The Consensus Hierarchy

1 Read/Write Registers, …

2 T&S, F&I, Swap, …

.
.
.

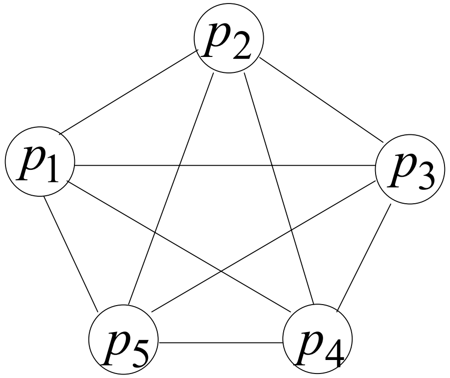
∞ CAS, …

Consensus #4
Synchronous Systems

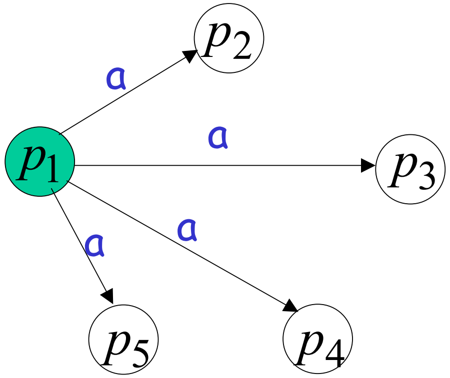
- In real systems, one can sometimes tell if a processor had crashed
  - Timeouts
  - Broken TCP connections
- Can one solve consensus at least in synchronous systems?

Communication Model

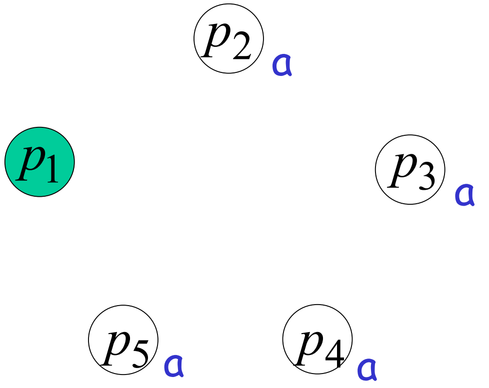
- Complete graph
- Synchronous



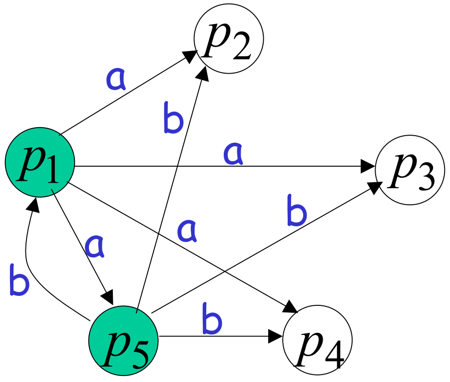
Send a message to all processors in one round: Broadcast



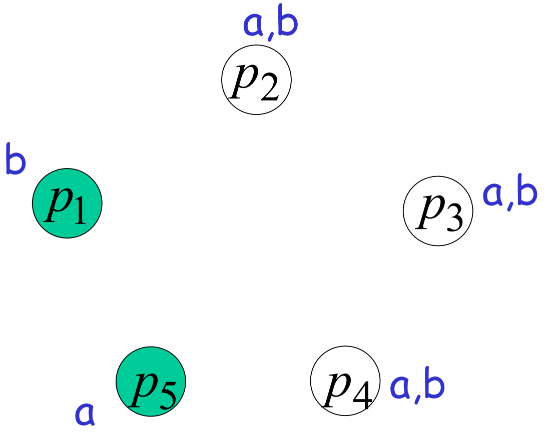
At the end of the round: everybody receives a



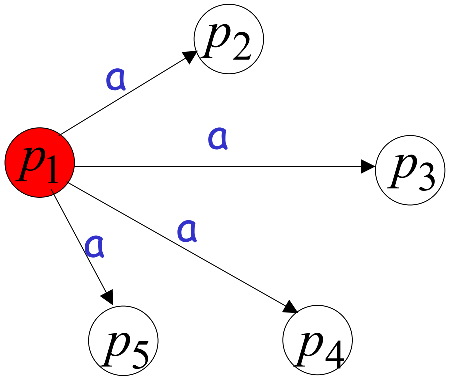
Broadcast: Two or more processes can broadcast in the same round



At end of round...

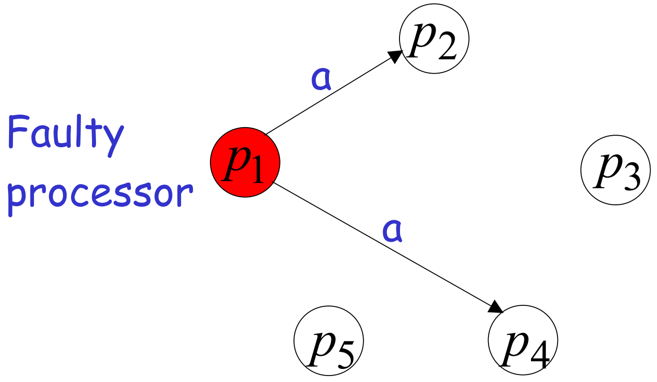


Crash Failures

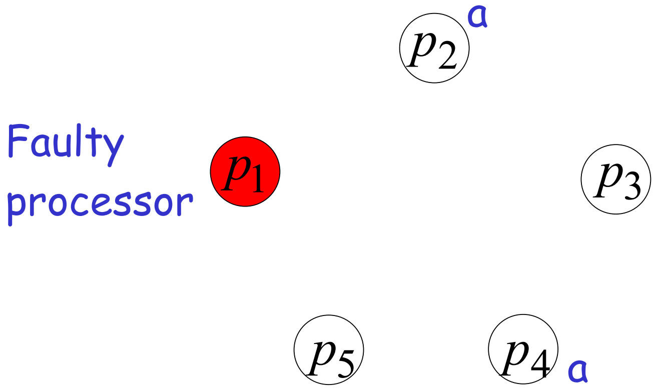


Faulty processor

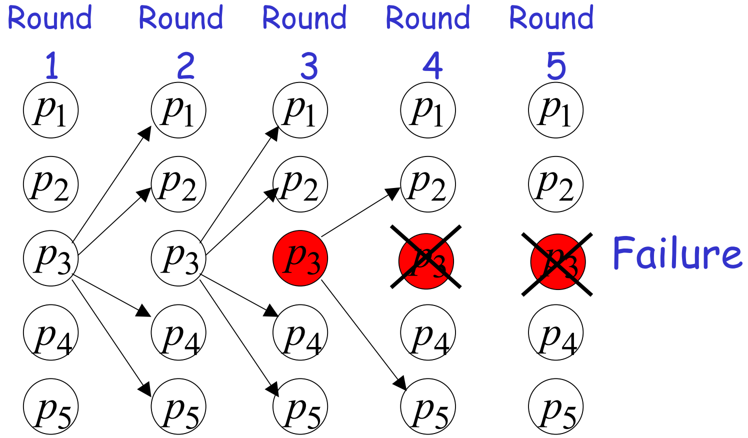
Some of the messages are lost, they are never received



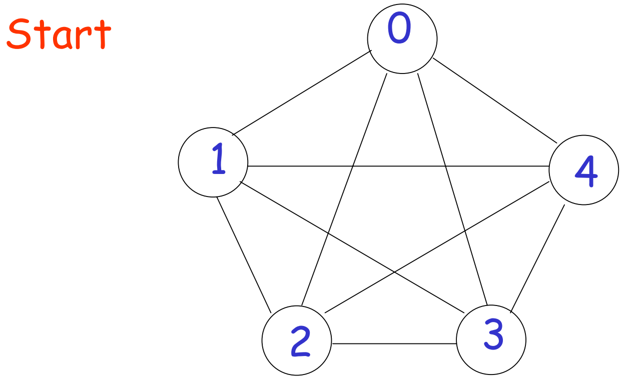
Effect



After a failure, the process disappears from the network

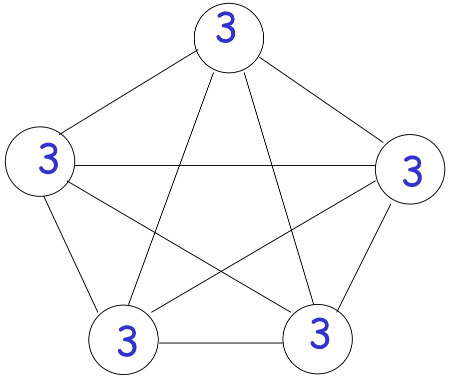


Consensus: Everybody has an initial value



Everybody must decide on the same value

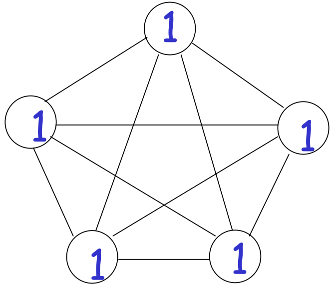
Finish



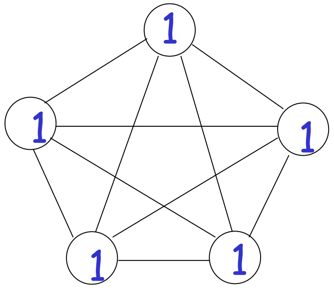
Validity condition:

