DD2451
Parallel and Distributed Computing

FDD3008
Distributed Algorithms

Lecture 7

Consensus, II

Mads Dam
Autumn/Winter 2011

Slides: Much material due to R. Wattenhofer, ETH

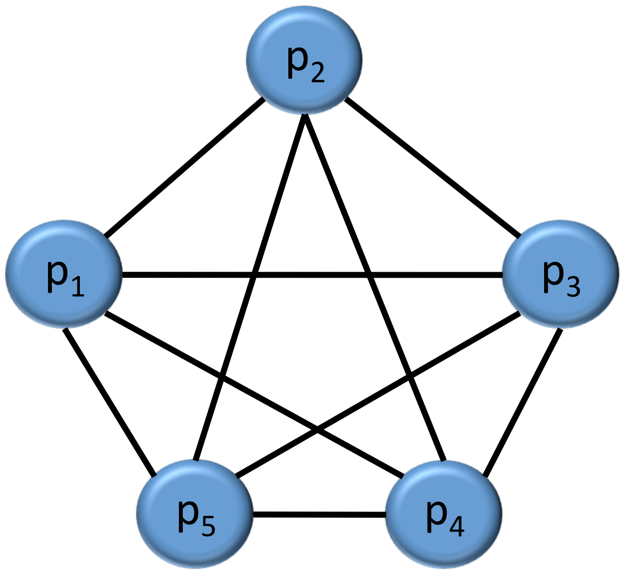
Previously . . .

- Consensus for shared memory
- Impossibility of consensus using atomic read-write registers
- Consensus hierarchy
- RMW instructions

- Today:
- Leave shared memory behind for a while
- Turn to message passing concurrency

Consensus #4: Synchronous Systems

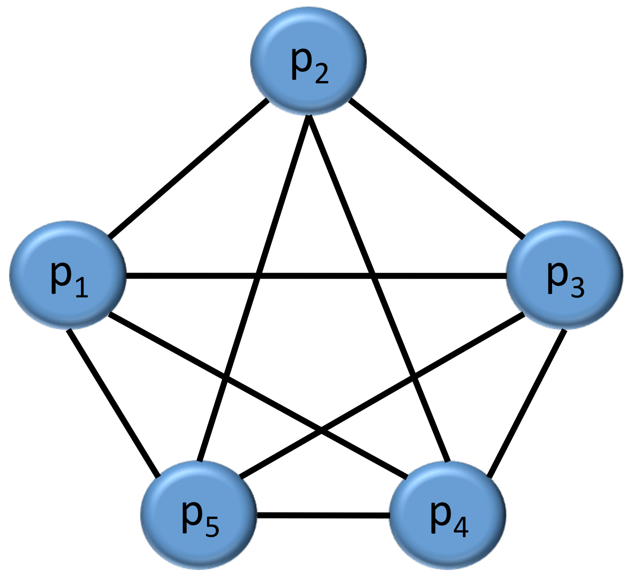
- One can sometimes tell if a processor had crashed
  - Timeouts
  - Broken TCP connections
  - Heartbeats
- Can one solve consensus at least in synchronous systems?
- Model
  - All communication occurs in synchronous rounds
  - Complete communication graph



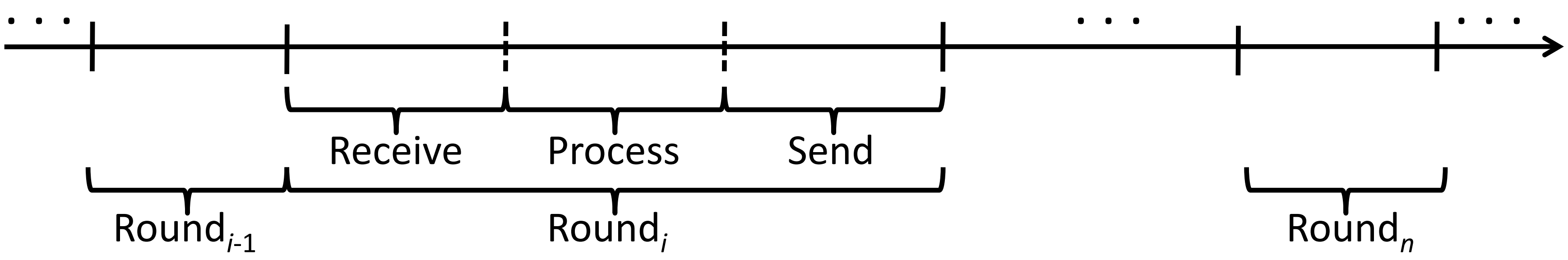
Reading: Attiya, Welch ch 5 until 5.3

Synchronous Systems - Model

- Model
  - All communication occurs in synchronous rounds
  - Complete communication graph

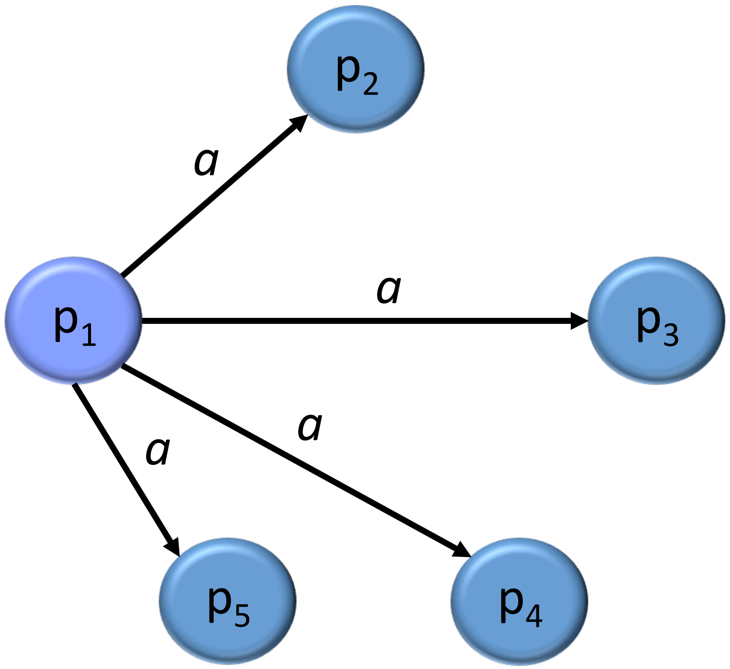


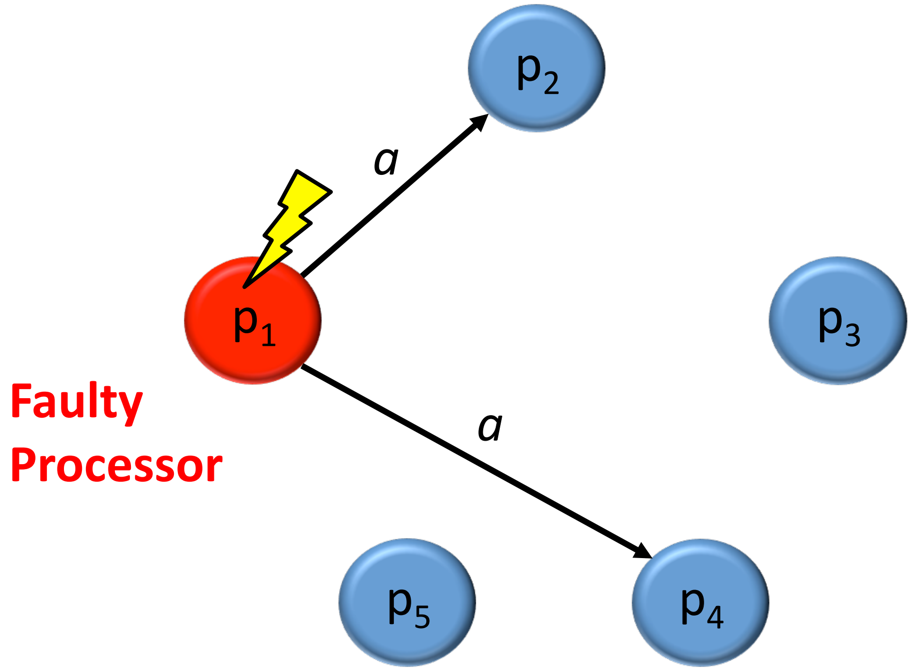
- Synchronous system:
  - Roughly synchronized rounds
  - Message passing, bounded delay
  - Each round: Receive, process, send



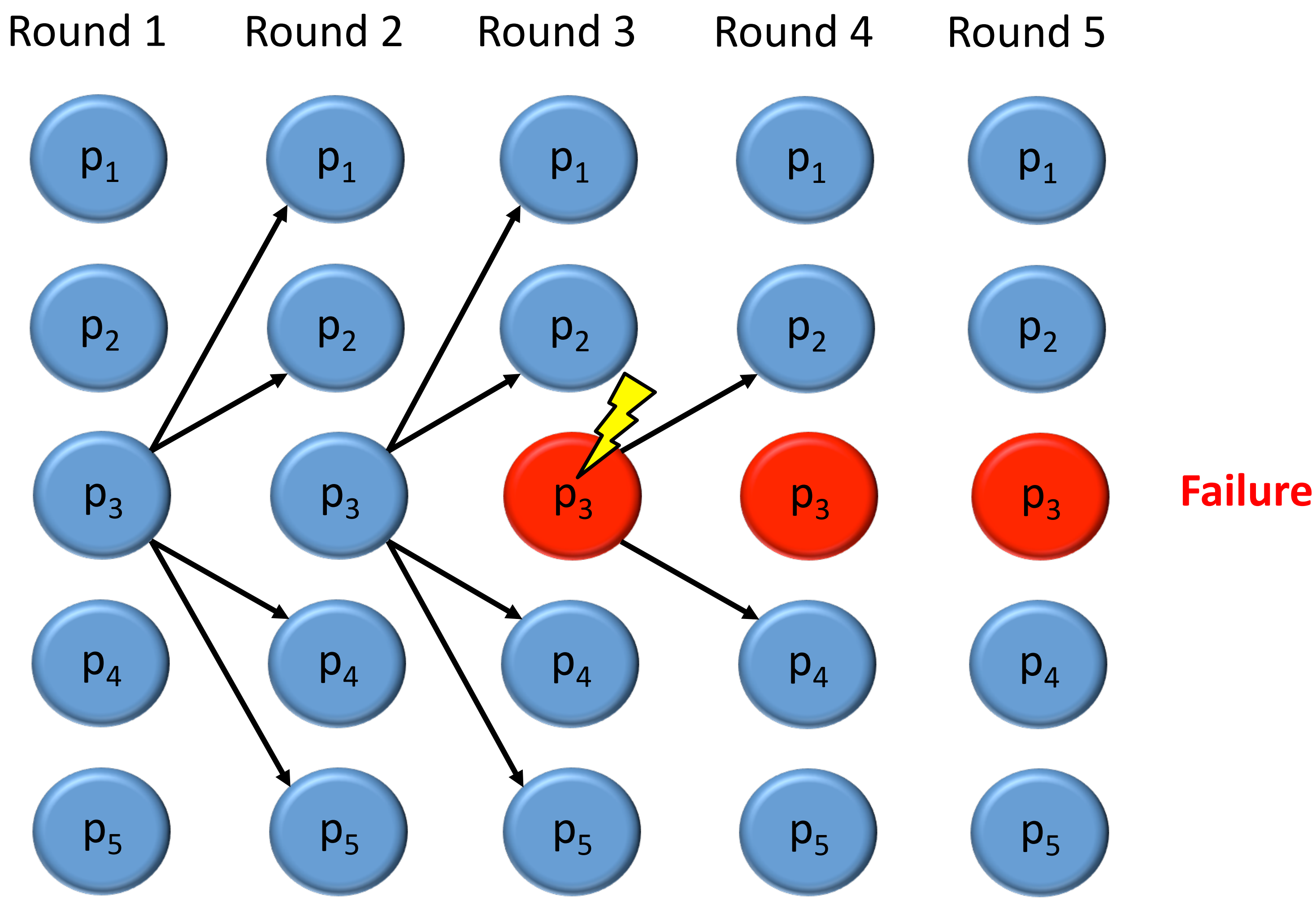
Crash Failures

- Broadcast: Send a Message to All Processes 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





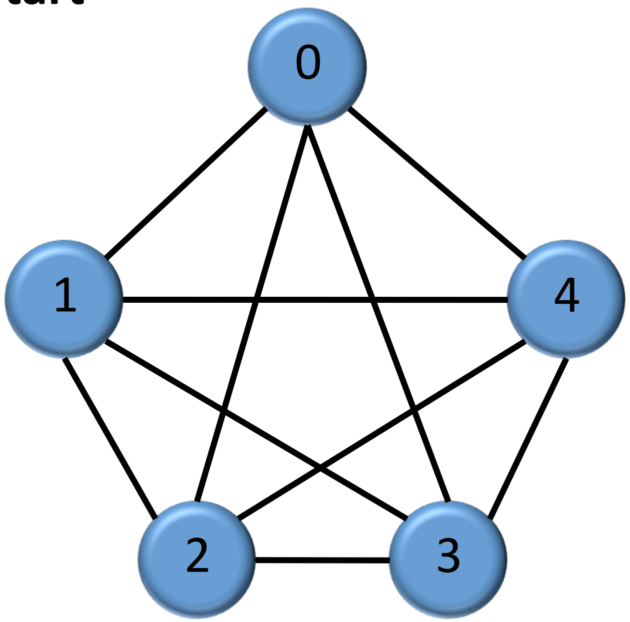
After a Failure, the Process Disappears from the Network



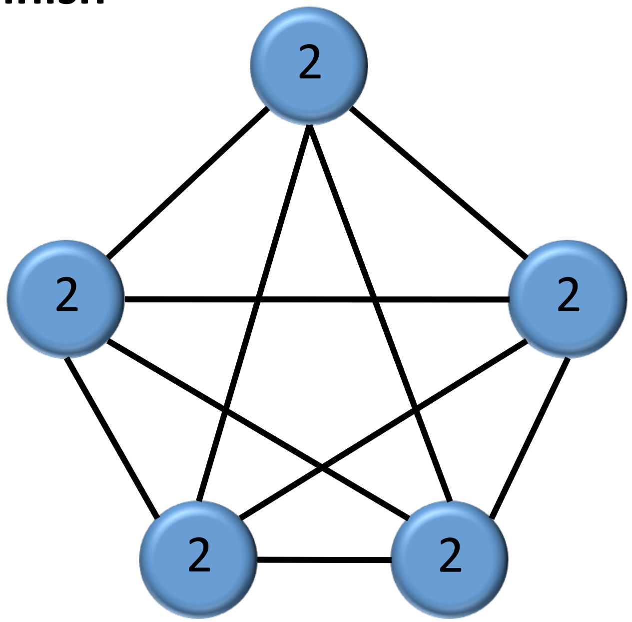
Consensus Definition

- Everybody has an initial value
- Everybody must decide on the same value

Start



Finish



- **Validity condition**:
  If everybody starts with the same value, they must decide on that value

A Simple Consensus Algorithm

Each process:

1. Broadcast own value

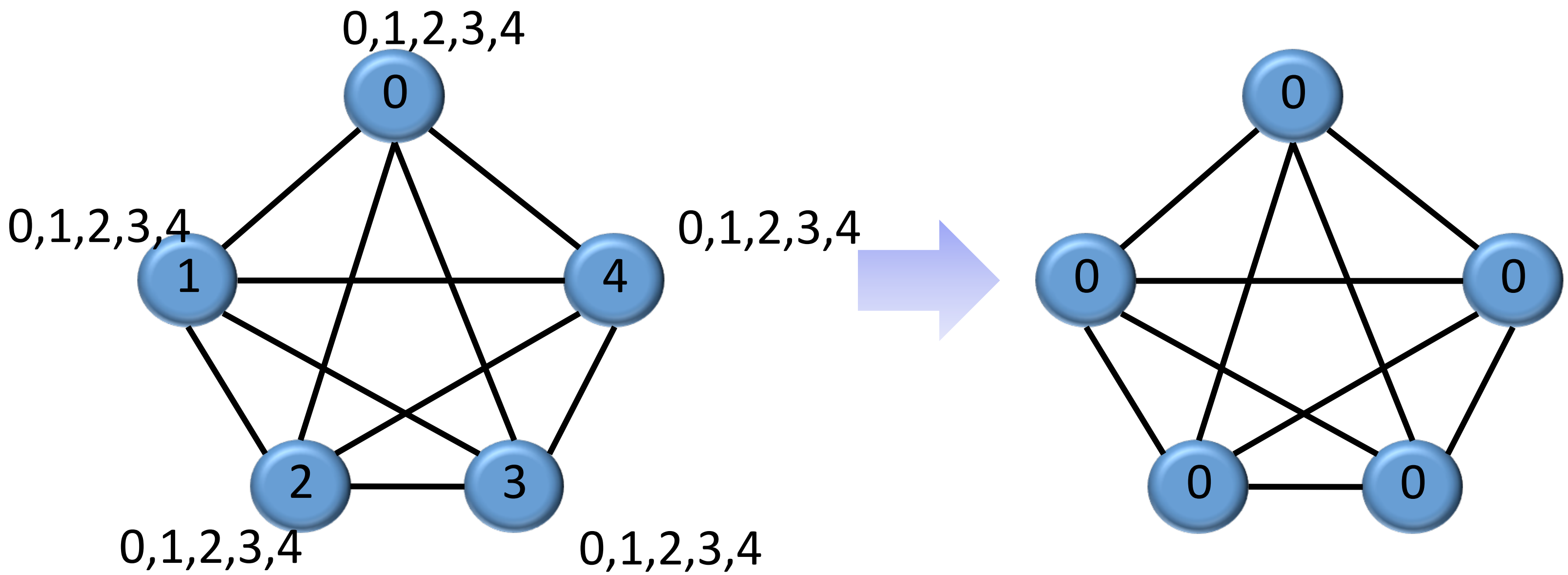
2. Decide on the minimum of all received values

Including the own value

Note that only one round is needed!

