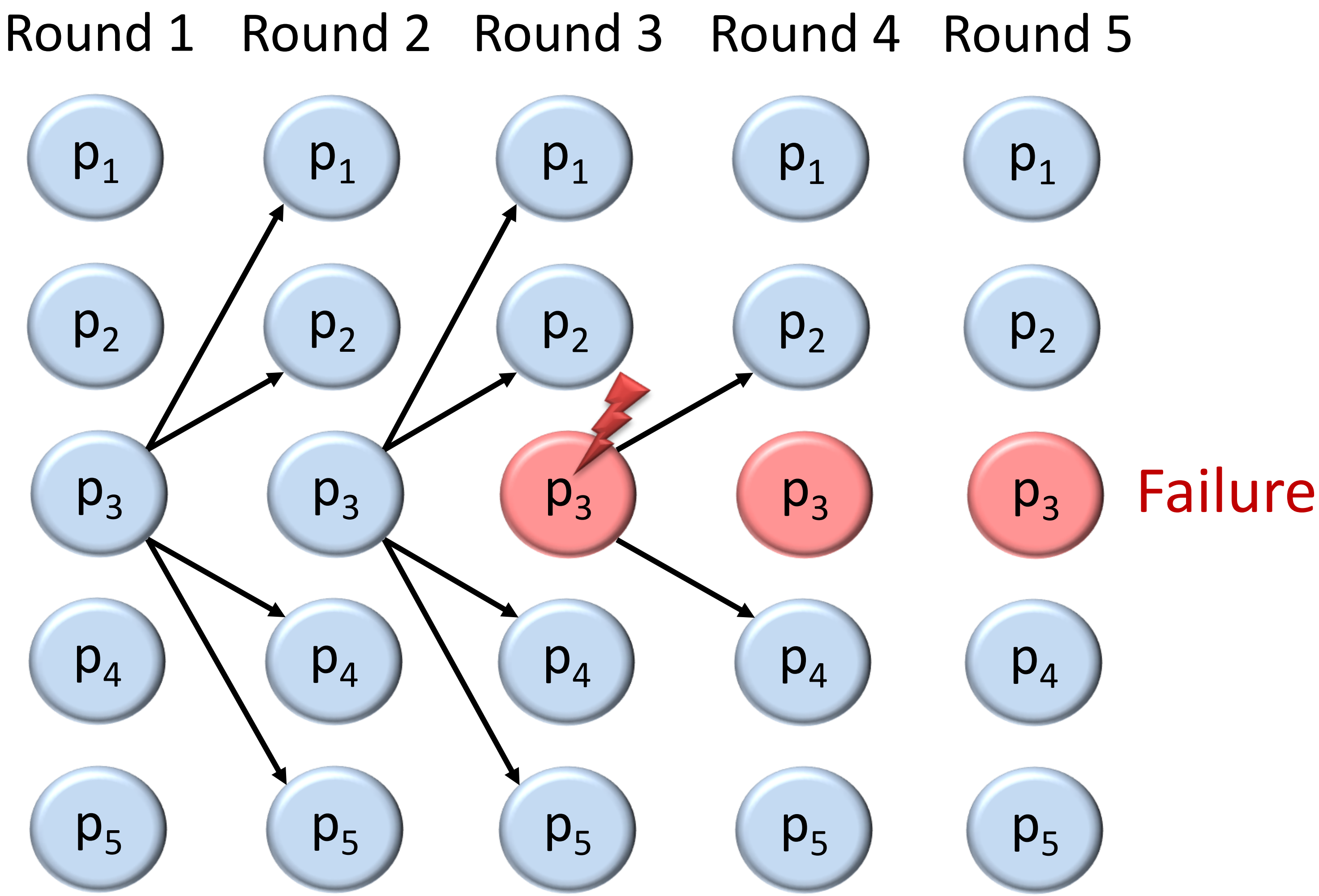


Crash Failures

- Broadcast: Send a message to all nodes in one round
  - At the end of the round everybody receives the message a
  - Every process can broadcast a value in each round
- Crash Failures: A broadcast can fail if a process crashes
  - Some of the messages may be lost, i.e., they are never received



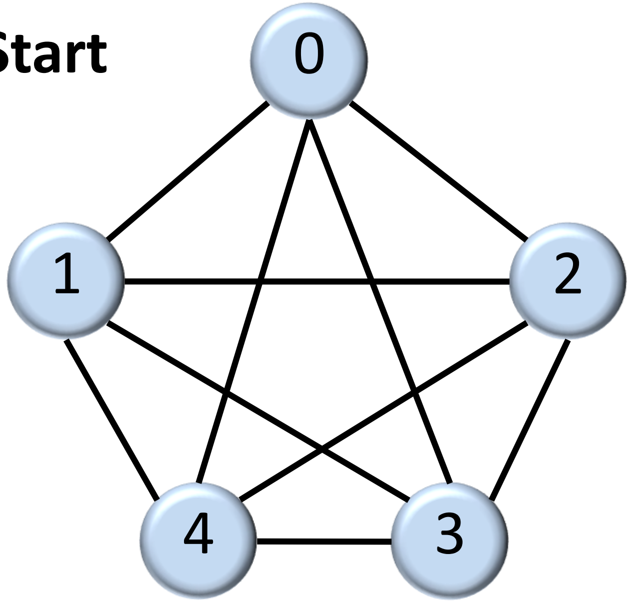
Process disappears after failure



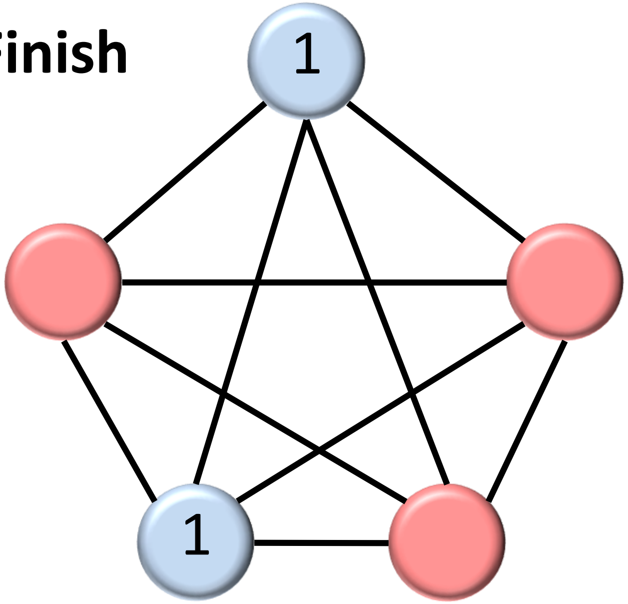
f-Resilient Consensus Algorithm

- If an algorithm solves consensus for f failed processes, we say it is an f-resilient consensus algorithm
- Example: The input and output of a 3-resilient consensus alg.

Start



Finish



- **Refined validity condition**:
  All processes decide on a value that is available initially

An f-Resilient Consensus Algorithm

Each process:

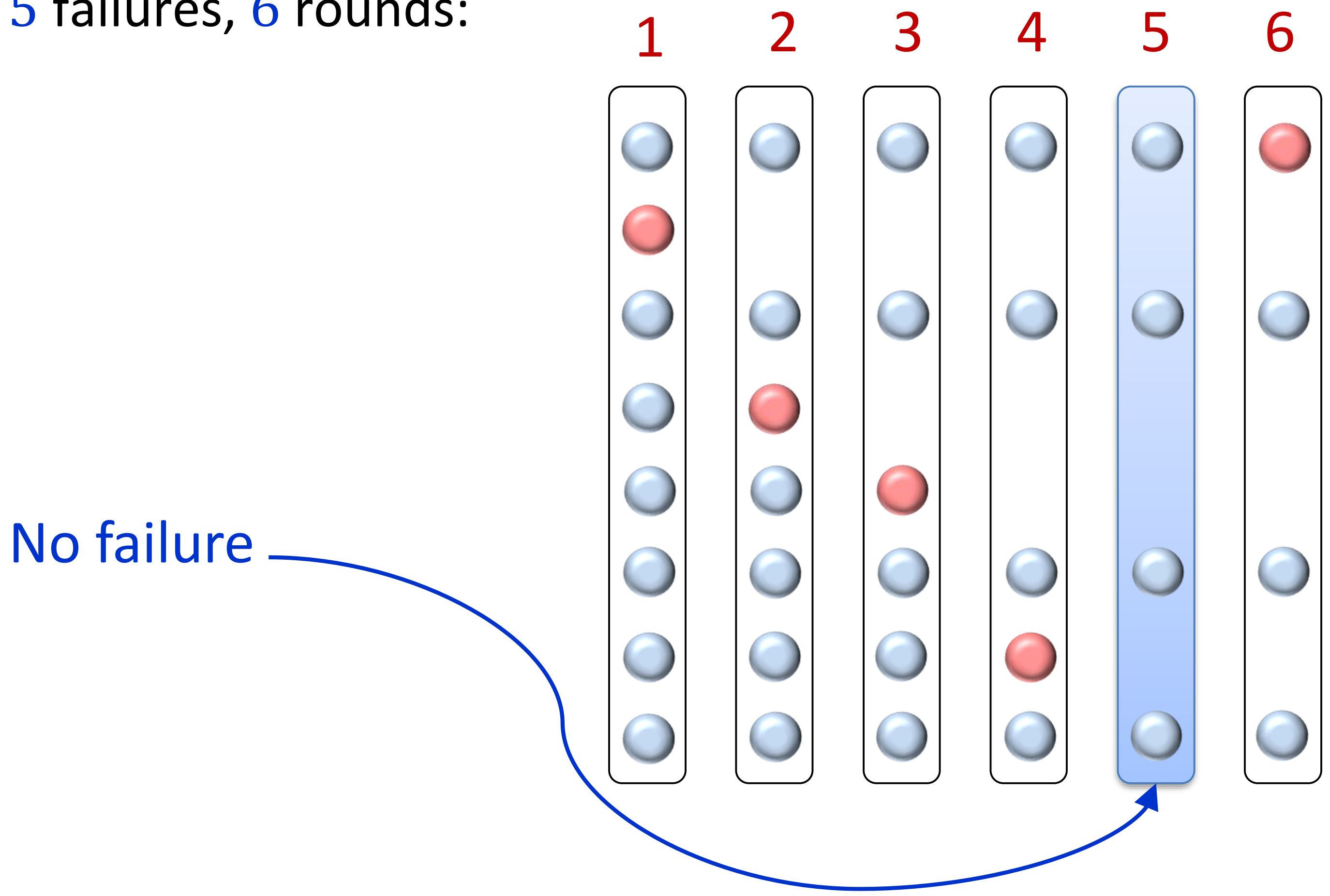
Round 1:
Broadcast own value

Round 2 to round $f + 1$:
Broadcast the minimum of the received values unless it has been sent before

End of round $f + 1$:
Decide on the minimum value received

Analysis

- If there are f failures and $f+1$ rounds, then there is a round with no failed process
- Example: 5 failures, 6 rounds:



Analysis

- At the end of the round with no failure
  - Every (non faulty) process knows about all the values of all the other participating processes
  - This knowledge doesn't change until the end of the algorithm
- Therefore, everybody will decide on the same value
- However, as we don't know the exact position of this round, we have to let the algorithm execute for $f + 1$ rounds

- **Validity:** When all processes start with the same input value, then consensus is that value

Theorem

Theorem

If at most $f \leq n-2$ of n nodes of a synchronous message passing system can crash, at least $f+1$ rounds are needed to solve consensus.

Proof idea:

- Show that f rounds are not enough if $n \geq f+2$
- Before proving the theorem, we consider a

  "worst-case scenario": In each round one of the processes fails

Lower Bound on Rounds: Proof

Recall from earlier in the course:

- For the impossibility proof of the two generals problem, we used an indistinguishability proof
- Execution E is indistinguishable from execution E' for some node v if v sees the same things in both executions.
  - same inputs and messages (schedule)
- If E is indistinguishable from E' for v, then v does the same thing in both executions.
  - We denoted this by $E|v = E'|v$

Similarity:

- Call E_i and E_j **similar** if $E_i|v = E_j|v$ for some node v

$$E_i \sim_v E_j \;\Leftrightarrow\; E_i|v = E_j|v$$

Lower Bound on Rounds: Proof

Similarity Chain:

- Consider a sequence of executions $E_1, E_2, E_3, \dots, E_T$ such that

$$\boldsymbol{\forall i \geq 1 \ : \quad E_i \sim_{v_i} E_{i+1}}$$

  – any two consecutive executions E_i and E_{i+1} are indistinguishable for some node v_i (we assume that v_i does not crash in E_i and E_{i+1})

- **Indistinguishability:**
  $\forall i \geq 1 \ :$ Node v_i decides on the same value in E_i and E_{i+1}

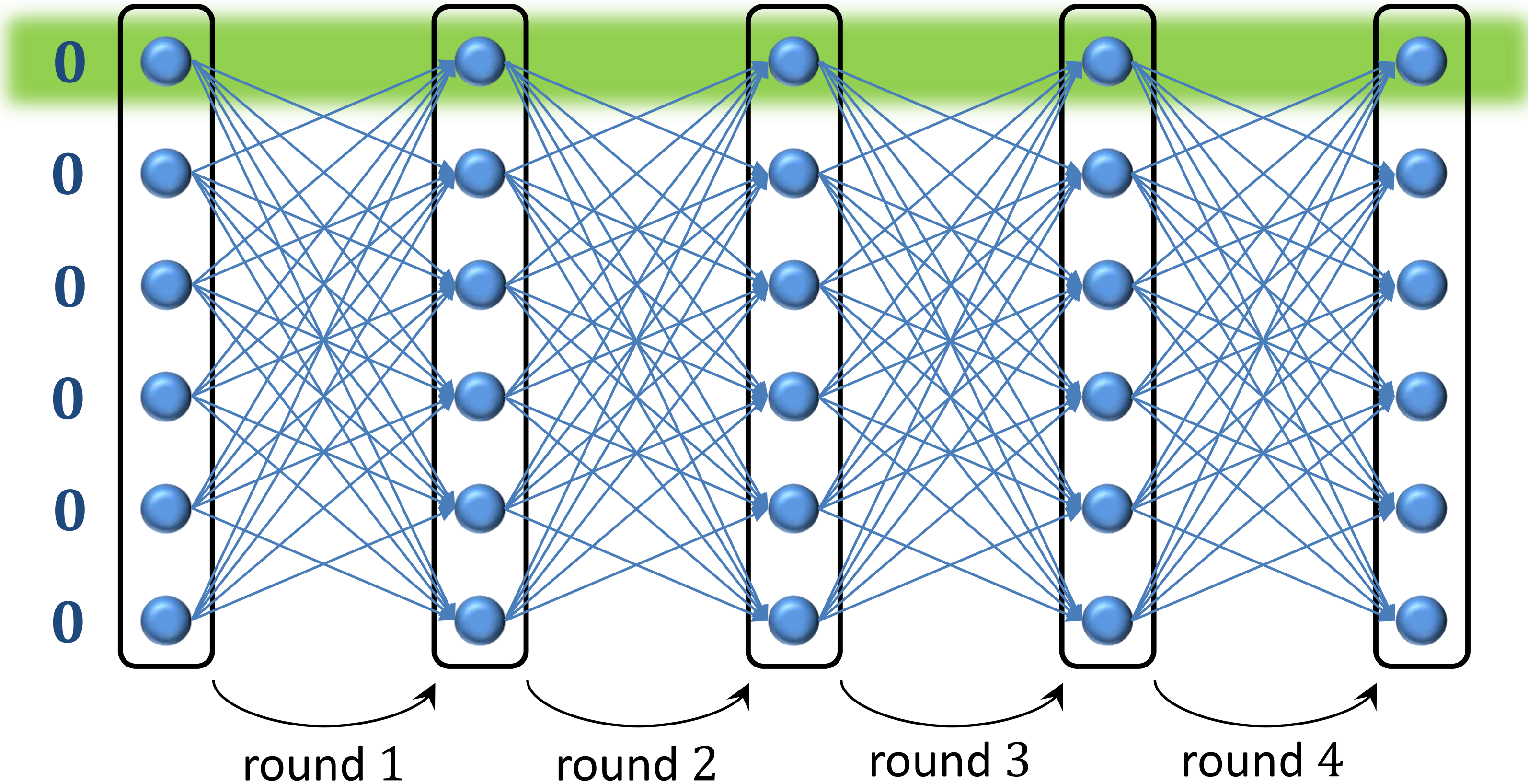
- **Agreement:**
  $\forall i \geq 1 \ :$ All nodes decide on the same value in E_i and E_{i+1}

- Hence, all executions $E_1, \dots, E_T$ have the same decision value!

- **Goal:**
  E_1: no crashes, all inputs are 0; E_T: no crashes, all inputs are 1

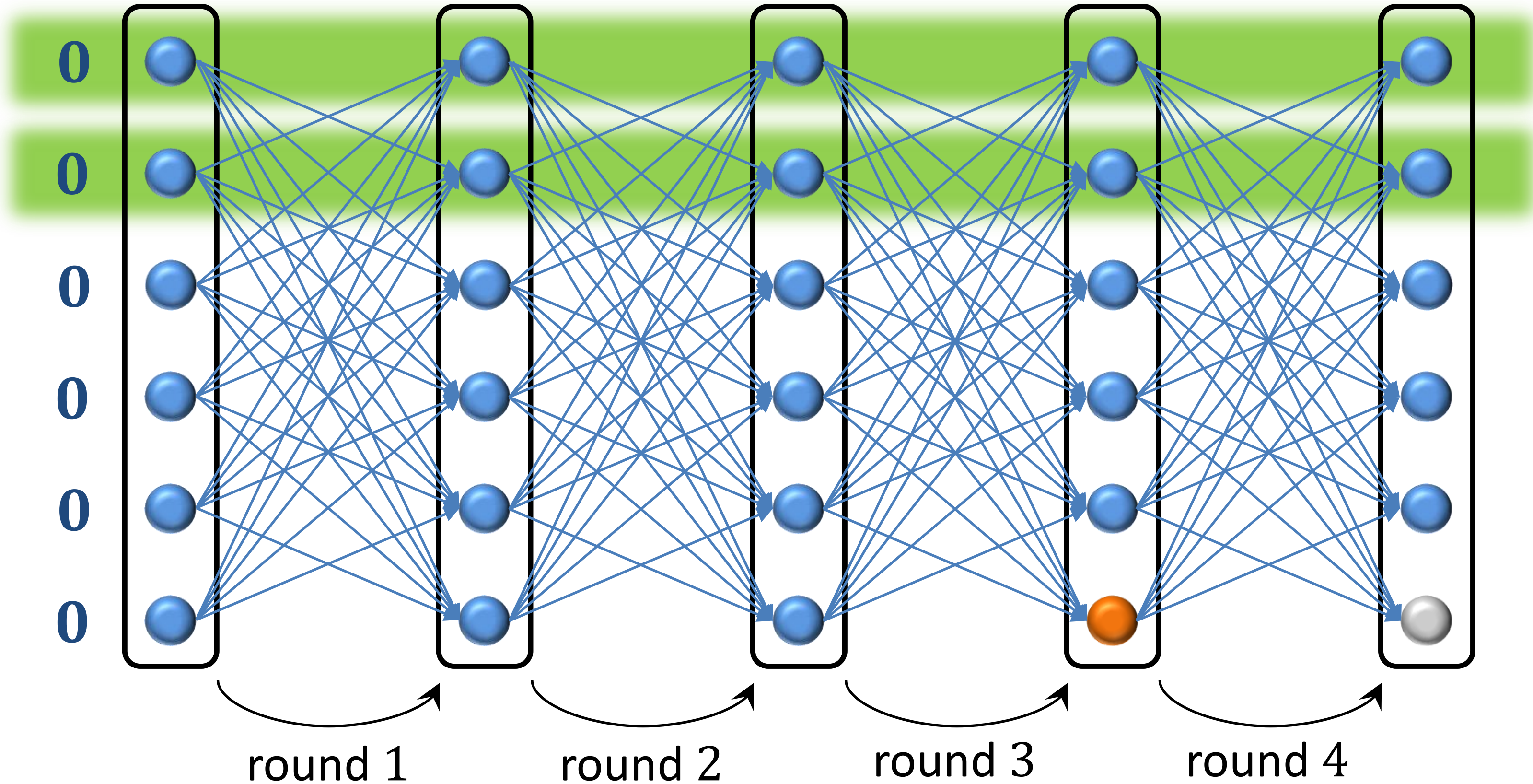
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