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

Start



Finish



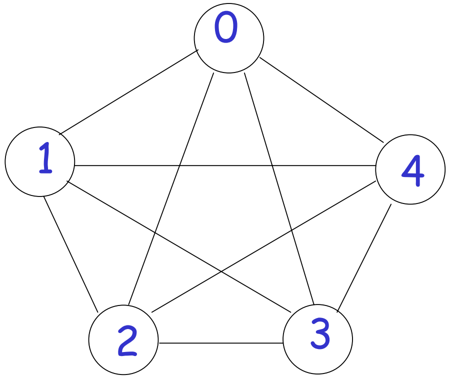
A simple algorithm

Each processor:

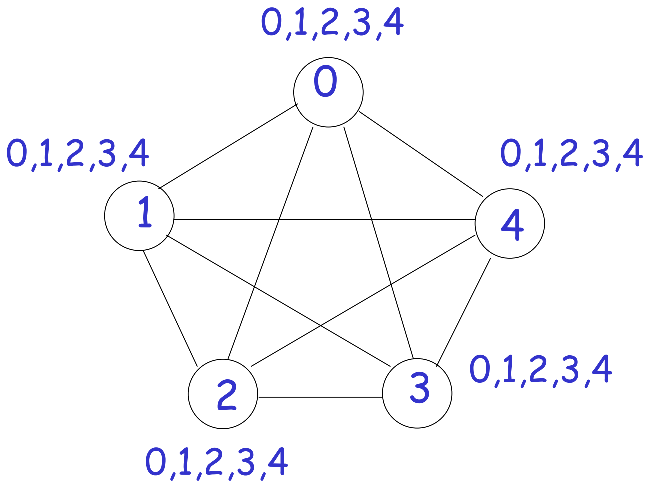
1. Broadcasts value to all processors
2. Decides on the minimum

(only one round is needed)

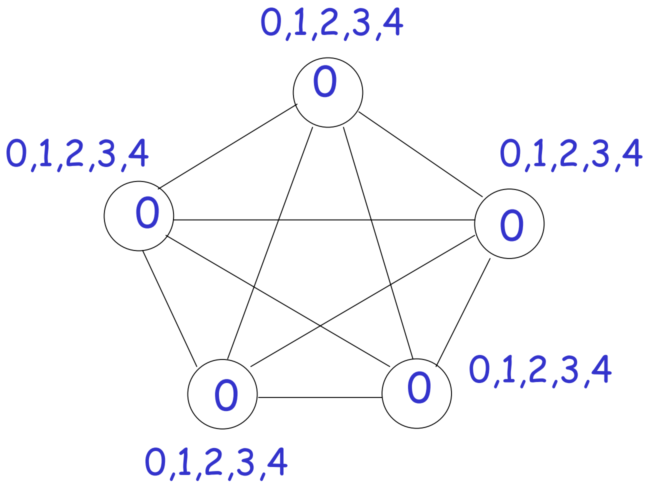
Start



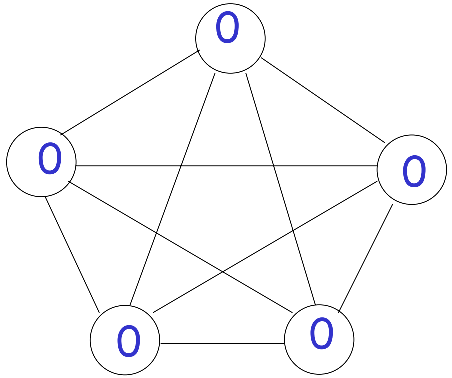
Broadcast values



Decide on minimum

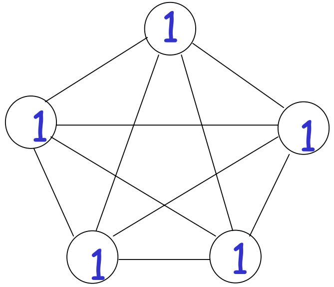


Finish

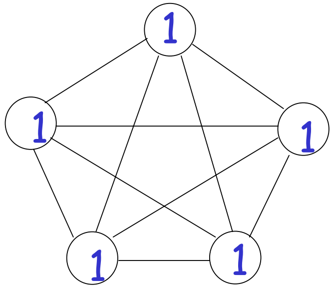


This algorithm satisfies the validity condition

Start



Finish



If everybody starts with the same initial value, everybody sticks to that value (minimum)

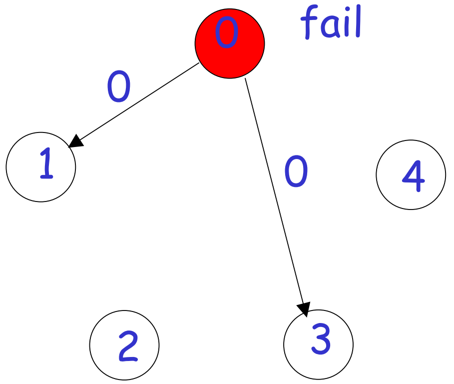
Consensus with Crash Failures

The simple algorithm doesn't work

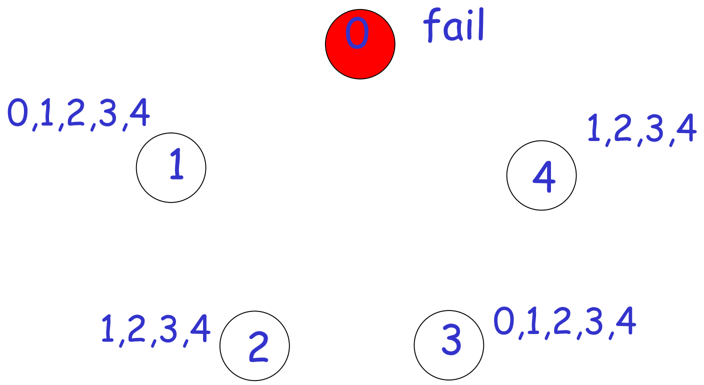
Each processor:

1. Broadcasts value to all processors
2. Decides on the minimum

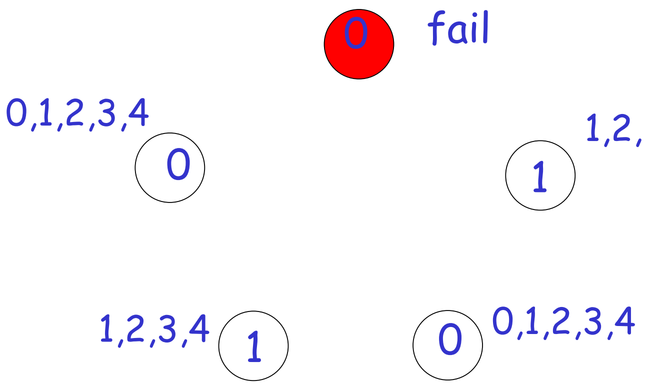
Start The failed processor doesn't broadcast its value to all processors



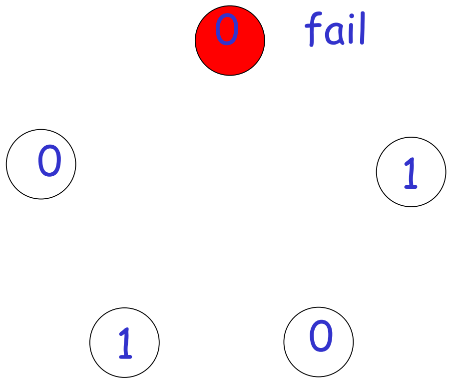
Broadcasted values



Decide on minimum



Finish - No Consensus!

