Part 2: Fault-Tolerance
Distributed Systems 2014

Roger Wattenhofer

Overview Part 2

- Lecture (Monday 9-11 & Friday 9-11, usually)
  - It's all about fault-tolerance
  - First theory, in particular consensus, models, algorithms, and lower bounds
  - Then large-scale practice, fault-tolerant systems
  - Finally small-scale practice, programming, multi-core

- Exercises (Monday 11-12 & Friday 11-12, sometimes)
  - There will be paper exercises, without grades
  - Exercises don't have to be handed in…
    …but you are strongly encouraged to solve them!
  - But: There will be a creative task that gives exam points (10 or 20).

- Personnel
  - Roger Wattenhofer, Christian Decker, Michael König, Jara Uitto
  - www.disco.ethz.ch
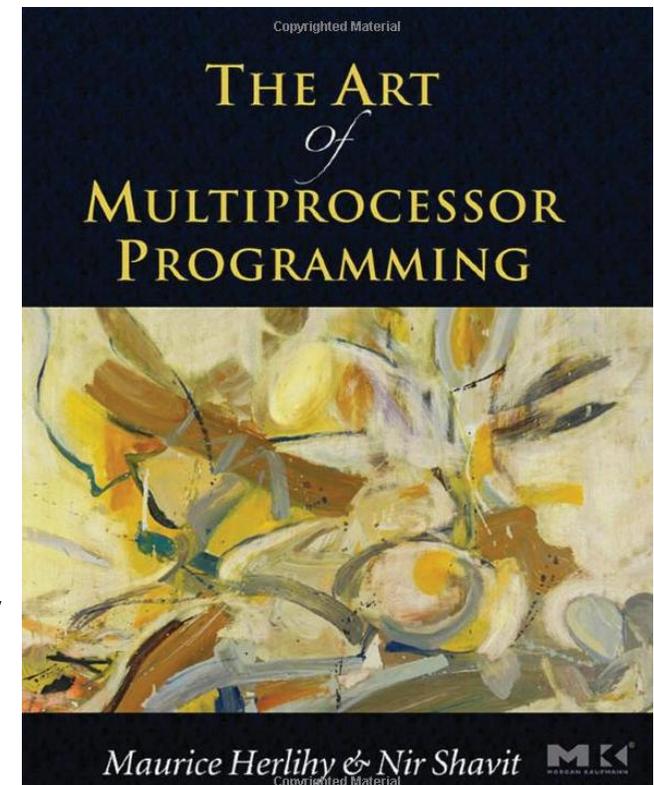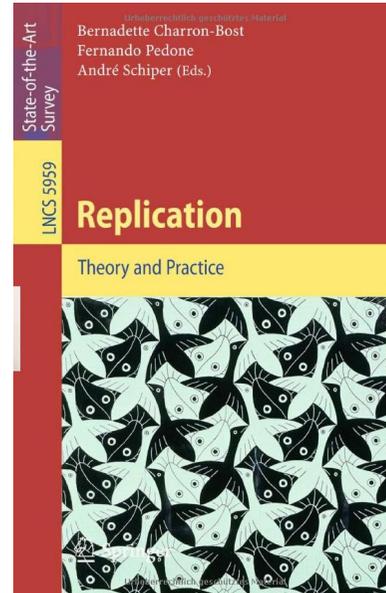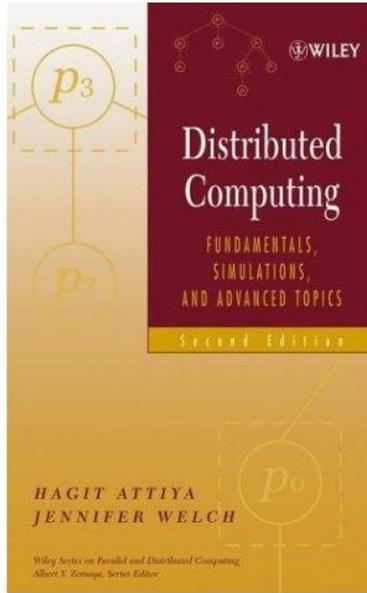
Why are we studying distributed systems?

- First, and most importantly: The world is distributed!
  - Companies with offices around the world
  - Computer networks, in particular the Internet

- Performance
  - Parallel high performance computing
  - Multi-core machines

- Fault-Tolerance
  - Availability
  - Reliability

Book for Part 2

- Great book
- Goes beyond class
- Fully covers Chapter 3
- Does not cover everything we do (but also some parts of Chapters 1 & 2)

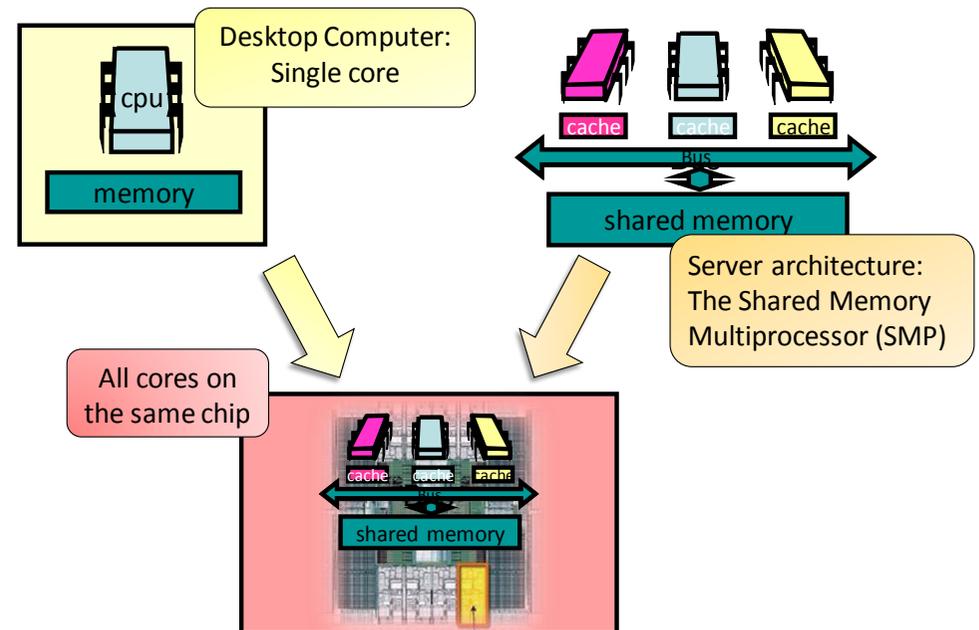- Some pictures on slides are from Maurice Herlihy

  (Thanks, Maurice!)