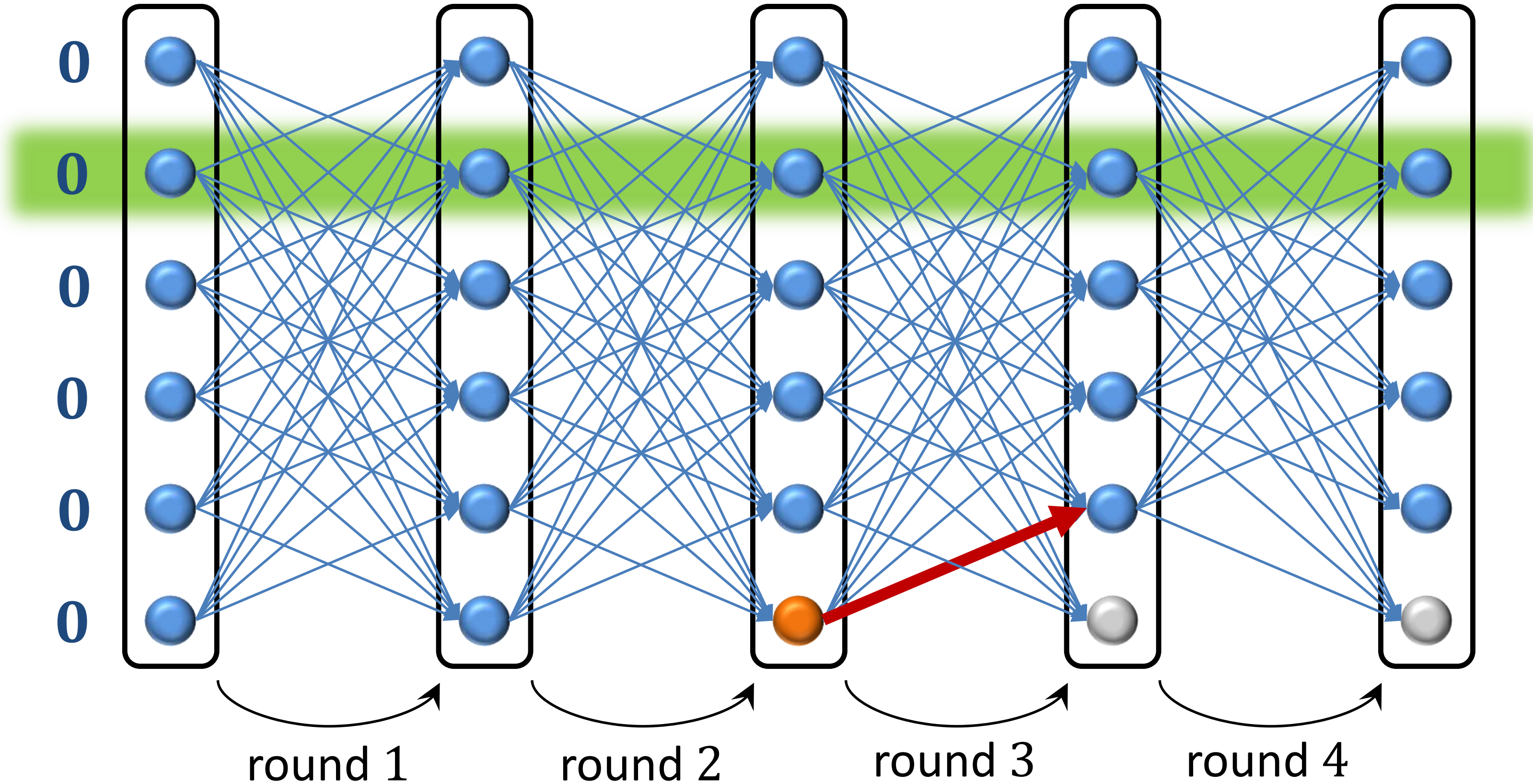
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



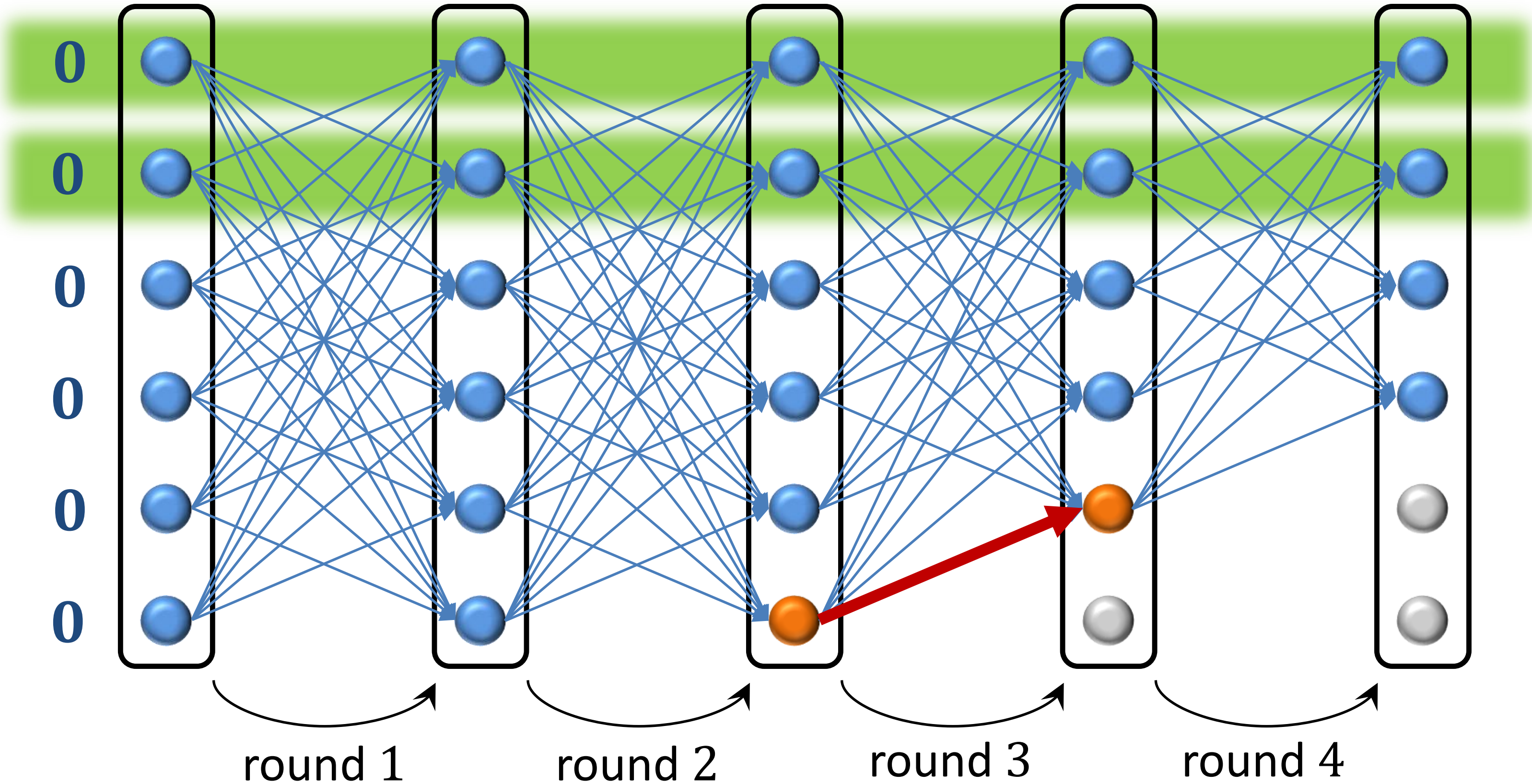
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



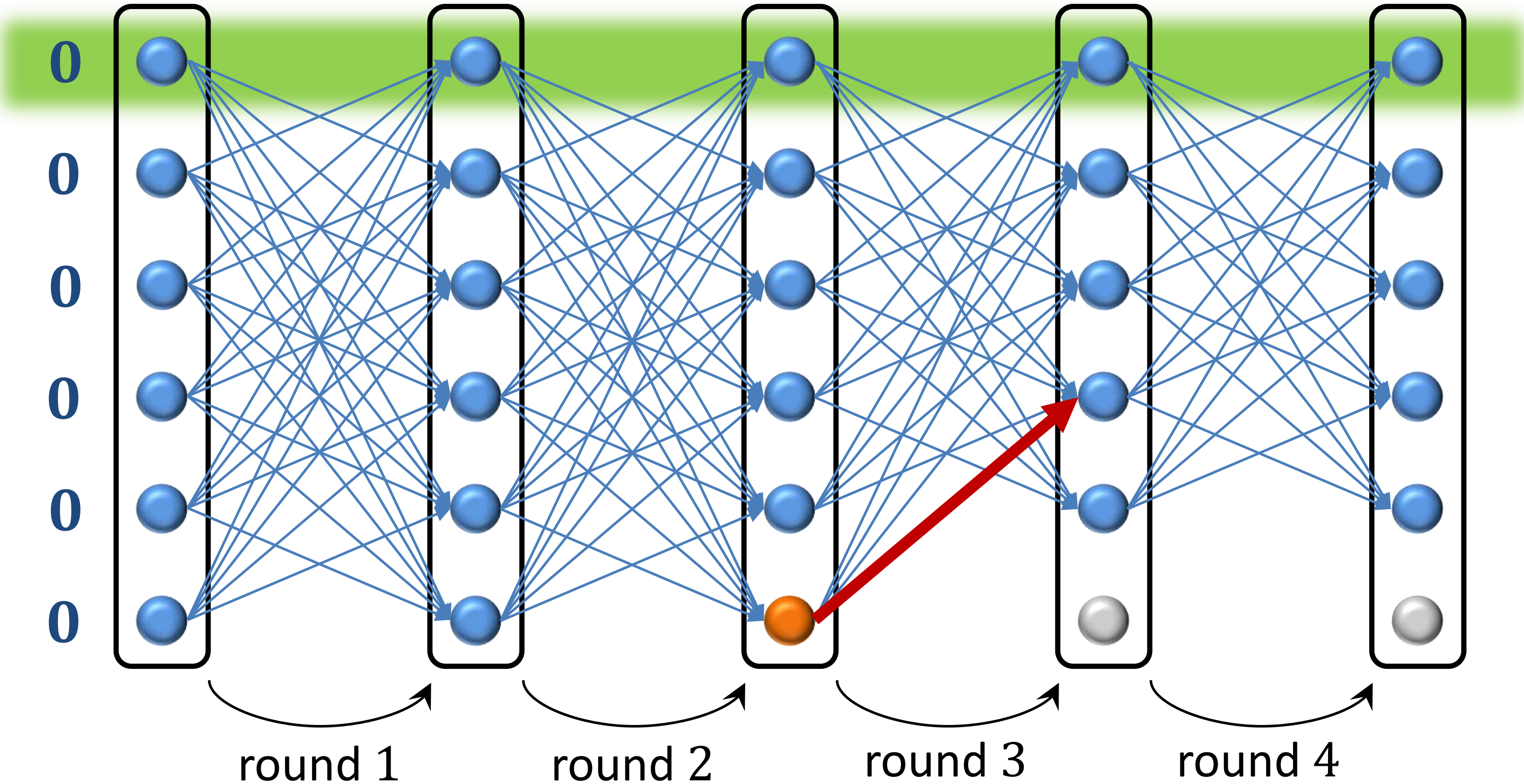
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



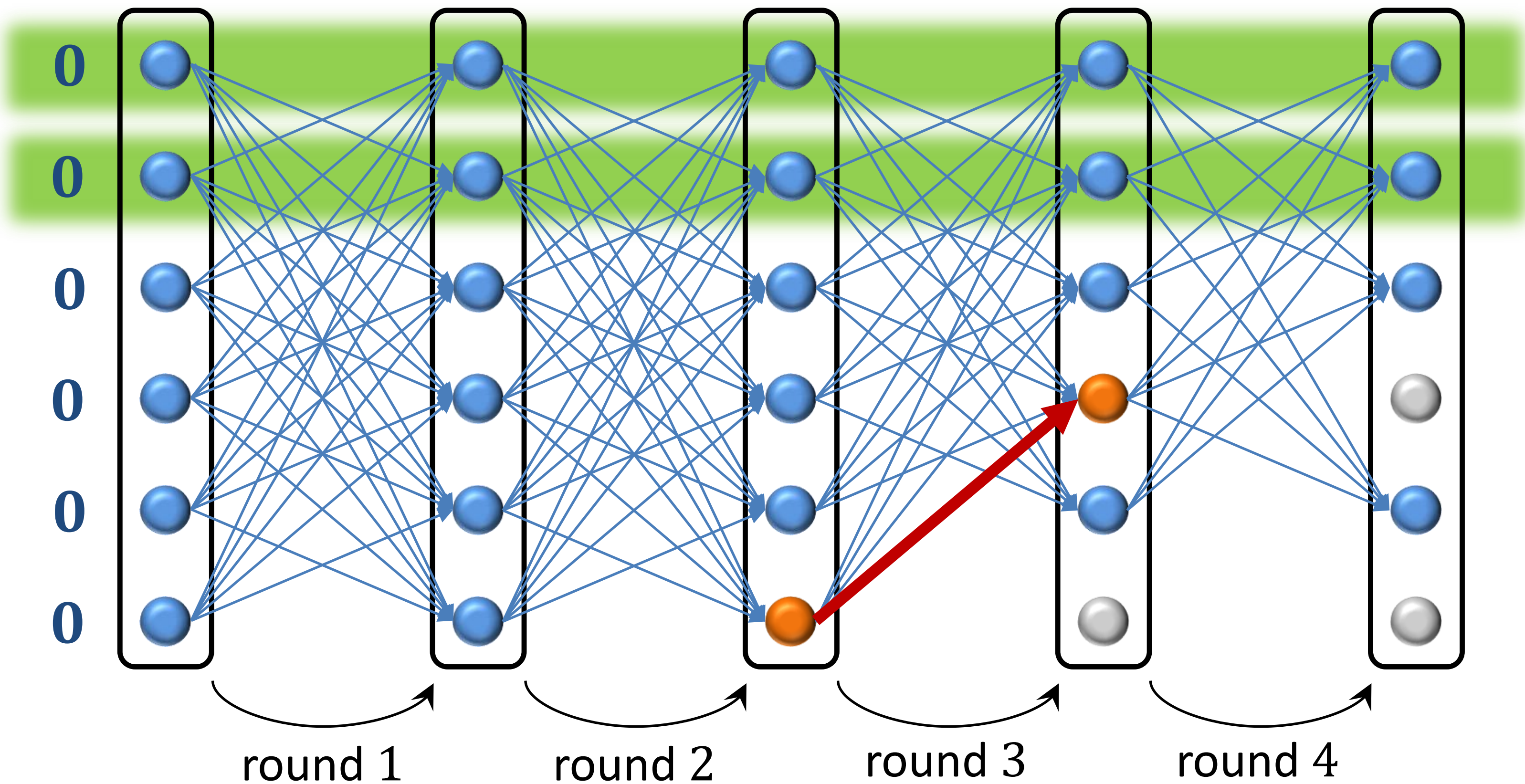
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



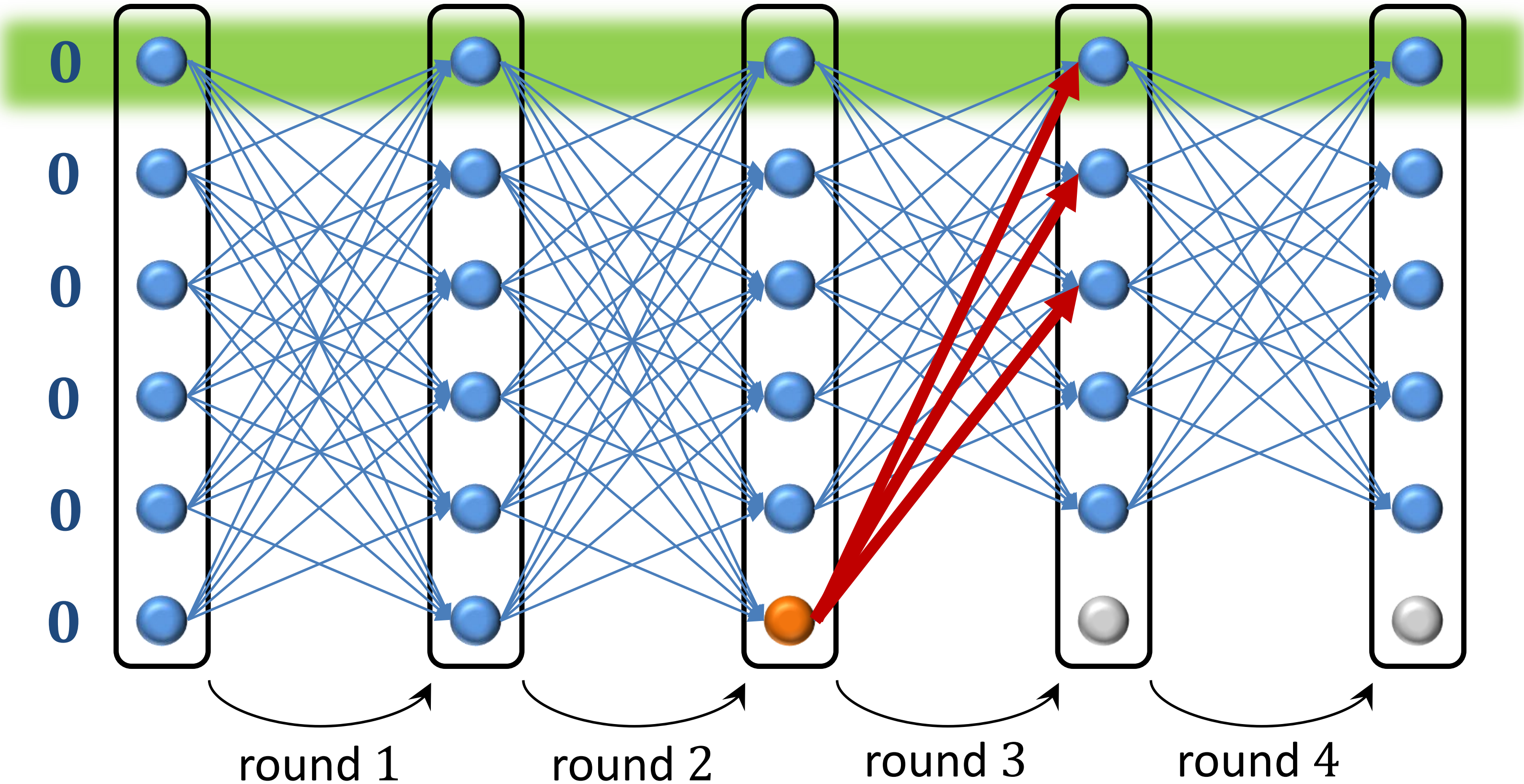
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



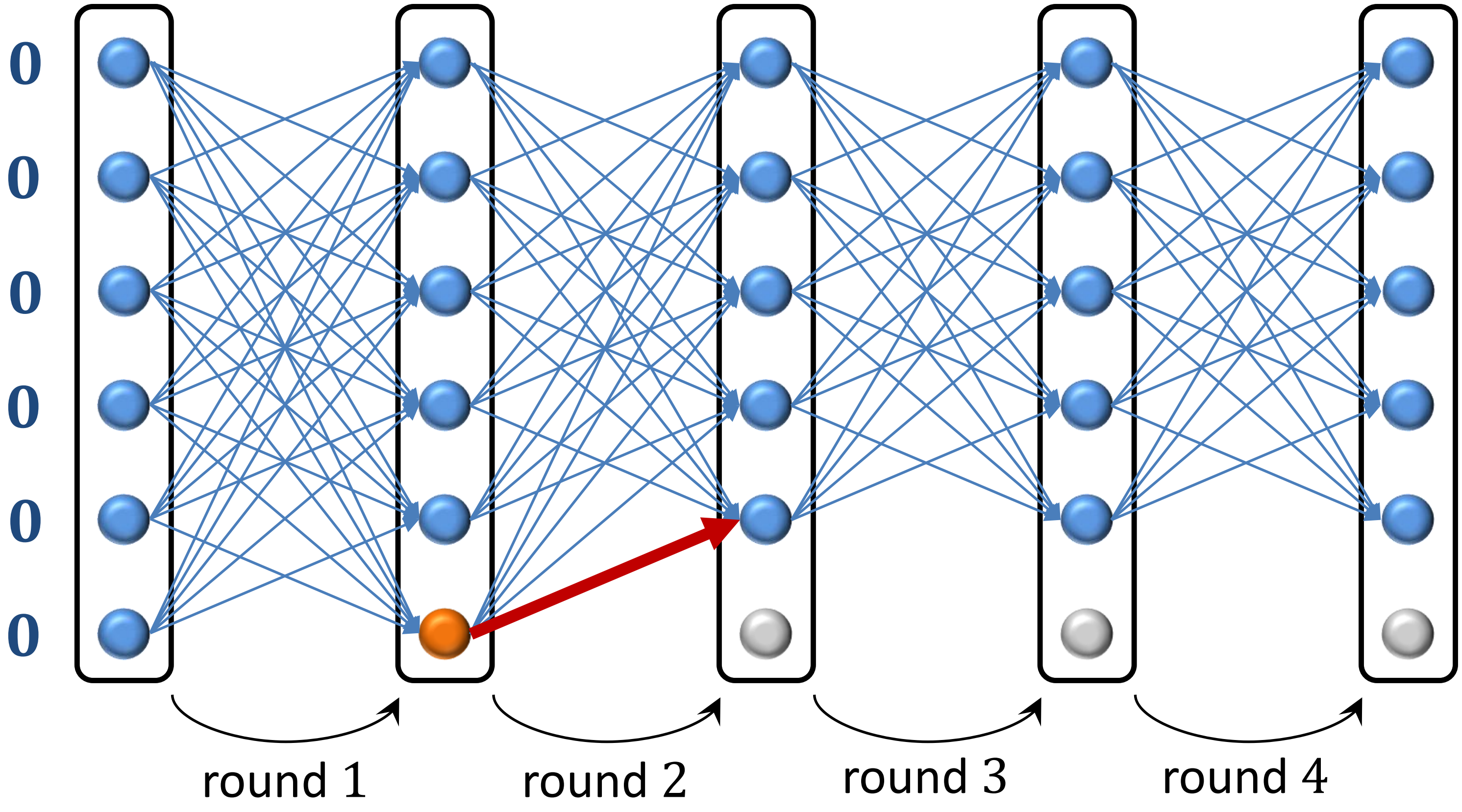
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