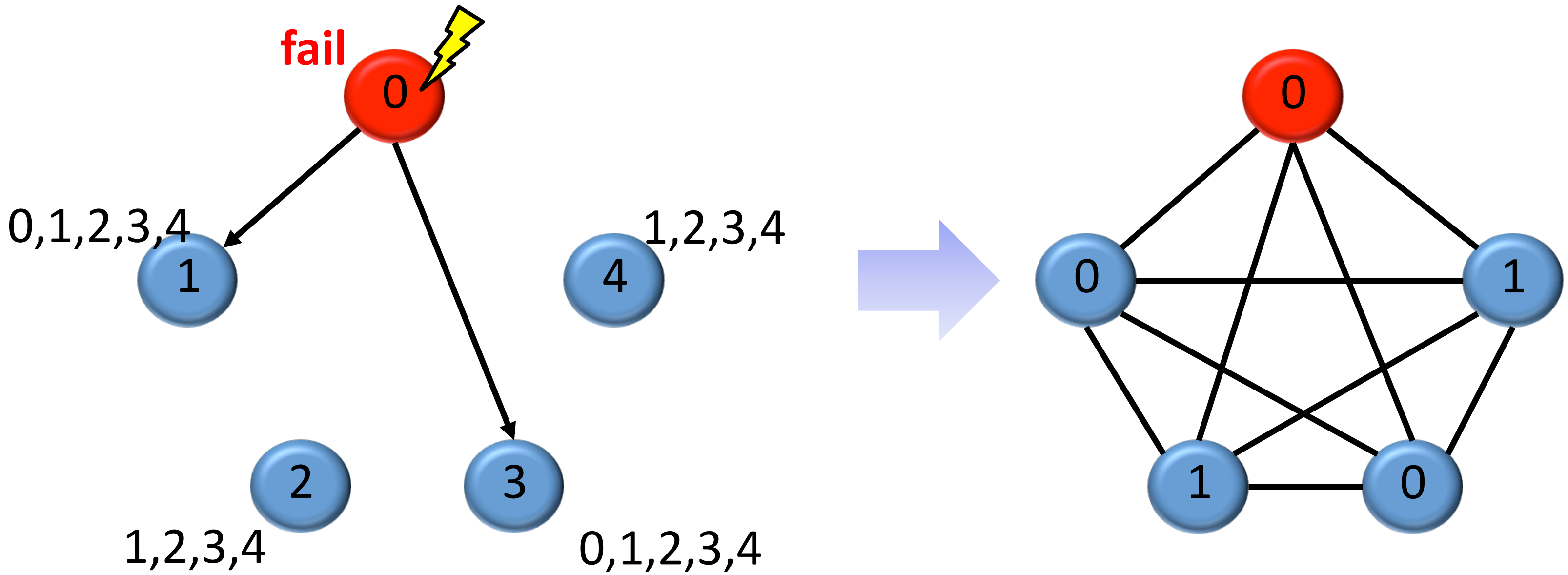
No Failures

- Broadcast values and decide on minimum → Consensus!
- Validity condition is satisfied: If everybody starts with the same initial value, everybody sticks to that value (minimum)



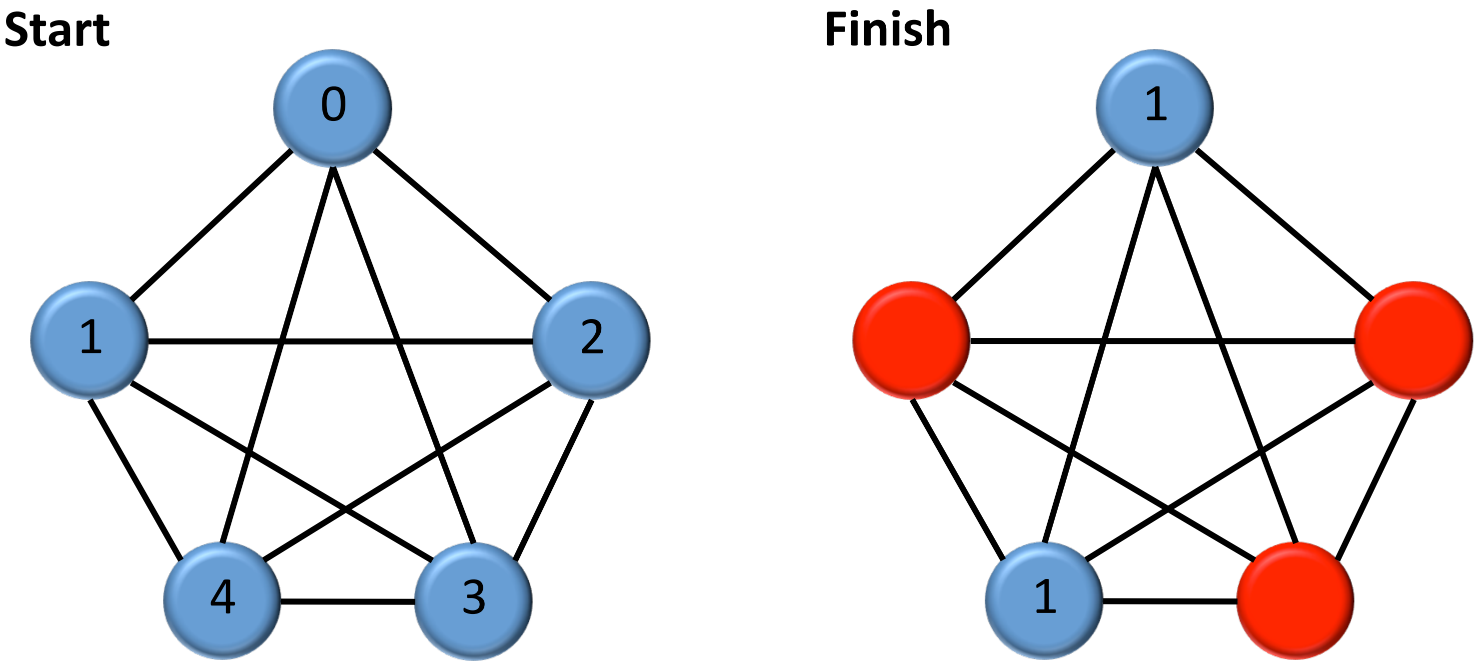
Failures

- The failed processor doesn't broadcast its value to all processors
- Decide on minimum - no consensus!



An *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 algorithm:



- **Refined validity condition**:

  If everybody starts with the same value, they must decide on that value

  All non-faulty processes eventually decide

An *f*-resilient Consensus Algorithm

Algorithm FloodSet:

Each process:

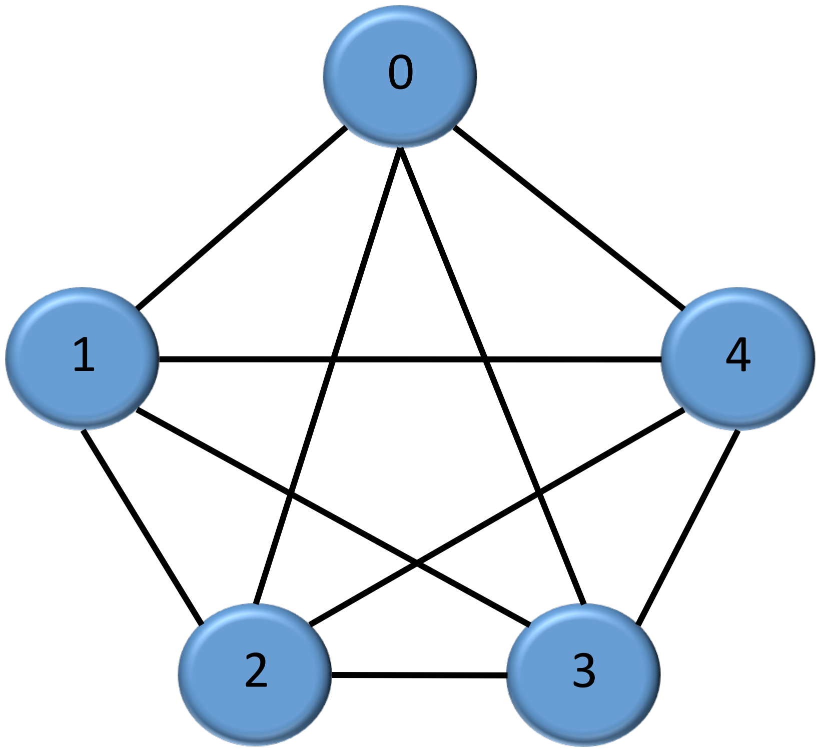
Round 1:
Broadcast own value

Round 2 to round *f*+1:
Broadcast all newly received values

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

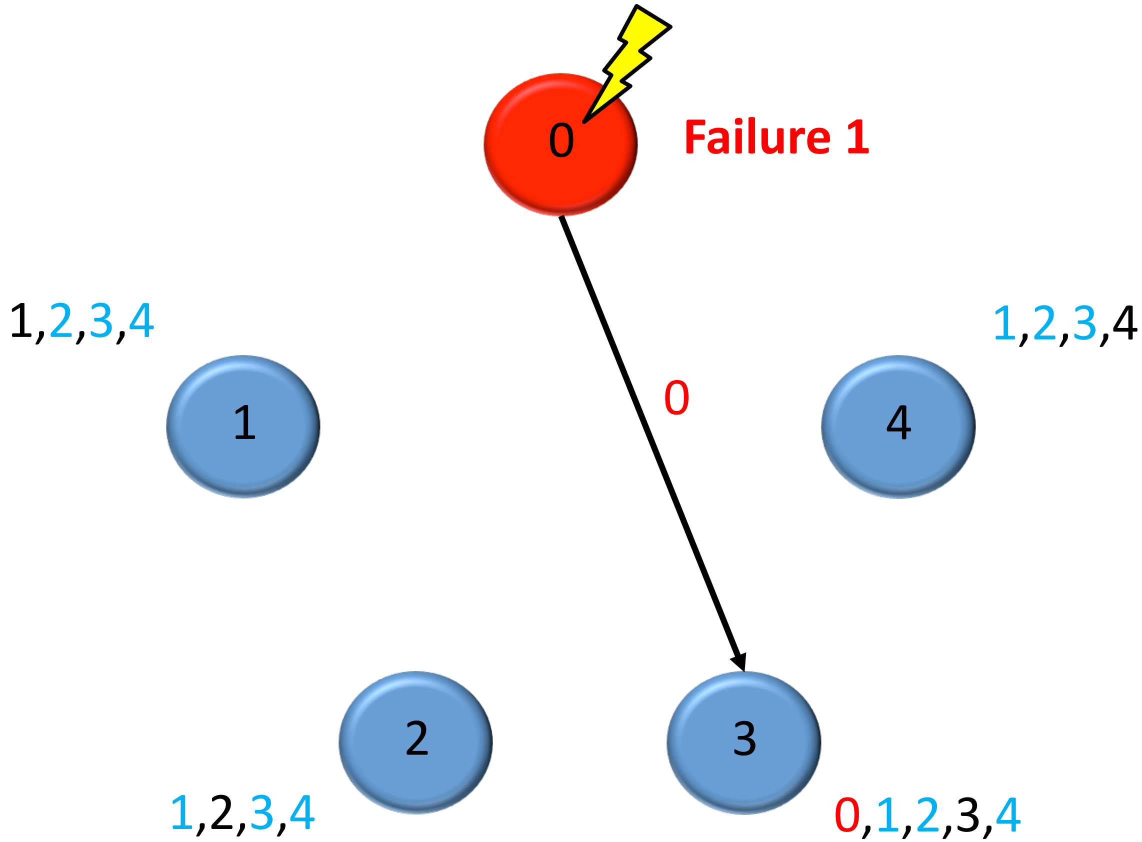
An *f*-resilient Consensus Algorithm

- Example: $f = 2$ failures, $f + 1 = 3$ rounds needed



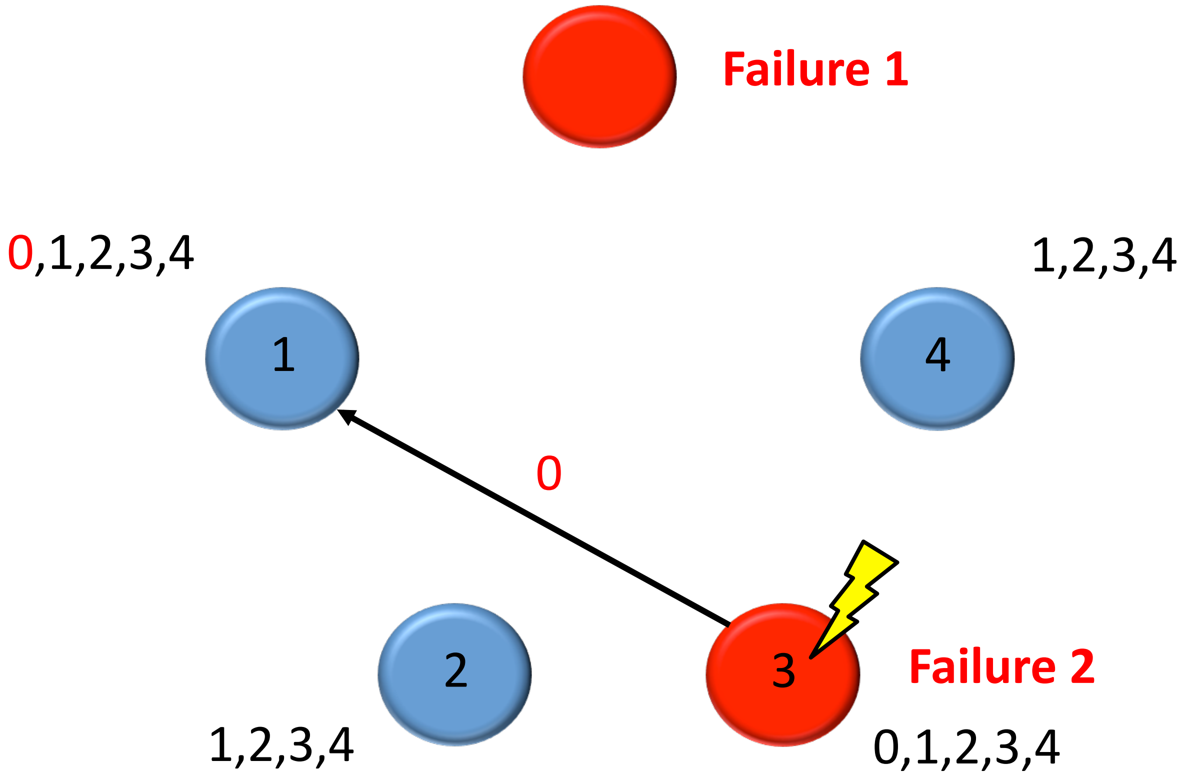
An *f*-resilient Consensus Algorithm

- Round 1: Broadcast all values to everybody



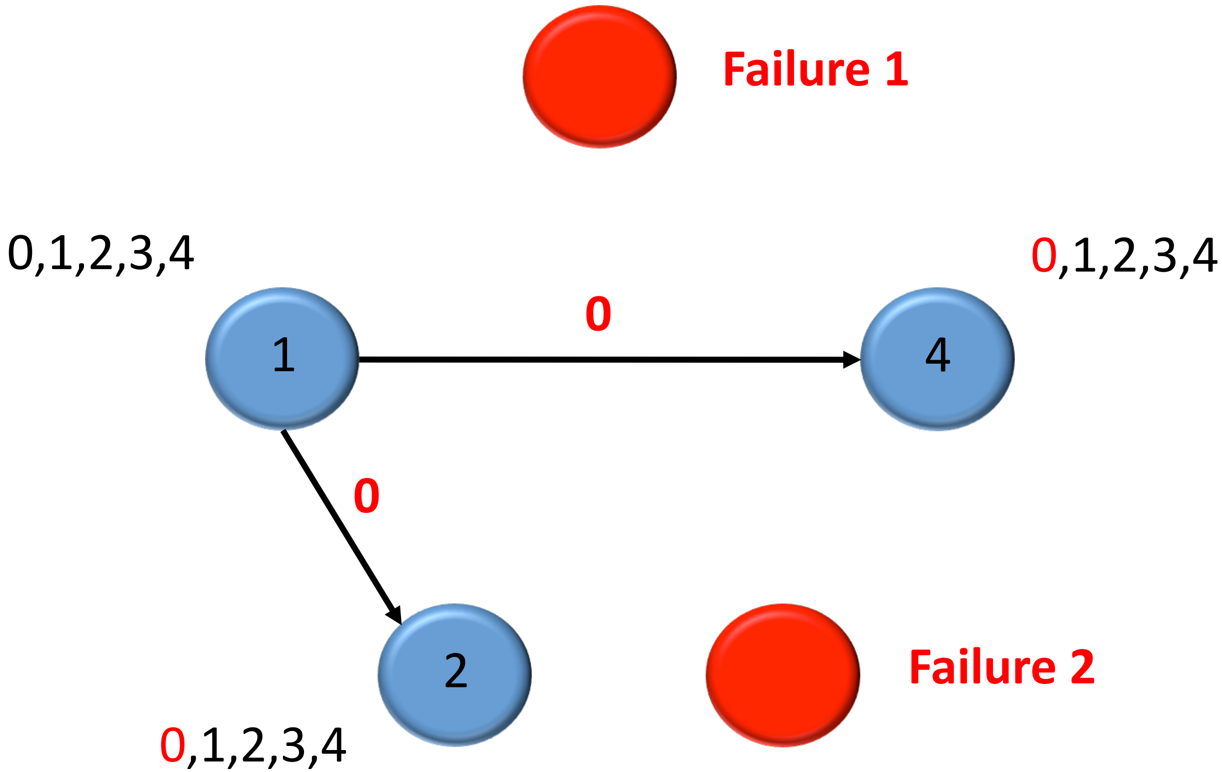
An *f*-resilient Consensus Algorithm

- Round 2: Broadcast all new values to everybody



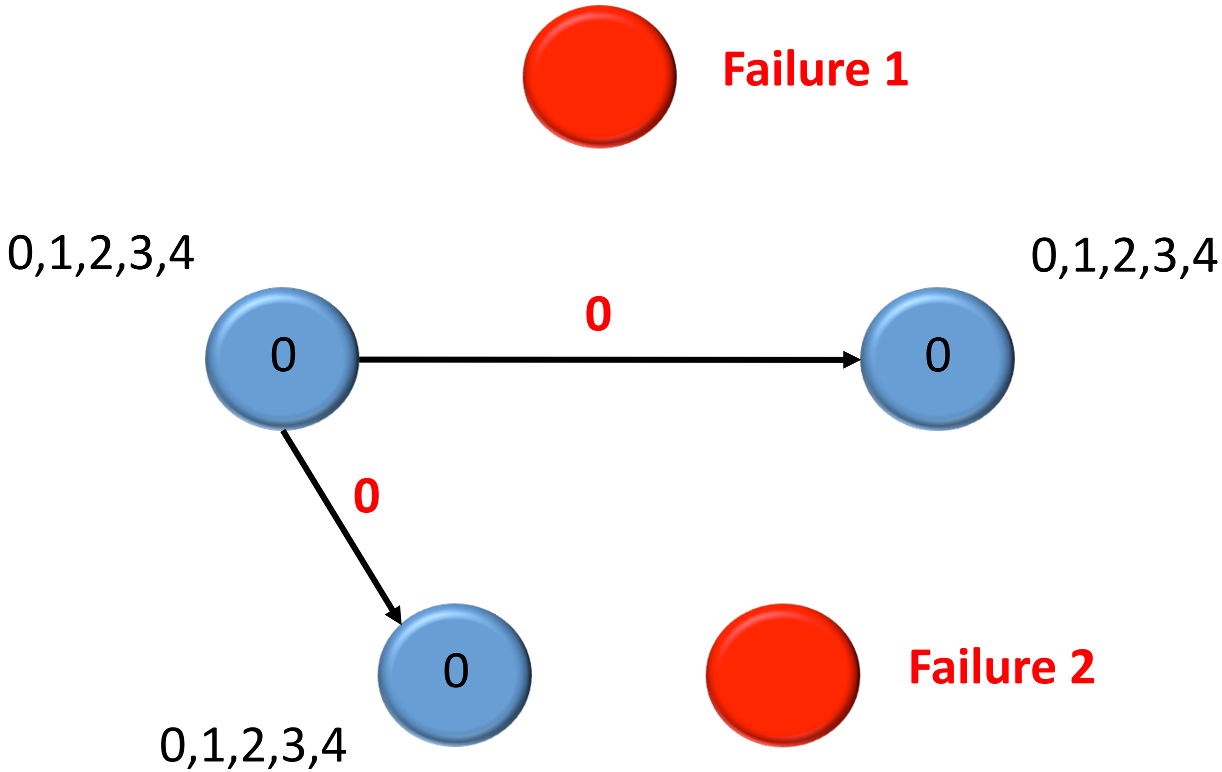
An *f*-resilient Consensus Algorithm

- Round 3: Broadcast all new values to everybody



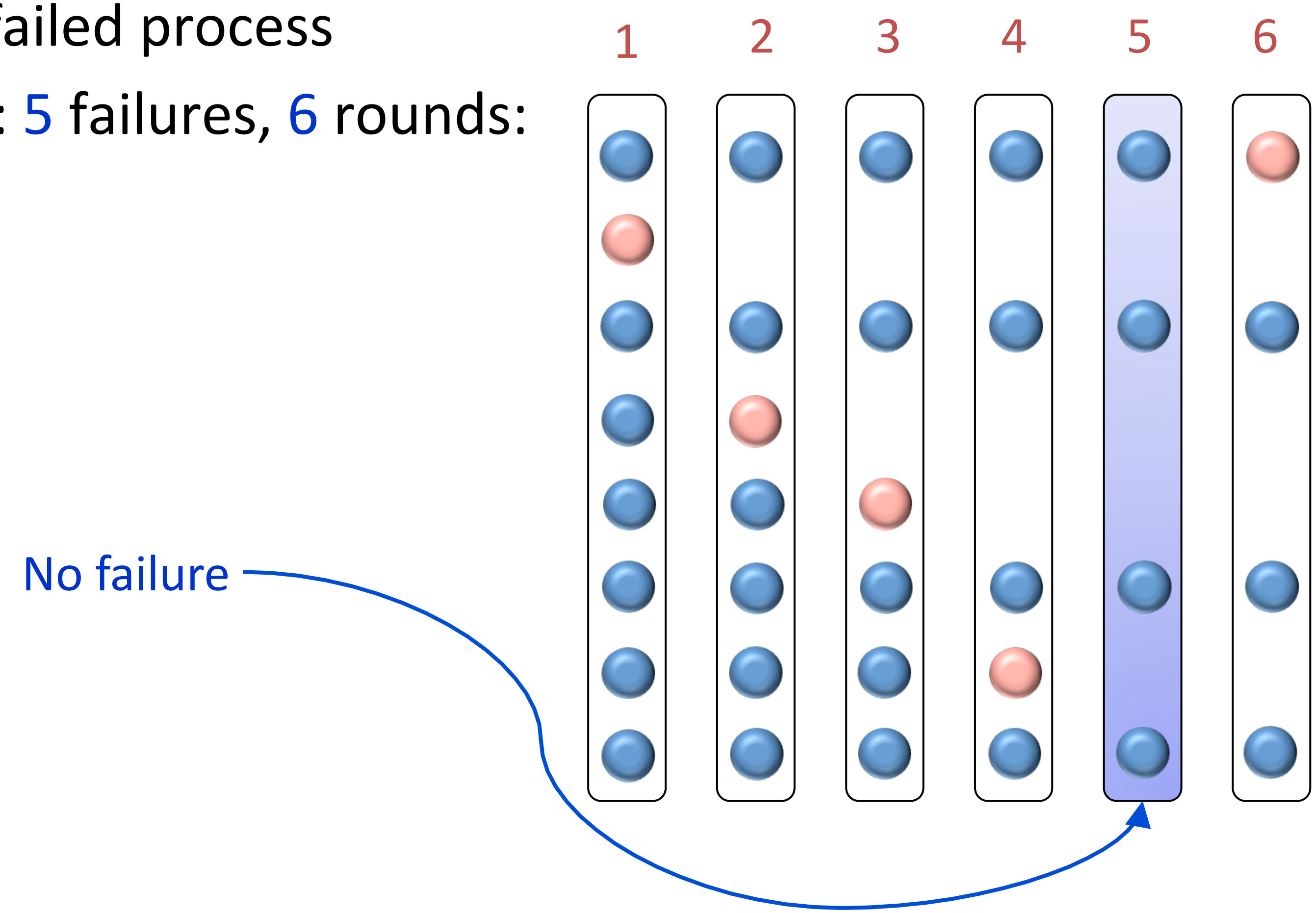
An *f*-resilient Consensus Algorithm

- Decide on minimum → Consensus!



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

Exercises

Exercise 1

- The message complexity of an algorithm is the number of messages passed along some link in the process graph
- What is the message complexity of the FloodSet algorithm?

Lower Bound, Crash Failures

Theorem

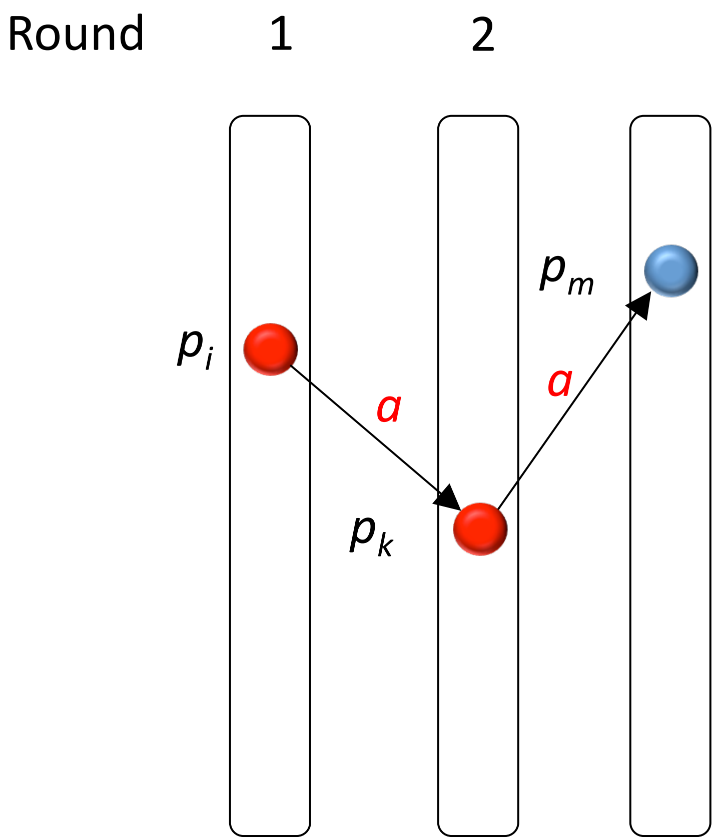
Any *f*-resilient consensus algorithm requires at least *f* + 1 rounds

Note that this is
not a formal proof!

Proof sketch:

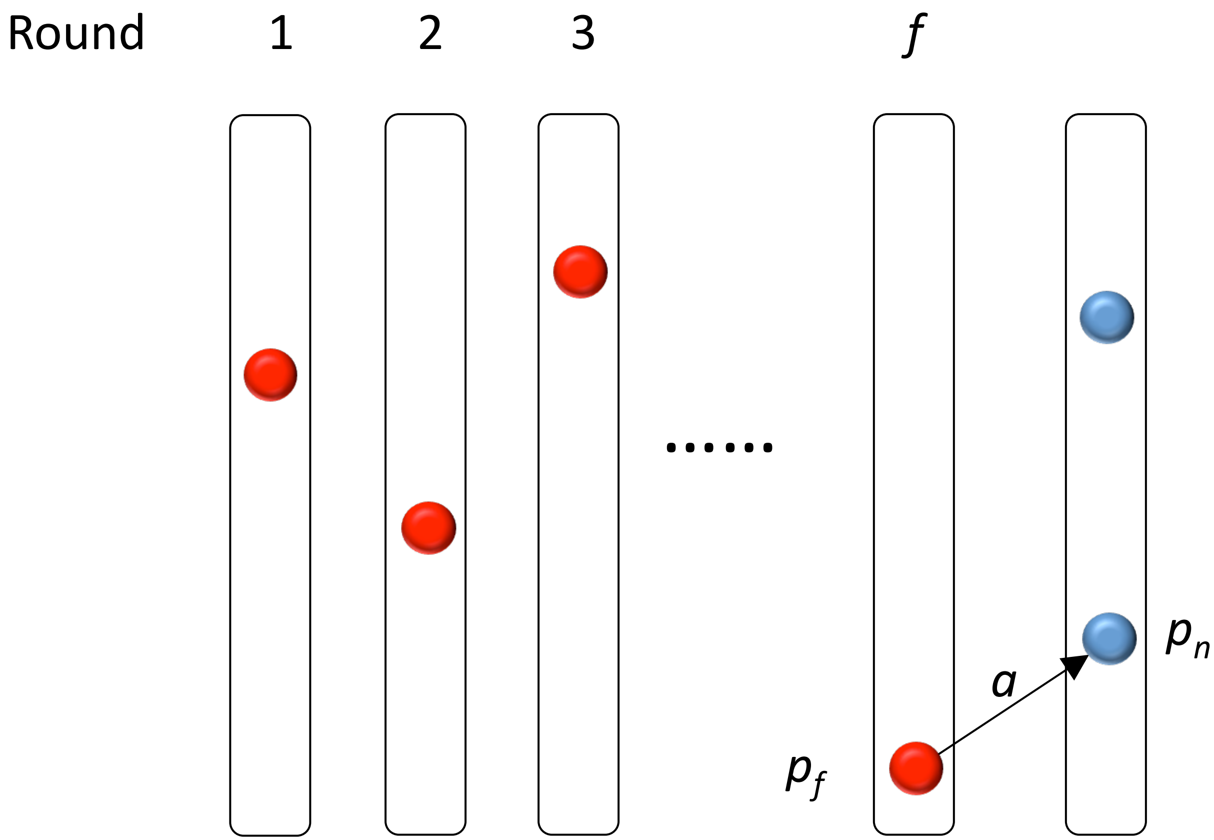
- Assume for contradiction that *f* or less rounds are enough
- Worst-case scenario: There is a process that fails in each round

Worst-case Scenario



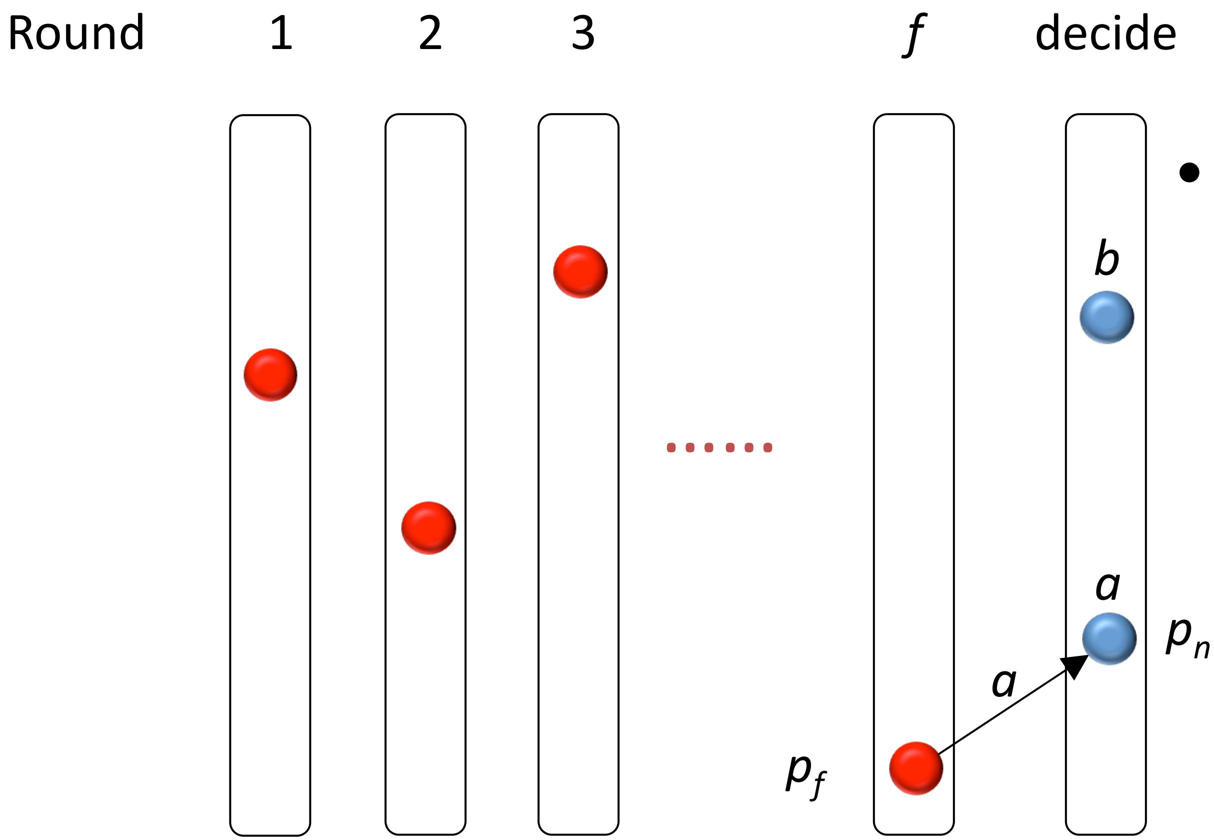
- Before process p_i fails, it sends its value a only to one process p_k
- Before process p_k fails, it sends its value a to only one process p_m