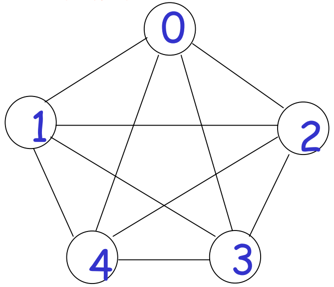


If an algorithm solves consensus for
f failed processes we say it is

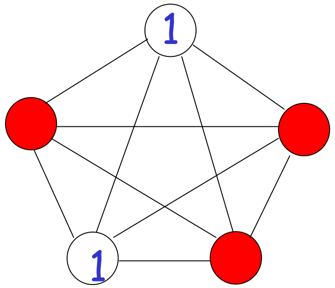
an f-resilient consensus algorithm

Example: The input and output of a
3-resilient consensus algorithm

Start



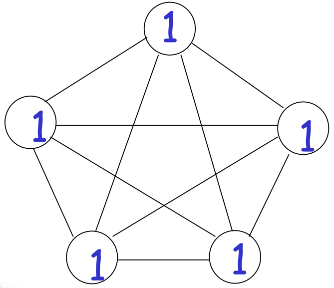
Finish



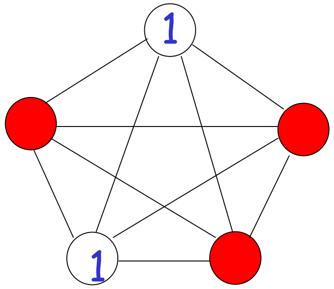
New validity condition:

all non-faulty processes decide on a value
that is available initially.

Start



Finish



An f-resilient algorithm

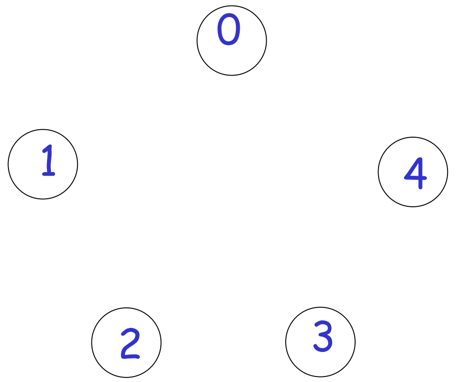
Round 1:
Broadcast my value

Round 2 to round f+1:
Broadcast any new received values

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

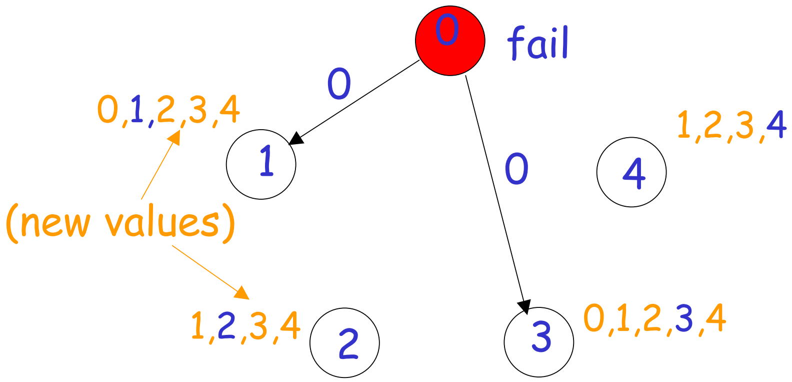
Example: f=1 failures, f+1=2 rounds needed

Start



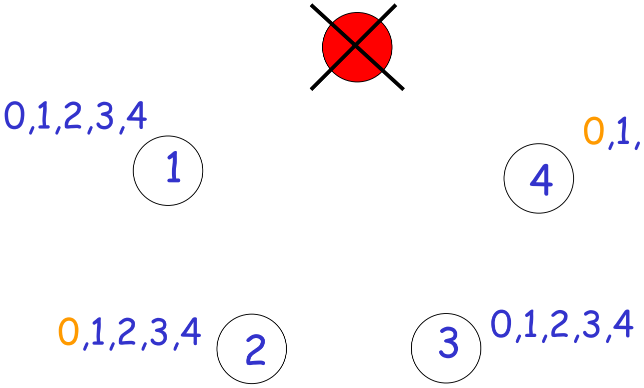
Example: f=1 failures, f+1 = 2 rounds needed

Round 1 Broadcast all values to everybody



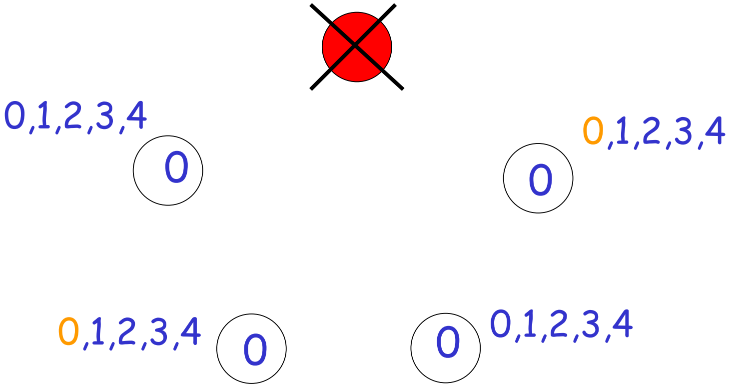
Example: f=1 failures, f+1 = 2 rounds needed

Round 2 Broadcast all new values to everybody



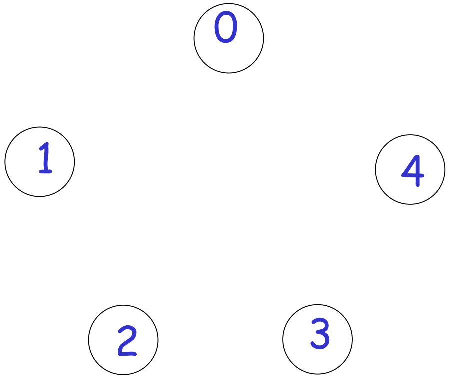
Example: f=1 failures, f+1 = 2 rounds needed

Finish Decide on minimum value



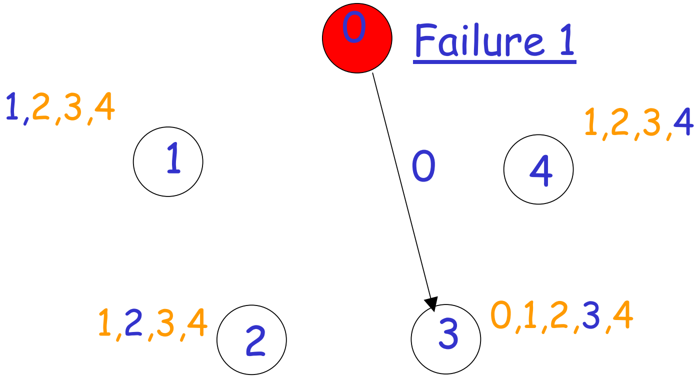
Example: f=2 failures, f+1 = 3 rounds needed

Start Example of execution with 2 failures



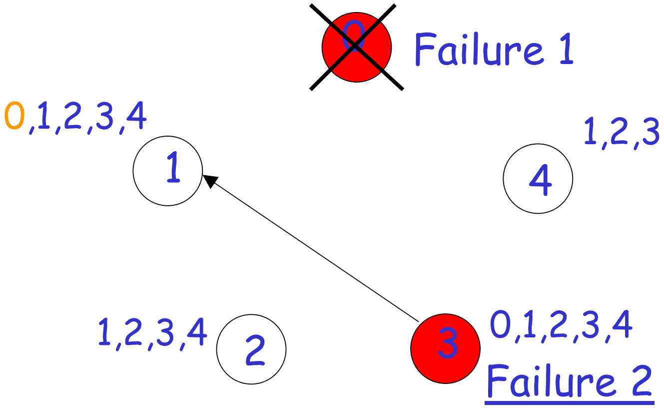
Example: f=2 failures, f+1 = 3 rounds needed

Round 1 Broadcast all values to everybody



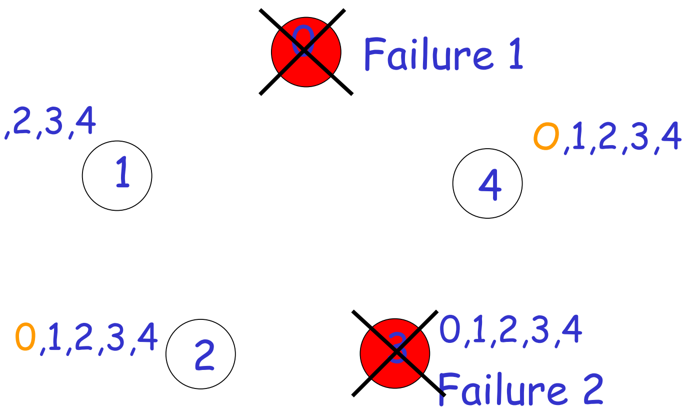
Example: f=2 failures, f+1 = 3 rounds needed