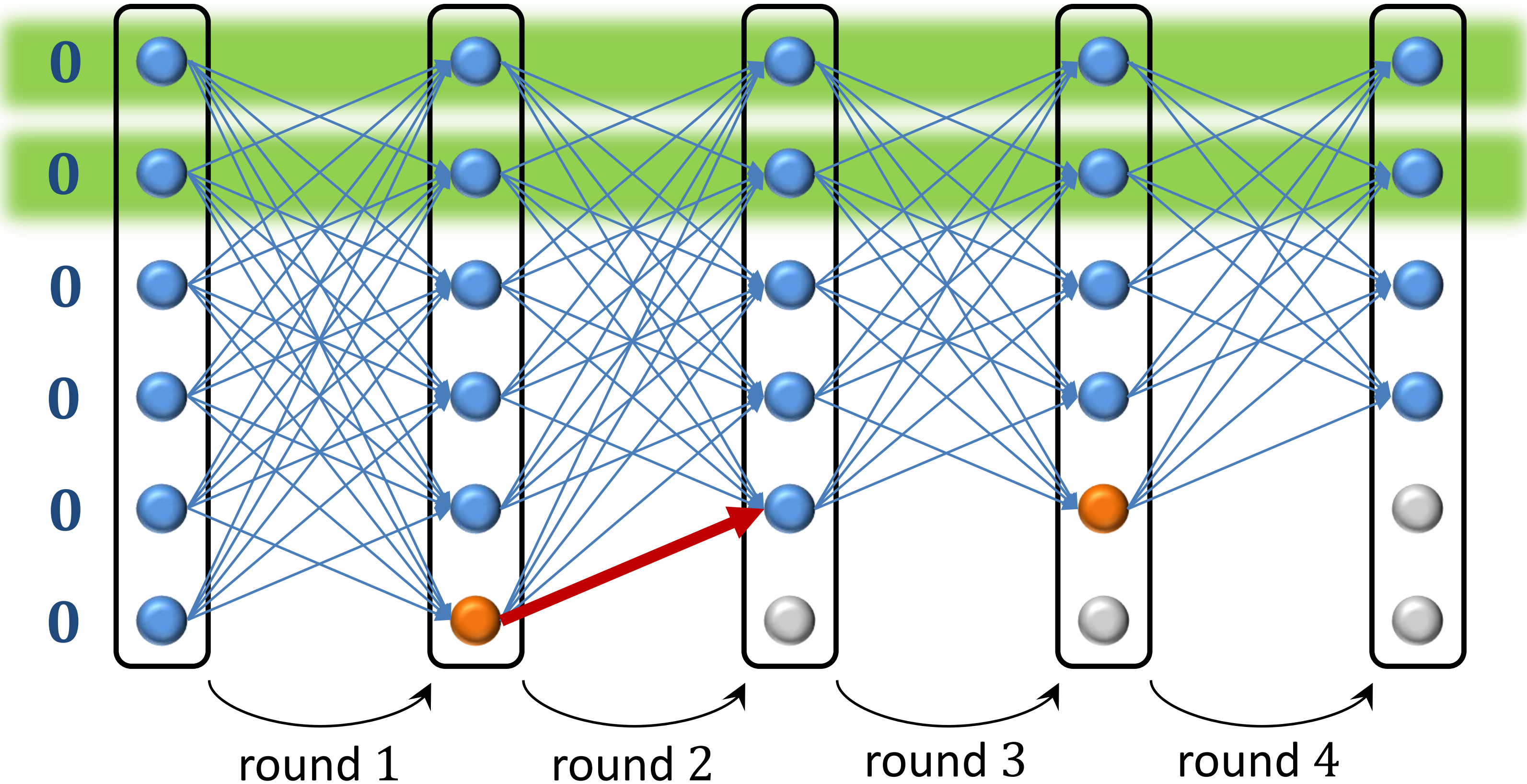
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



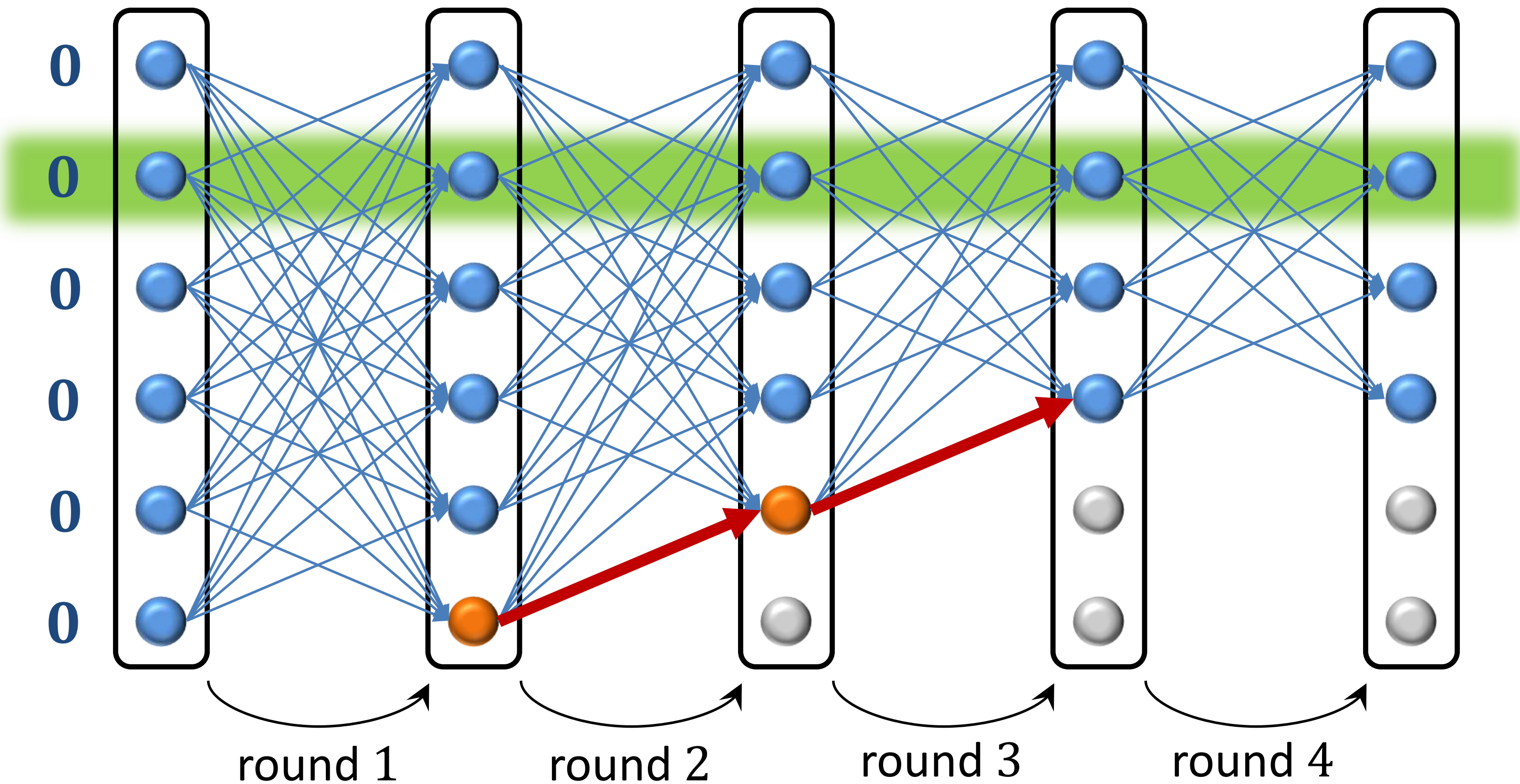
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



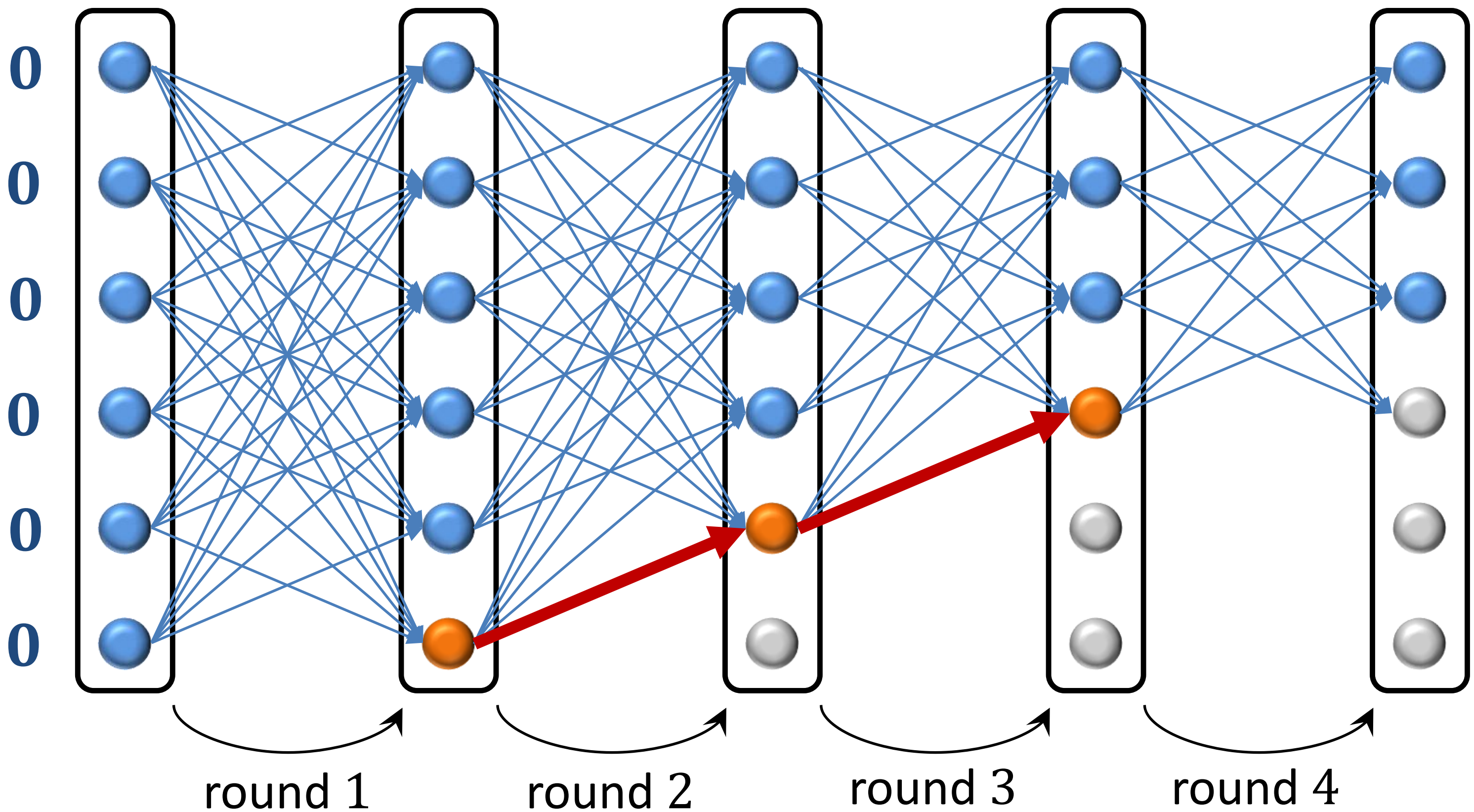
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



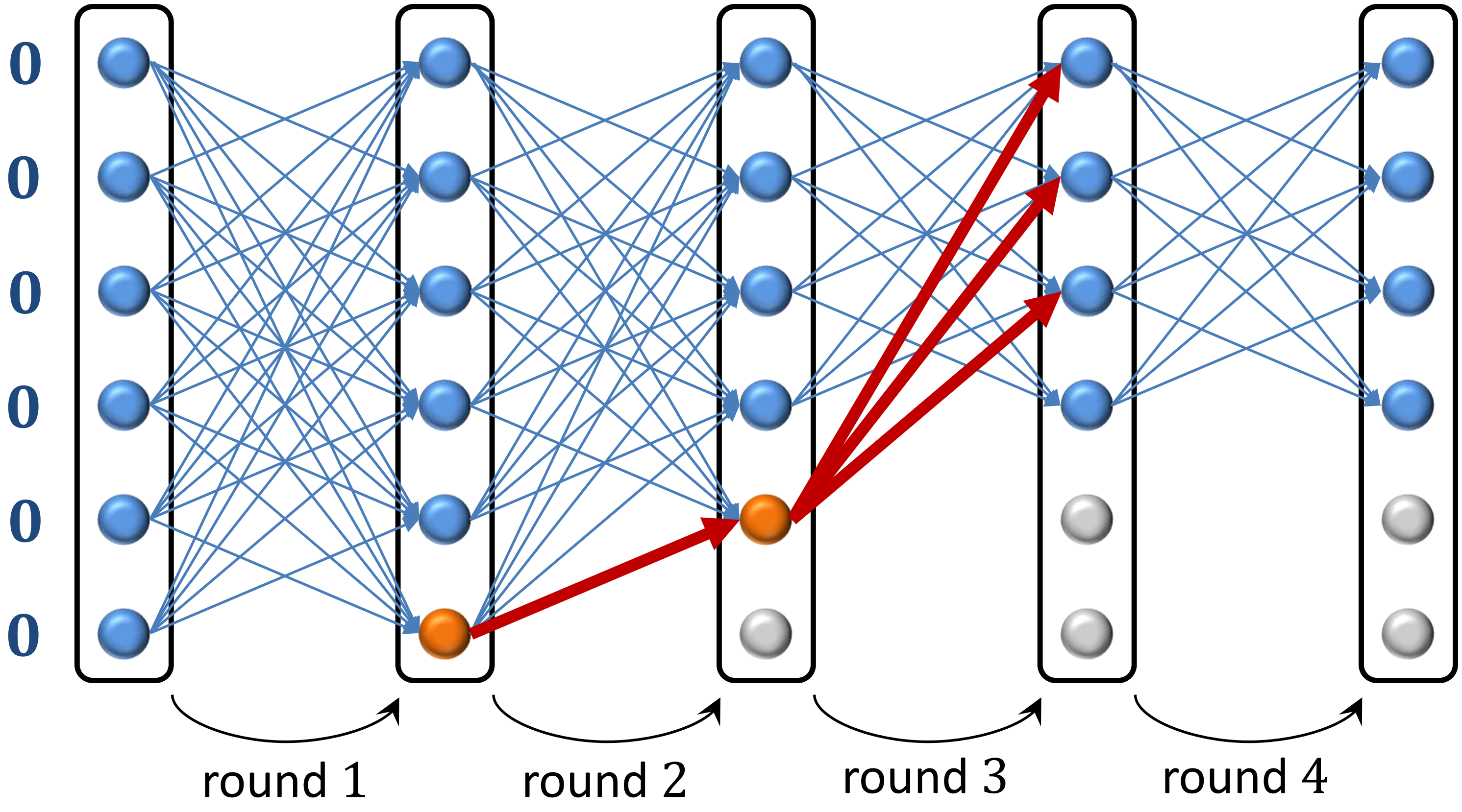
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ Need to show: 4 rounds are not enough



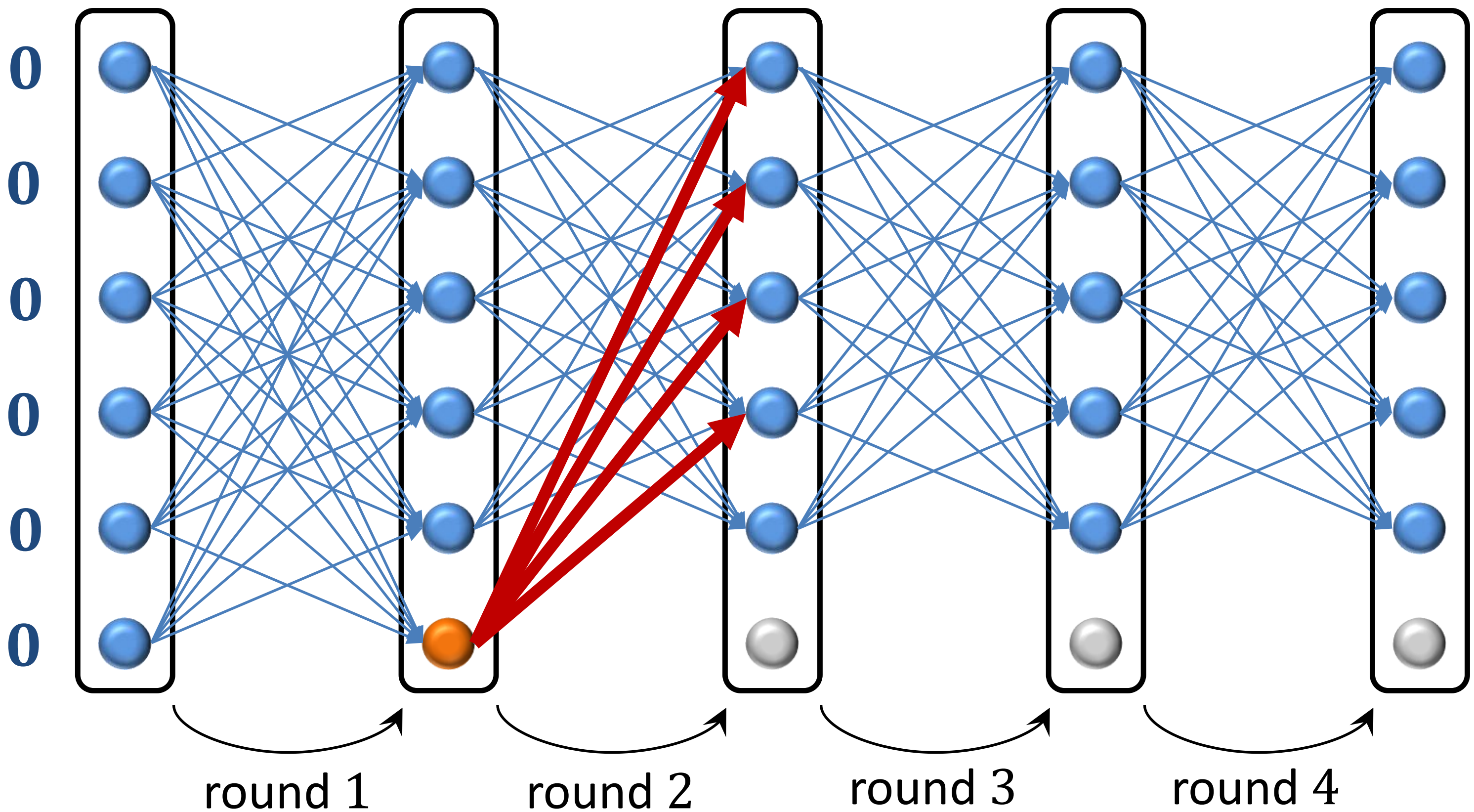
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



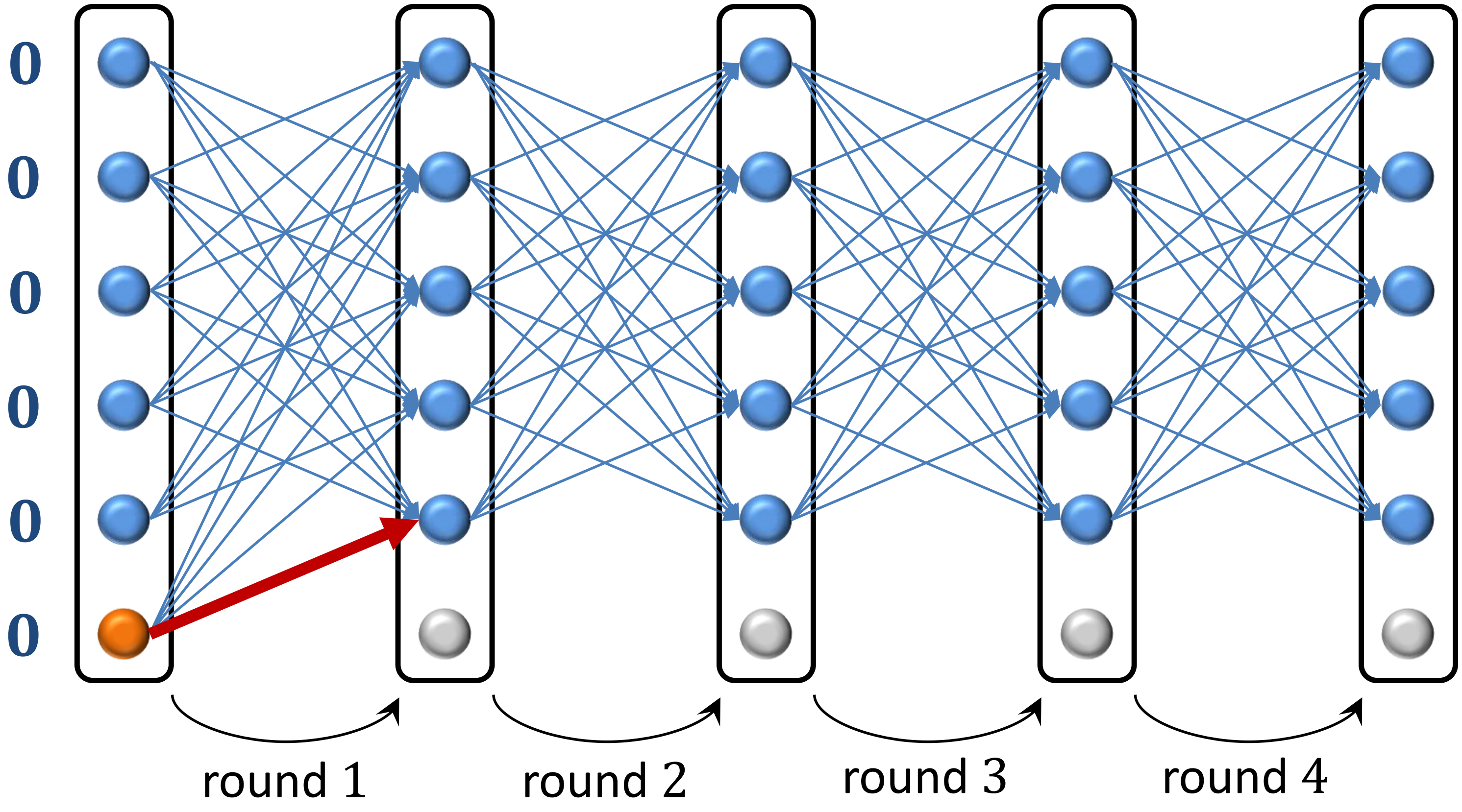
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