Worst-case Scenario



- At the end of round *f* only one process p_n knows about value *a*

Worst-case Scenario

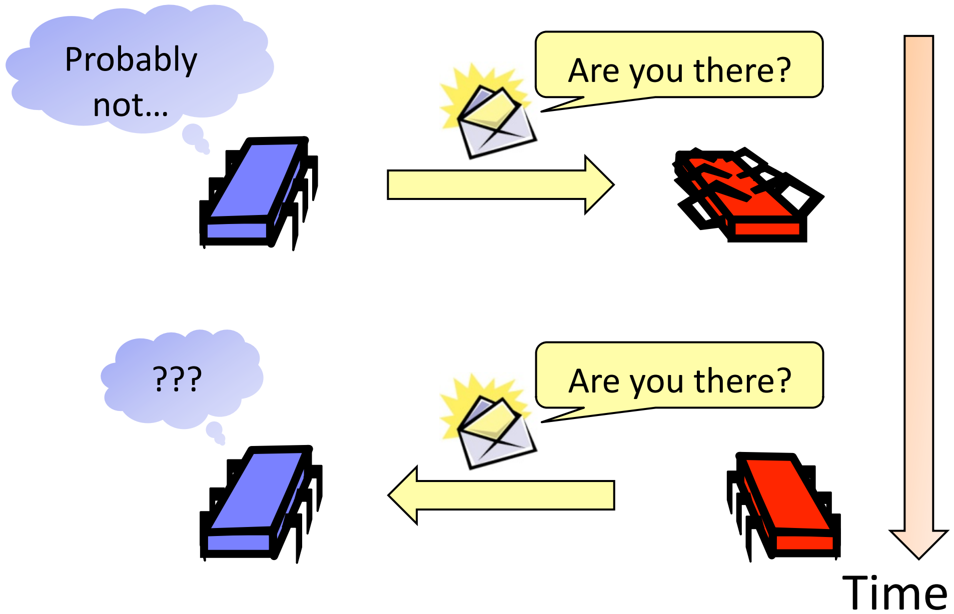


- Process p_n may decide on *a* and all other processes may decide on another value *b*

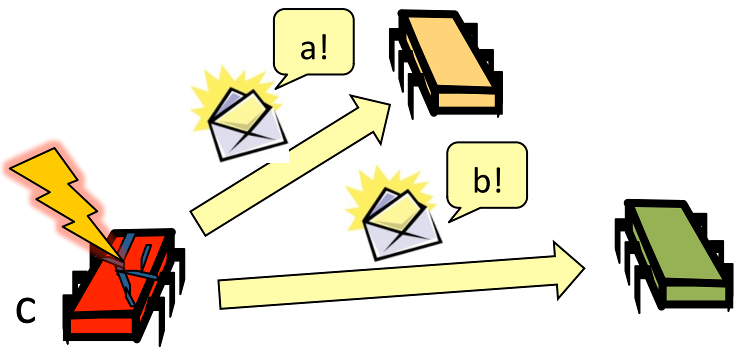
- Therefore *f* rounds are not enough → At least *f*+1 rounds are needed

Arbitrary Behaviour

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



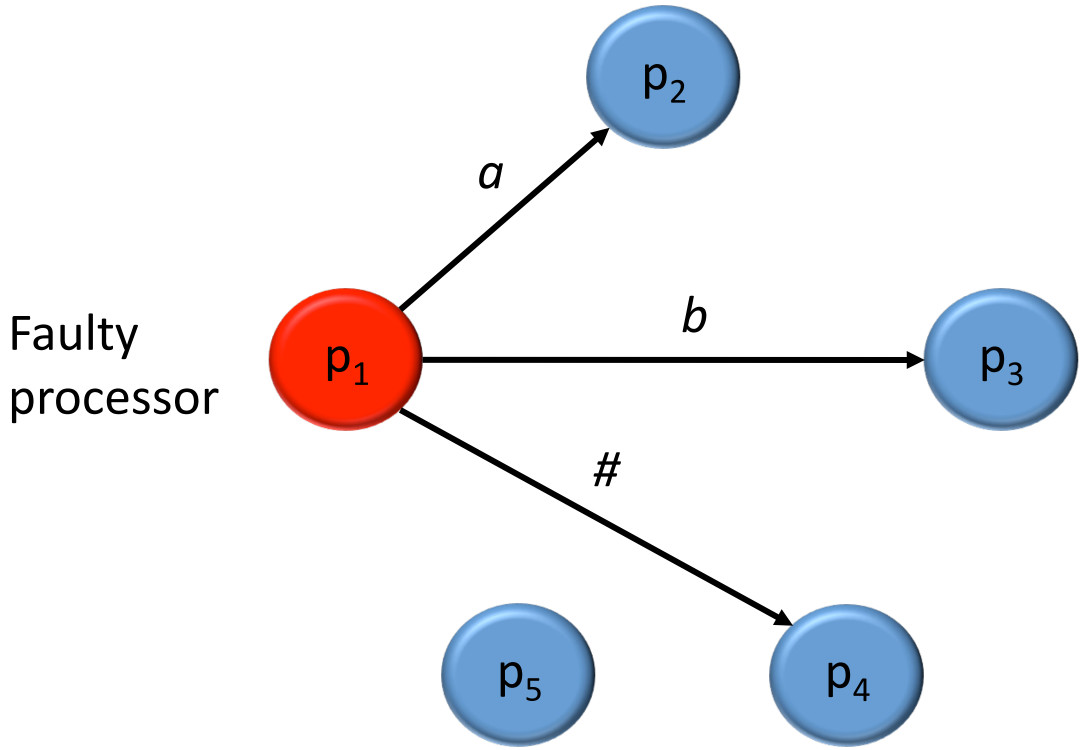
- Maybe the processes are damaged:



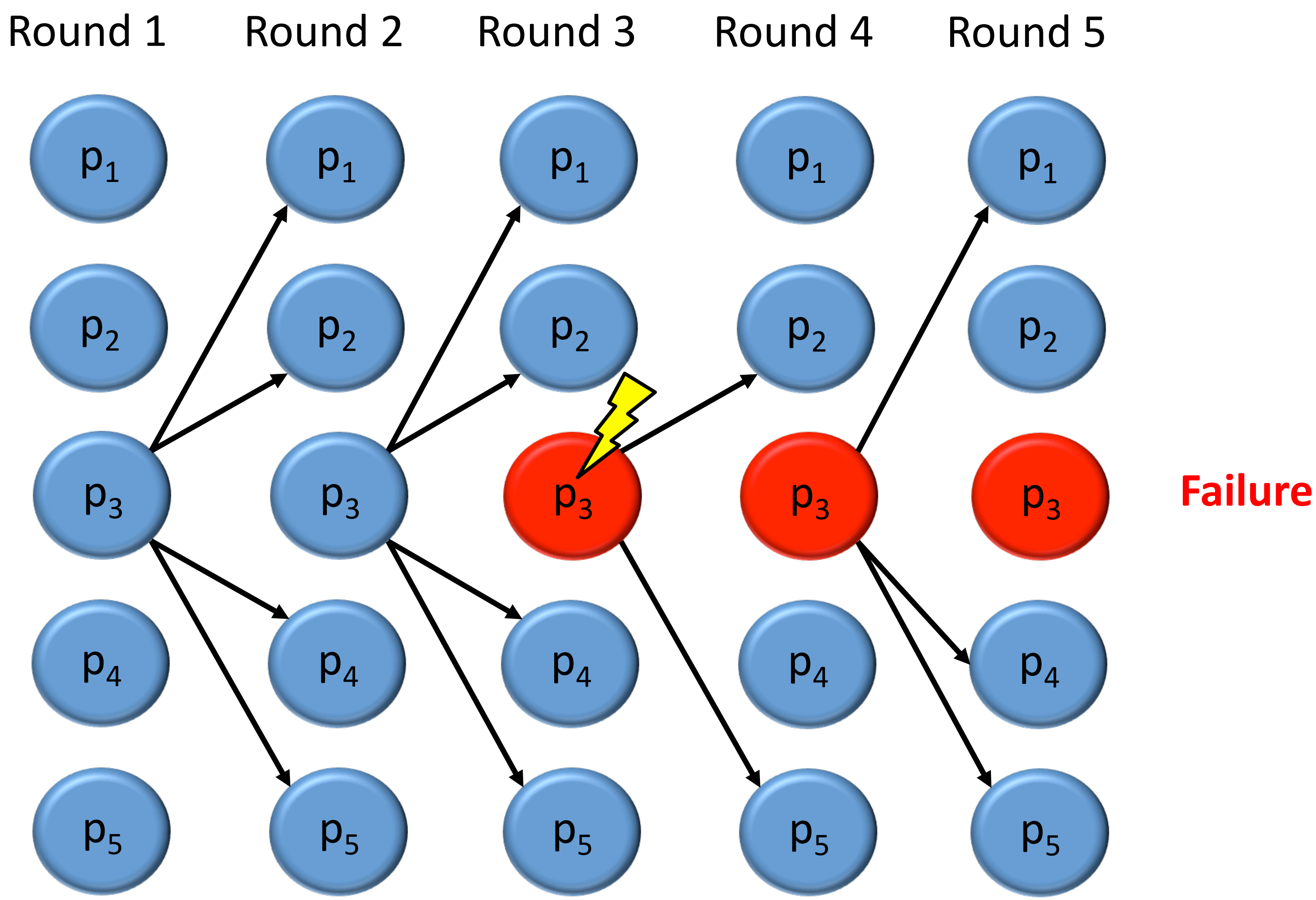
- Maybe the processes are malicious:

Consensus #5: Byzantine Failures

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



After a Failure, the Process Remains in the 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 condition: If all non-faulty processes start with the same value, then all non-faulty processes decide on that value

- 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)?

Lower Bound, Byzantine Failures

Theorem

There is no *f*-resilient algorithm for n processes, where $f \geq n/3$

Proof outline:

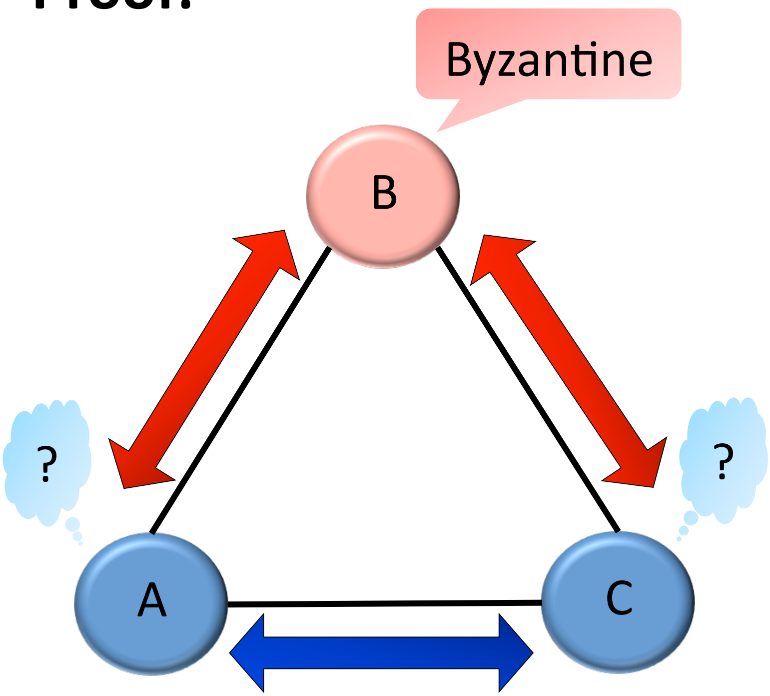
- First, we prove the 3 processes case
- The general case can be proved by reducing it to the 3 processes case

The 3 Processes Case

Lemma

There is no 1-resilient algorithm for 3 processes

Proof:

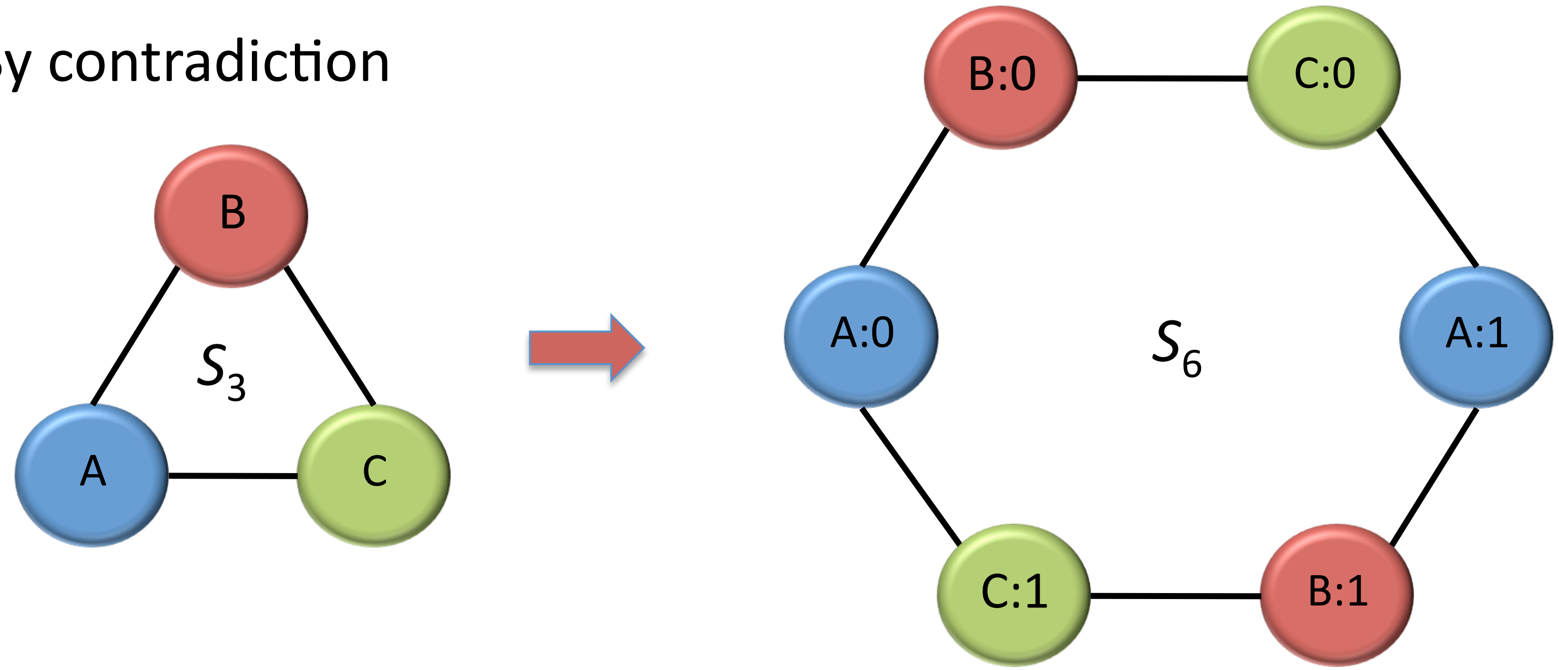


Intuition:

- Process A may also receive information from C about B's messages to C
- Process 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 of Lemma