Round 2 Broadcast new values to everybody



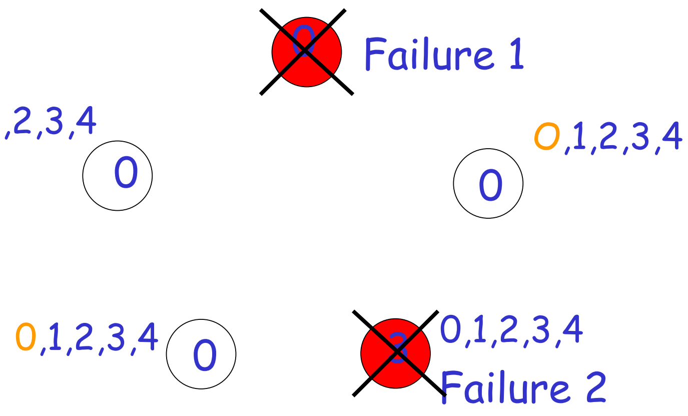
Example: f=2 failures, f+1 = 3 rounds needed

Round 3 Broadcast new values to everybody



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

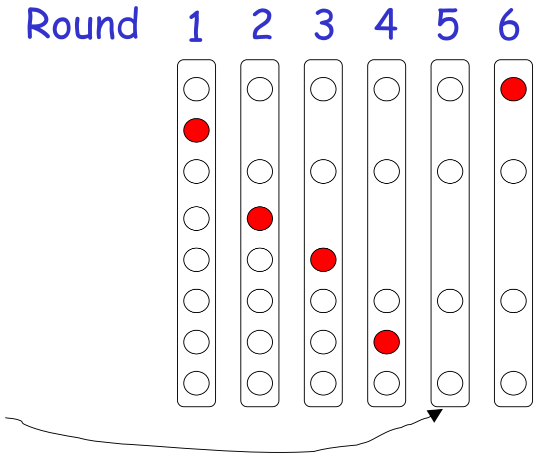
Finish Decide on the minimum value



If there are f failures and f+1 rounds then there is a round with no failed process

Example:
5 failures,
6 rounds

No failure



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, at the end of the round with no failure:

Everybody would 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 of algorithm:

when all processes start with the same input value then the consensus is that value

This holds, since the value decided from each process is some input value

---

A Lower Bound

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

---

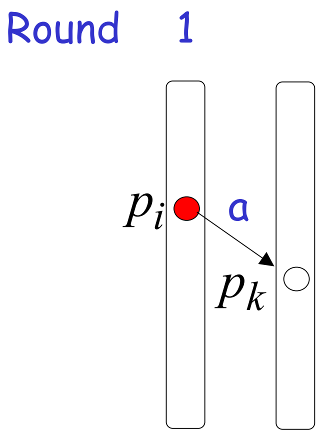
Proof sketch:

Assume for contradiction that f or less rounds are enough

Worst case scenario:

There is a process that fails in each round

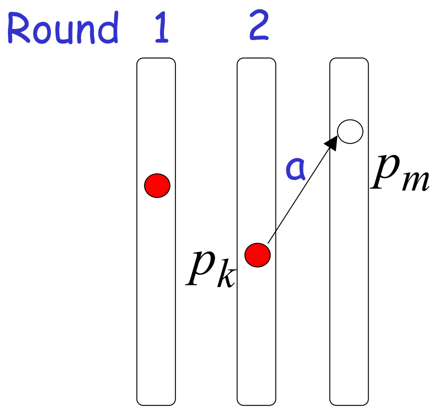
Worst case scenario



before process p_i fails, it sends its value **a** to only one process p_k

---

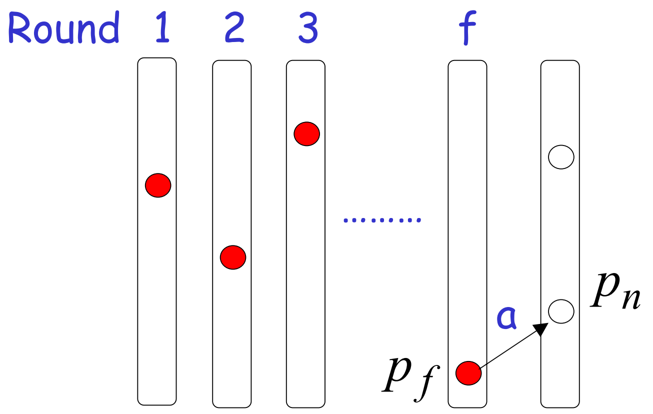
Worst case scenario



before process p_k fails, it sends

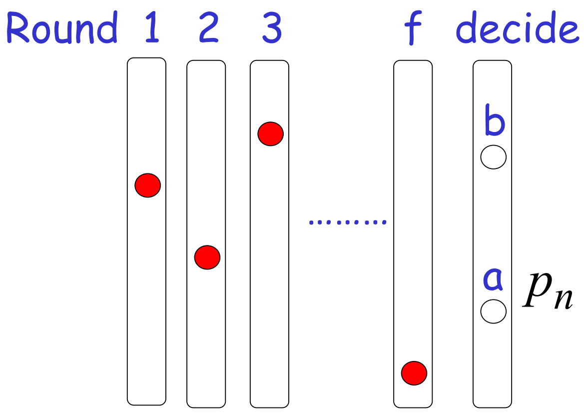
value **a** to only one process p_m

Worst case scenario



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

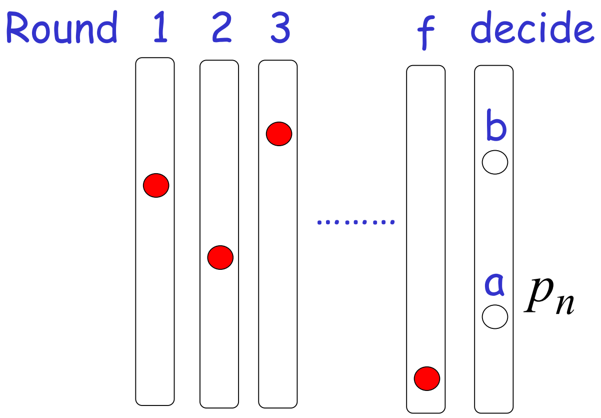
Worst case scenario



Process p_n may decide on **a**, and all other

processes may decide on another value **(b)**

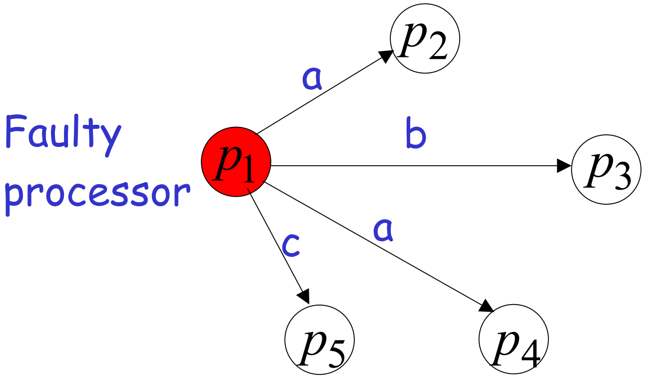
Worst case scenario



Therefore f rounds are not enough

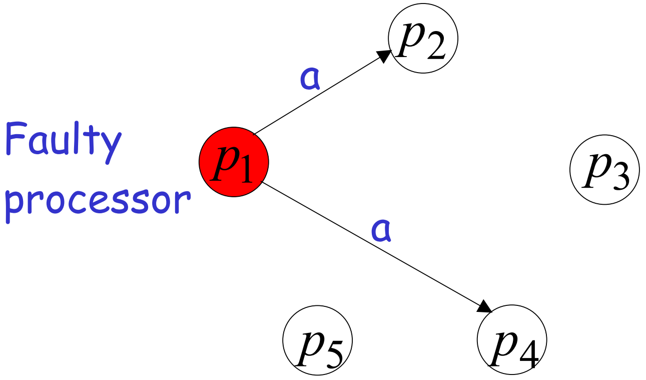
At least f+1 rounds are needed

Consensus #5
Byzantine Failures

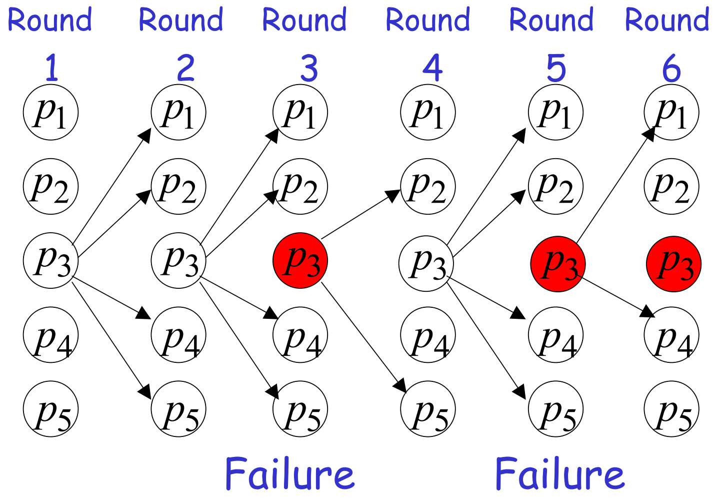


Different processes receive different values

Some messages may be lost



A Byzantine process can behave like a Crashed-failed process



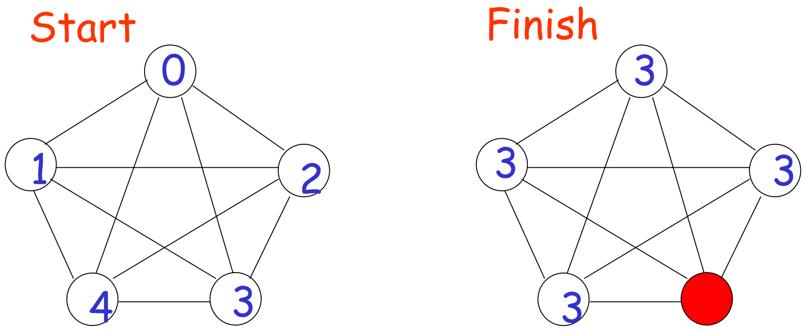
After failure the process continues functioning in the network

Consensus with Byzantine Failures

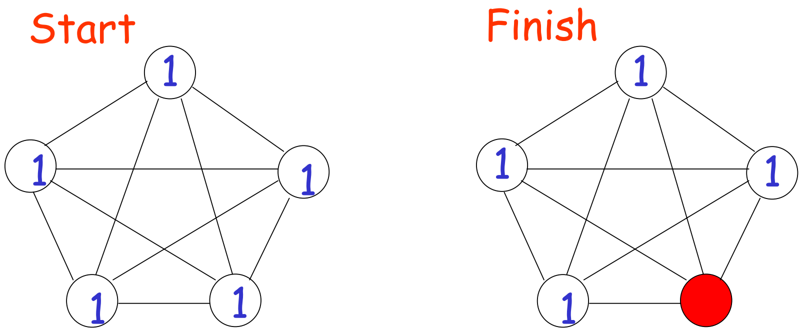
f-resilient consensus algorithm:

solves consensus for f failed processes

Example: The input and output of
a 1-resilient consensus algorithm



Validity condition:
if all non-faulty processes start with
the same value then all non-faulty processes
decide on that value



Lower bound on number of rounds

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

Proof: follows from the crash failure lower bound

Upper bound on failed processes

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

Plan: First we prove the 3 process case, and then the general case

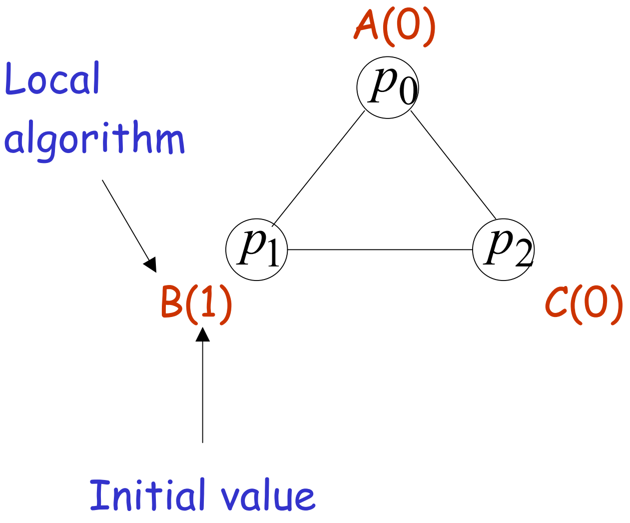
---

The 3 processes case

Lemma: There is no 1-resilient algorithm for 3 processes

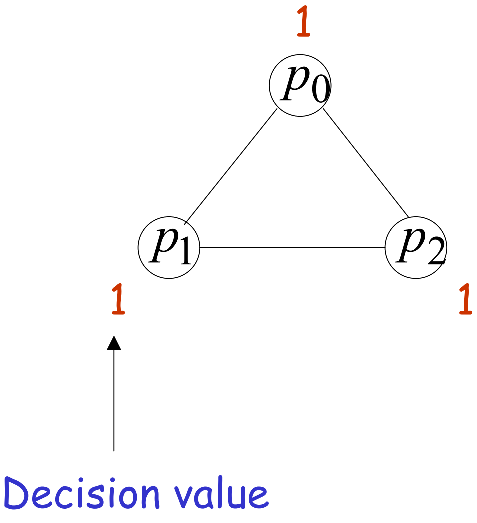
Proof: Assume for contradiction that there is a 1-resilient algorithm for 3 processes

---

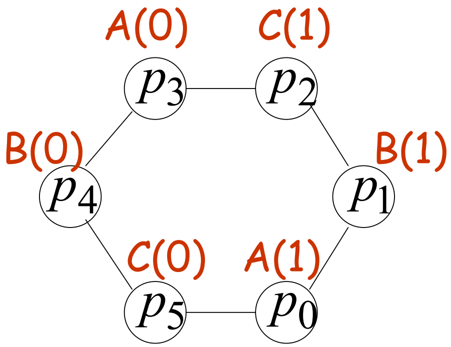


Local algorithm

Initial value

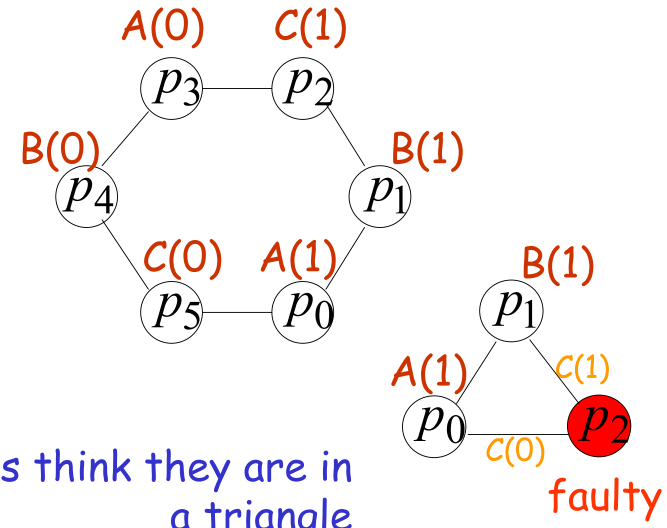


Decision value

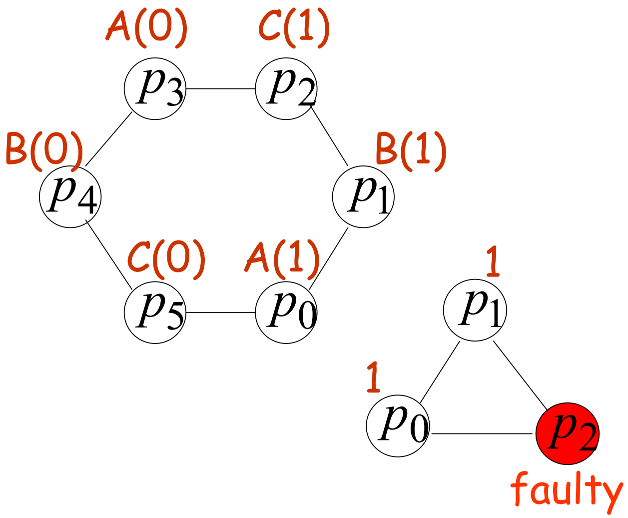


Assume 6 processes are in a ring

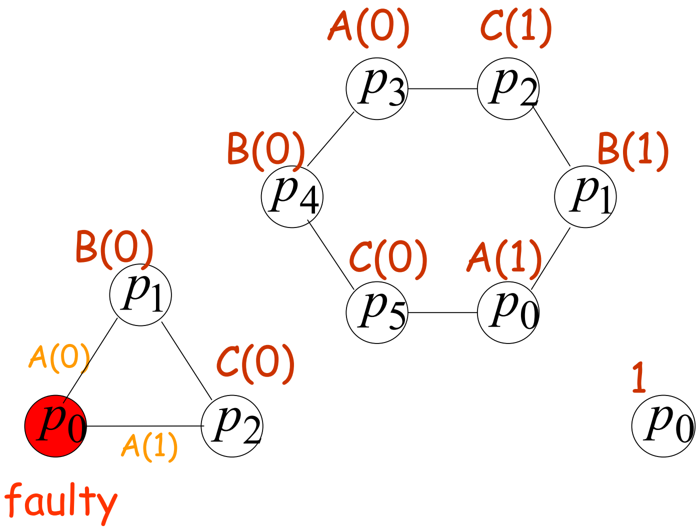
(just for fun)