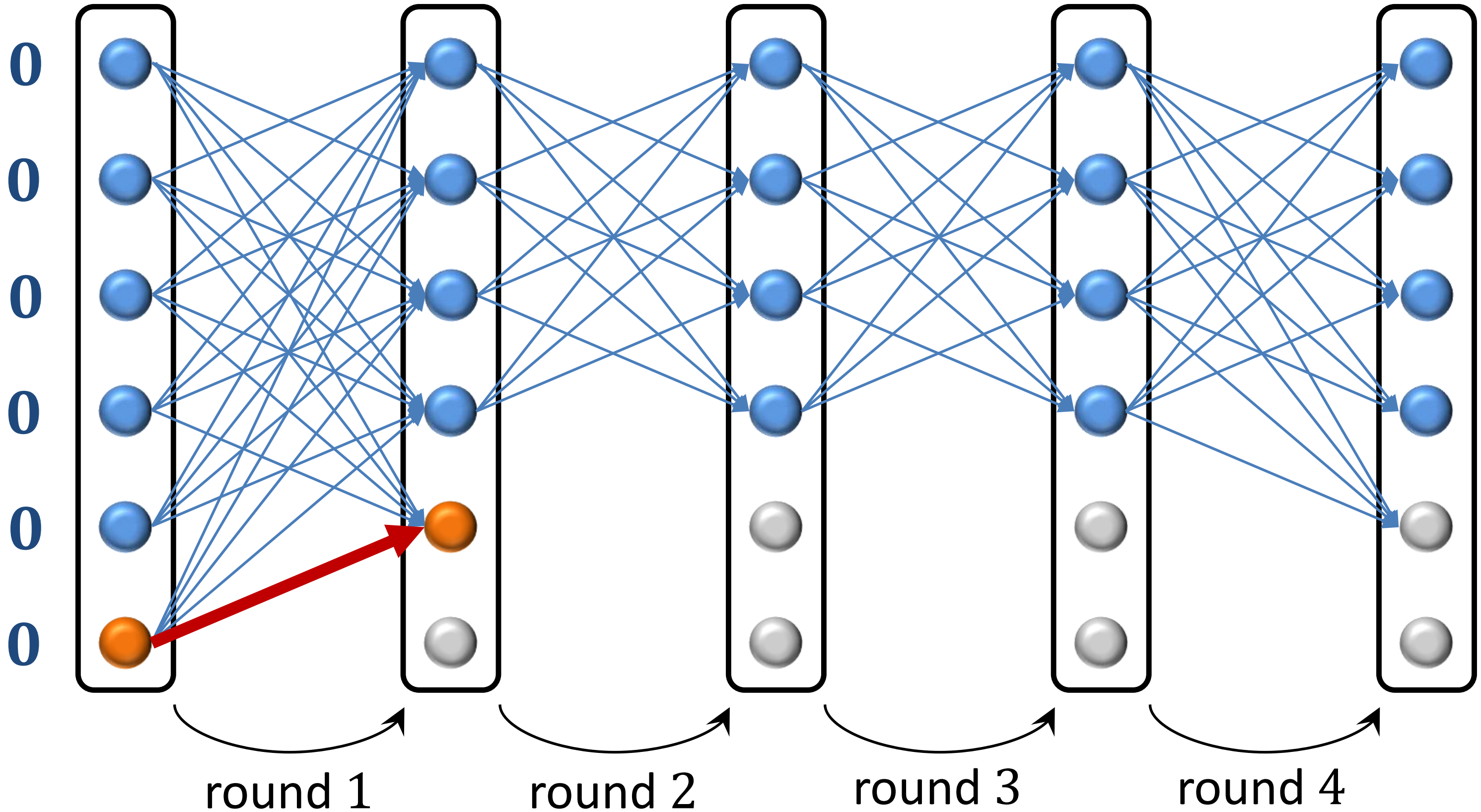
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



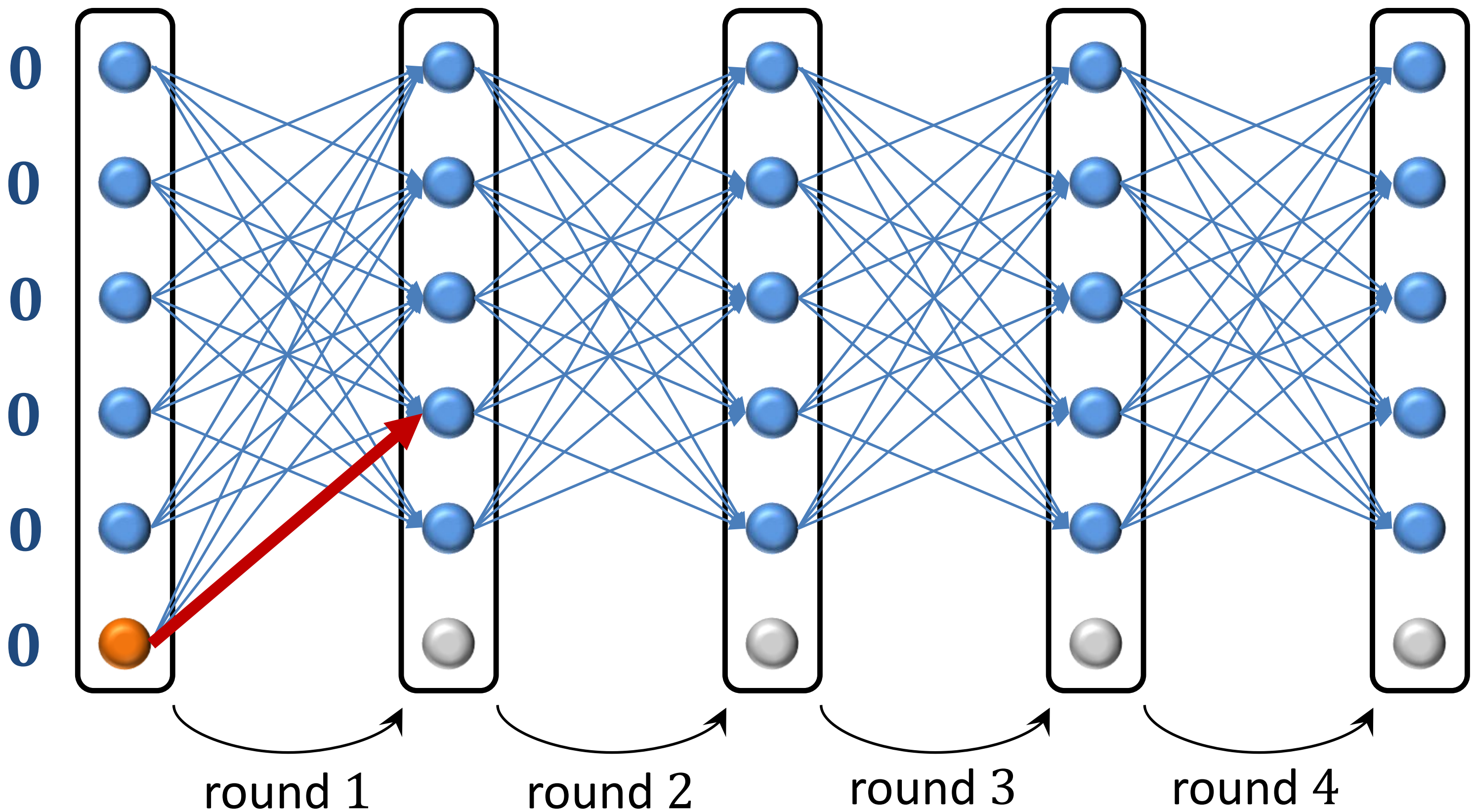
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



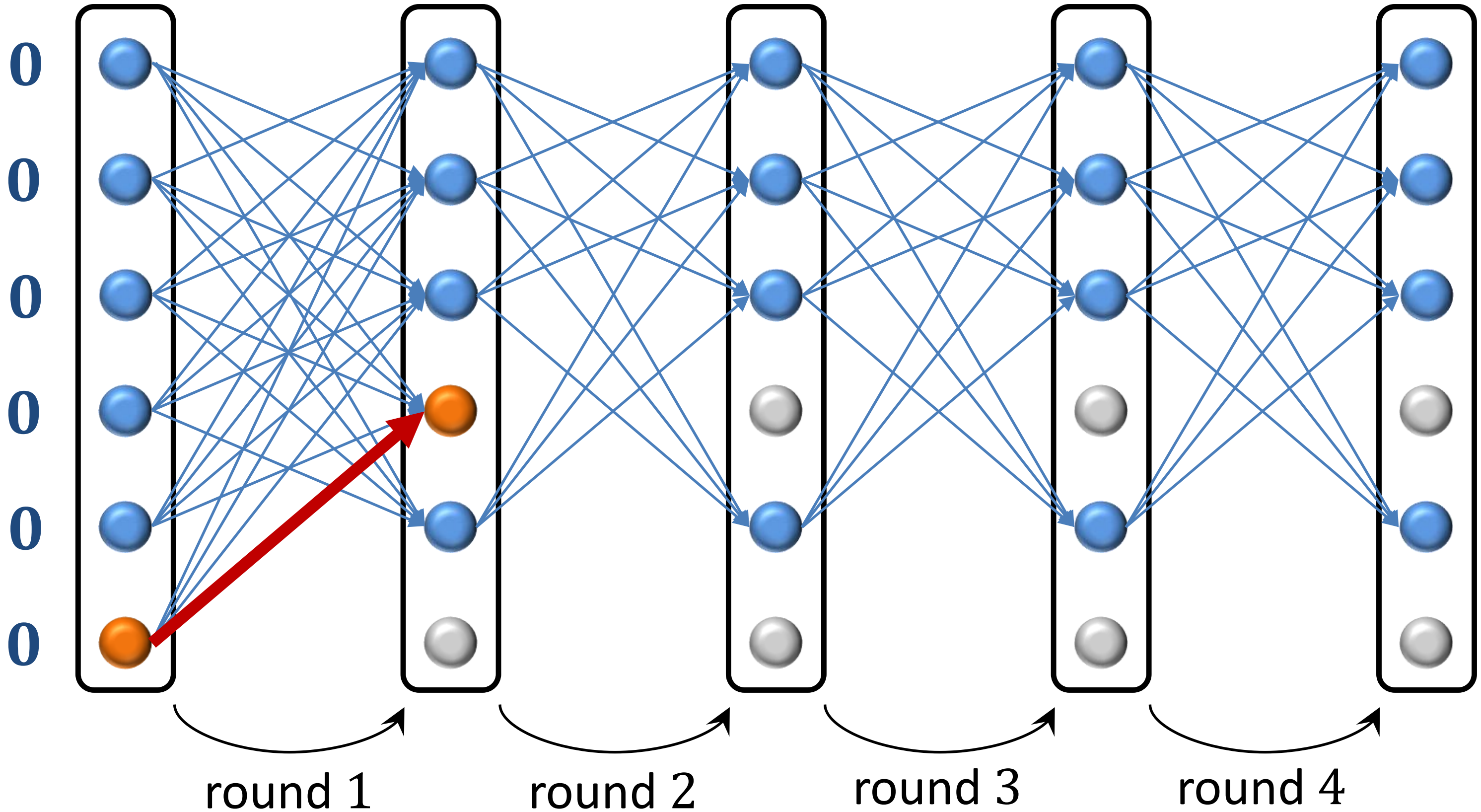
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



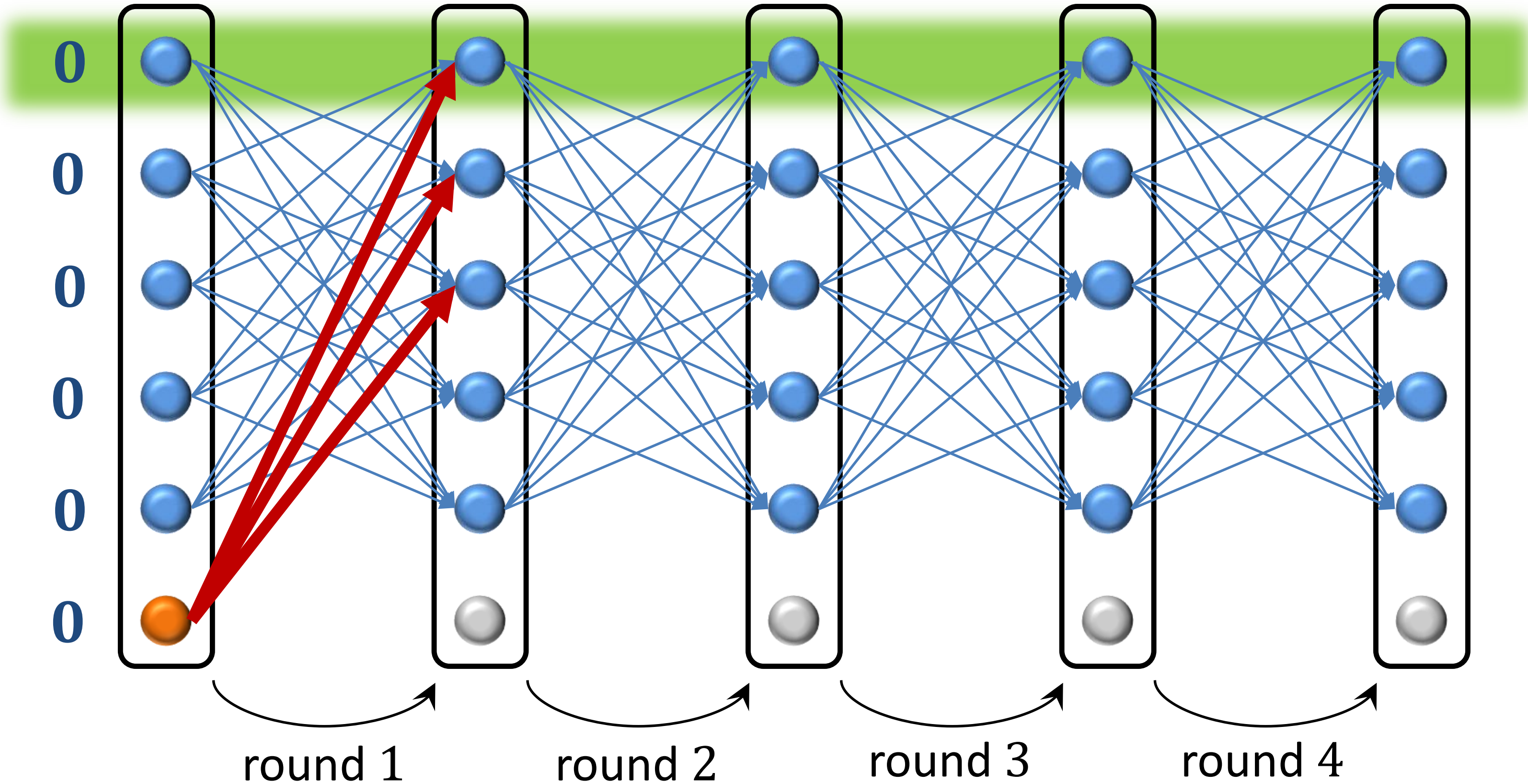
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



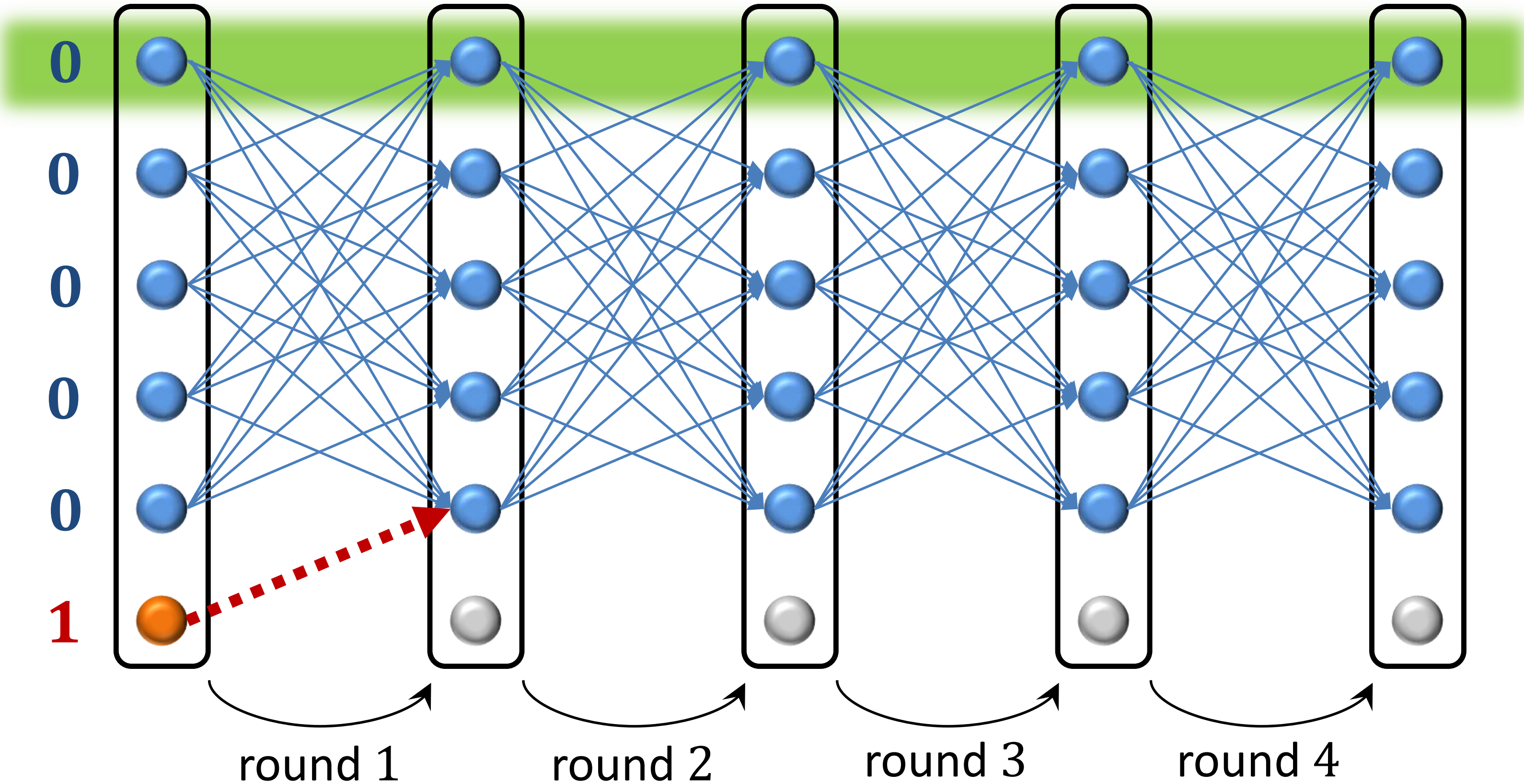
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



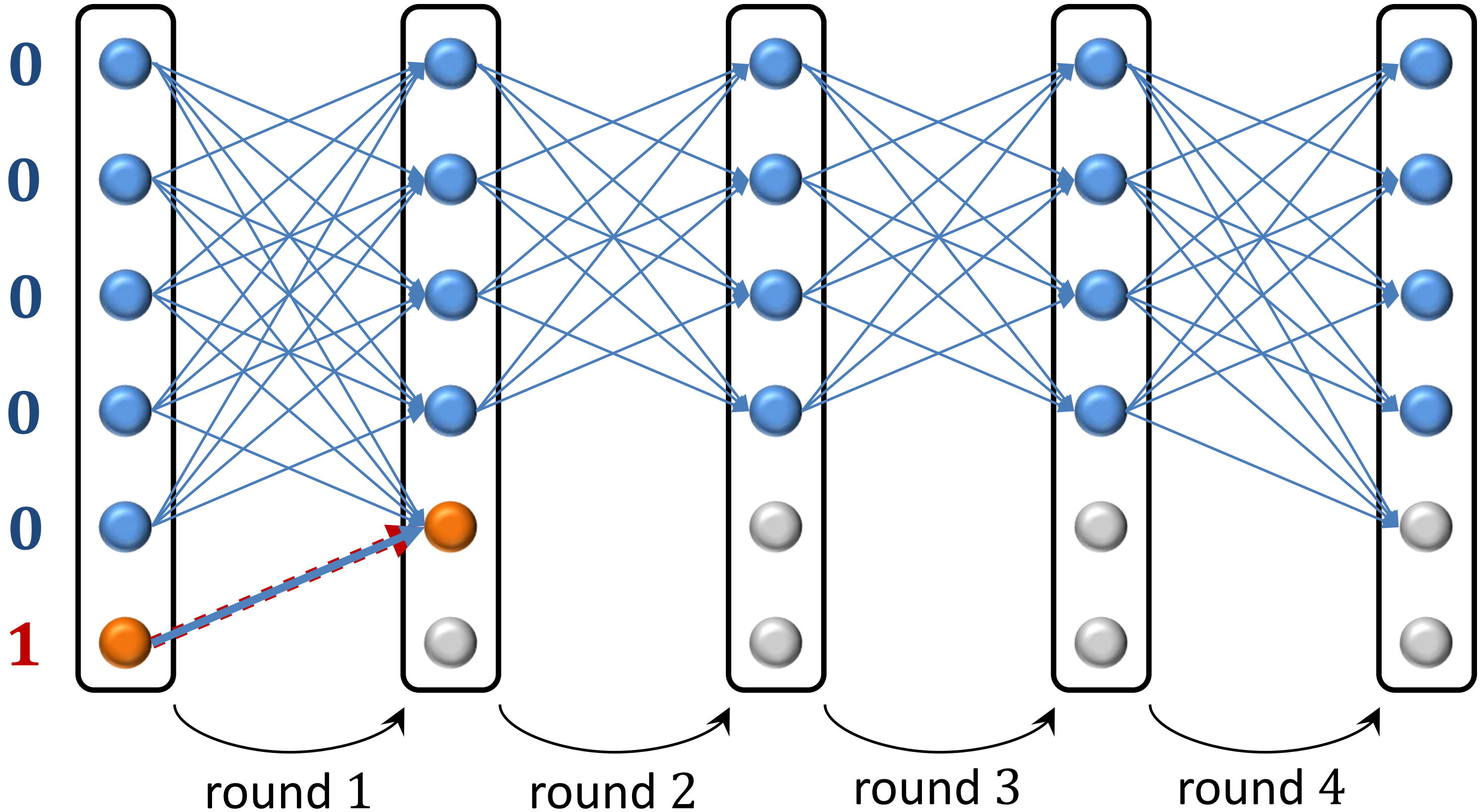
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