By contradiction

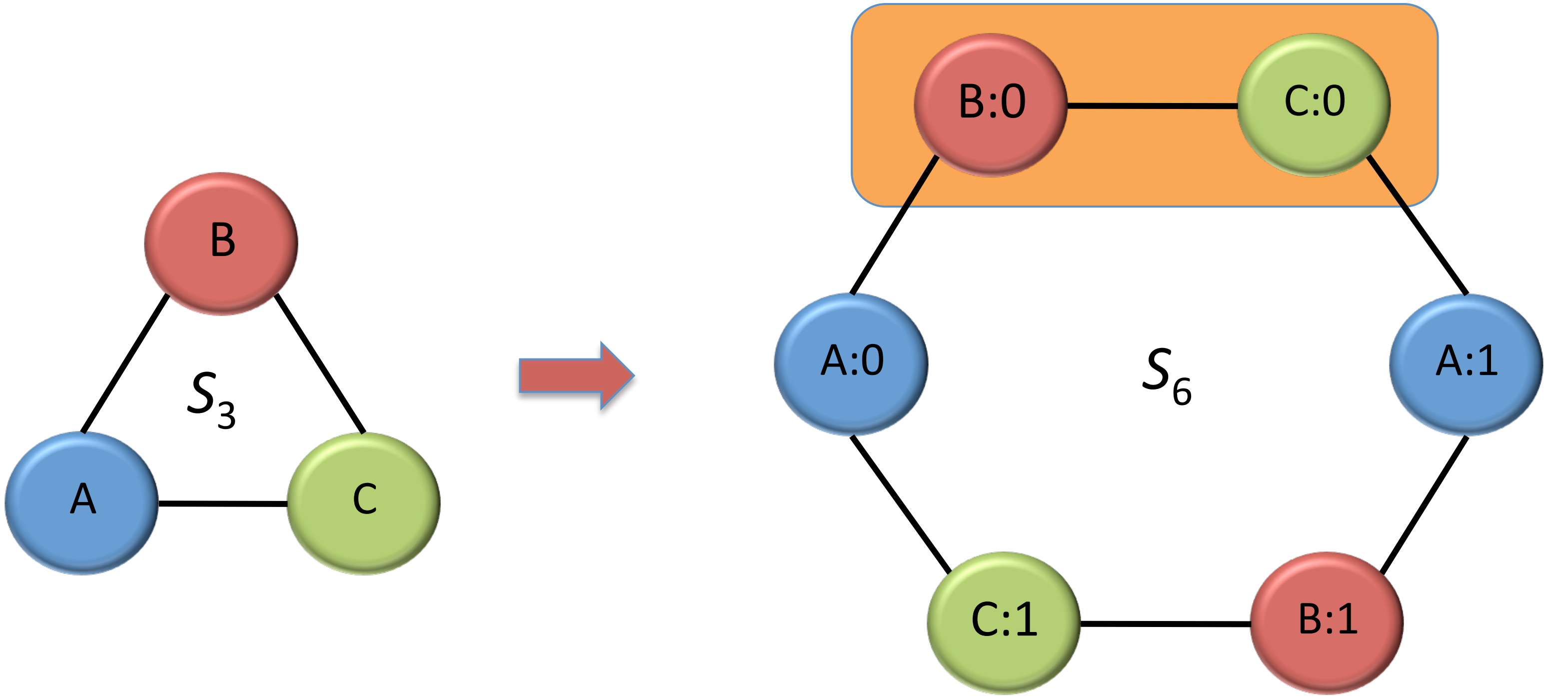


Assume three process algorithm exists, executed by A, B, C

Construct system S_6 by running each process with input 0 or 1

Let an execution of S_6 be given

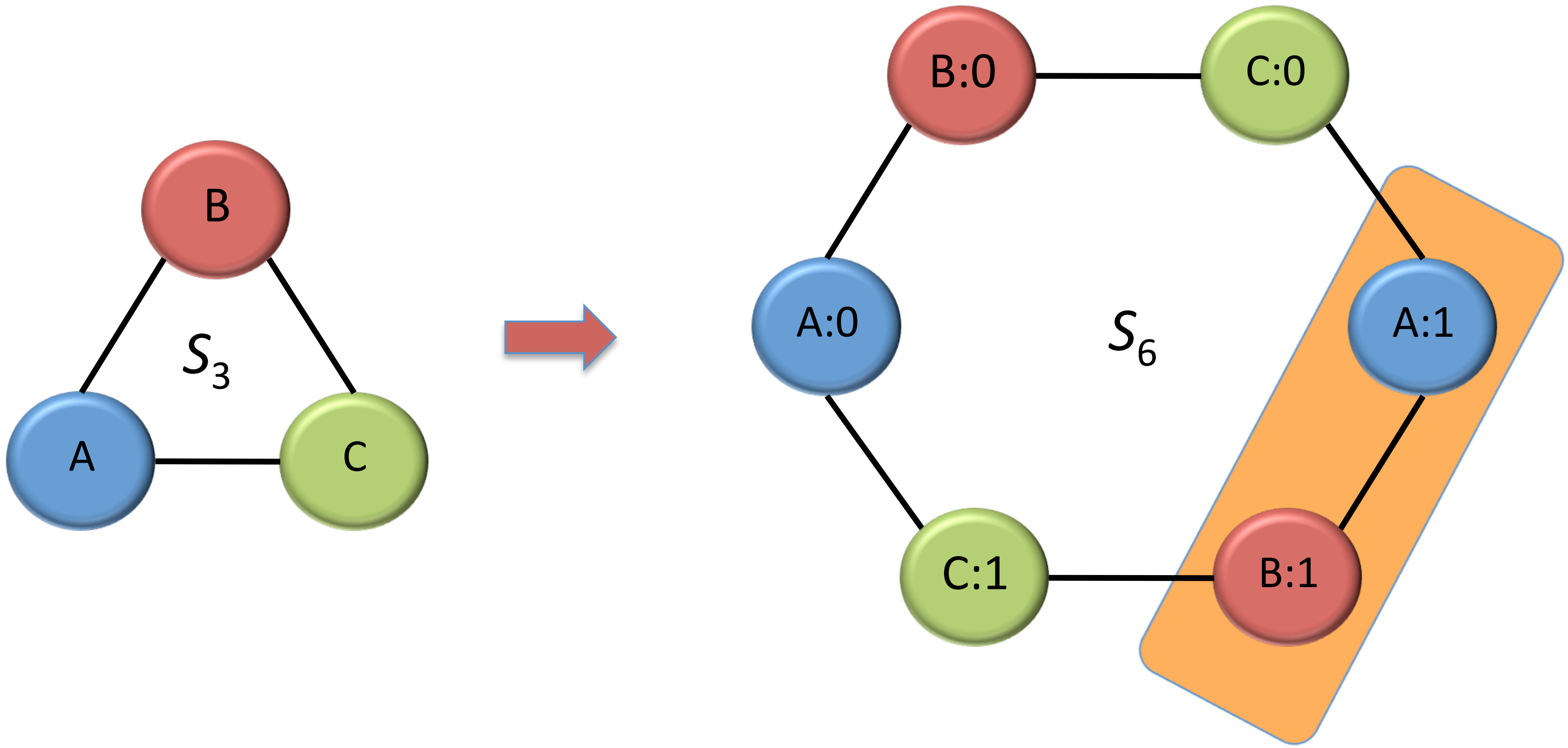
Proof of Lemma



To nodes B:0 and C:0 there is no difference between execution of S_3 and execution of S_6 – node A might be faulty

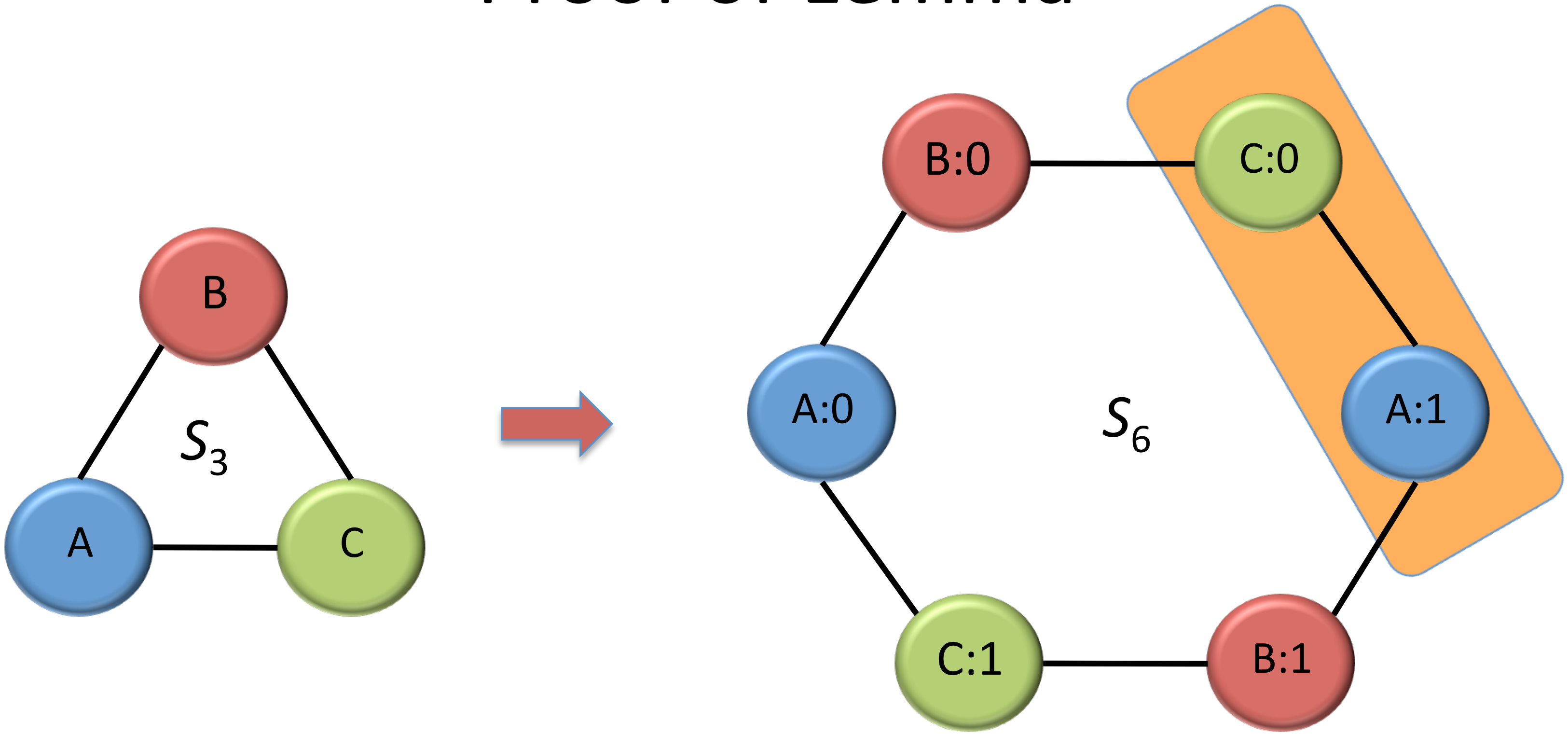
They must decide 0 in S_3 so they decide 0 in S_6 as well

Proof of Lemma



Similarly nodes A:1 and B:1 must decide 1

Proof of Lemma



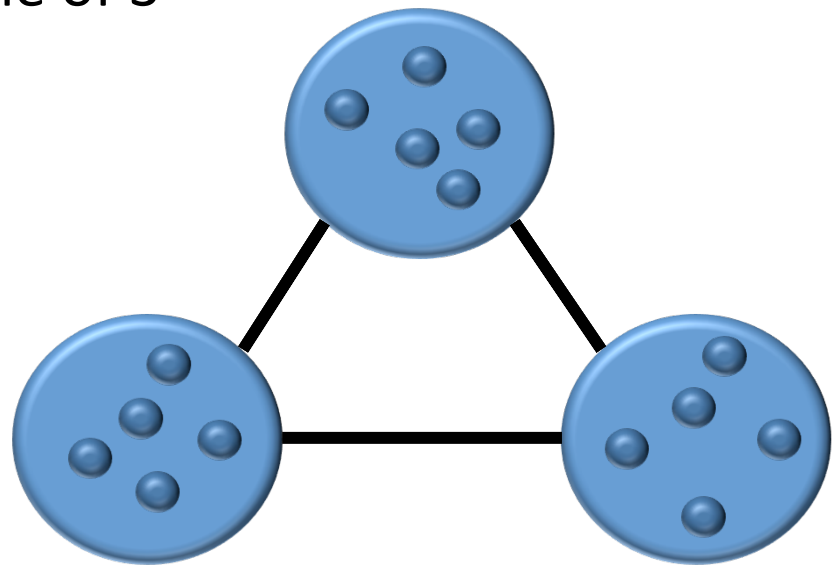
Also C:0 and A:1 cannot distinguish an execution of S_3 from an execution of S_6

S_3 solves byzantine agreement so C:0 and A:1 must decide and agree

But C:0 must decide 0 and A:1 must decide 1.

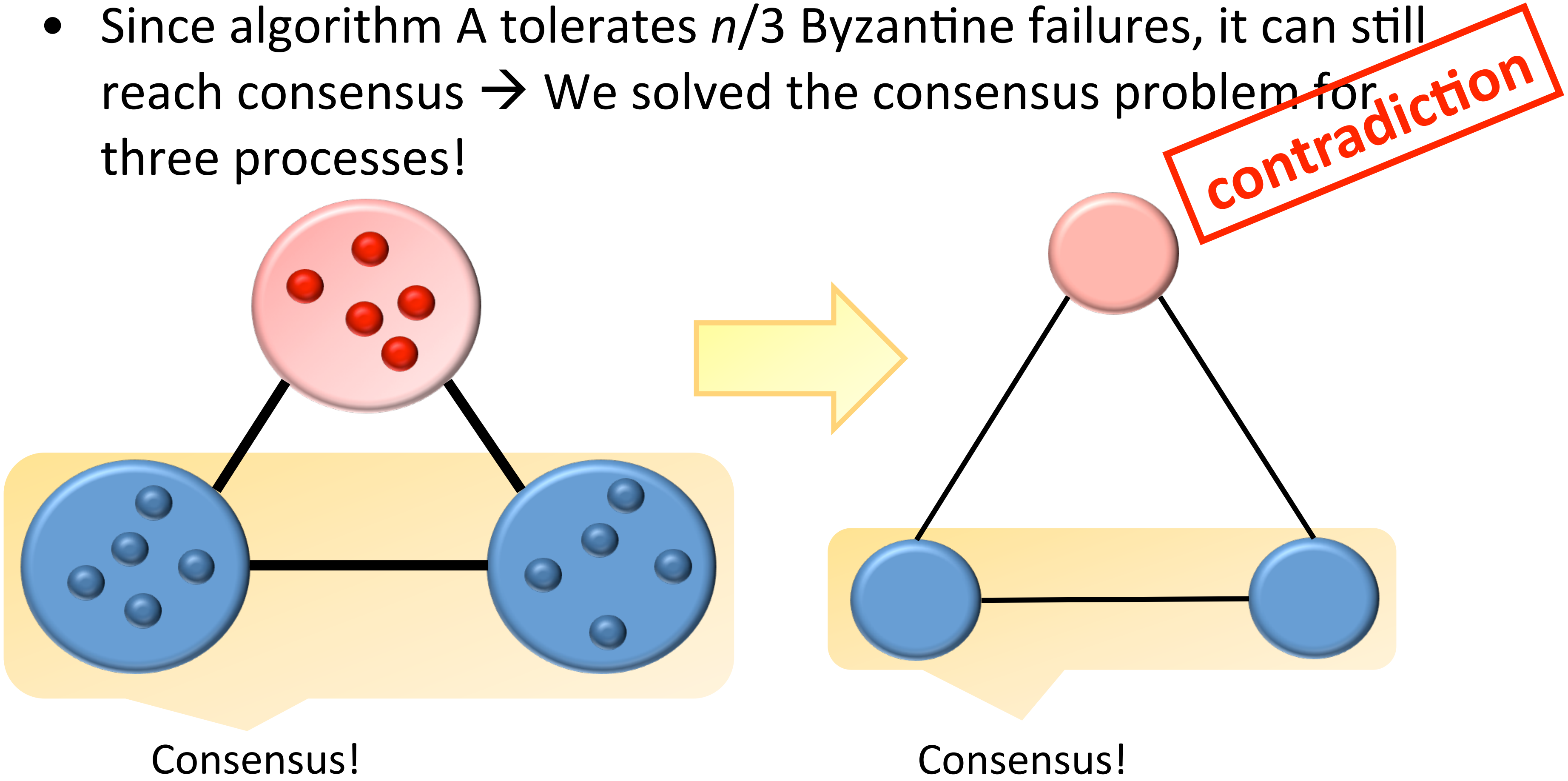
The General Case

- Assume for contradiction that there is an *f*-resilient algorithm A for *n* processes, where $f \geq n/3$
- We use this algorithm to solve the consensus algorithm for 3 processes where one process is Byzantine!
- If *n* is not evenly divisible by 3, we increase it by 1 or 2 to ensure that *n* is a multiple of 3
- We let each of the three processes simulate $n/3$ processes



The General Case

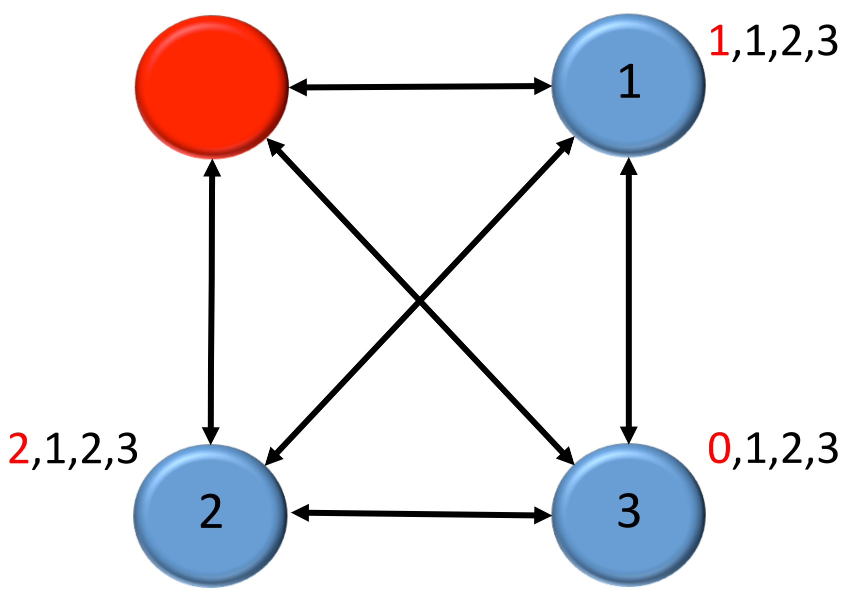
- One of the 3 processes is Byzantine → Its *n*/3 simulated processes may all behave like Byzantine processes
- Since algorithm A tolerates *n*/3 Byzantine failures, it can still reach consensus → We solved the consensus problem for three processes!