THE ART OF MULTIPROCESSOR PROGRAMMING

Maurice Herlihy & Nir Shavit

Books for Chapters 1 & 2

Consensus
Part 2, Chapter 1

Roger Wattenhofer

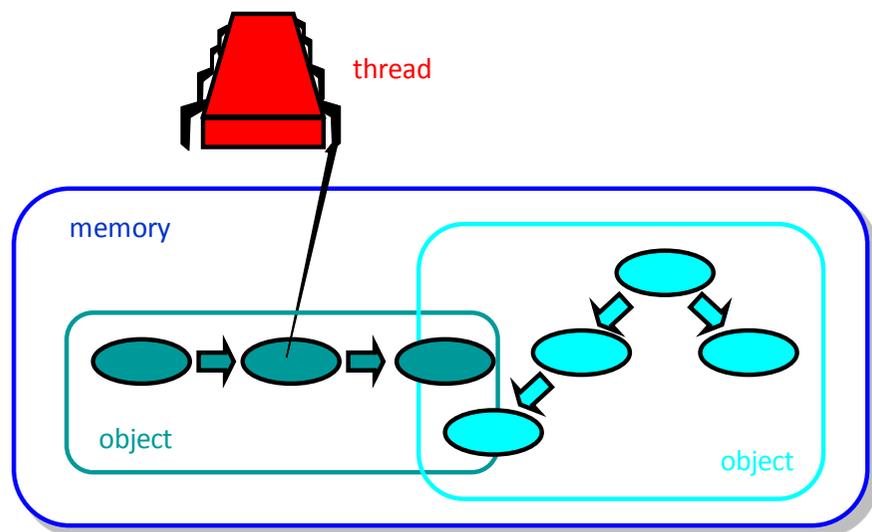*ETH Zurich – Distributed Computing – www.disco.ethz.ch*

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

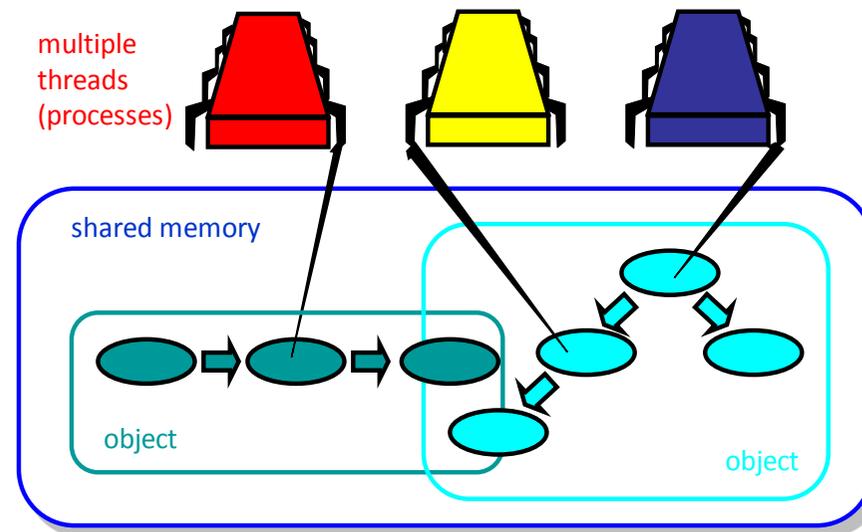## Introduction: From Single-Core to Multicore Computers

Desktop Computer: Single core

Server architecture: The Shared Memory Multiprocessor (SMP)
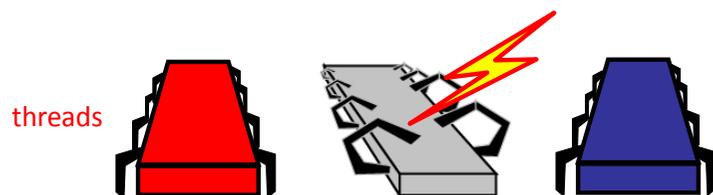
All cores on the same chip

Sequential Computation

Concurrent Computation

Fault Tolerance & Asynchrony

- Why fault-tolerance?
  - Even if processes do not die, there are "near-death experiences"
- Sudden unpredictable delays:
  - Cache misses (short)
  - Page faults (long)
  - Scheduling quantum used up (really long)

Road Map

- In this first part, we are going to focus on principles
  - Start with idealized models
  - Look at a simplistic problem
  - Emphasize correctness over pragmatism
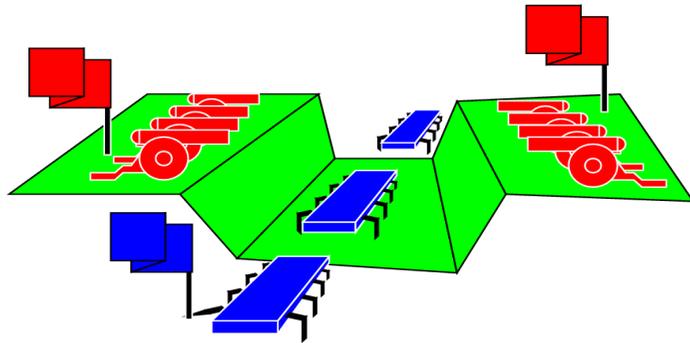 - "Correctness may be theoretical, but incorrectness has practical impact"

I'm no theory weenie! Why all the theorems and proofs?

- Distributed systems are hard
  - Failures
  - Concurrency
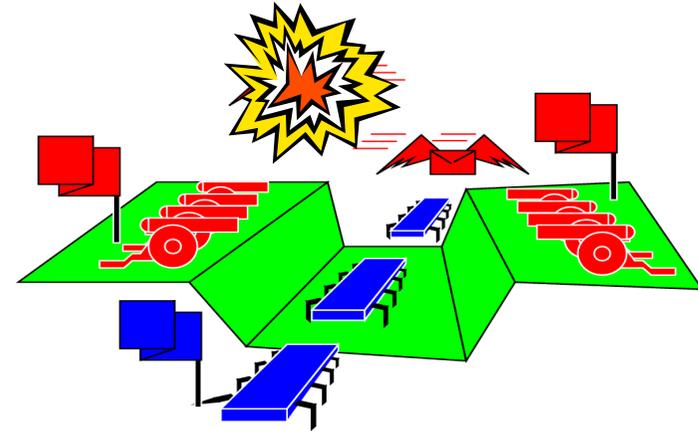
- Easier to go from theory to practice than vice-versa

The Two Generals

- Red army wins if both sides attack simultaneously
- Red army can communicate by sending messages…

Problem: Unreliable Communication

- … such as "let's attack tomorrow at 6am" …
- … but messages do not always make it!
- Task: Design a "red army protocol" that works despite message failures!