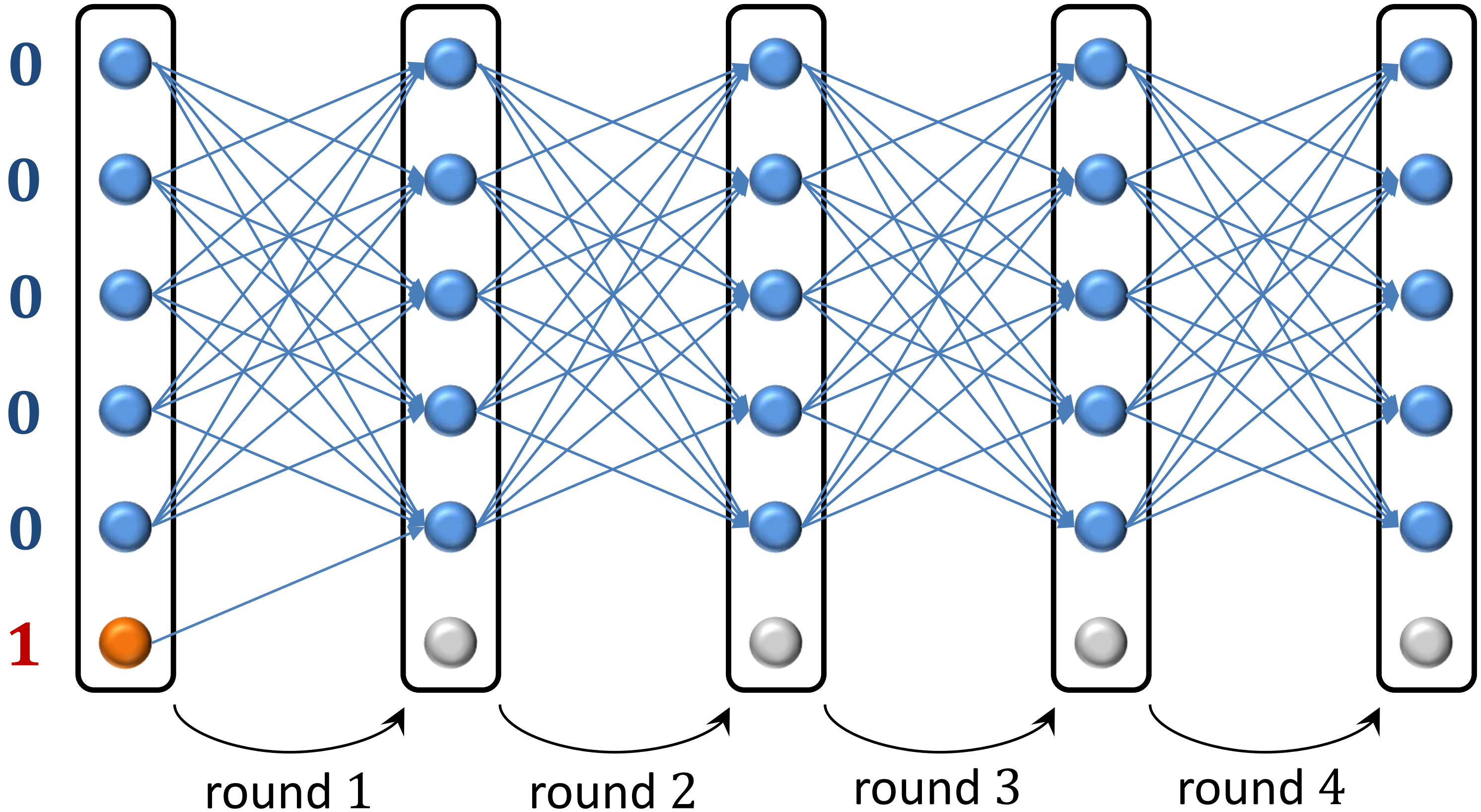
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



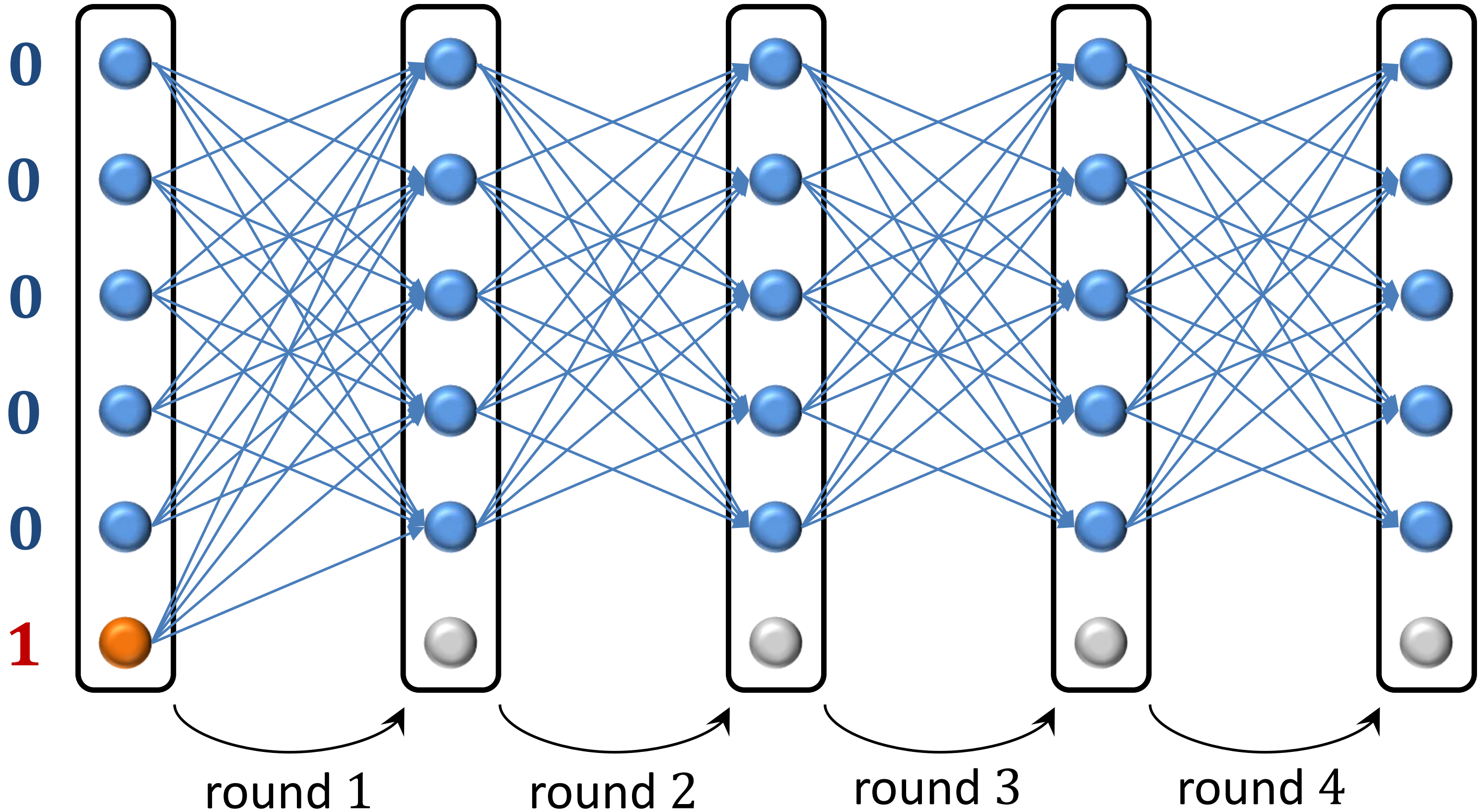
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



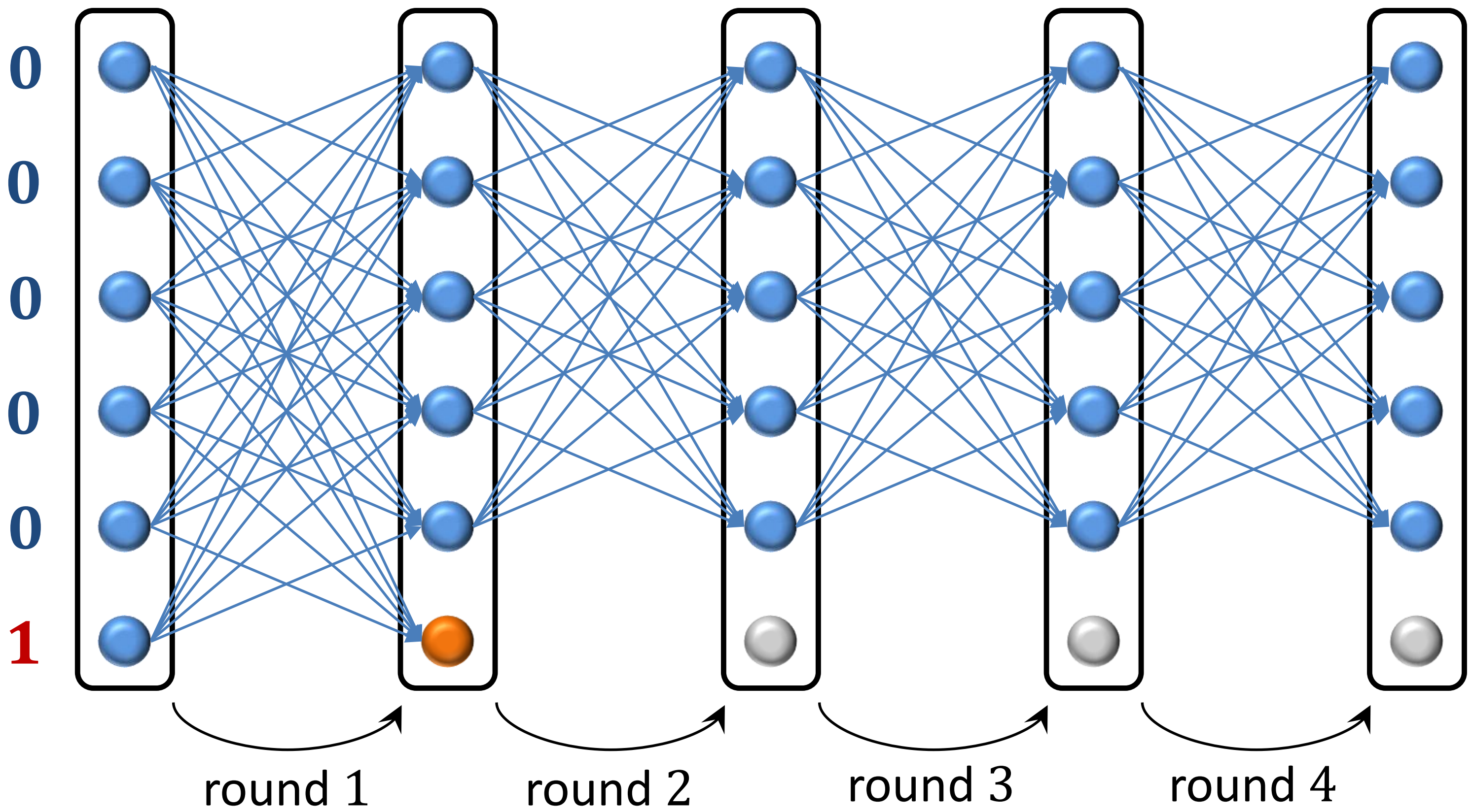
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



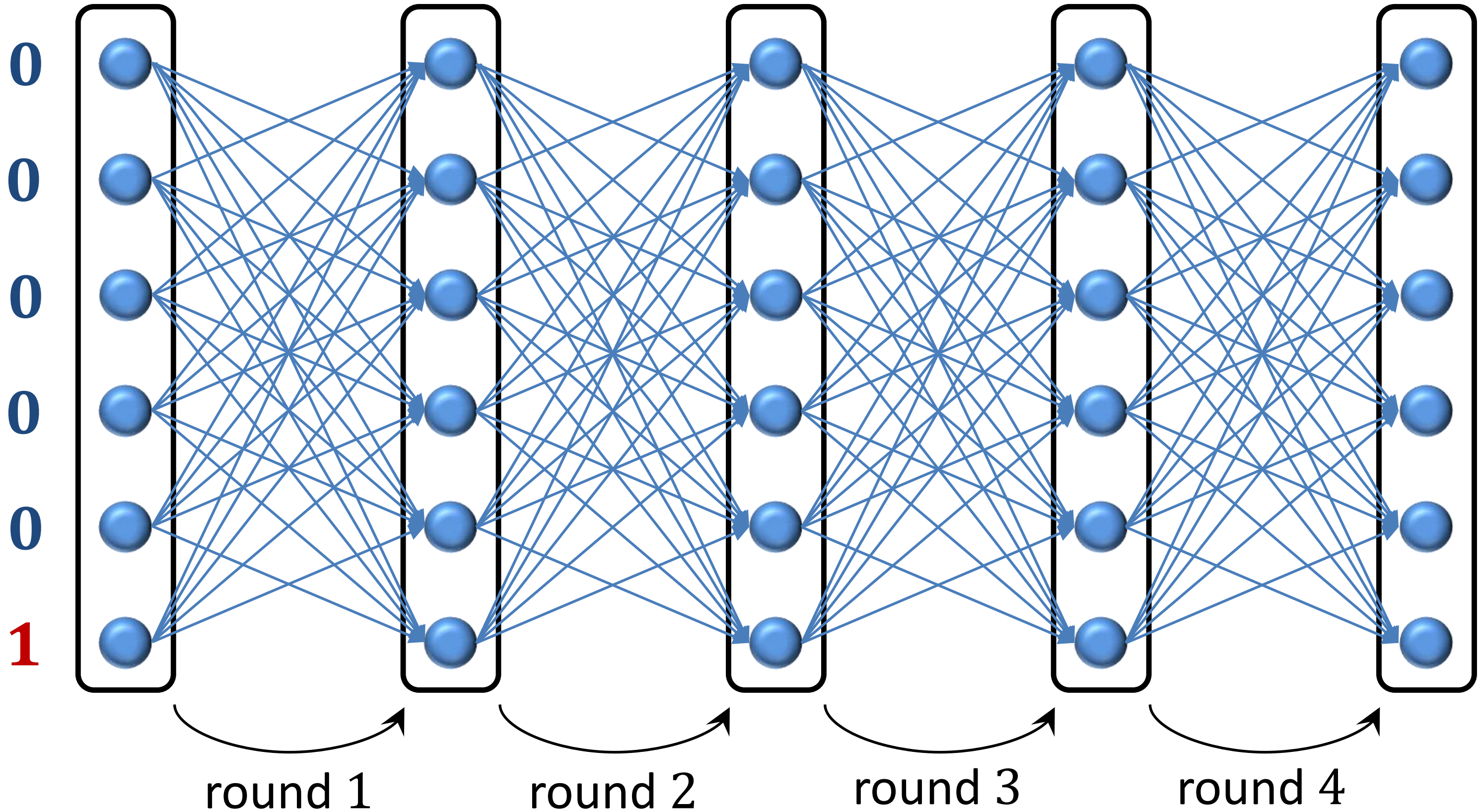
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



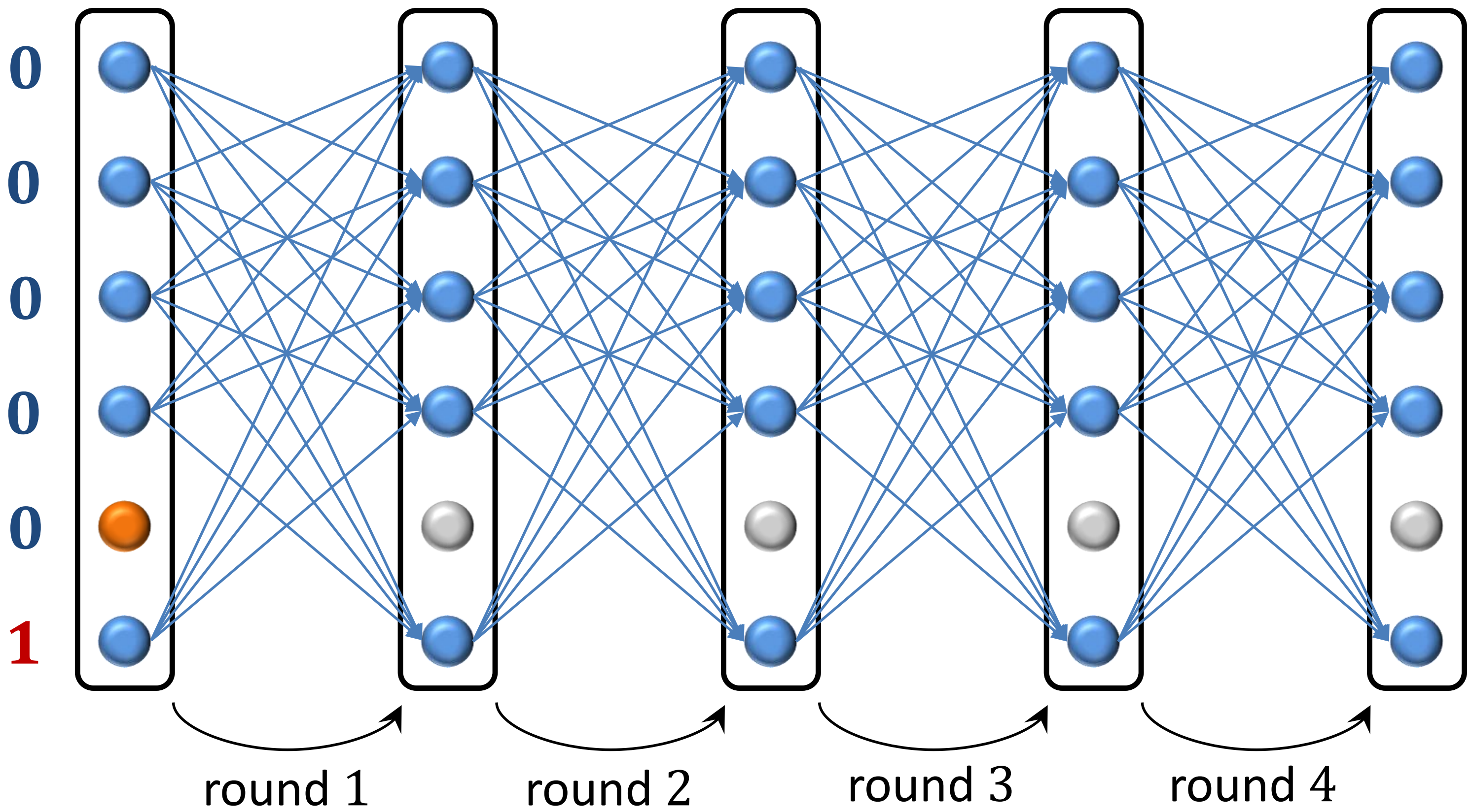
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



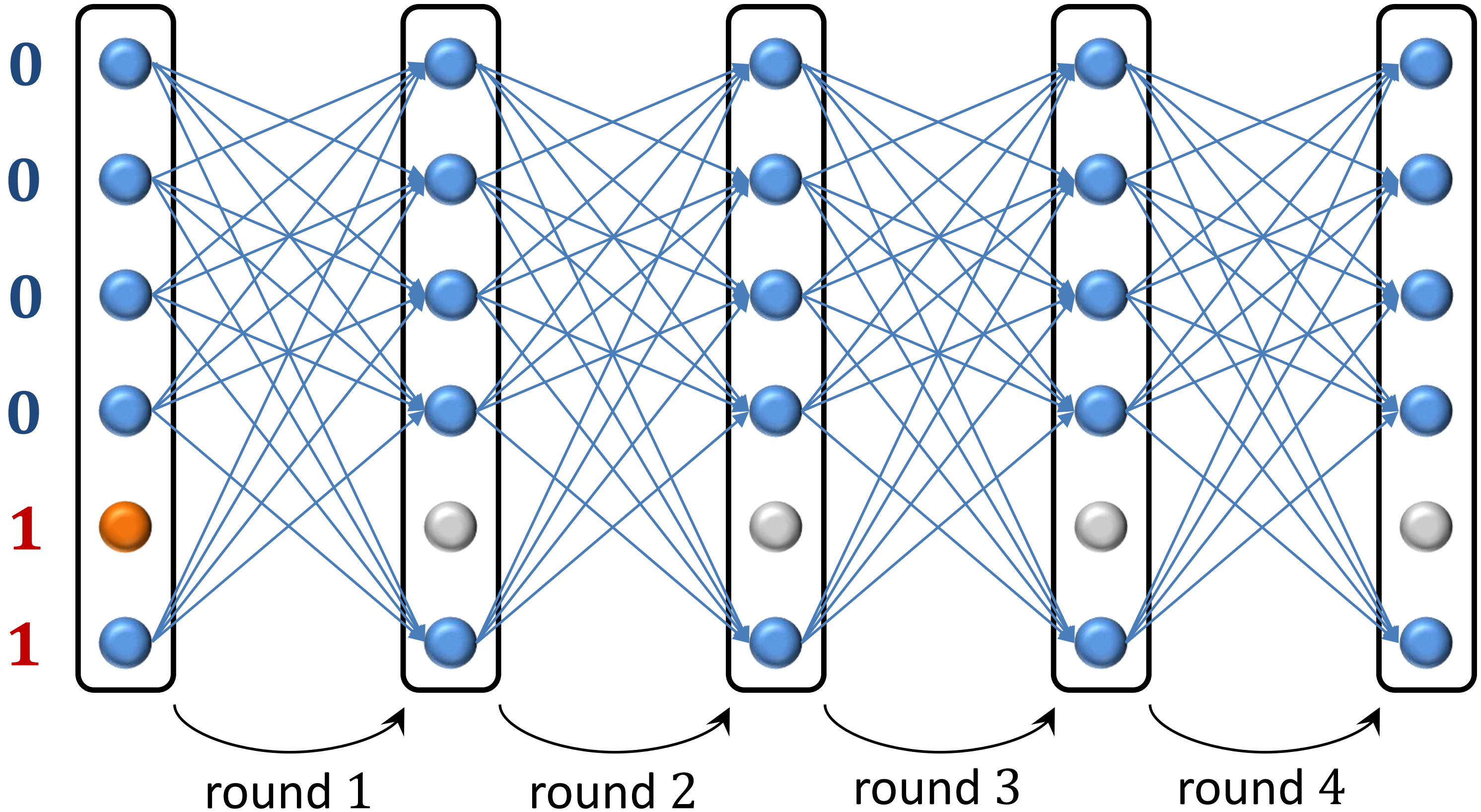
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