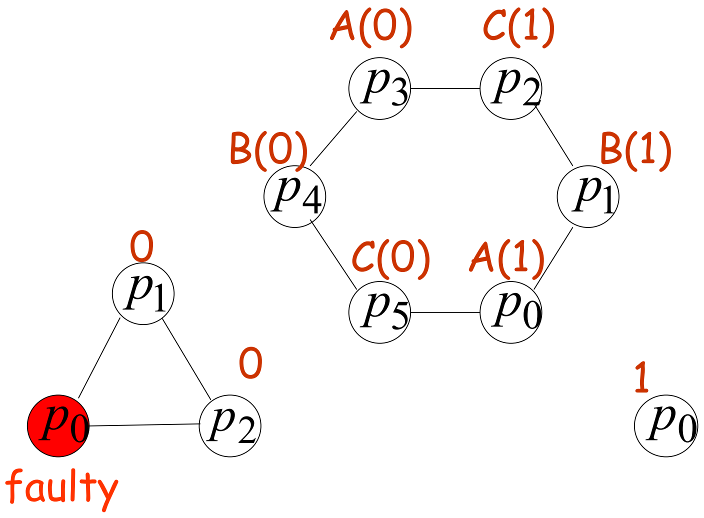


Processes think they are in a triangle

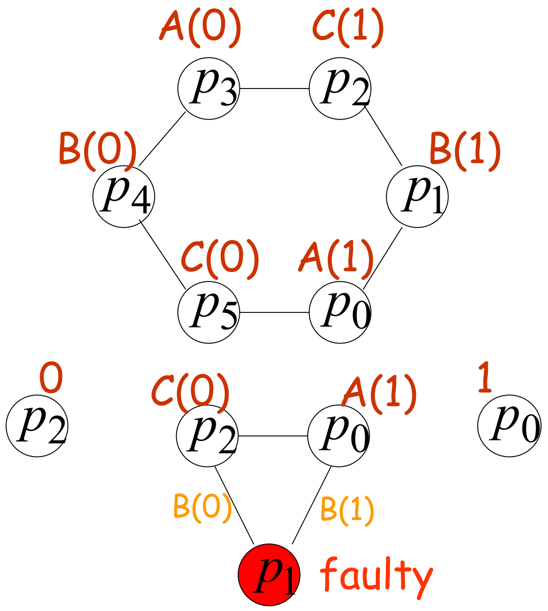


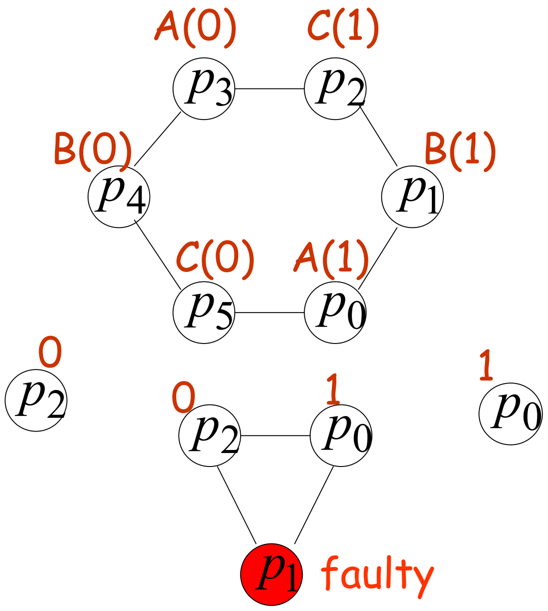
(validity condition)



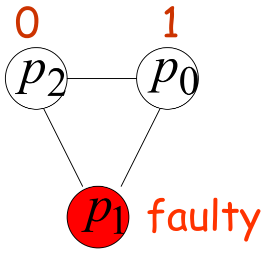


(validity condition)





Impossibility



Conclusion

There is no algorithm that solves consensus for 3 processes in which 1 is a byzantine process

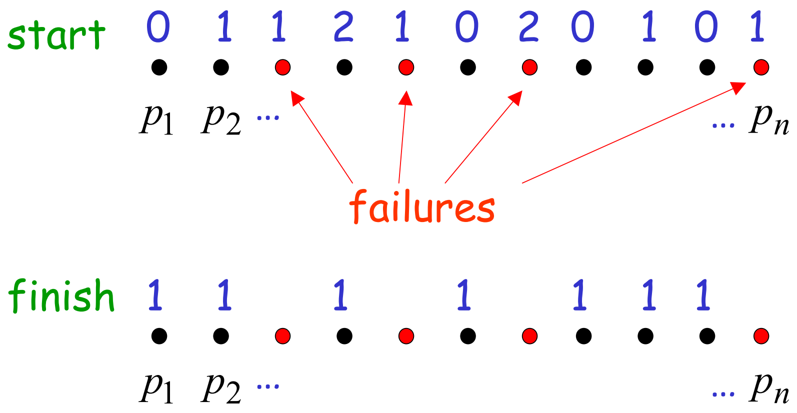
---

The n processes case

Assume for contradiction that there is an f-resilient algorithm A for n processes, where $f \geq n/3$

We will use algorithm A to solve consensus for 3 processes and 1 failure (which is impossible, thus we have a contradiction)

Algorithm A



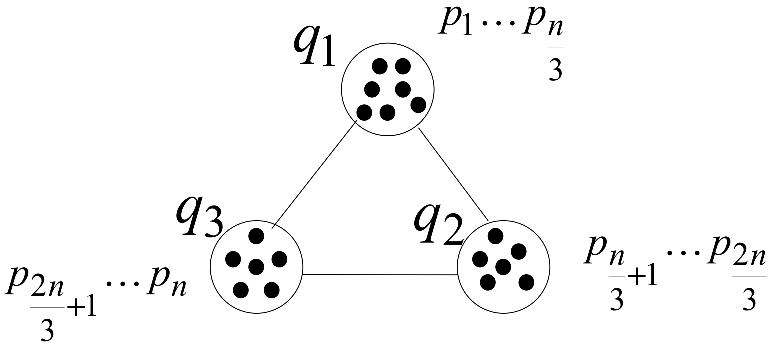
start 0 1 1 2 1 0 2 0 1 0 1

p_1 p_2 … … p_n

failures

finish 1 1 1 1 1 1 1

p_1 p_2 … … p_n

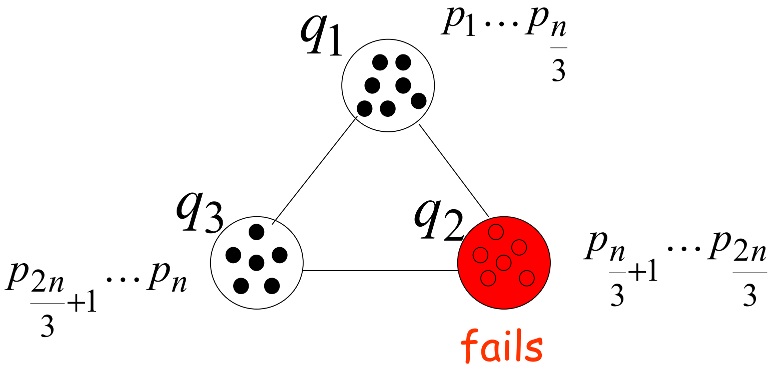


q_1 $p_1 \cdots p_{\frac{n}{3}}$

q_2 $p_{\frac{n}{3}+1} \cdots p_{\frac{2n}{3}}$

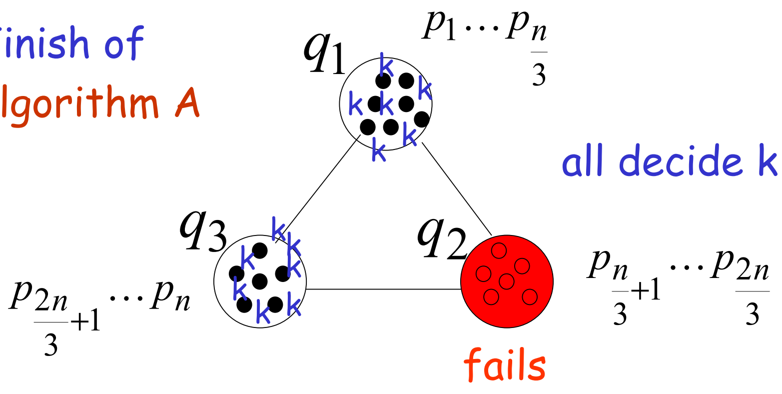
q_3 $p_{\frac{2n}{3}+1} \cdots p_n$

Each process q simulates algorithm A on $n/3$ of "p" processes



When a single *q* is byzantine, then *n*/3 of the "*p*" processes are byzantine too.