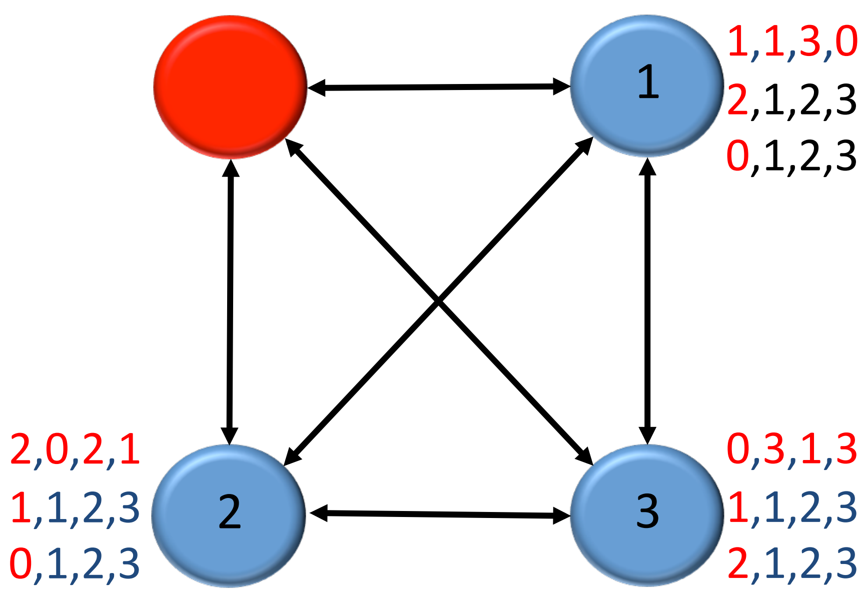
Consensus #6: A Simple Algorithm for Byzantine Agreement

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

Round 1: Exchange all values

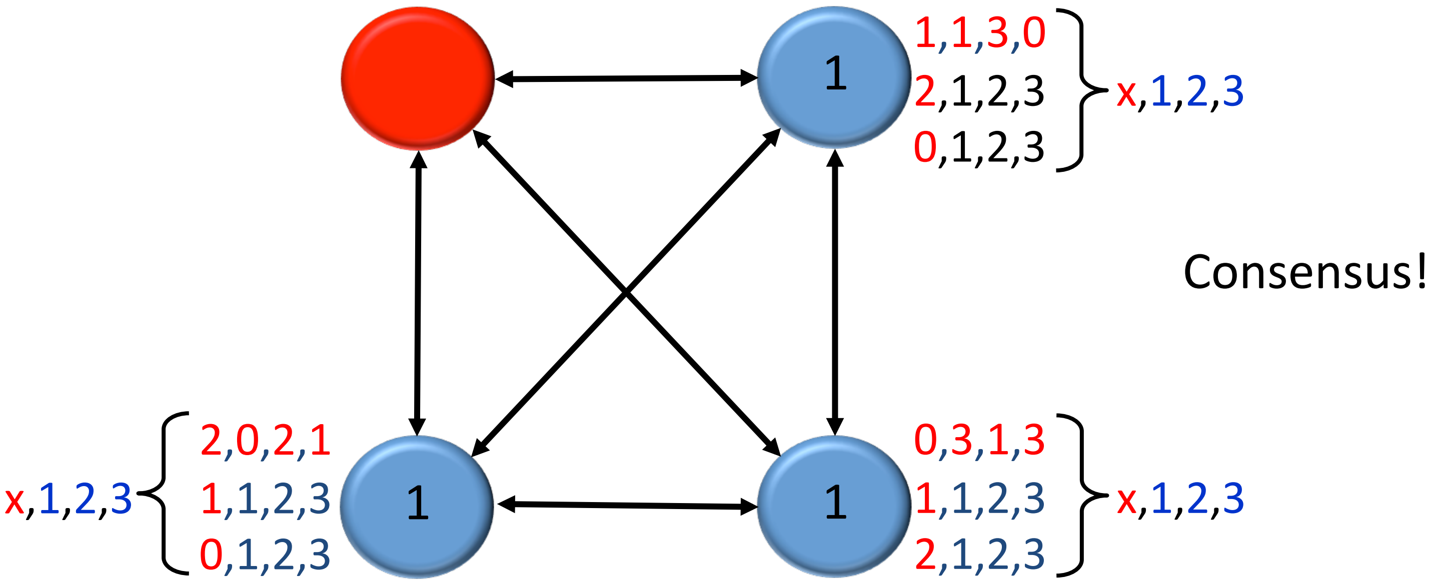


Round 2: Exchange the received info



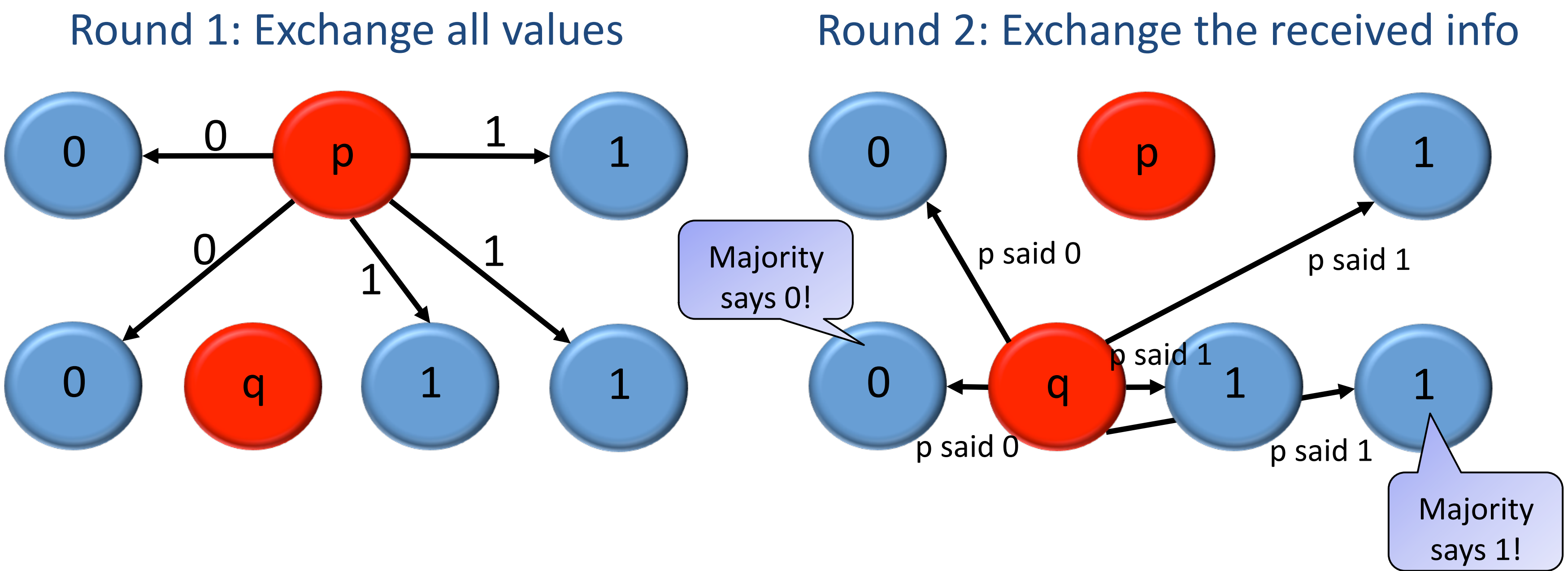
A Simple Algorithm for Byzantine Agreement

- After the second round each node has received 12 values, 3 for each of the 4 input values. If at least 2 of 3 values are equal, this value is accepted. If all 3 values are different, the value is discarded
- The node then decides on the minimum accepted value



A Simple Algorithm for Byzantine Agreement

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



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

A Simple Algorithm for Byzantine Agreement

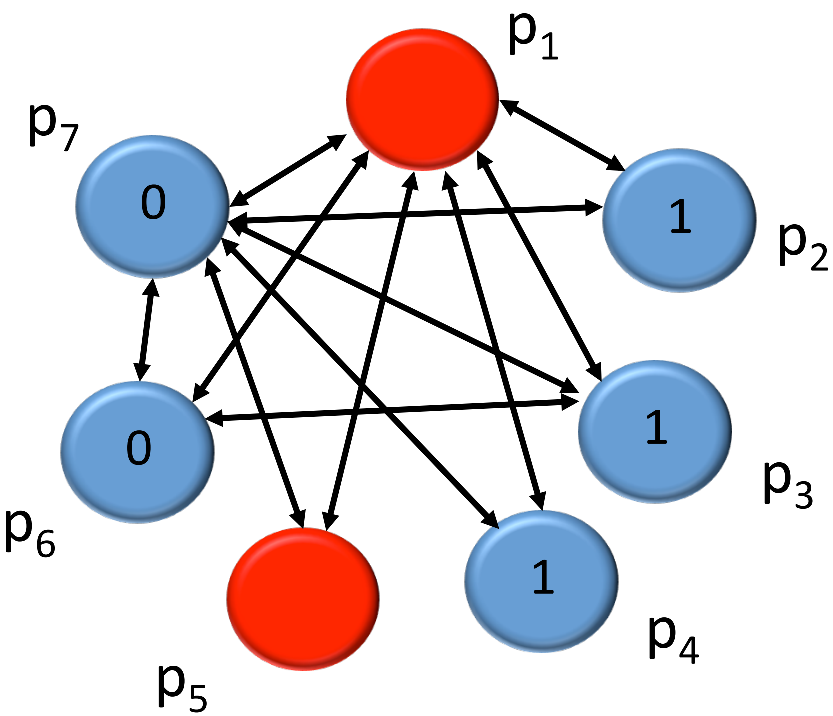
- The solution is simple: Again exchange all information!
- This way, the processes learn that a majority thinks that q gave inconsistent information about p → q can be excluded, and also p if it also gave inconsistent information (about q).
- If f=2 and n > 6, consensus can be reached in 3 rounds!
- In fact, the algorithm

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

solves the problem for any f and any $n > 3f$

Pease, Shostak, Lamport: Reaching Agreement in the Presence of Faults, JACM vol 27, 1980

Round 1: Exchange All Values



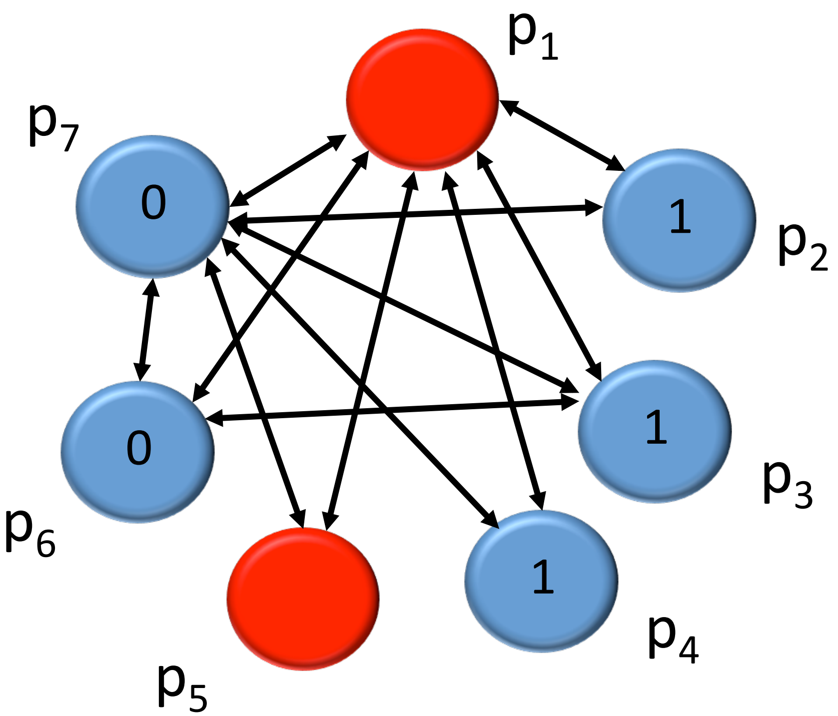
Etc. complete graph

p_2: p_1 said 1
p_3 said 1
p_4 said 1
p_5 said 1
p_6 said 0
p_7 said 0

p_7: p_1 said 0
p_2 said 1
p_3 said 1
p_4 said 1
p_5 said 0
p_6 said 0
p_7 said 0

… etc. …

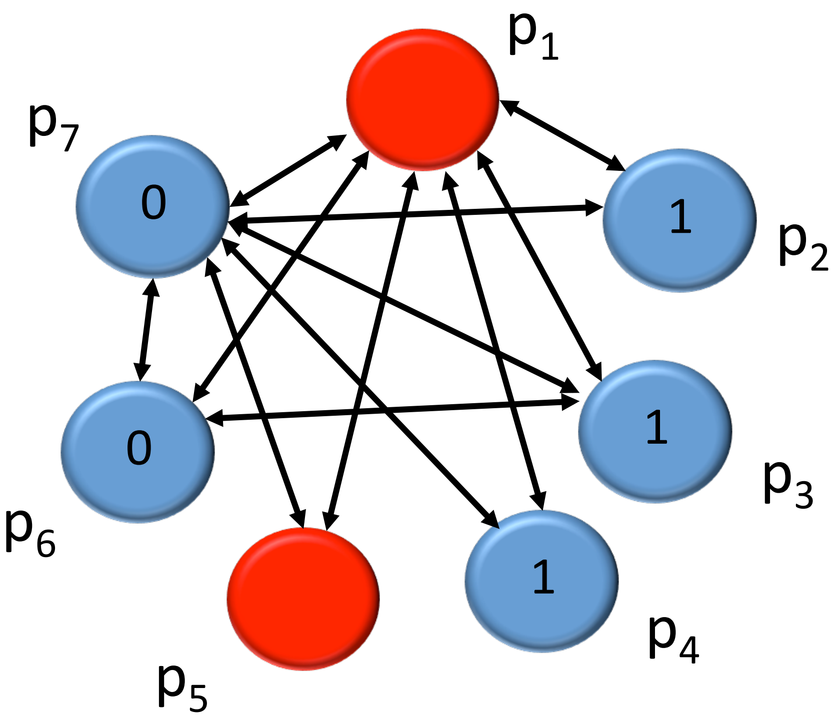
Round 2: Exchange All Values



Etc. complete graph

p_2: p_1 said p_1 said 1
p_1 said p_3 said 1
p_1 said p_4 said 1
p_1 said p_5 said 1
p_1 said p_6 said 1
p_1 said p_7 said 1
p_3 said p_1 said 1
p_3 said p_2 said 1
. . .
P_7: . . .
p_1 said p_3 said 0
p_1 said p_4 said 0
p_1 said p_5 said 0
p_1 said p_6 said 0
… etc. …

Round 3: Exchange All Values



Etc. complete graph

p_2: . . .

p_3 said p_1 said p_1 said 1

p_3 said p_1 said p_3 said 1

p_3 said p_1 said p_4 said 1

. . .

P_7: . . .

p_3 said p_1 said p_3 said 0

p_1 said p_3 said p_2 said 0

… etc. …

Simple Byzantine Agreement - Analysis

p must decide if *q* has provided inconsistent information:

- Is there subset *P* of {*p1*,…,*p7*} of size > $(n + f)/2 = 4.5$ and value *v* such that *p* said *p2* said … said *p7* said *q* said *v* ?

All sequences of length <= *f*

- If *q* is correct: Yes there is, as we can choose only correct nodes for *P*: $n - f > (n - f)/2 + (n - f)/2 > (n - f)/2 + f = (n + f)/2$ (recall: $n > 3f$)
- If *q* is incorrect:
  - Suppose both *p* and *p'* finds such a set *P* and value *v* for *q*
  - The sets have > $2((n+f)/2) - n = f$ common members
  - One of those is correct, so said the same of *q* in both cases
  - So *p* and *p'* agree that *q* said *v*

Simple Byzantine Agreement - Analysis

p must decide if *q* has provided inconsistent information:

- Is there subset *P* of {*p1*,…,*p7*} of size > (*n* + *f*)/2 = 4.5 and value *v* such that *p* said *p2* said … said *p7* said *q* said *v* ?