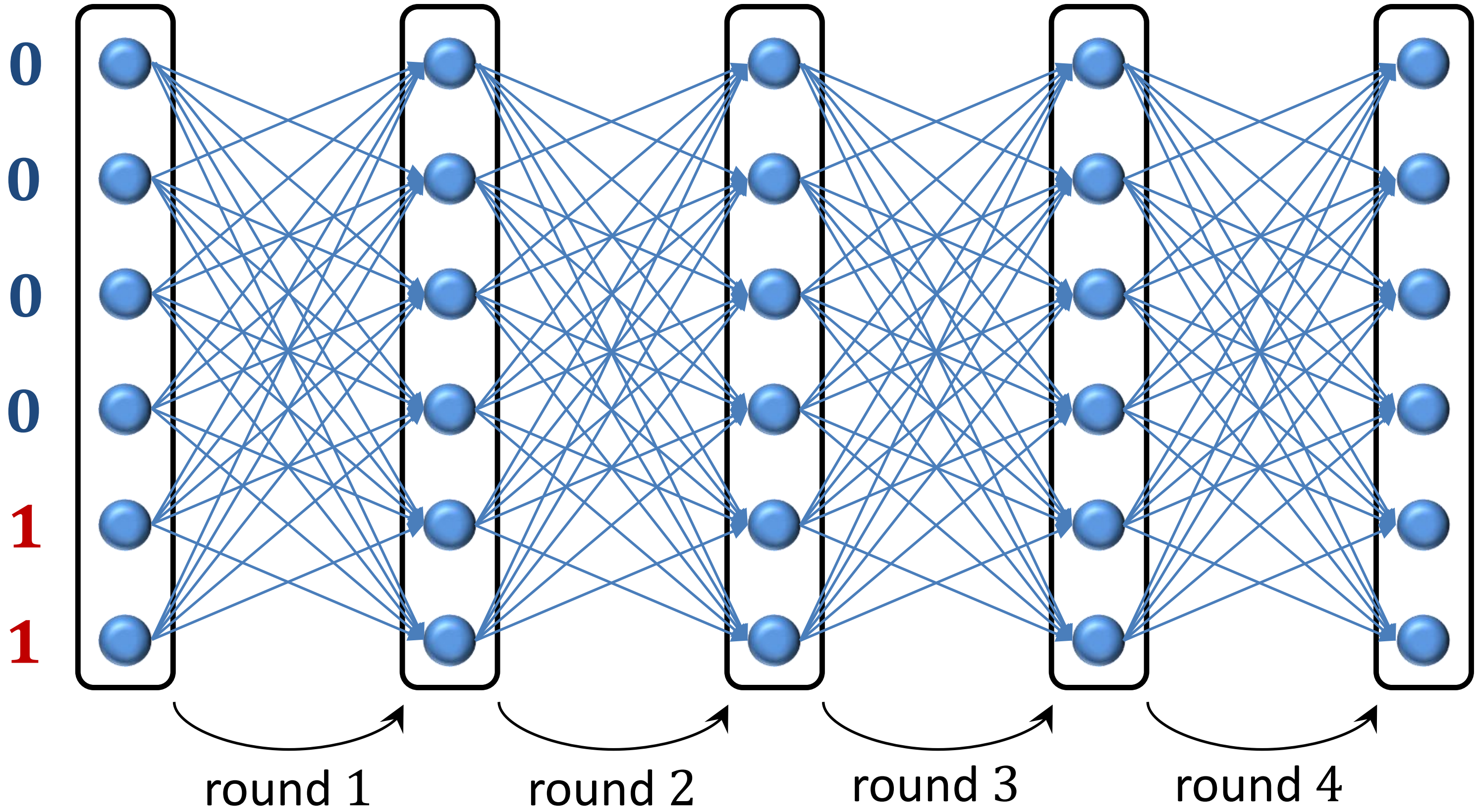
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show:** **4 rounds are not enough**



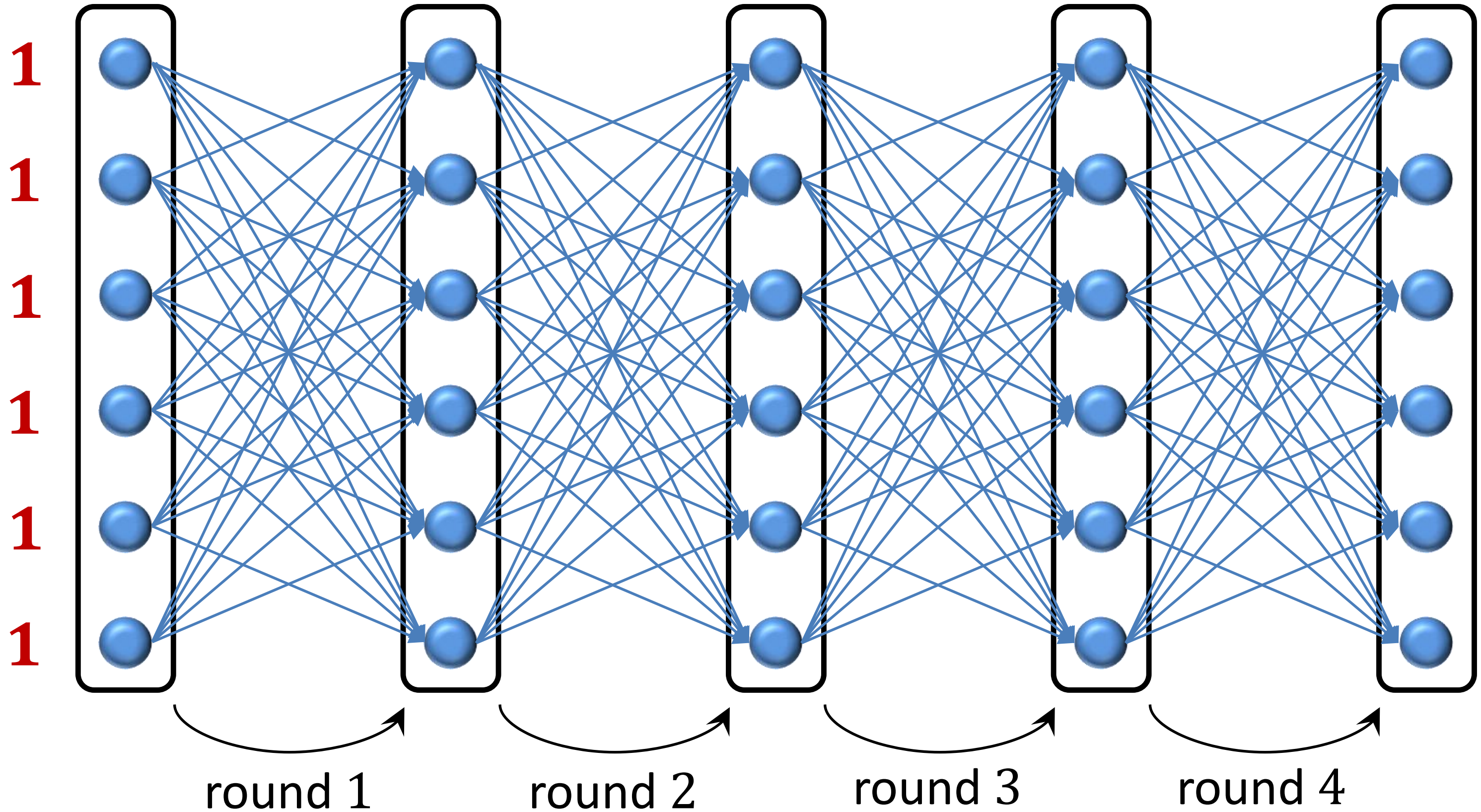
Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



Lower Bound on Rounds: Proof

Example: $f = 4, n = 6$ **Need to show: 4 rounds are not enough**



Lower Bound on Rounds

Theorem

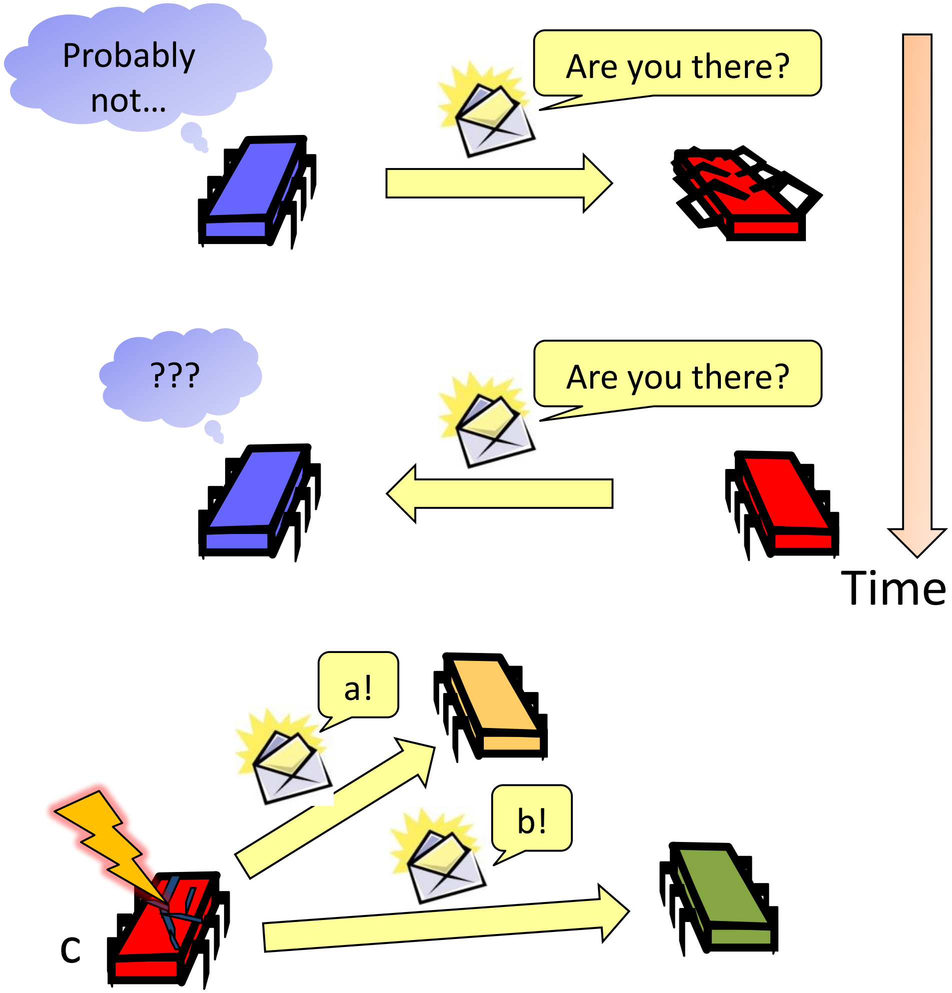
If at most $f \leq n - 2$ of n nodes of a synchronous message passing system can crash, at least $f + 1$ rounds are needed to solve consensus.

Proof:

- Similarity chain starting with fault-free all-zeroes execution and ending with fault-free all-ones execution
- In all executions, at most one crash per round
- Construction works as long as there are at least 2 non-faulty nodes in each execution ($n \geq f + 2$)
- **Validity:** all-zeroes $\Longrightarrow$ decision 0; all-ones $\Longrightarrow$ decision 1
  Similarity Chain: same decision in all executions

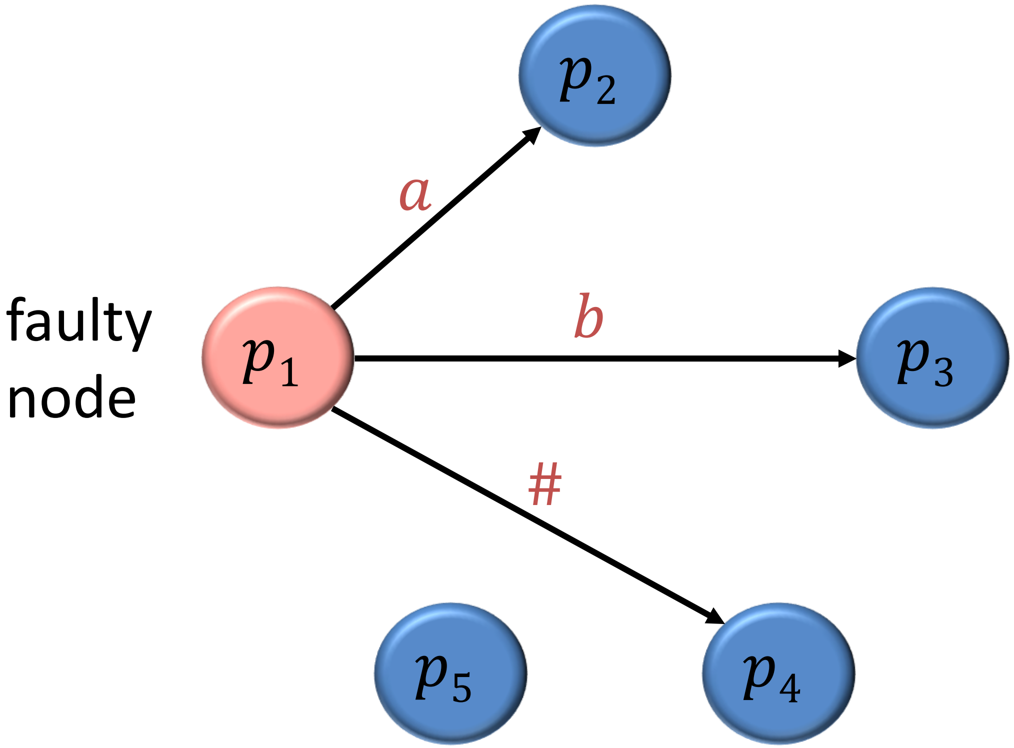
Arbitrary Behavior

- The assumption that processes crash and stop forever is sometimes too optimistic
- Maybe the processes fail and recover:
- Maybe the processes are damaged:

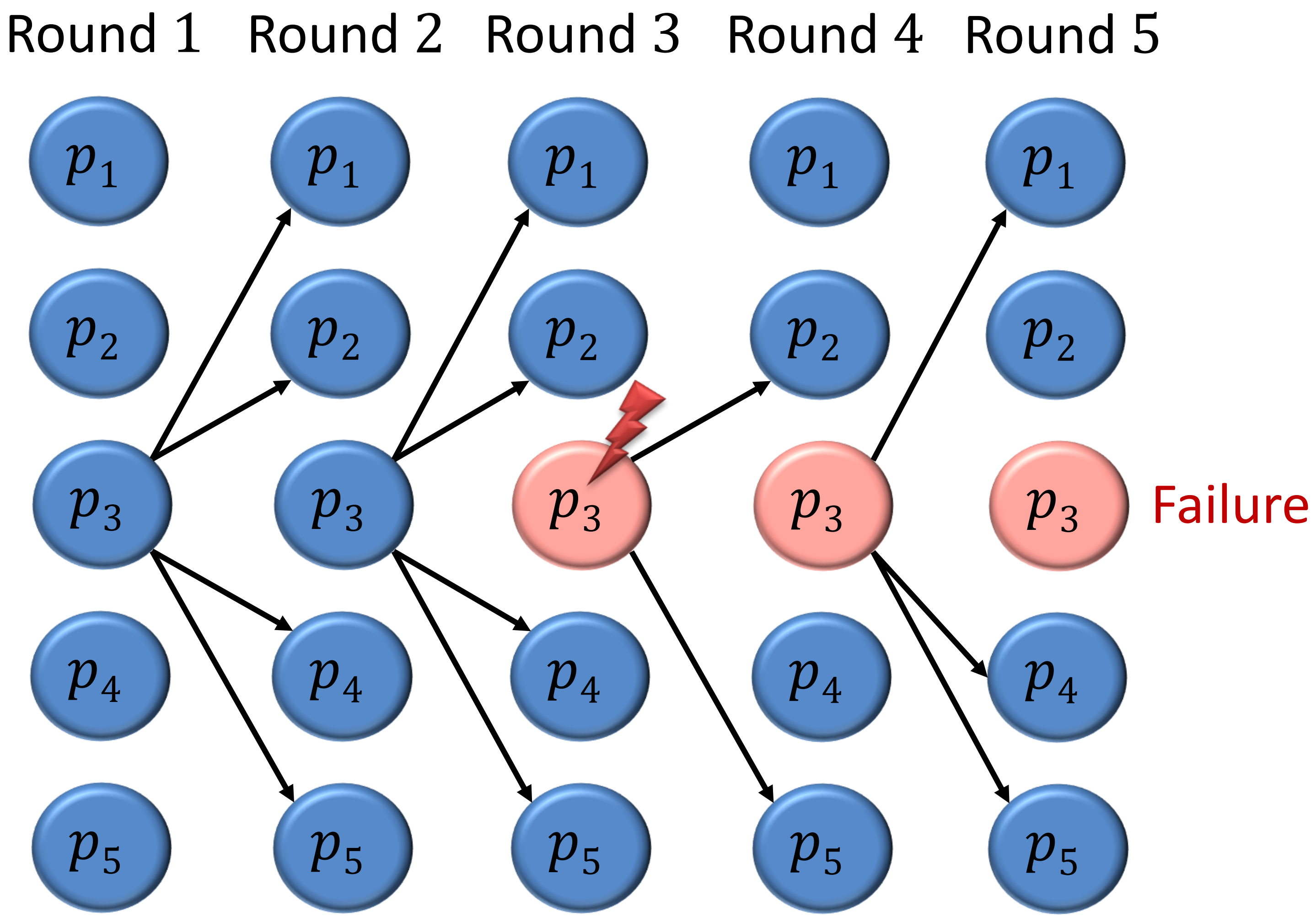


Consensus #5: Byzantine Failures

- Different processes may receive different values
- A Byzantine process can behave like a crash-failed process



After Failure, Node Remains in Network



Consensus with Byzantine Failures

- Again: If an algorithm solves consensus for f failed processes, we say it is an f-resilient consensus algorithm
- **Validity:** If all non-faulty processes start with the same value, then all non-faulty processes decide on that value
  - Note that in general this validity condition does not guarantee that the final value is an input value of a non-Byzantine process
  - However, if the input is binary, then the validity condition ensures that processes decide on a value that at least one non-Byzantine process had initially
- Obviously, any f-resilient consensus algorithm requires at least $f + 1$ rounds (follows from the crash failure lower bound)
- How large can f be...? Can we reach consensus as long as the majority of processes is correct (non-Byzantine)?

Impossibility

Theorem

There is no f-resilient Byzantine consensus algorithm for n nodes for $f \geq n/3$

Proof outline

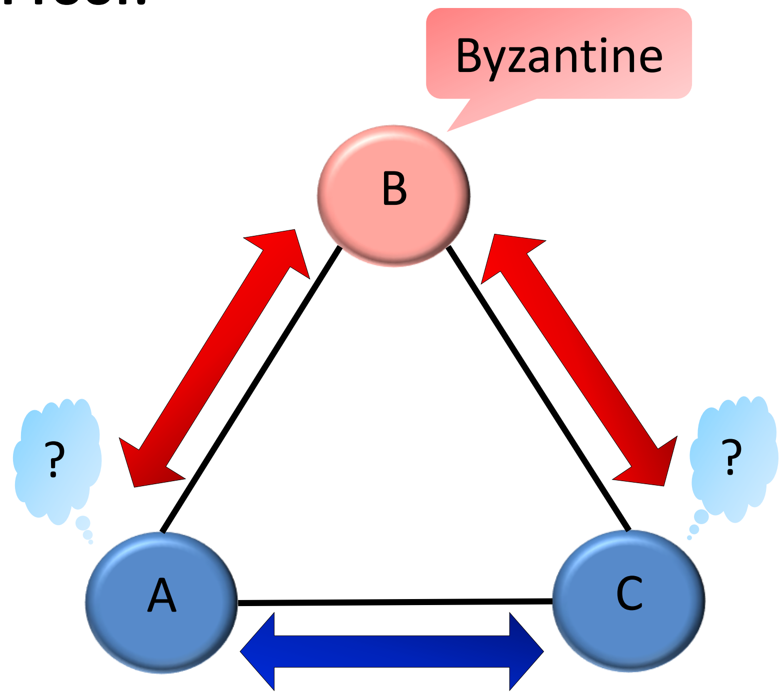
- First, we discuss the 3 node case
  - not possible for $f = 1$
- The general case can then be proved by reduction from the 3 node case
  - Given an algorithm for n node and f faults for $f \geq n/3$, we can construct a 1-resilient 3-node algorithm

The 3 Node Case

Lemma

There is no 1-resilient algorithm for 3 nodes

Proof:

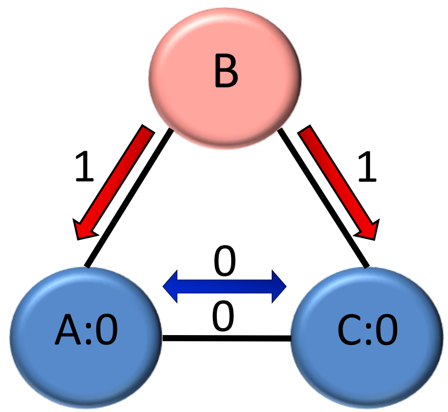


Intuition:

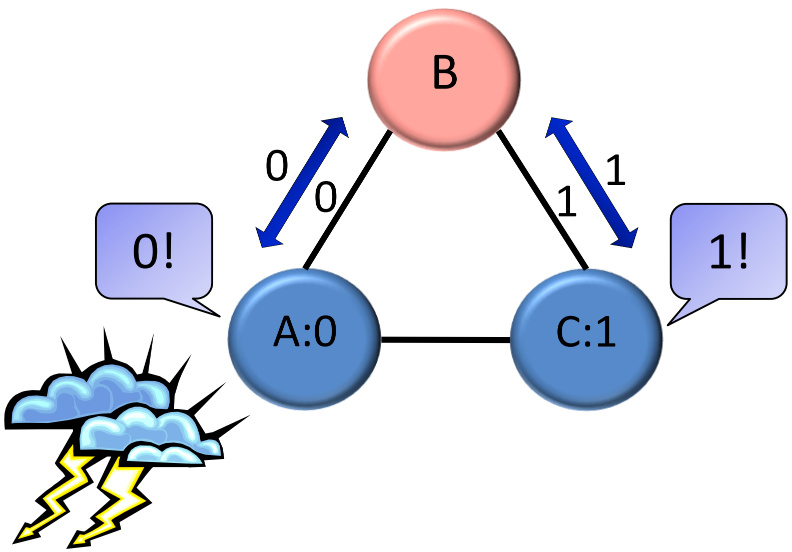
- Node A may also receive information from C about B's messages to C
- Node A may receive conflicting information about B from C and about C from B (the same for C!)
- It is impossible for A and C to decide which information to base their decision on!