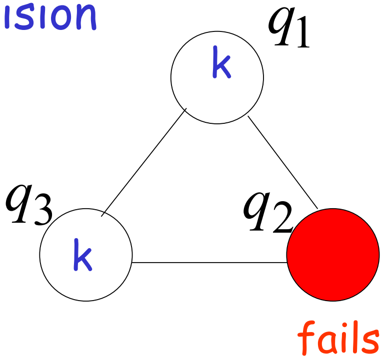
Finish of algorithm A



all decide k

algorithm A tolerates *n*/3 failures

Final decision



We reached consensus with 1 failure

Impossible!!!

Conclusion

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

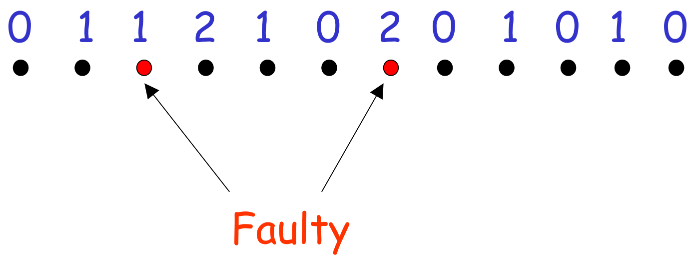
The King Algorithm

solves consensus with n processes and f failures where $f < n/4$ in $f+1$ "phases"

There are f+1 phases
Each phase has two rounds
In each phase there is a different king

Example: 12 processes, 2 faults, 3 kings

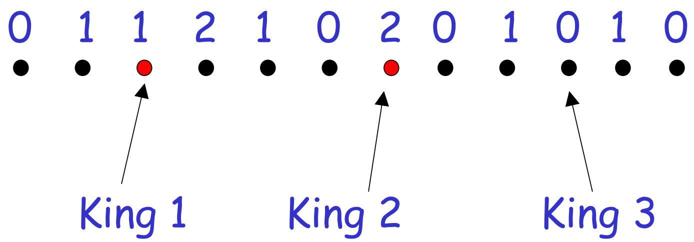
initial values


0 1 1 2 1 0 2 0 1 0 1 0

Faulty

Example: 12 processes, 2 faults, 3 kings

initial values


0 1 1 2 1 0 2 0 1 0 1 0

King 1 King 2 King 3

Remark: There is a king that is not faulty

The **King** algorithm

Each processor p_i has a preferred value v_i

In the beginning, the preferred value is set to the initial value

The **King** algorithm: Phase k

Round 1, processor p_i :

- Broadcast preferred value v_i
- Set v_i to the majority of values received

The **King** algorithm: Phase k

Round 2, king p_k :

- Broadcast new preferred value v_k

Round 2, process p_i :

- If v_i had majority of less than $\frac{n}{2}+f$ then set v_i to v_k

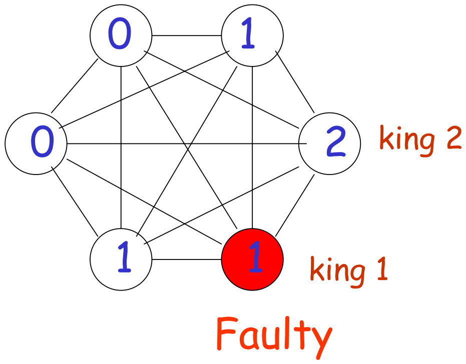
---

The **King** algorithm

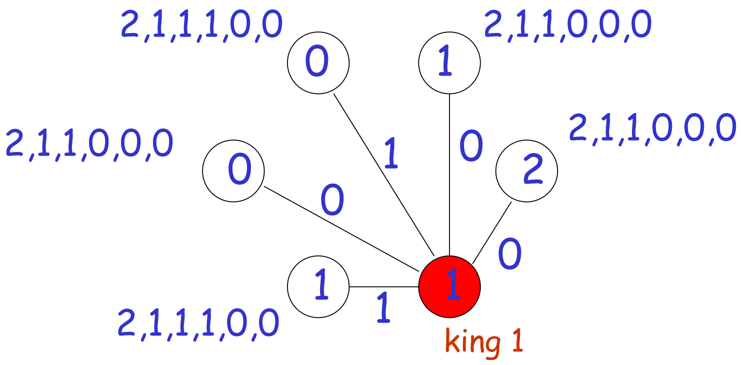
End of Phase f+1:

Each process decides on preferred value

Example: 6 processes, 1 fault



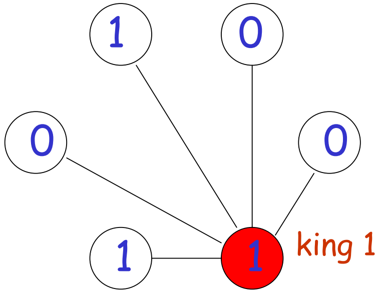
Phase 1, Round 1



Everybody broadcasts

Phase 1, Round 1

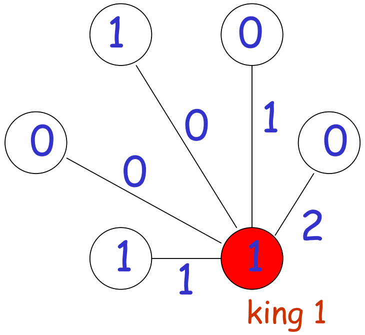
Choose the majority



Each majority population was $3 \le \frac{n}{2} + f = 4$

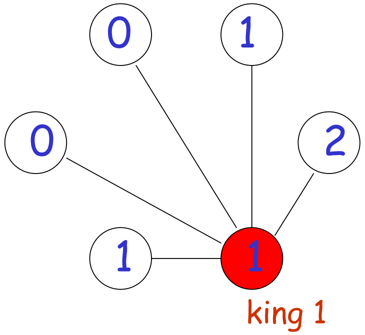
On round 2, everybody will choose the king's value

Phase 1, Round 2



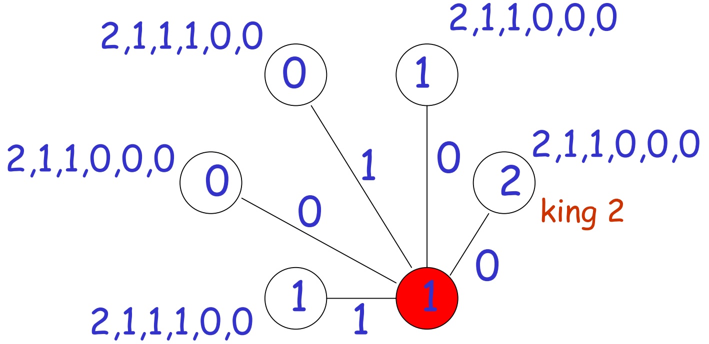
The king broadcasts

Phase 1, Round 2



Everybody chooses the king's value

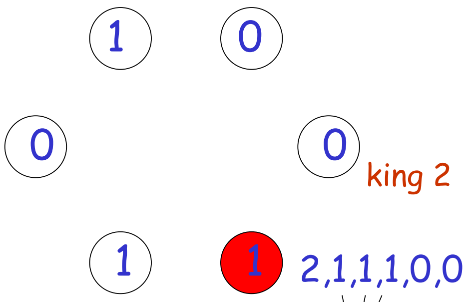
Phase 2, Round 1



Everybody broadcasts

Phase 2, Round 1

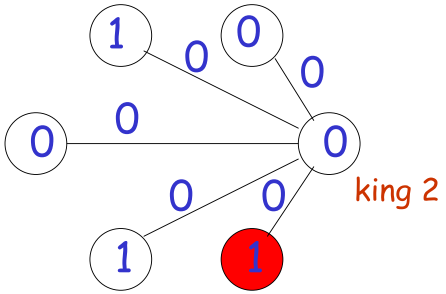
Choose the majority



Each majority population is $3 \le \frac{n}{2} + f = 4$

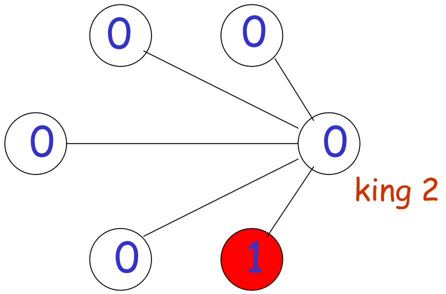
On round 2, everybody will choose the king's value

Phase 2, Round 2



The king broadcasts

Phase 2, Round 2



Everybody chooses the king's value

Final decision

Invariant / Conclusion

In the round where the king is non-faulty, everybody will choose the king's value **v**

After that round, the majority will remain value **v** with a majority population which is at least $n - f > \frac{n}{2} + f$

Exponential Algorithm

solves consensus with n processes and f failures where $f < n/3$ in $f+1$ "phases"

But: uses messages with exponential size

Consensus #6
Randomization

- So far we looked at deterministic algorithms only. We have seen that there is no asynchronous algorithm.
- Can one solve consensus if we allow our algorithms to use randomization?