All sequences of length <= *f*

- What if p does not find a set P?
- Answer:
  - p knows that q has delivered inconsistent information
  - Drop q and recurse using n – 1 nodes and f – 1 byzantine nodes
  - Drop all strings of shape q1 said … qm said **q** said p’ said v from consideration
  - For each q that is not dropped in this way, by induction p finds a set P
  - Why? Eventually all nodes that provided inconsistent information are dropped

Exercise

2. Write down the algorithm in pseudocode and complete the proof sketched above

   Be clear on what the inductive statement is and how it is proved

Simple Byzantine Agreement: Summary

- 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 a considerable disadvantage:
  - \- The size of the messages increases exponentially!

- Can we 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* processes and *f* failures where $n > 4f$ 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 processes choose the same value and that in future rounds the processes do not change their values anymore

Berman, Garay, Perry: Towards optimal distributed consensus, FOCS 1989 (also #8)

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

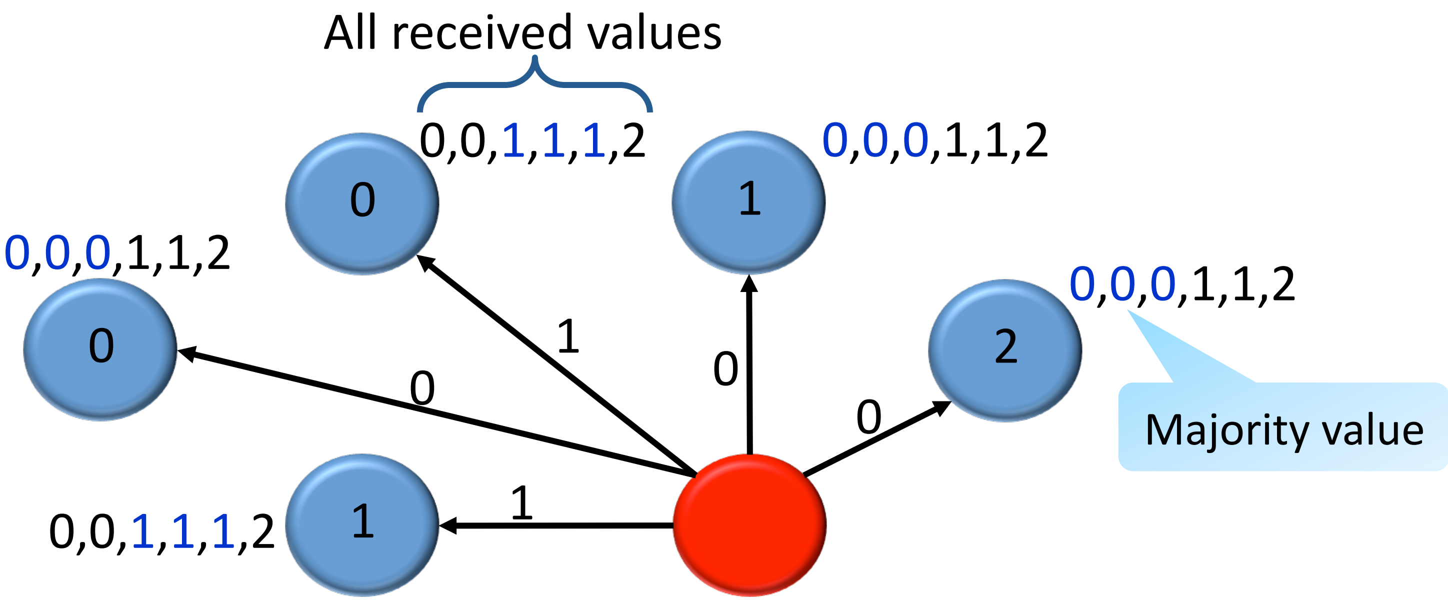
If not supporting any value

    set own value to the queen's value

The Queen Algorithm: Example

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

No process supports a value

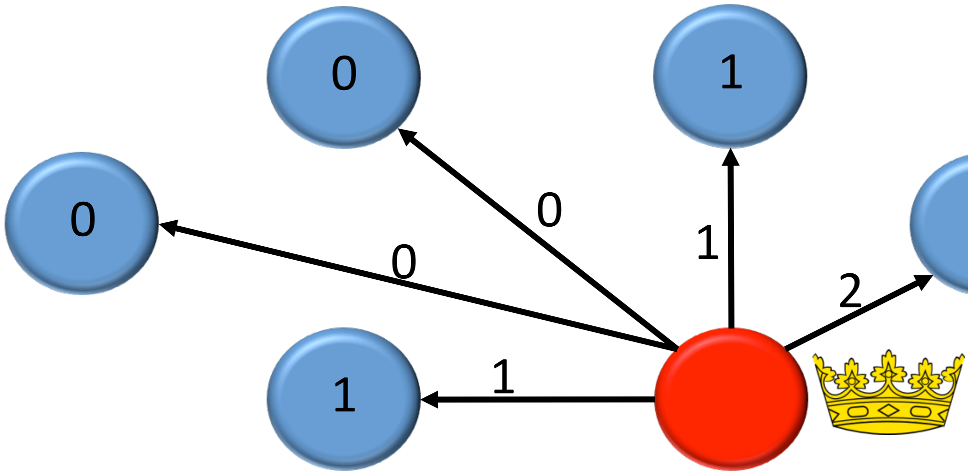


Broadcast own value
Set own value to the value that was received most often
If own value appears > $n/2+f$ times support this value
else do not support any value

The Queen Algorithm: Example

- Phase 1, round 2 (Queen broadcasts):

All processes choose the queen's value

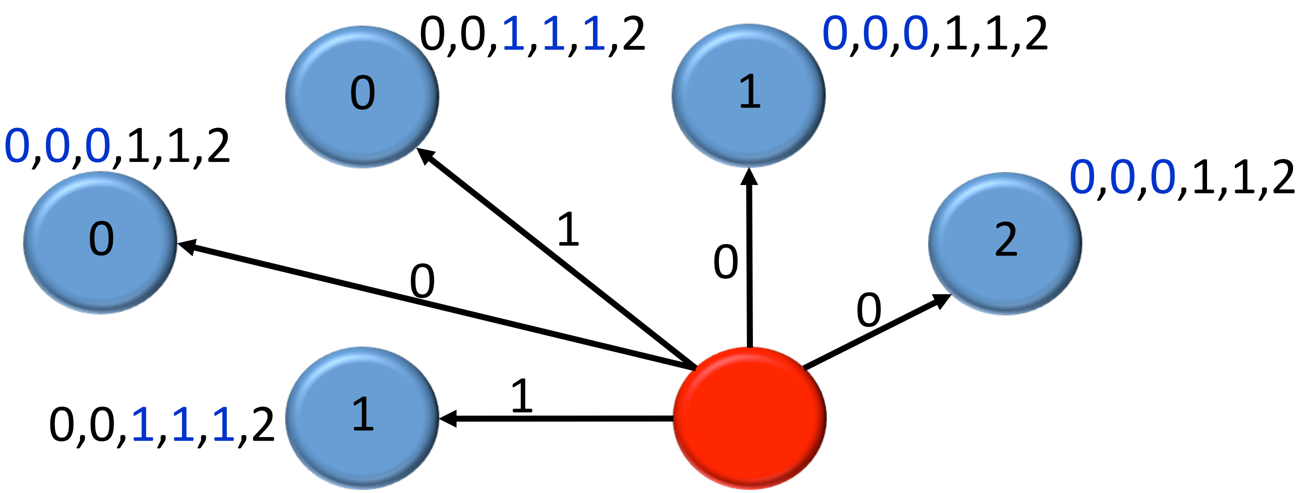


The queen broadcasts its value
If not supporting any value
    set own value to the queen's value

The Queen Algorithm: Example

- Phase 2, round 1 (All broadcast)

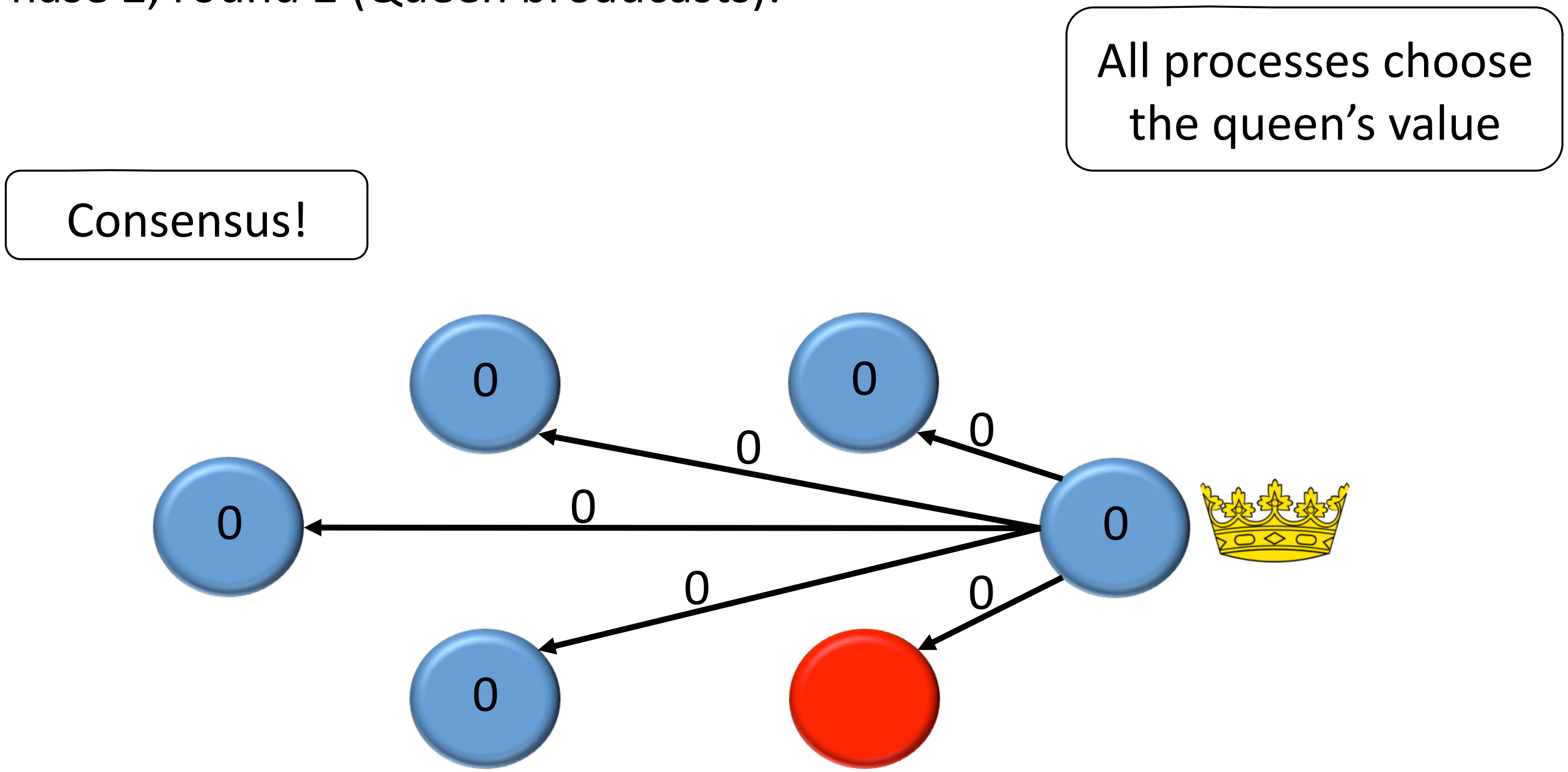
No process supports a value



Broadcast own value
Set own value to the value that was received most often
If own value appears > n/2+*f* times support this value
else do not support any value

The Queen Algorithm: Example

- Phase 2, round 2 (Queen broadcasts):



The queen broadcasts its value
If not supporting any value
set own value to the queen's value

The Queen Algorithm: Analysis

- After the phase where the queen is correct, all correct processes have the same value
  - If all processes change their values to the queen's value, obviously all values are the same
  - If some process does not change its value to the queen's value, it received a value $> n/2+f$ times → All other correct processes (including the queen) received this value $> n/2$ times and thus all correct processes share this value
- In all future phases, no process changes its value
  - In the first round of such a phase, processes receive their own value from at least $n\text{-}f > n/2$ processes and thus do not change it
  - The processes do not accept the queen's proposal if it differs from their own value in the second round because the processes received their own value at least $n\text{-}f = (n\text{-}f)/2 + (n\text{-}f)/2 > n/2+f$ times. Thus, all correct processes 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: processes 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 processes!
    Is it possible to get an algorithm that works with $f < n/3$ Byzantine processes 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 processes only propose a value if many processes have this value, and a value is only accepted if many processes propose this value

The King Algorithm

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

Round 1:
Broadcast own value

Round 2:
If some value x appears $\geq n\text{-}f$ 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 $< n\text{-}f$ proposals
Set own value to the king's value

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

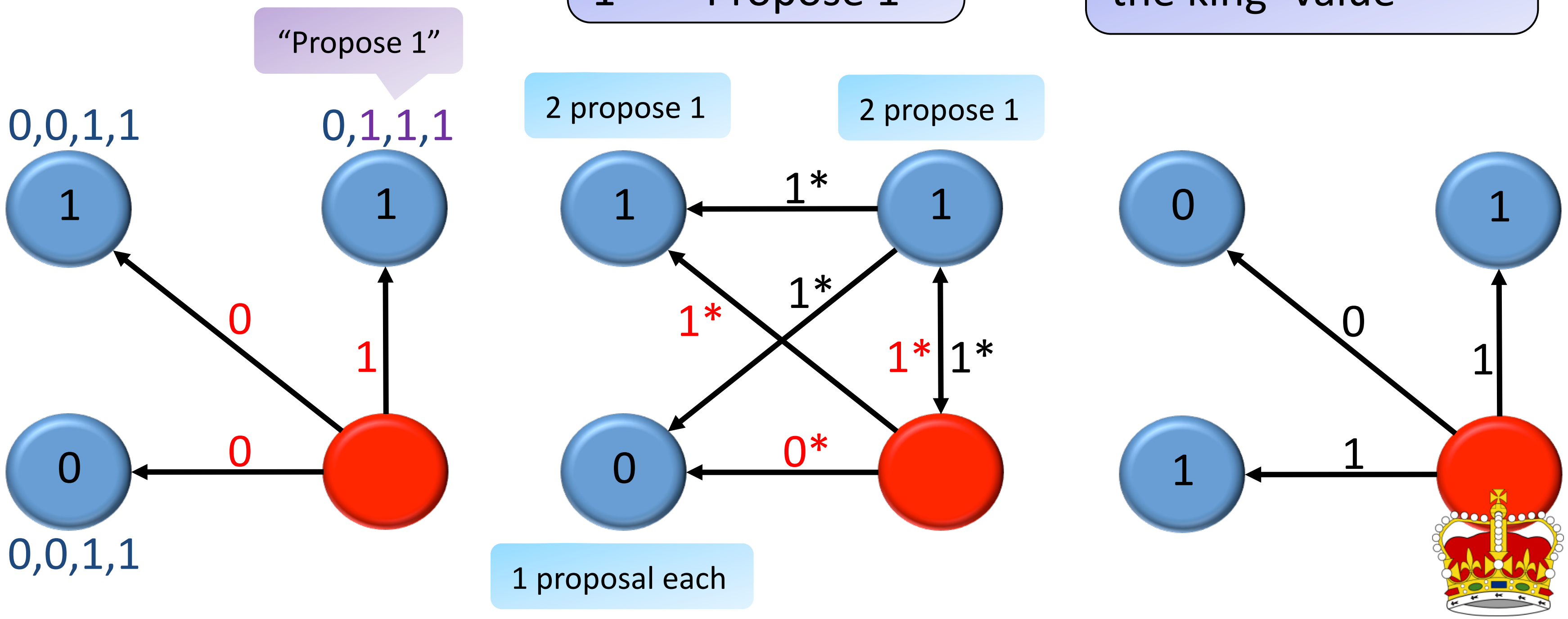
Also send own value to oneself

The King Algorithm: Example

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

0* = "Propose 0"
1* = "Propose 1"