Real World Generals

```
Date: Wed, 11 Dec 2002 12:33:58 +0100
From: Friedemann Mattern <mattern@inf.ethz.ch>
To: Roger Wattenhofer <wattenhofer@inf.ethz.ch>
Subject: Vorlesung


Sie machen jetzt am Freitag, 08:15 die Vorlesung
Verteilte Systeme, wie vereinbart. OK? (Ich bin
jedenfalls am Freitag auch gar nicht da.) Ich
uebernehme das dann wieder nach den
Weihnachtsferien.
```

Real World Generals

```
Date: Mi 11.12.2002 12:34
From: Roger Wattenhofer <wattenhofer@inf.ethz.ch>
To: Friedemann Mattern <mattern@inf.ethz.ch>
Subject: Re: Vorlesung


OK. Aber ich gehe nur, wenn sie diese Email nochmals
bestaetigen... :-)
```

Real World Generals

Date: Wed, 11 Dec 2002 12:53:37 +0100
From: Friedemann Mattern <mattern@inf.ethz.ch>
To: Roger Wattenhofer <wattenhofer@inf.ethz.ch>
Subject: Naechste Runde: Re: Vorlesung

Das dachte ich mir fast. Ich bin Praktiker und mache
es schlauer: Ich gehe nicht, unabhaengig davon, ob
Sie diese email bestaetigen (beziehungsweise
rechtzeitig erhalten). (:-)

Real World Generals

Date: Mi 11.12.2002 13:01
From: Roger Wattenhofer <wattenhofer@inf.ethz.ch>
To: Friedemann Mattern <mattern@inf.ethz.ch>
Subject: Re: Naechste Runde: Re: Vorlesung ...

Ich glaube, jetzt sind wir so weit, dass ich diese
Emails in der Vorlesung auflegen werde...

Real World Generals

Date: Wed, 11 Dec 2002 18:55:08 +0100
From: Friedemann Mattern <mattern@inf.ethz.ch>
To: Roger Wattenhofer <wattenhofer@inf.ethz.ch>
Subject: Re: Naechste Runde: Re: Vorlesung ...

Kein Problem. (Hauptsache es kommt raus, dass der
Prakiker am Ende der schlauere ist... Und der
Theoretiker entweder heute noch auf das allerletzte
Ack wartet oder wissend das das ja gar nicht gehen
kann alles gleich von vornherein bleiben laesst...
(:-))

Theorem

Theorem
There is no non-trivial protocol that ensures
that the red armies attack simultaneously

Proof:

1. Consider the protocol that sends the fewest messages
2. It still works if the last message is lost
3. So just don't send it (messengers' union happy!)
4. But now we have a shorter protocol!
5. Contradicting #1

Fundamental limitation: We need an unbounded number of
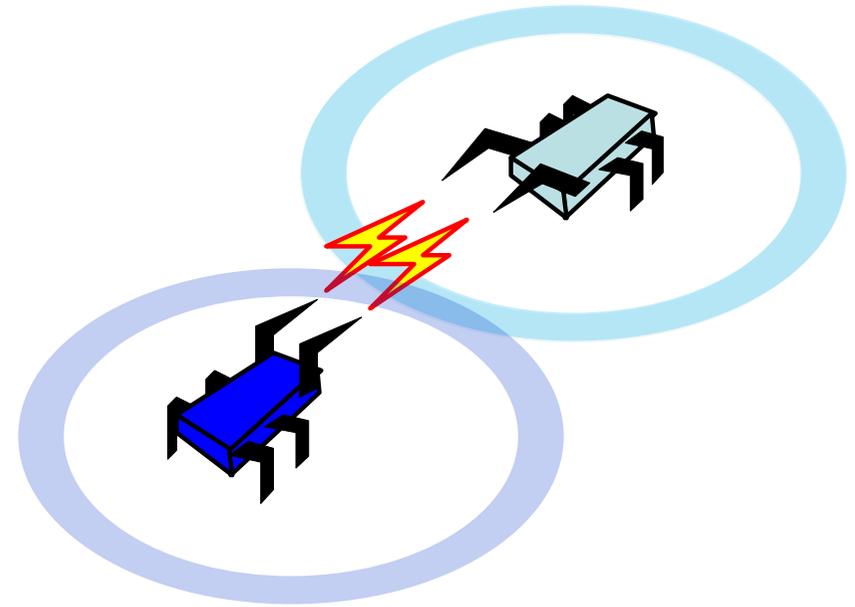messages, otherwise it is possible that no attack takes place!

Consensus Definition: Each Thread has a Private Input
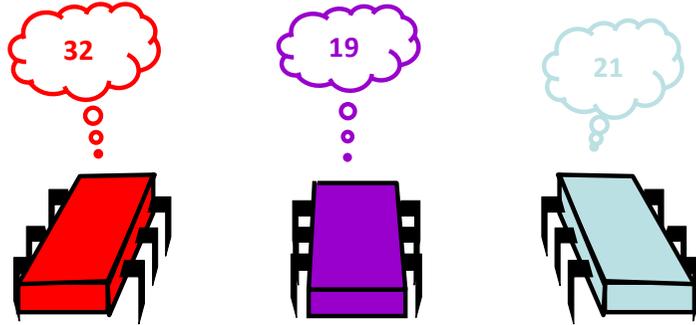
Consensus Definition: The Threads Communicate

Consensus Definition: They Agree on Some Thread's Input
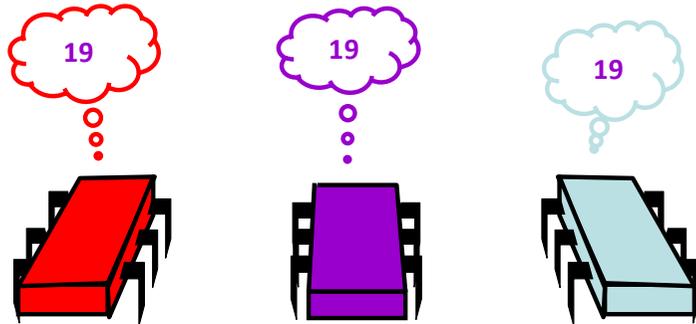
Consensus is Important

- With consensus, you can implement anything you can imagine…

- Examples:
  - With consensus you can decide on a leader,
  - implement mutual exclusion,
  - or solve the two generals problem
  - and much more…

- We will see that in some models, consensus is possible, in some other models, it is not

- The goal is to learn whether for a given model consensus is possible or not … and prove it!