Yes, we can!

- We tolerate some processes to be faulty (at most f stop failures)
- General idea: Try to push your initial value; if other processes do not follow, try to push one of the suggested values randomly.

Randomized Algorithm

- At most f stop-failures (assume n > 9f)
- For process p_i with initial input $x \in \{0,1\}$:

1. Broadcast Proposal(x, round)
2. Wait for n-f Proposal messages.
3. If at least n-2f messages have value v, then x := v, else x := undecided.

Randomized Algorithm

4. Broadcast Bid(x, round).
5. Wait for n-f Bid messages.
6. If at least n-2f messages have value v,
   then decide on v.
   If at least n-4f messages have value v,
   then x := v.
   Else choose x randomly ($p(0) = p(1) = \frac{1}{2}$)
7. Go back to step 1 (next round).

What do we want?

- Agreement: Non-faulty processes decide non-conflicting values.
- Validity: If all have the same input, that input should be decided.
- Termination: All non-faulty processes *eventually* decide.

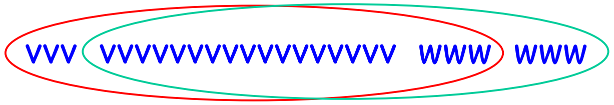
All processes have same input

- Then everybody will agree on that input in the very first round already.
- Validity follows immediately

- If not, then any decision is fine!
- Validity follows too (in any case).

What if process i decides in step 6a (Agreement)…?

- Then process i has received at least n-2f Bid messages with value v.



- Then everybody else has received at least n-3f messages will value v, and thus everybody will propose v next round, and thus decide v.

What about termination?

- We have seen that if a process decides in step 6a, all others will follow in the next round at latest.
- If in step 6b/c, all processes choose the same value (with probability 2^{-n}), all give the same bid, and terminate in the next round.

Byzantine & Asynchronous?

- The presented protocol is in fact already working in the Byzantine case!
- (That's why we have "n-4f" in the protocol and "n-3f" in the proof.)

But termination is awfully slow…

- In expectation, about the same number of processes will choose 1 or 0 in step 6c.
- The probability that a strong majority of processes will propose the same value in the next round is exponentially small.

Naïve Approach

- In step 6c, all processes should choose the same value! (Reason: validity is not a problem anymore since for sure there exist 0's and 1's and therefore we can savely always propose the same…)
- Replace 6c by: "choose x := 1"!

Problem of Naïve Approach

- What if a majority of processes bid 0 in round 4? Then some of the processes might go into 6b (setting x=0), others into 6c (setting x=1). Then the picture is again not clear in the next round
- Anyway: Approach 1 is deterministic! We know (#2) that this doesn't work!

Shared/Common Coin

- The idea is to replace 6c with a subroutine where 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.

Shared Coin Algorithm

Code for process i:

1. Set local coin c_i := 0 with probability 1/n, else (w.h.p.) c_i := 1.
2. Use reliable broadcast* to tell all processes about your local coin c_i.
3. If you receive a local coin c_j of another process j, add j to the set coins_i, and memorize c_j.

Shared Coin Algorithm

4. If you have seen exactly n-f local coins then copy the set coins$_i$ into the set seen$_i$ (but do not stop extending coins$_i$ if you see new coins)
5. Use reliable broadcast to tell all processes about your set seen$_i$.

Shared Coin Algorithm

6. If you have seen at least n-f seen$_j$ which satisfy seen$_j$ $\subseteq$ coins$_i$, then terminate with:
7. If you have seen at least a single local coin with $c_j = 0$ then return 0, else (if you have seen 1-coins only) then return 1.

Why does the shared coin algorithm terminate?

- For simplicity we look at f crash failures only, assuming that 3f < n.
- Since at most f processes crash you will see at least n-f local coins in step 4.
- For the same reason you will see at least n-f seen sets in step 6.
- Since we used reliable broadcast, you will eventually see all the coins that are in the other's sets.

Why does the algorithm work?

- Looks like magic at first…
- General idea: a third of the local coins will be seen by all the processes! If there is a "0" among them we're done. If not, chances are high that there is no "0" at all.
- Proof details: next few slides…

Proof: Matrix

- Let i be the first process to terminate (reach step 7)
- For process i we draw a matrix of all the sets $seen_j$ (columns) and local coins c_k (rows) process i has seen.
- We draw an "X" in the matrix if and only if set $seen_j$ includes coin c_k.

Proof: Matrix (f=2, n=7, n-f=5)

	$seen_1$	$seen_3$	$seen_5$	$seen_6$	$seen_7$
$coin_1$	X	X	X	X	X
$coin_2$			X	X	X
$coin_3$	X	X	X	X	X
$coin_5$	X	X	X		X
$coin_6$	X	X	X	X	
$coin_7$	X	X		X	X

- Note that there are at least $(n-f)^2$ X's in this matrix ($\geq$n-f rows, n-f X's in each row).

Proof: Matrix

- Lemma 1: There are at least f+1 rows where at least f+1 cells have an "X".
- Proof: Suppose by contradiction that this is not the case. Then the number of X is bounded from above by $f \cdot (n-f) + (n-f) \cdot f$, …

Few rows have many X

All other rows have at most f X

Proof: Matrix

$|X| \leq 2f(n-f)$

we use $3f < n \rightarrow 2f < n-f$

$< (n-f)^2$

but we know that $|X| \geq (n-f)^2$

$\leq |X|$.

A contradiction!

Proof: The set W

- Let W be the set of local coins where the rows in the matrix have more than f X's.
- Lemma 2: All local coins in the set W are seen by all processes (that terminate).
- Proof: Let $w \in W$ be such a local coin. With Lemma 1 we know that w is at least in f+1 seen sets. Since each process must see at least n-f seen sets (before terminating), these sets overlap, and w will be seen.

Proof: End game

- Theorem: With constant probability all processes decide 0, with constant probability all processes decide 1.
- Proof: With probability $(1-1/n)^n \approx 1/e$ all processes choose $c_i = 1$, and therefore all will decide 1.
- With probability $1-((1-1/n)^{|W|})$ there is at least one 0 in the set W. Since $|W| \approx n/3$ this probability is constant. Using Lemma 2 we know that in this case all processes will decide 0.

Back to Randomized Consensus

- Plugging the shared coin back into the randomized consensus algorithm is all we needed.
- If some of the processes go into 6b and, the others still have a constant chance that they will agree on the same shared coin.
- The randomized consensus protocol finishes in a constant number of rounds!

Improvements

- For crash-failures, there is a constant expected time algorithm which tolerates f failures with $2f < n$.
- For Byzantine failures, there is a constant expected time algorithm which tolerates f failures with $3f < n$.
- Similar algorithms have been proposed for the shared memory model.

Databases et al.

- Consensus plays a vital role in many distributed systems, most notably in distributed databases:
  - Two-Phase-Commit (2PC)
  - Three-Phase-Commit (3PC)

Summary

- We have solved consensus in a variety of models; particularly we have seen
  - algorithms
  - wrong algorithms
  - lower bounds
  - impossibility results
  - reductions
  - etc.

Credits

- The impossibility result (#2) is from Fischer, Lynch, Patterson, 1985.
- The hierarchy (#3) is from Herlihy, 1991.
- The synchronous studies (#4) are from Dolev and Strong, 1983, and others.
- The Byzantine studies (#5) are from Lamport, Shostak, Pease, 1980ff., and others.
- The first randomized algorithm (#6) is from Ben-Or, 1983.

Questions?

Distributed
Computing
Group
Roger Wattenhofer