All processes choose the king' value



Round 1

Broadcast own value

Round 2

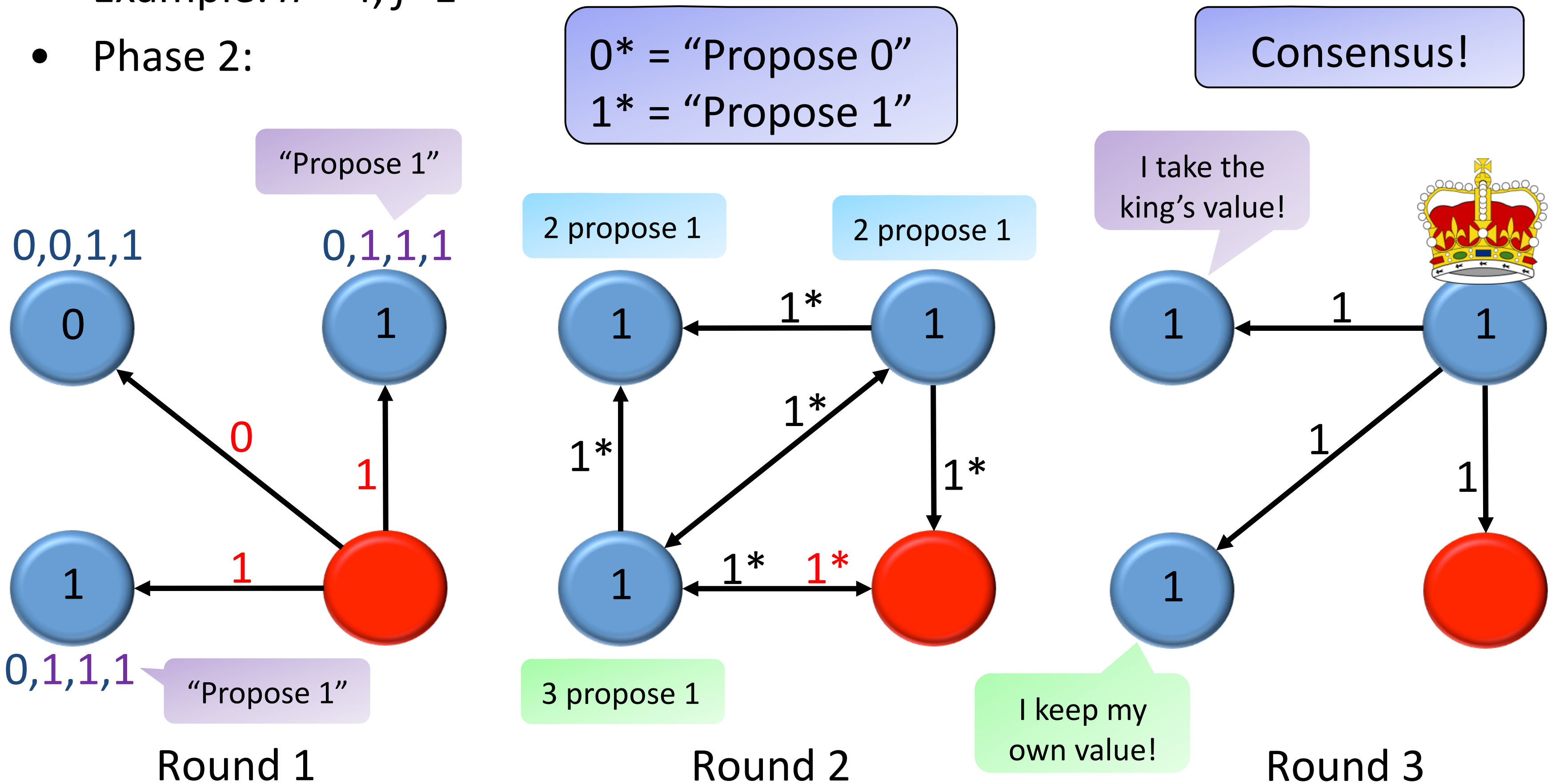
If some value x appears ≥ n-f times
Broadcast "Propose x"
If some proposal received > f times
Set own value to this proposal

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 2:



Round 1:

Broadcast own value

Round 2:

If some value x appears ≥ 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: Analysis

- Observation: If some correct process proposes x, then no other correct process proposes $y \neq x$
  - Both processes would have to receive $\geq n - f$ times the same value
  - $\geq n - 2f$ of the sending processes are non-faulty
  - Then there must be $\geq 2(n - 2f) + f = 2n - 3f > n$ processes

We used that $f < n/3$!

- The validity condition is satisfied
  - If all correct processes start with the same value, all correct processes receive this value $\geq n - f$ times and propose it
  - All correct processes receive $\geq n - f$ proposals, i.e., no correct process 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 $\geq n\text{-}f$ times → $\geq n\text{-}2f$ correct processes broadcast this proposal and all correct processes receive it $\geq n\text{-}2f > f$ 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

Exercises

3. Some networks are organized as a hypercube. There are $n = 2^m$ processes and each process can communicate with m other processes.
   a) Modify the King algorithm so that it works in a hypercube. Optimize the algorithm according to resilience.
   b) How many failures can your algorithm handle? (Assume Byzantine processes can neither forge nor alter source or destination of a message.)
   c) How many rounds does this algorithm require?

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

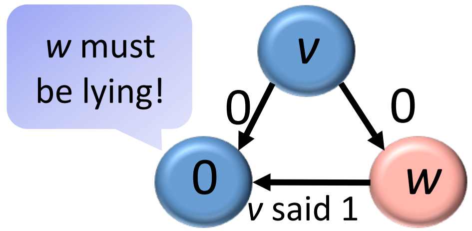
- However, it also has a disadvantage:
  - \- The algorithm requires f+1 phases consisting of 3 rounds each

    This is three times as much as an optimal algorithm

    Is it possible to get an algorithm that uses small messages and requires fewer rounds of communication?

Consensus #9: Byzantine Agreement Using Authentication

- *Unforgeability condition*: If a process *p* never sends a message *m*, then no correct process ever accepts *m* (as coming from *p*)



- Why is this condition helpful?
  - A Byzantine process cannot convince a correct process that some other correct processes voted for a certain value if they did not!

Idea:

- There is a designated process P. The goal is to decide on P's value
- Assume binary input. The default value is 0, i.e., if P cannot convince the processes that P's input is 1, all correct processes choose 0

D. Dolev, R. Strong: Polynomial algorithms for byzantine agreement, Proc. 14th STOC, 1982

Byzantine Agreement Using Authentication

If I am P and own input is 1

    value :=1

    broadcast “P has 1”

else

    value := 0

In each round r ϵ 1...*f+1*:

If value = 0 and accepted r messages “P has 1” in total including a message from P itself

    value := 1

    broadcast “P has 1” plus the r accepted messages that caused the local value to be set to 1

> In total r+1 authenticated “P has 1” messages

After *f*+1 rounds:

Decide value

Byzantine Agreement Using Authentication: Intuition

So what's going on?

- The goal: If one correct P decides 1 (0) then all correct processes decide 1 (0), at the latest in round $f + 1$
- Since messages are authenticated, "P has 1" sent from node *i* is different from "P has 1" sent from node *j*
- If a correct node p receives an authentic message "P has 1" from P can it then decide 1?
- If so, it can then terminate the following round – then all other processes will have received the same messages p received and decide 1
- But what if P (e.g.) waits until round f+1 to tell a correct node that it has 1?

Byzantine Agreement Using Authentication: Analysis

Case 1: P is correct

- P's input is 1: All correct processes accept P's message in round 1 and set value to 1. No process ever changes its value back to 0
- P's input is 0: P never sends a message "P has 1", thus no correct process ever sets its value to 1

Byzantine Agreement Using Authentication: Analysis

Case 2: P is Byzantine

- P tries to convince some correct processes that its input is 1
- Assume a correct process p sets value = 1 in round $r < f+1$: Process p has accepted r messages including the message from P. Therefore, all other correct processes accept the same r messages plus p's message and set their values to 1 as well in round $r+1$
- Assume that a correct process p sets its value to 1 in round $f+1$: In this case, p accepted $f+1$ messages. At least one of those is sent by a correct process, which must have set its value to 1 in an earlier round. We are again in the previous case, i.e., all correct processes decide 1!

Exercises

4. Modify the algorithm such that it handles arbitrary input. The processes may also agree on a “sender faulty” value. Prove that your algorithm is correct.

Byzantine Agreement Using Authentication: Summary

- Using authenticated messages has several advantages:
  - \+ It works for any number of Byzantine processes!
  - \+ It only takes *f+1* rounds, which is optimal
  - \+ Small messages: processes send at most f+1 "short" messages to all other processes in a single round

sub-exponential length

- However, it also has some disadvantages:
  - \- If P is Byzantine, the processes may agree on a value that is not in the original input
  - \- It only works for binary input
  - \- The algorithm requires authenticated messages…

Byzantine Agreement Using Authentication: Improvements

- Can we modify the algorithm so that it satisfies the validity condition?
  - Yes! Run the algorithm in parallel for $2f$+1 “masters” P. Either 0 or 1 is decided at least f+1 times, i.e., at least one correct process had this value. Decide on this value!
  - Alas, this modified protocol only works if $f < n/2$
- Can we get rid of the authentication?
  - Yes! Use *consistent-broadcast*. This technique is not discussed
  - This modified protocol works if $f < n/3$, which is optimal
  - However, each round is split into two → The total number of rounds is $2f+2$

Consensus #10: A Randomized Algorithm

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

- Can one solve consensus in *asynchronous* systems if we allow randomization?

Asynchronous system: Messages may be delayed indefinitely

- The answer is yes!
- The basic idea of the algorithm is to push the initial value. If other processes 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 $f < n/9$ processes are Byzantine

M. Ben-Or: Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. PODC 1983

Randomized Algorithm

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

In each round *r* = 1,2,…:

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 P[*x*=0] = P[*x*=1] = ½
Broadcast proposal(*x*, *r*)
If decided on a value → stop

Randomized Algorithm - Validity

x := own input; *r* = 0

n – f correct processes have same x

Broadcast proposal(*x*, *r*)

n – f correct processes broadcast x

In each round *r* = 1,2,…:

Wait for *n-f* proposals

If at least *n-2f* proposals have some value *y*

x := *y*; decide on *y*

All correct processes receive n – 2f proposals for x

else if at least *n-4f* proposals have some value *y*

x := *y*;

else

choose *x* randomly with P[*x*=0] = P[*x*=1] = ½

Broadcast proposal(*x*, *r*)

If decided on a value → stop

Randomized Algorithm - Agreement

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

In each round *r* = 1,2,…:

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 P[*x*=0] = P[*x*=1] = ½
Broadcast proposal(*x*, *r*)
If decided on a value → stop

Some correct process decides x

Randomized Algorithm - Agreement

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

n – 3f correct processes proposed x

In each round r = 1,2,…:

Wait for n-f proposals
If at least n-$2f$ proposals have some value y
 x := y; decide on y

Some correct process decides x

else if at least n-$4f$ proposals have some value y
 x := y;
else
 choose x randomly with P[x=0] = P[x=1] = ½
Broadcast proposal(x, r)
If decided on a value → stop

Randomized Algorithm - Agreement

x := own input; *r* = 0

Broadcast proposal(*x*, *r*)

n – 3f correct processes proposed x

In each round *r* = 1,2,…:

Wait for *n-f* proposals

If at least *n-2f* proposals have some value *y*

x := *y*; decide on *y*

Some correct process decides x

else if at least *n-4f* proposals have some value *y*

x := *y*;

n – 4f correct processes proposed x

So: all n – f correct processes take x

All decide x next round

else

choose *x* randomly with P[*x*=0] = P[*x*=1] = ½

Broadcast proposal(*x*, *r*)

If decided on a value → stop

Randomized Algorithm - Termination

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

In each round r = 1,2,…:

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 $P[x=0] = P[x=1] = ½$
Broadcast proposal(x, r)
If decided on a value → stop

Some correct process does not set x randomly

Randomized Algorithm - Termination

x := own input; *r* = 0

Broadcast proposal(*x*, *r*)

In each round *r* = 1,2,…:

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 P[*x*=0] = P[*x*=1] = ½

Broadcast proposal(*x*, *r*)

If decided on a value → stop

n > 9f

n – 5f correct processes proposed x =>
no correct process proposed y != x

Some correct process does not set x randomly

Randomized Algorithm - Termination

x := own input; r = 0

Broadcast proposal(x, r)

In each round r = 1,2,…:

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 P[x=0] = P[x=1] = ½

Broadcast proposal(x, r)

If decided on a value → stop

Worst case: All choose randomly
Prob(all choose i) = $2^{-(n-f)}$
Termination in expectation < 2^n

$n > 9f$

$n - 5f$ correct processes proposed x =>
no correct process proposed y != x

Some correct process does not set x randomly

Randomized Algorithm: Analysis

- Validity condition (as before)
  - If all correct processes 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 processes decide, they agree on the value)
  - Assume that some correct process decides on *x*. This process must have received *x* from *n-3f* correct processes. Every other correct process must have received *x* at least *n-4f* times, i.e., all correct processes set their local value to *x*, and propose and decide on *x* in the next round

Randomized Algorithm: Analysis

Termination (all correct processes eventually decide)

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

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

- Thus, in the worst case all n-f correct processes need to choose the same bit randomly, which happens with probability $(½)^{(n-f)}$
- Hence, all correct processes eventually decide. The expected running time is smaller than 2^n

Exercises

5. Explain why it does not work by just setting $x = 1$ instead of choosing x randomly

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

- Replace:

  choose x randomly with P[x=0] = P[x=1] = ½

  with a subroutine in which all the processes compute a so-called shared (a.k.a. common, “global”) coin
- A shared coin is a shared random binary variable that is 0 with constant probability, and 1 with constant probability
- And: with constant probability some processes see 0 and some see 1
- For the sake of simplicity, we assume that there are at most $f < n/3$ crash failures (no Byzantine failures!!!)

Bracha, G. (1984). An asynchronous $\lfloor(n-1)/3\rfloor$-resilient consensus protocol. PODC 1984

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 3*f*+1 = n!

Shared Coin Algorithm - Termination

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

All correct processes receive n – f coins

All correct processes receive n – f coin sets

Shared Coin: Analysis

Termination:

- All correct processes broadcast their coins.
- It follows that all correct processes receive at least n-f coins
- All correct processes broadcast their coin sets.
- It follows that all correct processes receive at least n-f coin sets and the subroutine terminates

Shared Coin: Analysis

- We will now show that at least 1/3 of all coins are seen by everybody

A coin is *seen* if it is in at least one received coin set

- More precisely: We will show that at least f+1 coins are in at least f+1 coin sets
  - Recall that $f < n/3$
  - Since f+1 coins are in at least f+1 coin sets
  - and all processes receive n-f coin sets:
  - all correct processes see these coins!

Shared Coin: Analysis

- Proof that at least *f*+1 coins are in at least *f*+1 coin sets
  - Draw the coin sets and the contained coins as a matrix
  - Example: n=7, f=2

x means coin c_i is in set s_j

	s_1	s_3	s_5	s_6	s_7
c_1	x	x	x	x	x
c_2		x	x		
c_3	x	x	x	x	x
c_4		x	x		x
c_5	x			x	
c_6	x		x	x	x
c_7	x	x		x	x

Shared Coin: Analysis

At least f+1 rows (coins) have at least f+1 x's (are in at least f+1 coin sets)

- First, there are exactly $(n-f)^2$ x's in this matrix
- Assume that the statement is wrong: Then at most f rows may be full and contain n-f x's. And all other rows (at most $n-f$) have at most f x's
- Thus, in total we have at most $f(n-f)+(n-f)f = 2f(n-f)$ x's
- But $2f(n-f) < (n-f)^2$ because $2f < n-f$ (recall again; $3f < n$)

	s_1	s_3	s_5	s_6	s_7
c_1	x	x	x	x	x
c_2		x	x		
c_3	x	x	x	x	x
c_4		x	x		x
c_5	x			x	
c_6	x		x	x	x
c_7	x	x		x	x

Shared Coin

Theorem

All processes decide 0 with constant probability, and all processes decide 1 with constant probability

Proof:

- With probability $(1\text{-}1/n)^n \approx 1/e \approx 0.37$ all processes choose 1. Thus, all correct processes return 1
- There are at least n/3 coins seen by all correct processes. The probability that at least one of these coins is set to 0 is at least
  $1\text{-}(1\text{-}1/n)^{n/3} \approx 1\text{-}(1/e)^{1/3} \approx 0.28$

Back to Randomized Consensus

- If this shared coin subroutine is used, there is a constant probability that the processes agree on a value
- Some nodes may not want to perform the subroutine because they received the same value *x* at least *n*-4*f* times. However, there is also a constant probability that the result of the shared coin toss is *x*!
- Of course, all nodes must take part in the execution of the subroutine
- This randomized algorithm terminates in a constant number of rounds (in expectation)!

Randomized Algorithm: Summary

The randomized algorithm has several advantages:

+ It only takes a constant number of rounds in expectation
+ It can handle crash failures even if communication is asynchronous

However, it also has some disadvantages:

- It works only if there are $f < n/9$ crash failures.
- It doesn't work if there are Byzantine processes
- It only works for binary input

There are similar algorithms for the shared memory model

Can it be improved?

- There is a constant expected time algorithm that tolerates $f < n/2$ crash failures
- There is a constant expected time algorithm that tolerates $f < n/3$ Byzantine failures