Consensus #1: Shared Memory

- $n > 1$ processors
- Shared memory is memory that may be accessed simultaneously by multiple threads/processes.
- Processors can atomically *read* or *write* (not both) a shared memory cell

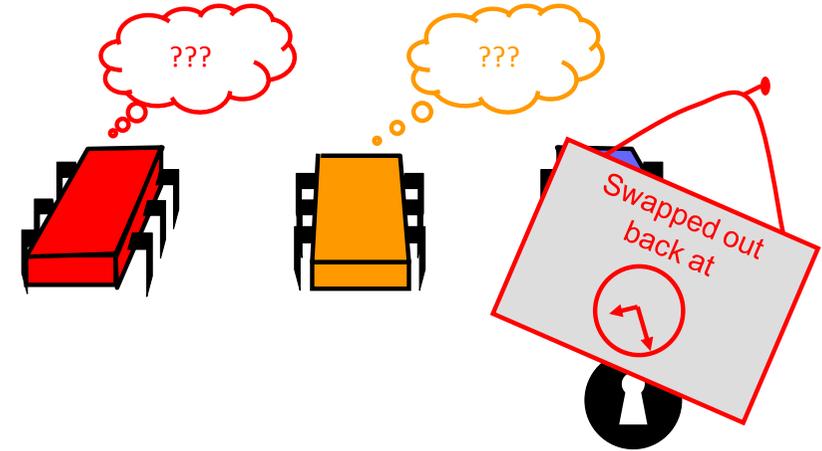**Protocol:**

- There is a designated memory cell c.
- Initially c is in a special state "?"
- Processor 1 writes its value v_1 into c, then decides on v_1.
- A processor $j \neq 1$ reads c until j reads something else than "?", and then decides on that.
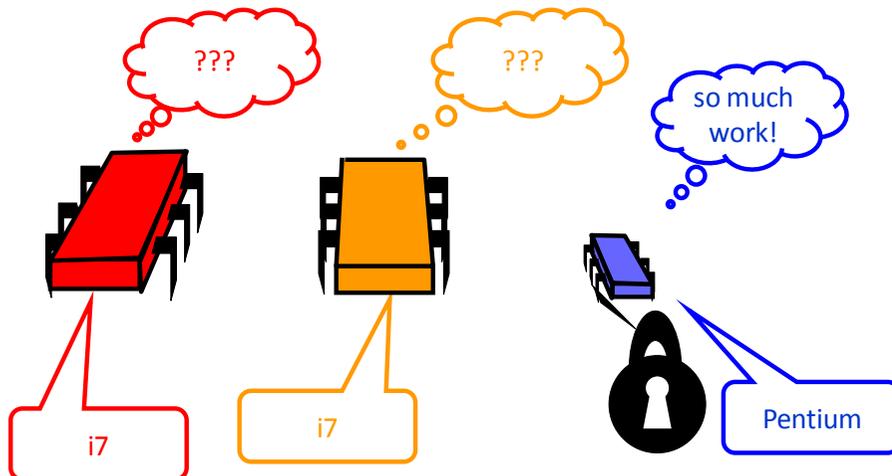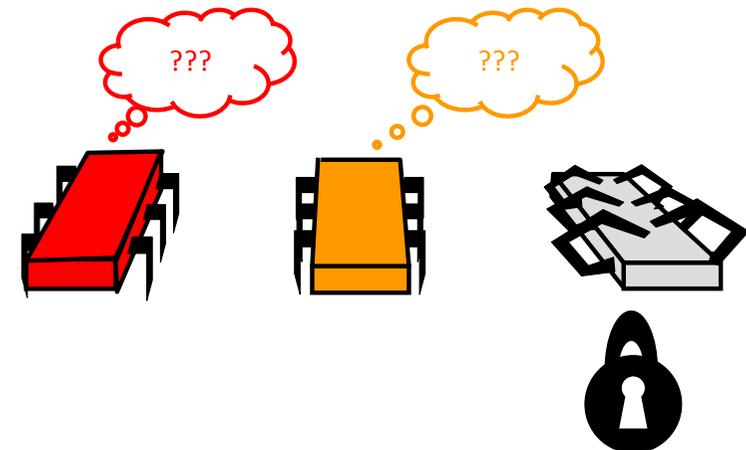
- Problems with this approach?
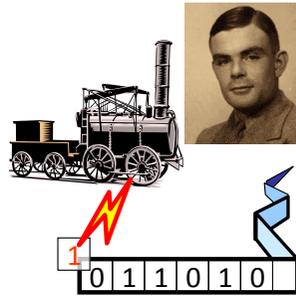
Unexpected Delay

Heterogeneous Architectures

Fault-Tolerance

Computability

- Definition of computability
  - Computable usually means Turing-computable, i.e., the given problem can be solved using a Turing machine
  - Strong mathematical model!

| 0 | 1 | 1 | 0 | 1 | 0 |

- Shared-memory computability
  - Model of asynchronous concurrent computation
  - Computable means it is wait-free computable on a multiprocessor
  - Wait-free…?
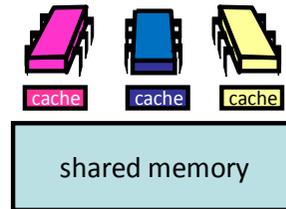
shared memory

Consensus #2: Wait-free Shared Memory

- $n > 1$ processors
- Processors can atomically *read* or *write* (not both) a shared memory cell
- Processors might crash (stop… or become very slow…)

Wait-free implementation:

- Every process (method call) completes in a finite number of steps
- Implies that locks cannot be used → The thread holding the lock may crash and no other thread can make progress
- We assume that we have wait-free atomic registers (that is, reads and writes to same register do not overlap)

A Wait-free Algorithm

- There is a cell c, initially c="?"
- Every processor i does the following:

```
r = Read(c);
if (r == "?") then
    Write(c, vᵢ); decide vᵢ;
else
    decide r;
```

- Is this algorithm correct…?