Proof Sketch

- Assume that both A and C have input 0. If they decided 1, they could violate the validity condition → A and C must decide 0 independent of what B says
- Similary, A and C must decide 1 if their inputs are 1
- We see that the processes must base their decision on the majority vote

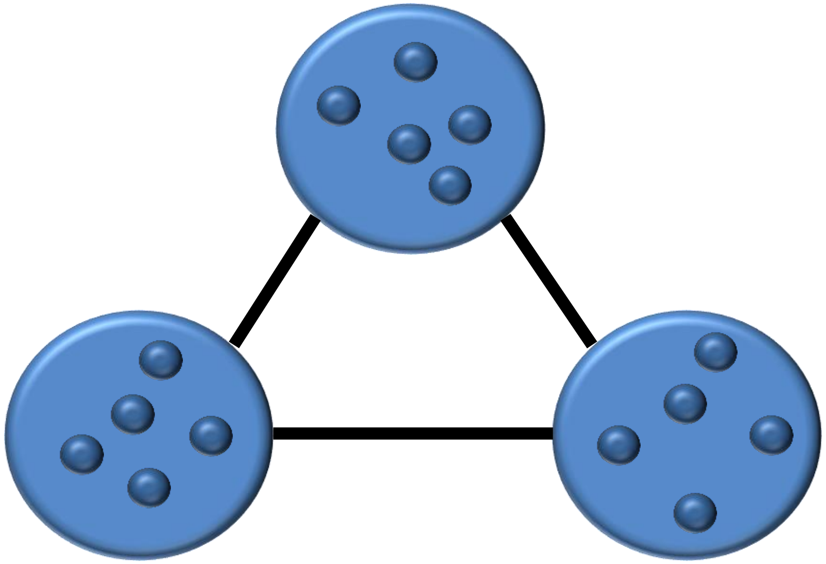


- If A's input is 0 and B tells A that its input is 0 → A decides 0
- If C's input is 1 and B tells C that its input is 1 → C decides 1



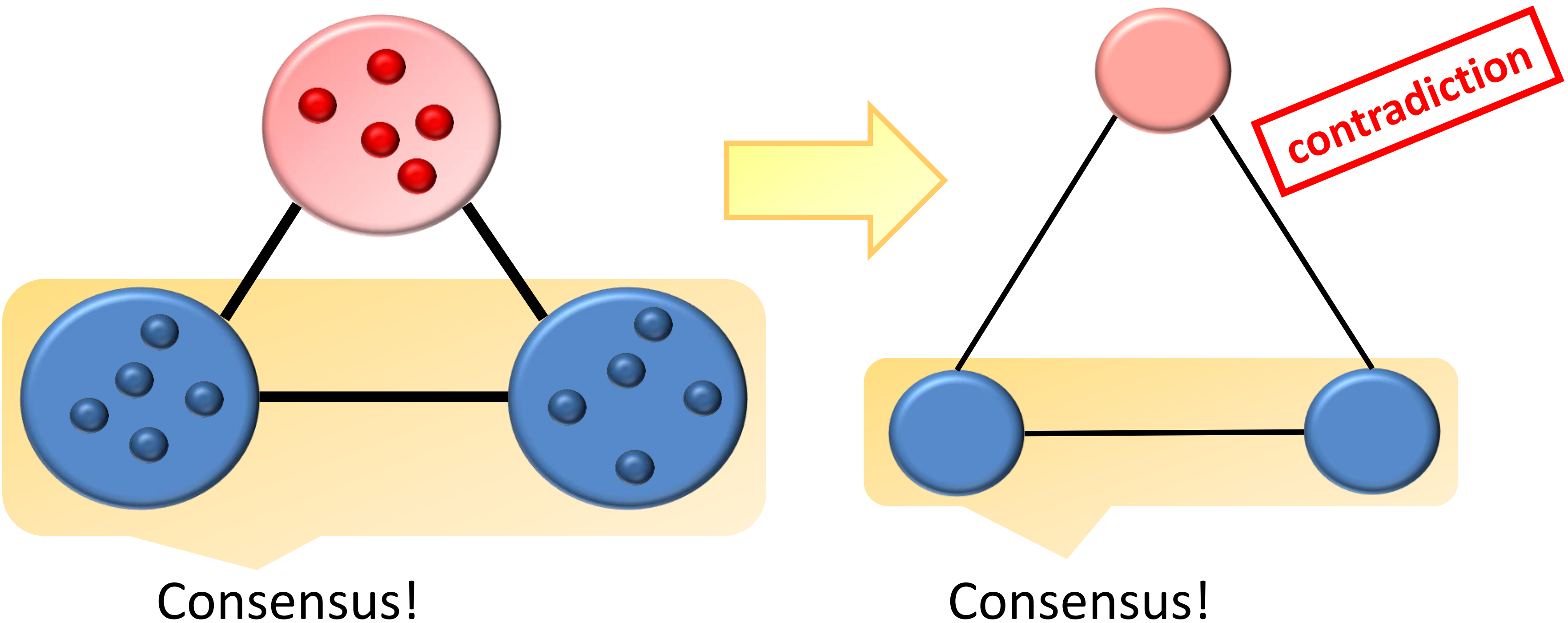
The General Case

- Assume for contradiction that there is an f-resilient algorithm A for n nodes, where $f \geq n/3$
- We use this algorithm to solve consensus for 3 nodes where one node is Byzantine!
- For simplicity assume that n is divisible by 3
- We let each of the three processes simulate $n/3$ processes



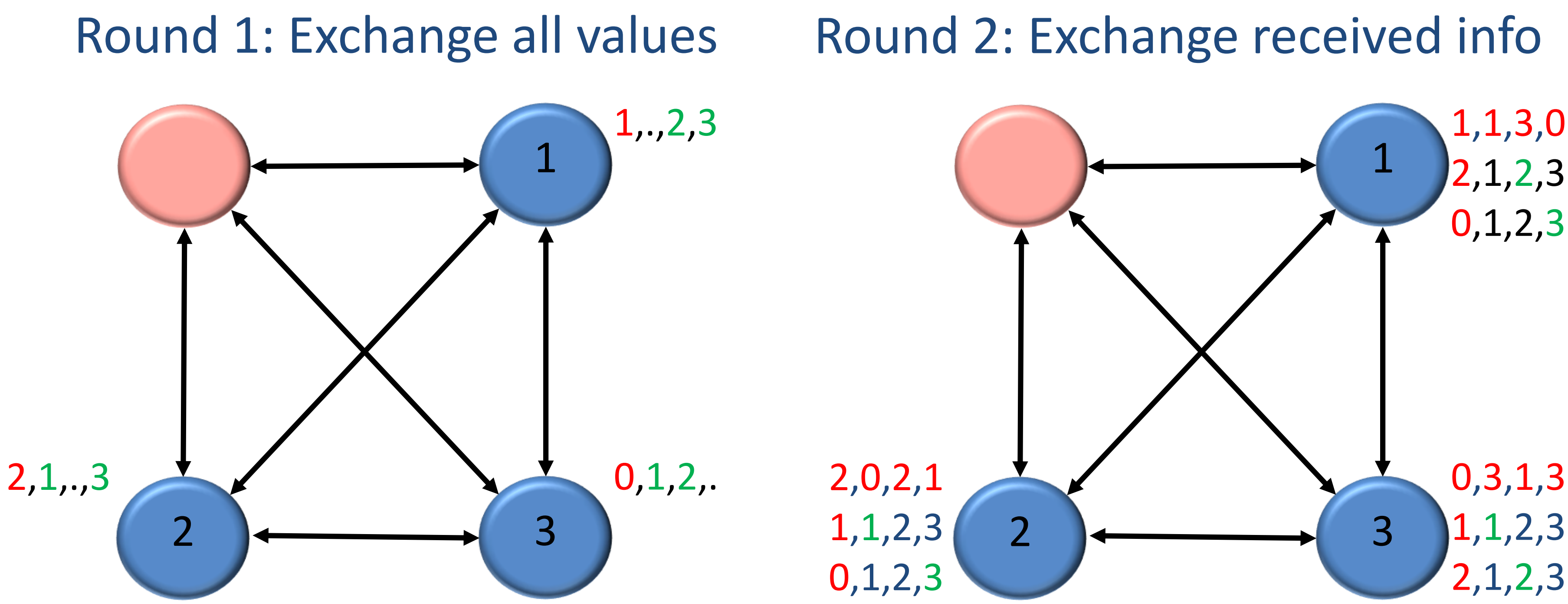
The General Case

- One of the 3 nodes is Byzantine $\Longrightarrow$ its $n/3$ simulated nodes may all behave like Byzantine nodes
- Since algorithm A tolerates $n/3$ Byzantine failures, it can still reach consensus
  $\Longrightarrow$ We solved the consensus problem for three processes!



Cons. #6: Simple Byzantine Agreement Alg.

- Can the nodes reach consensus if $n > 3f$?
- A simpler question: What if $n = 4$ and $f = 1$?
- The answer is yes. It takes two rounds:



[matrix: one column for each original value, one row for each neighbor]

Simple Byzantine Agreement Algorithm

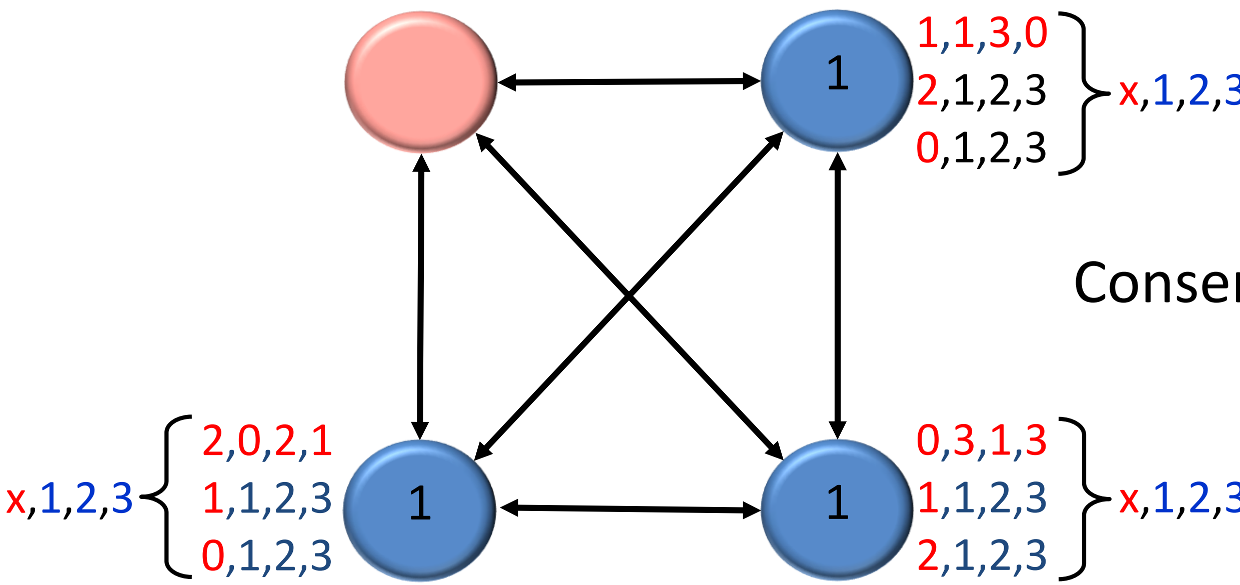
- After round 2, each node has received 12 values, 3 for each of the 4 input values (columns). If at least 2 of the 3 values of a column are equal, this value is accepted, otherwise it is discarded.
  - Values of honest nodes are accepted

Simple Byzantine Agreement Algorithm

- After round 2, each node has received 12 values, 3 for each of the 4 input values (columns). If at least 2 of the 3 values of a column are equal, this value is accepted, otherwise it is discarded.
  - Values of honest nodes are accepted
  - The value of the Byzantine node is accepted iff it sends the same value to at least two nodes in the first round.

Simple Byzantine Agreement Algorithm

- After round 2, each node has received 12 values, 3 for each of the 4 input values (columns). If at least 2 of the 3 values of a column are equal, this value is accepted, otherwise it is discarded.
  - Values of honest nodes are accepted
  - The value of the Byzantine node is accepted iff it sends the same value to at least two nodes in the first round.
- Decide on most frequently accepted value!

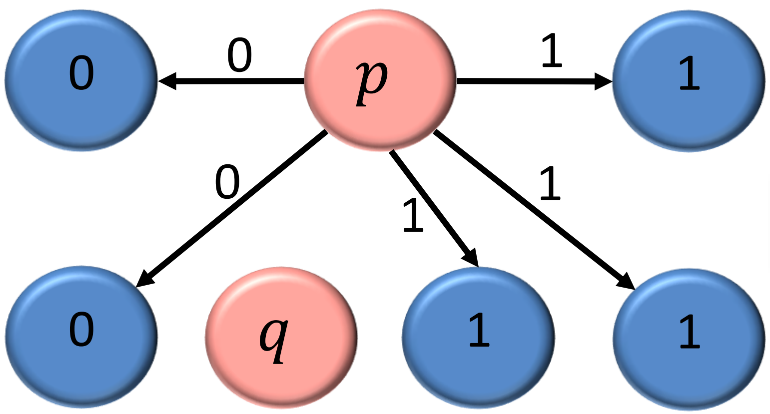


Consensus!

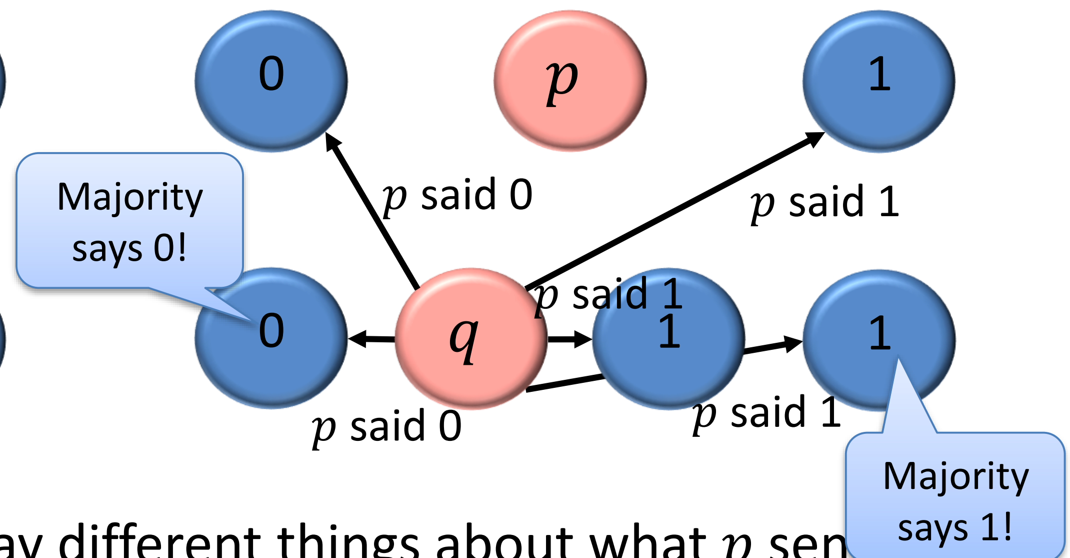
Simple Byzantine Agreement Algorithm

- Does the algorithm still work in general for any f and $n > 3f$?
- The answer is no. Try $f = 2$ and $n = 7$:

Round 1: Exchange all values



Round 2: Exchange received info



- The problem is that q can say different things about what p sen
  - What is the solution to this problem?

Simple Byzantine Agreement Algorithm

- The solution is simple: Again exchange all information!
- This way, the nodes can learn that q gave inconsistent information about p
- Hence, q can be excluded, and also p if it also gave inconsistent information (about q).
- If $f = 2$ and $n > 6$, consensus can be reached in 3 rounds!
- In fact, the following "algorithm" solves the problem for any f and any $n > 3f$:

Exchange all information for $f + 1$ rounds
Ignore all nodes that provided inconsistent information
Let all nodes decide based on the same input

Simple Byzantine Agreement Algorithm

The proposed algorithm has several advantages:

+ It works for any f and $n > 3f$, which is optimal
+ It only takes $f + 1$ rounds. This is even optimal for crash failures!
+ It works for any input and not just binary input

However, it has some considerable disadvantages:

– "Ignoring all nodes that provided inconsistent information" is not easy to formalize

– The **size of the messages increases exponentially**! This is a severe problem. It is therefore worth studying whether it is possible to solve the problem with small(er) messages

Consensus #7: The Queen Algorithm

- The Queen algorithm is a simple Byzantine agreement algorithm that uses small messages
- The Queen algorithm solves consensus with n nodes and f failures where $f < n/4$ in $f + 1$ phases

A phase consists of 2 rounds

Idea:

- There is a different (a priori known) queen in each phase
- Since there are $f + 1$ phases, in one phase the queen is not Byzantine
- Make sure that in this round all nodes choose the same value and that in future rounds the nodes do not change their values anymore

The Queen Algorithm

In each phase $i \in \{1, \ldots, f+1\}$:

At the end of phase $f+1$, decide on own value

Round 1:

Broadcast own value

Also send own value to oneself

Set own value to the value that was received most often

If several values have the same (highest) frequency, choose any value, e.g., the smallest

If own value appears $> n/2 + f$ times

  support this value

else

  do not support any value

Round 2:

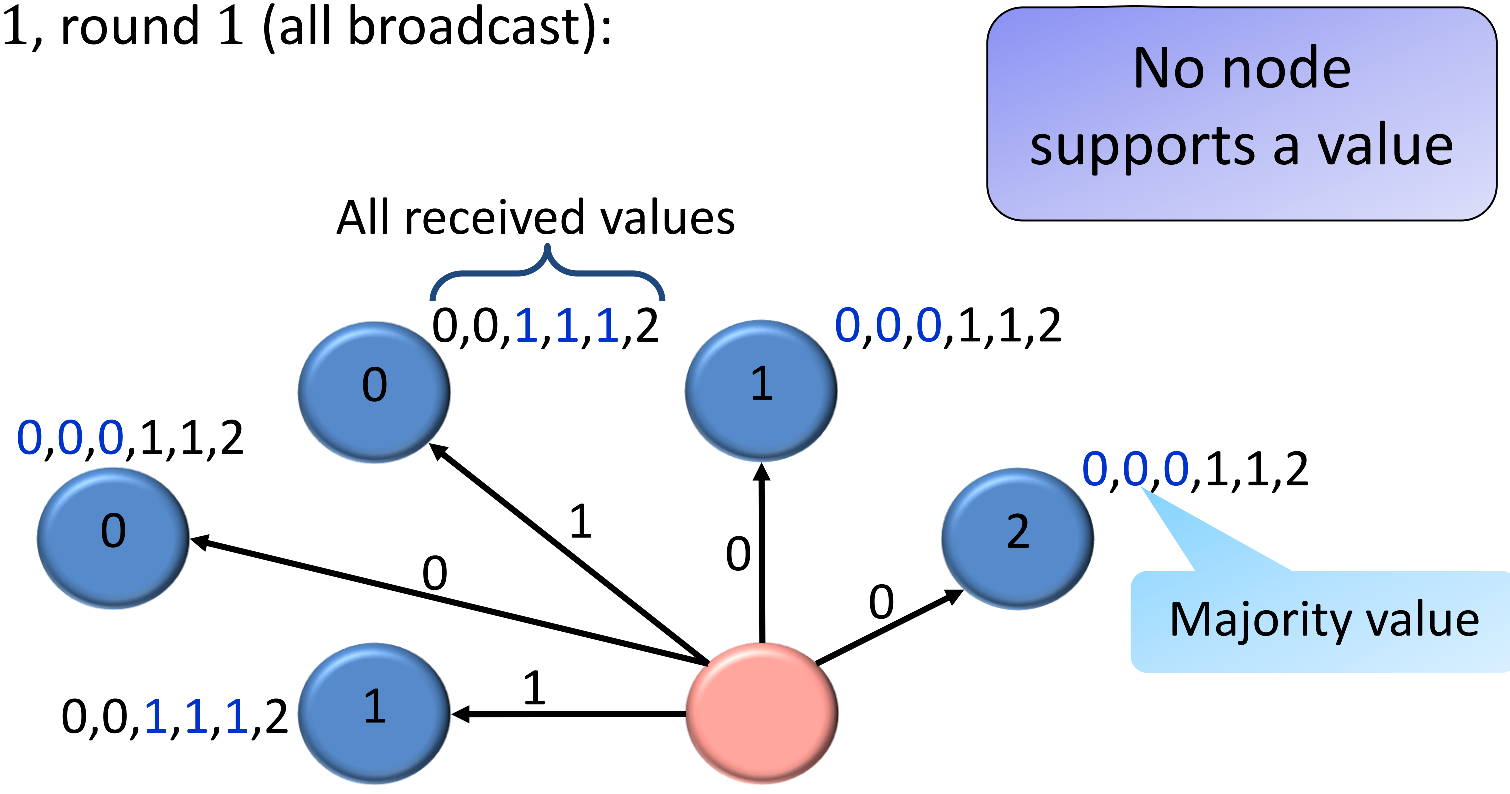
The queen broadcasts its value

If not supporting any value

  set own value to the queen's value

The Queen Algorithm: Example

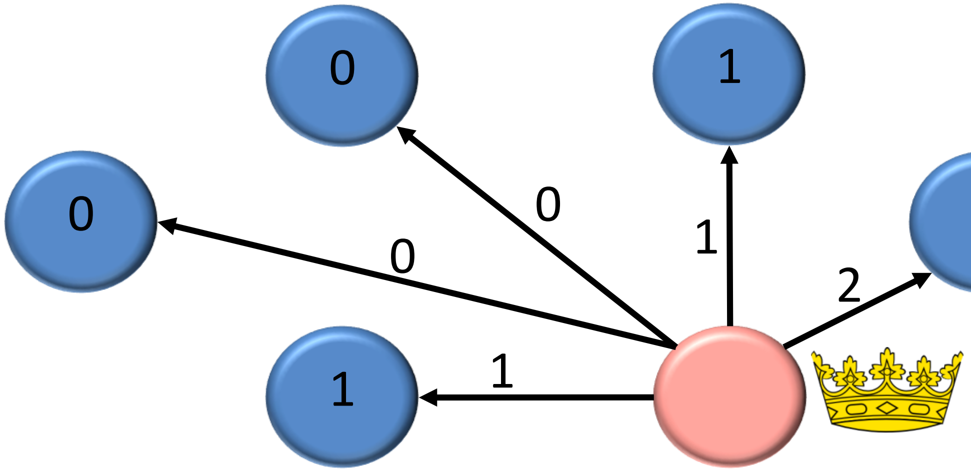
- Example: $n = 6, f = 1$
- Phase 1, round 1 (all broadcast):



The Queen Algorithm: Example

- Example: $n = 6, f = 1$
- Phase 1, round 2 (queen broadcasts):

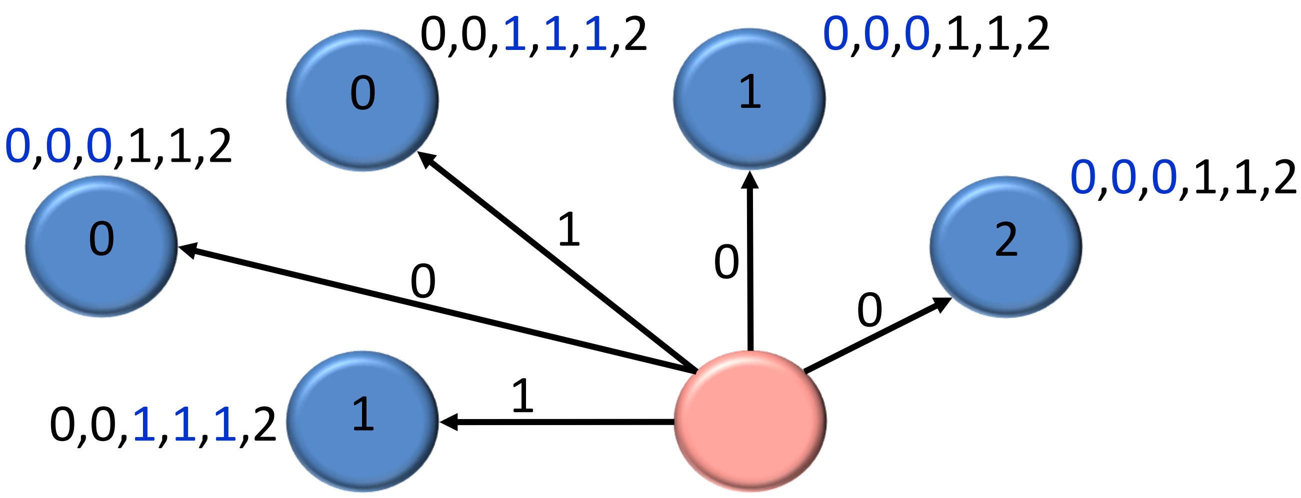
All nodes choose the queen's value



The Queen Algorithm: Example

- Example: $n = 6, f = 1$
- Phase 2, round 1 (all broadcast):

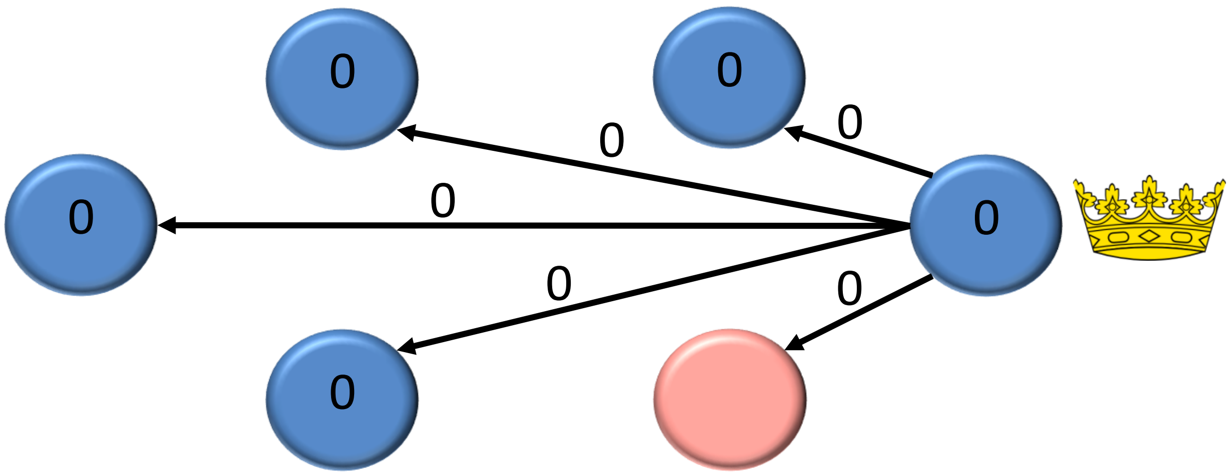
No node supports a value



The Queen Algorithm: Example

- Example: $n = 6, f = 1$
- Phase 2, round 2 (queen broadcasts):

All nodes choose the queen's value



Consensus!

The Queen Algorithm: Analysis

- After the phase where the queen is correct, all correct nodes have the same value
  - If all nodes change their values to the queen's value, obviously all values are the same
  - If some node does not change its value to the queen's value, it received a value $> n/2 + f$ times → All other correct nodes (including the queen) received this value $> n/2$ times and thus all correct nodes share this value
- In all future phases, no node changes its value
  - In the first round of such a phase, nodes receive their own value from at least $n - f > n/2$ nodes and thus do not change it
  - The nodes do not accept the queen's proposal if it differs from their own value in the second round because the nodes received their own value at least $n - f > n/2 + f$ times. Thus, all correct nodes support the same value

That's why we need $f < n/4$!

The Queen Algorithm: Summary

The Queen algorithm has several advantages:

+ The messages are small: nodes only exchange their current values
+ It works for any input and not just binary input

However, it also has some disadvantages:

– The algorithm requires $f + 1$ phases consisting of 2 rounds each ... this is twice as much as an optimal algorithm
– It only works with $f < n/4$ Byzantine nodes!

- Is it possible to get an algorithm that works with $f < n/3$ Byzantine nodes and uses small messages?

Consensus #8: The King Algorithm

- The King algorithm is an algorithm that tolerates $f < n/3$ Byzantine failures and uses small messages
- The King algorithm also takes $f + 1$ phases

A phase now consists of 3 rounds

Idea:

- The basic idea is the same as in the Queen algorithm
- There is a different (a priori known) king in each phase
- Since there are $f + 1$ phases, in one phase the king is not Byzantine
- The difference to the Queen algorithm is that the correct nodes only propose a value if many nodes have this value, and a value is only accepted if many nodes propose this value

The King Algorithm

In each phase $\boldsymbol{i} \in \{\mathbf{1} \dots \boldsymbol{f} + \mathbf{1}\}$:

At the end of phase $f + 1$, decide on own value

Round 1:

Broadcast own value

Also send own value to oneself

Round 2:

If some value x appears $\geq n - f$ times

Broadcast "Propose x"

If some proposal received $> f$ times

Set own value to this proposal

Round 3:

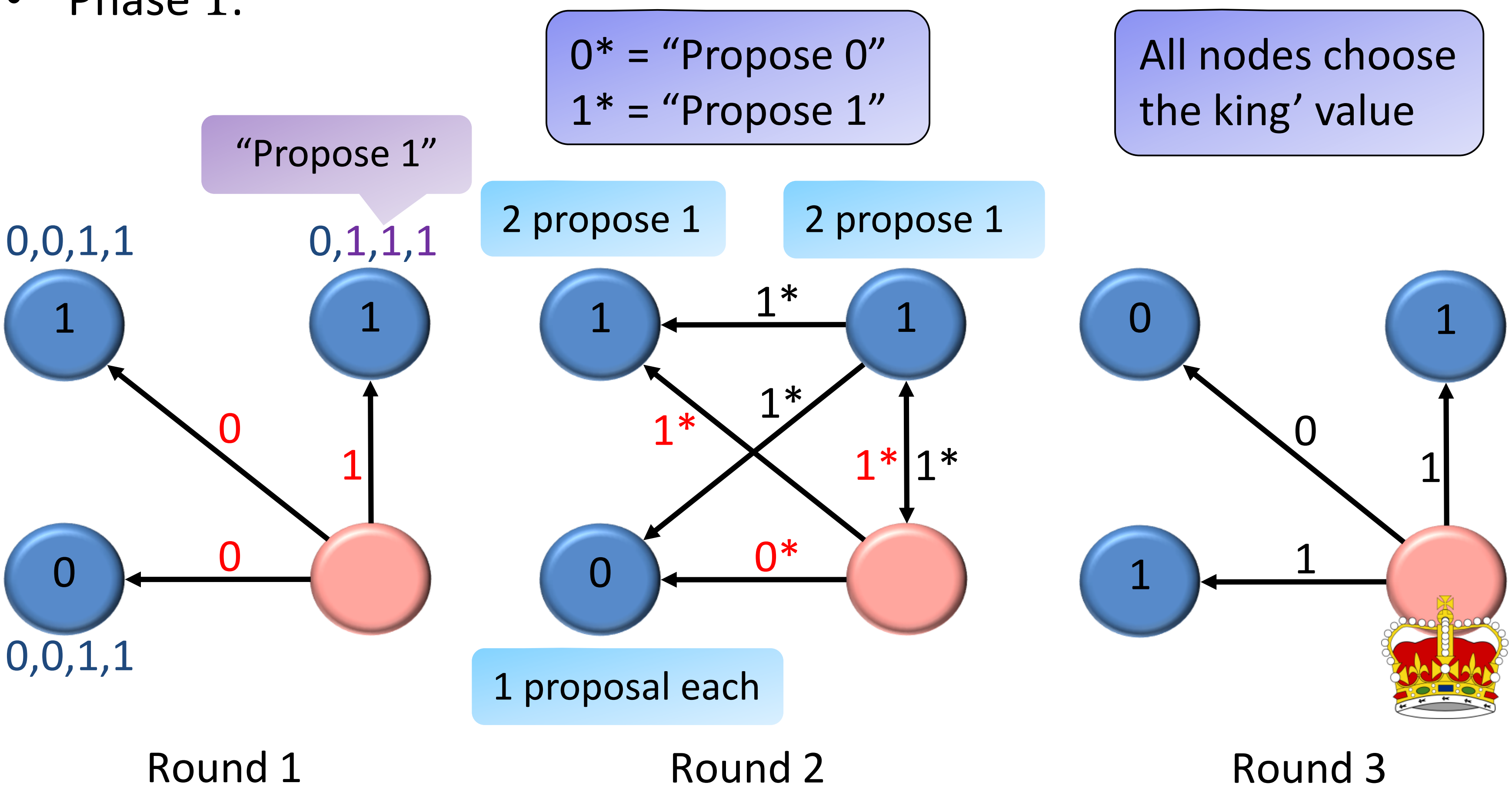
The king broadcasts its value

If own value received $< n - f$ proposals

Set own value to the king's value

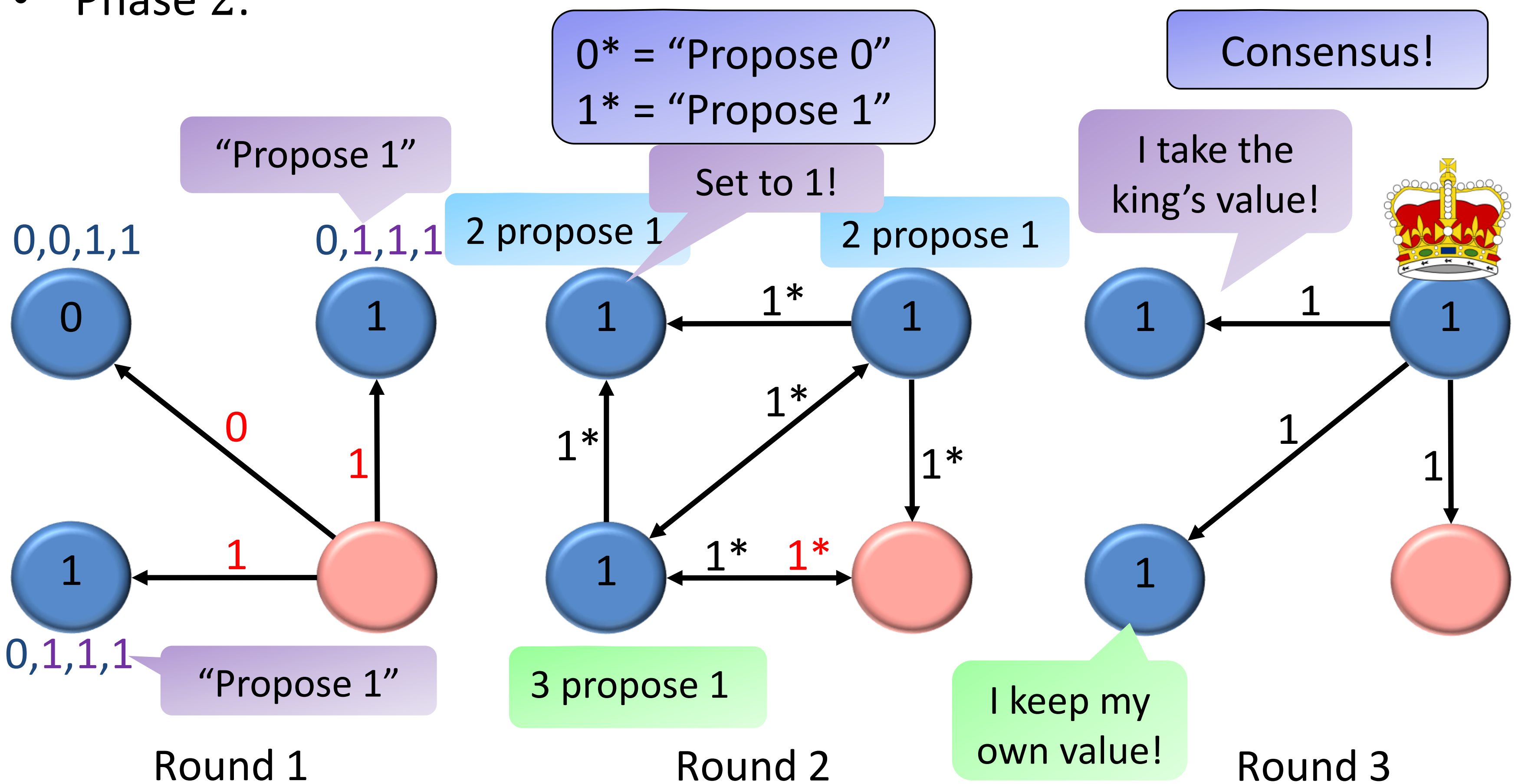
The King Algorithm: Example

- Example: $n = 4$, $f = 1$
- Phase 1:



The King Algorithm: Example

- Example: $n = 4$, $f = 1$
- Phase 2:



The King Algorithm: Analysis

- Observation: If some correct node proposes x, then no other correct node proposes $y \neq x$
  - Both nodes would have to receive $\geq n - f$ times the same value, i.e., both nodes received their value from $\geq n - 2f$ distinct correct nodes
  - In total, there must be $\geq 2(n - 2f) + f > n$ nodes, a contradiction!

We used that $f < n/3$!

- The validity condition is satisfied
  - If all correct nodes start with the same value, all correct nodes receive this value $\geq n - f$ times and propose it
  - All correct nodes receive $\geq n - f$ times proposals, i.e., no correct node will ever change its value to the king's value

The King Algorithm: Analysis

- After the phase where the king is correct, all correct processes have the same value
  - If all processes change their values to the king's value, obviously all values are the same
  - If some process does not change its value to the king's value, it received a proposal $\geq n - f$ times → $\geq n - 2f$ correct processes broadcast this proposal and all correct processes receive it $\geq n - 2f > f$ times
    → All correct processes set their value to the proposed value. Note that only one value can be proposed $> f$ times, which follows from the observation on the previous slide
- In all future phases, no process changes its value
  - This follows immediately from the fact that all correct processes have the same value after the phase where the king is correct and the validity condition

The King Algorithm: Summary

The King algorithm has several advantages:

+ It works for any f and $n > 3f$, which is optimal
+ The messages are small: processes only exchange their current values
+ It works for any input and not just binary input

However, it also has a disadvantage:

– The algorithm requires $f + 1$ phases consisting of 3 rounds each
  This is three times as much as an optimal algorithm

Consensus #9: A Randomized Algorithm

- So far we mainly tried to reach consensus in synchronous systems. The reason is that no deterministic algorithm can guarantee consensus in asynchronous systems even if only one process may crash

Synchronous system: Communication proceeds in synchronous rounds

- Can one solve consensus in asynchronous systems if we allow our algorithms to use randomization?

Asynchronous system: Messages are delayed indefinitely

- The answer is yes!
- The basic idea of the algorithm is to push the initial value. If other nodes do not follow, try to push one of the suggested values randomly
- For the sake of simplicity, we assume that the input is binary and at most $f < n/9$ nodes are Byzantine

Randomized Algorithm

$x :=$ own input; $r := 0$
Broadcast proposal(x, r)

In each round $r = 1, 2, \ldots$:

Wait for $n - f$ proposals
If at least $n - 2f$ proposals have some value y
    $x := y$; decide on y
else if at least $n - 4f$ proposals have some value y
    $x := y$;
else
    choose x randomly with $\Pr[x = 0] = \Pr[x = 1] = 1/2$
Broadcast proposal(x, r)
If decided on a value → stop

Randomized Algorithm: Analysis

Validity condition (If all have the same input, all choose this value)

- If all correct nodes have the same initial value x, they will receive $n - 2f$ proposals containing x in the first round and they will decide on x

Agreement (if the nodes decide, they agree on the same value)

- Assume that some correct node decides on x. This node must have received x from $n - 3f$ correct nodes. Every other correct node must have received x at least $n - 4f$ times, i.e., all correct nodes set their local value to x, and propose and decide on x in the next round

> The processes broadcast at the end of a phase to ensure that the processes that have already decided broadcast their value again!

Randomized Algorithm: Analysis

Termination (all correct processes eventually decide)

- If some nodes do not set their local value randomly, they set their local value to the same value.
  Proof: Assume that some nodes set their value to 0 and some others to 1, i.e., there are $\geq n - 5f$ correct nodes proposing 0 and $\geq n - 5f$ correct processes proposing 1.
  Then, in total there are $\geq 2(n - 5f) + f > n$ nodes. Contradiction!

  That's why we need $f < n/9$!

  – Thus, in the worst case all $n - f$ correct nodes need to choose the same bit randomly, which happens with probability $1/2^{n-f}$
  – Hence, all correct processes eventually decide. The expected running time is smaller than 2^n
- The running time is awfully slow. Is there a clever way to speed up the algorithm?
- What about simply setting $x := 1$?! (Why doesn't it work?)

Can we do this faster?! Yes, with a Shared Coin

- A better idea is to replace

  choose x randomly with $\Pr[x = 0] = \Pr[x = 1] = 1/2$

  with a subroutine in which all the processes compute a so-called shared (a.k.a. common, “global”) coin

- A shared coin is a random binary variable that is 0 with constant probability and 1 with constant probability
- For the sake of simplicity, we assume that there are at most $f < n/3$ crash failures (no Byzantine failures!)

All correct nodes know the outcome of the shared coin toss after each execution of the subroutine

Shared Coin Algorithm

Code for process i:

Set local coin $c_i := 0$ with probability $1/n$, else $c_i := 1$
Broadcast c_i
Wait for exactly $n - f$ coins and collect all coins in the local coin set s_i
Broadcast s_i
Wait for exactly $n - f$ coin sets
If at least one coin is 0 among all coins in the coin sets
    return 0
else
    return 1

Assume the worst case:
Choose f so that $3f + 1 = n$!