An Execution

Atomic read/write register

cell c

| ? |
| ? |
| ? |
| **32** |
| **17** |

32!   17!

time

Execution Tree

Initial state

?/?

read — read

?/? — ?/?

write — read — read — write

32/? — ?/? — ?/? — ?/17

read — write — write — write — write — read

32/32 — 32/? — ?/17 — 32/? — ?/17 — 17/17

✔

write — write — write — write

32/17 — 32/17 — 32/17 — 32/17

✔
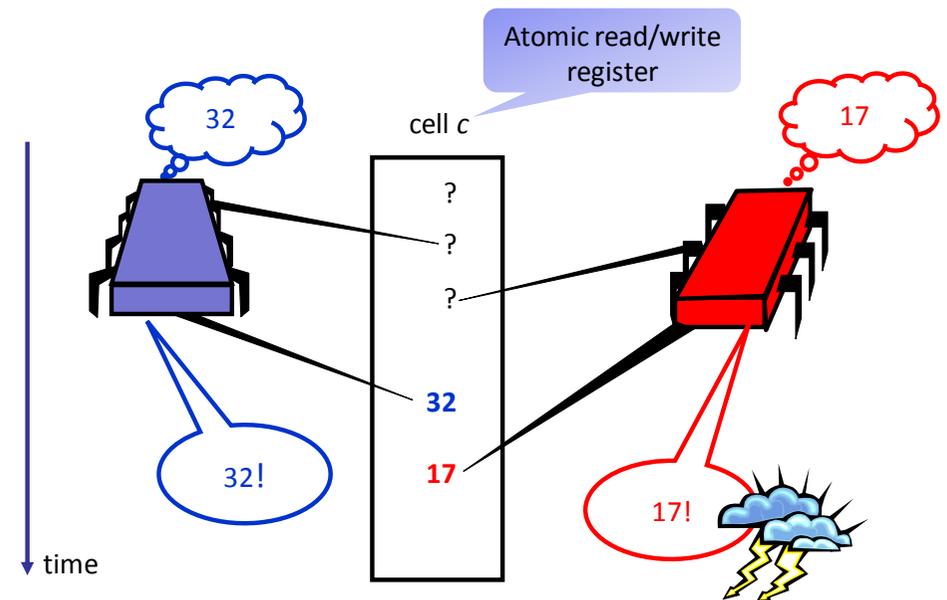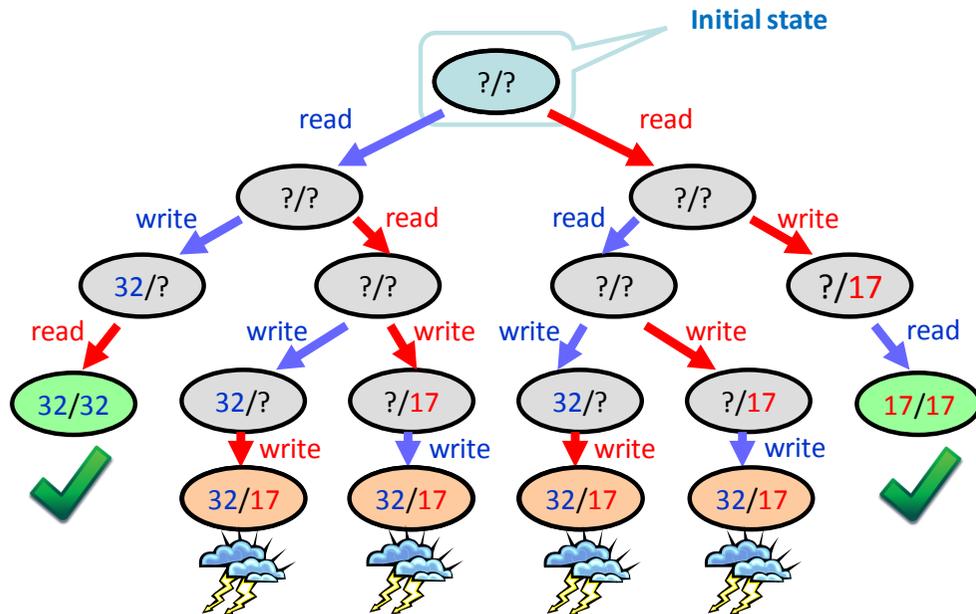
Theorem

Theorem
> There is no wait-free consensus algorithm using read/write atomic registers

??? ???
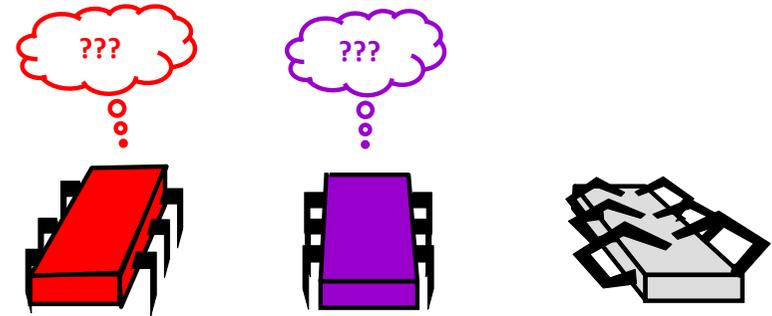
Proof

- Make it simple
  - There are only two threads A and B and the input is binary
- Assume that there is a protocol
- In this protocol, either A or B "moves" in each step
- Moving means
  - Register read
  - Register write

A moves — B moves
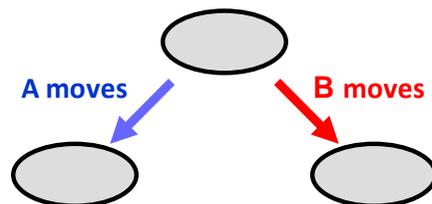
Execution Tree (of abstract but "correct" algorithm)

bivalent

Initial state

critical (univalent with the next step)

univalent

(0-valent) — (1-valent)

1 — 0 — 0 — 1 — 1 — 0

Final states (decision values)
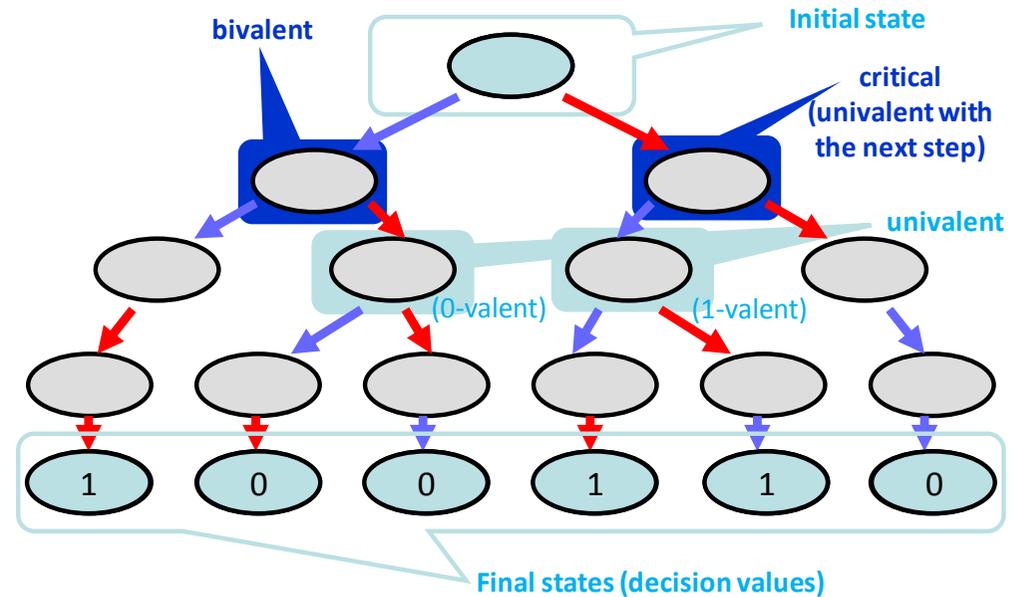
Bivalent vs. Univalent

- Wait-free computation is a tree
- Bivalent system states
  - Outcome is not fixed
- Univalent states
  - Outcome is fixed
  - Maybe not "known" yet
  - 1-valent and 0-valent states

- Claim
  - Some initial system state is bivalent
  - This means that the outcome is not always fixed from the start

Proof of Claim: A 0-Valent Initial State

- All executions lead to the decision 0
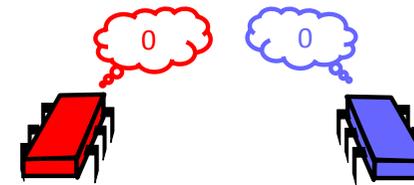
Similarly, the decision is always 1 if both threads start with 1!

- Solo executions also lead to the decision 0
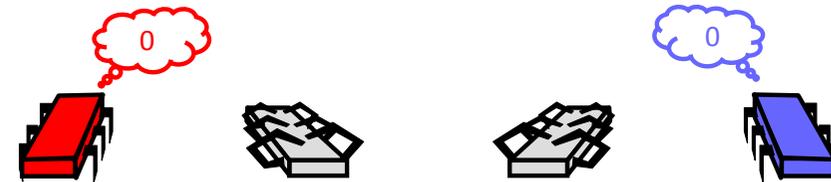
Proof of Claim: Indistinguishable Situations

- These two situations are indistinguishable → The outcome must be the same

The decision is 0!

The decision is 0!

Similarly, the decision is 1 if the red thread crashed!
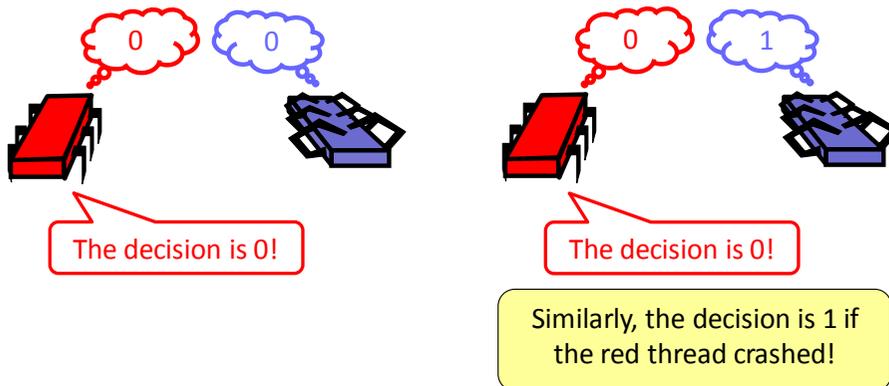
Proof of Claim: A Bivalent Initial State

Decision: 0

Decision: 1

Decision: 0

Decision: 1

Decision: 1?

Decision: 0?

This state is bivalent!

Critical States

- Starting from a bivalent initial state
- The protocol must reach a critical state
  - Otherwise we could stay bivalent forever
  - And the protocol is not wait-free
- The goal is now to show that the system can always remain bivalent
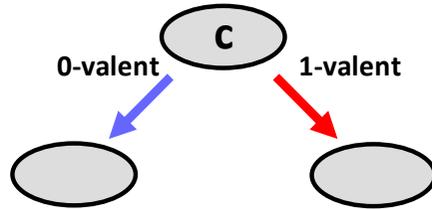
> A state is critical if the next state is univalent

0-valent   c   1-valent

Reaching a Critical State

- The system can remain bivalent forever if there is always an action that prevents the system from reaching a critical state:

b
A moves   B moves
b   1   1-valent
A moves   B moves
1   b   1-valent
A moves   B moves
b   0   0-valent
A moves   B moves
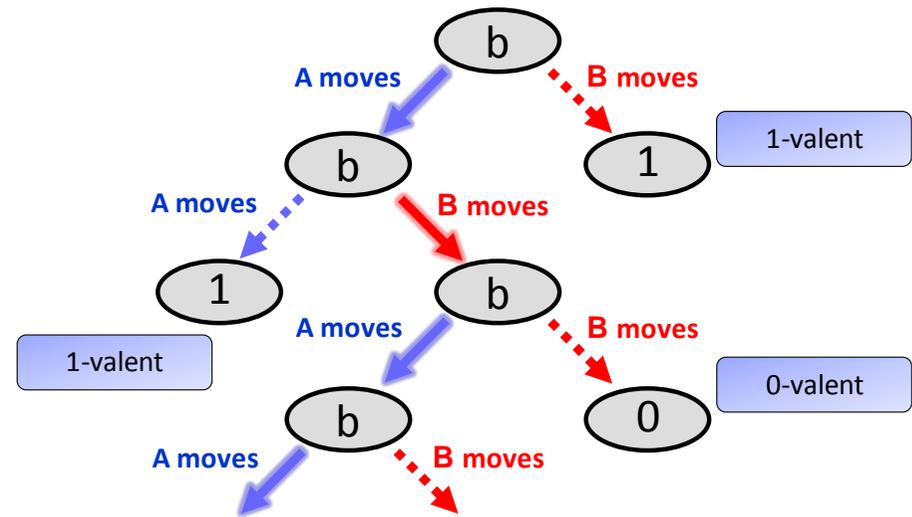
Model Dependency

- So far, everything was memory-independent!

- True for
  - Registers
  - Message-passing
  - Carrier pigeons
  - Any kind of asynchronous computation

- Threads
  - Perform reads and/or writes
  - To the same or different registers
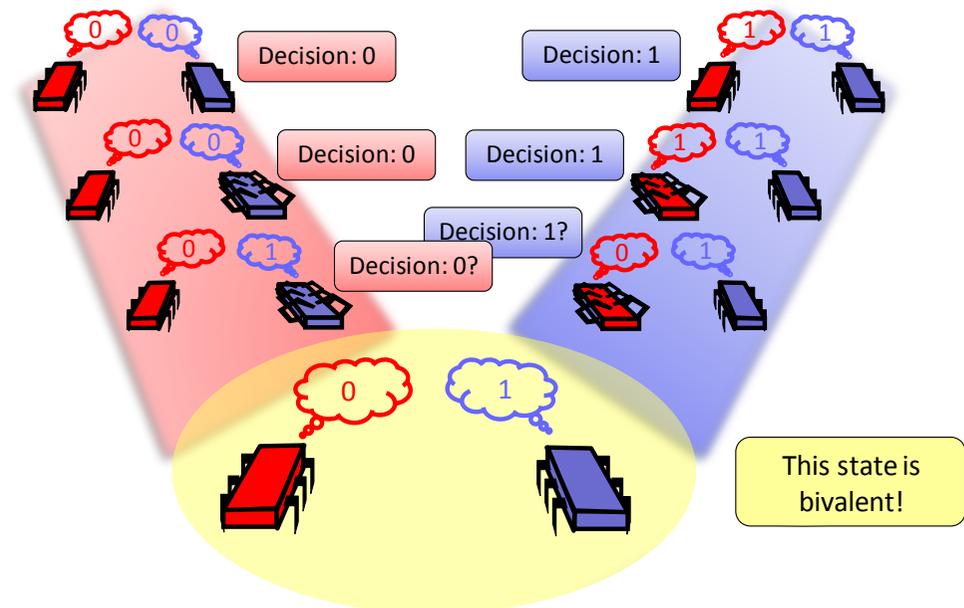 - Possible interactions?

Possible Interactions

	x.read()	y.read()	x.write()	y.write()
x.read()	?	?	?	?
y.read()	?	?	?	?
x.write()	?	?	?	?
y.write()	?	?	?	?

> A reads x

> B writes y

Reading Registers
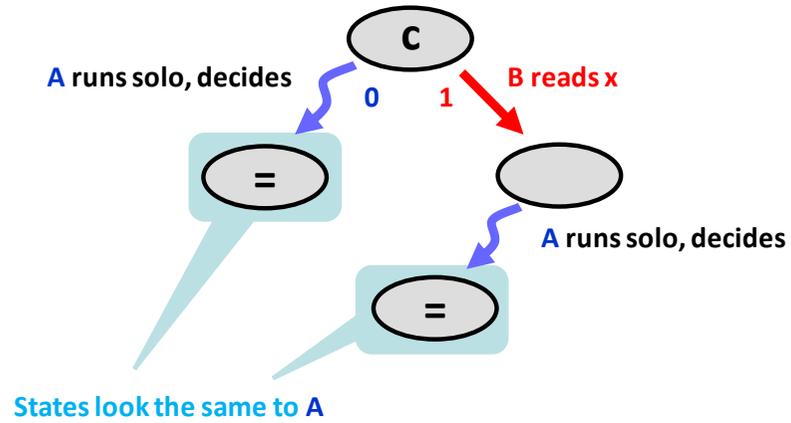
A runs solo, decides

B reads x

A runs solo, decides

States look the same to A

Possible Interactions

	x.read()	y.read()	x.write()	y.write()
x.read()	no	no	no	no
y.read()	no	no	no	no
x.write()	no	no	?	?
y.write()	no	no	?	?

Writing Distinct Registers
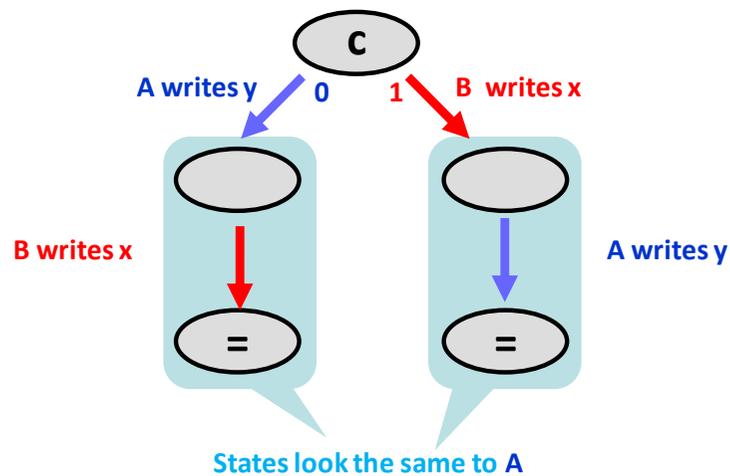
A writes y

B writes x

B writes x

A writes y

States look the same to A

Possible Interactions

	x.read()	y.read()	x.write()	y.write()
x.read()	no	no	no	no
y.read()	no	no	no	no
x.write()	no	no	?	no
y.write()	no	no	no	?

Writing Same Registers

A writes x   **0**   **1**   **B writes x**

A runs solo, decides

A writes x

A runs solo, decides

=

=

States look the same to A
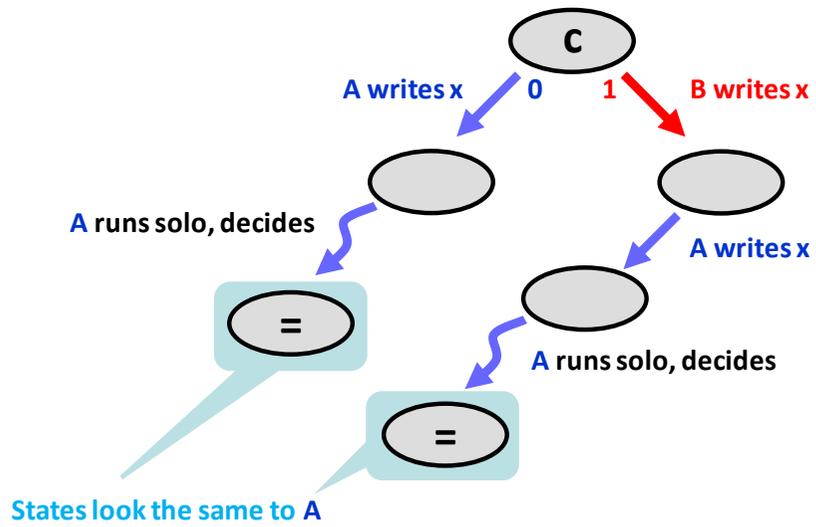
That's All, Folks!

	x.read()	y.read()	x.write()	y.write()
x.read()	no	no	no	no
y.read()	no	no	no	no
x.write()	no	no	no	no
y.write()	no	no	no	no

What Does Consensus Have to Do With Distributed Systems?

- We want to build a concurrent FIFO Queue with multiple dequeuers
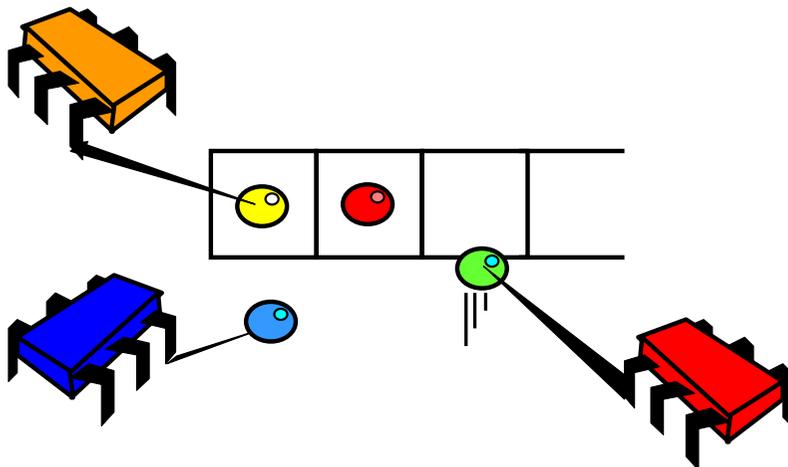
A Consensus Protocol

- Assume we have such a FIFO queue and a 2-element array

2-element array

FIFO Queue with red and black balls

Coveted red ball   **Dreaded black ball**

A Consensus Protocol

- Thread i writes its value into the array at position i
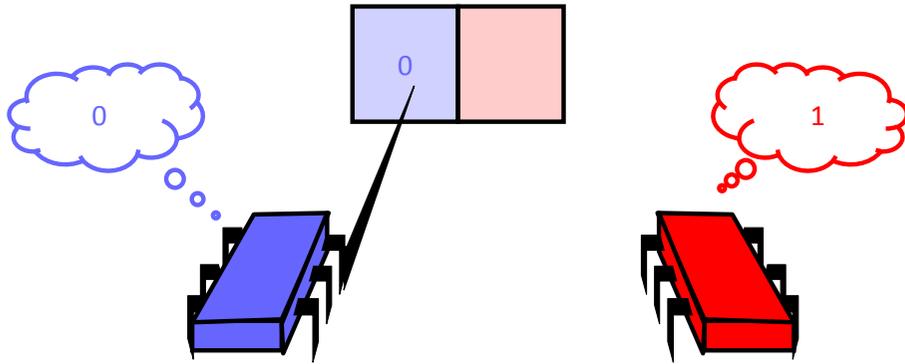
A Consensus Protocol

- Then, the thread takes the next element from the queue

A Consensus Protocol

I got the coveted red ball, so I will decide my value

I got the dreaded black ball, so I will decide the other's value from the array

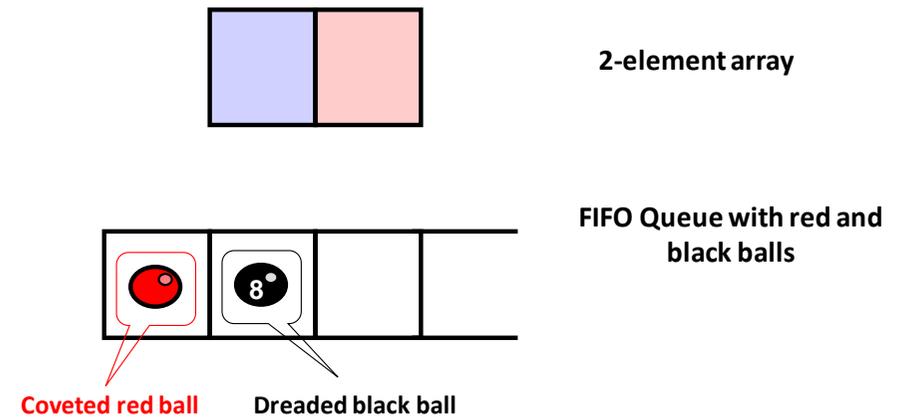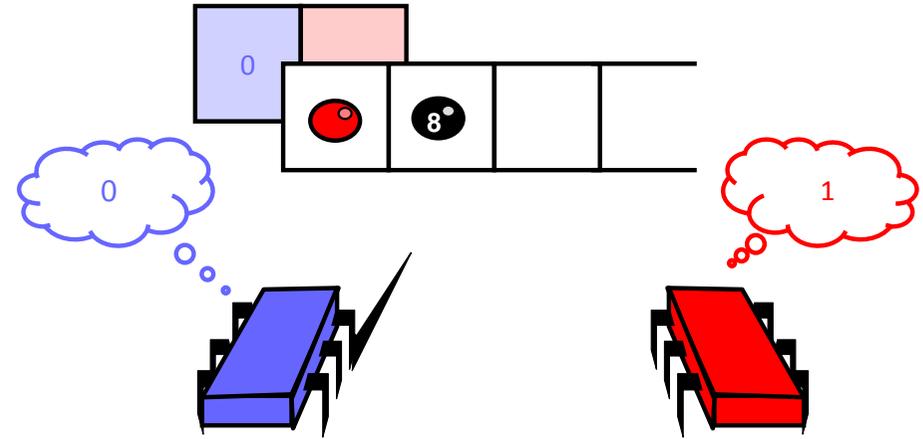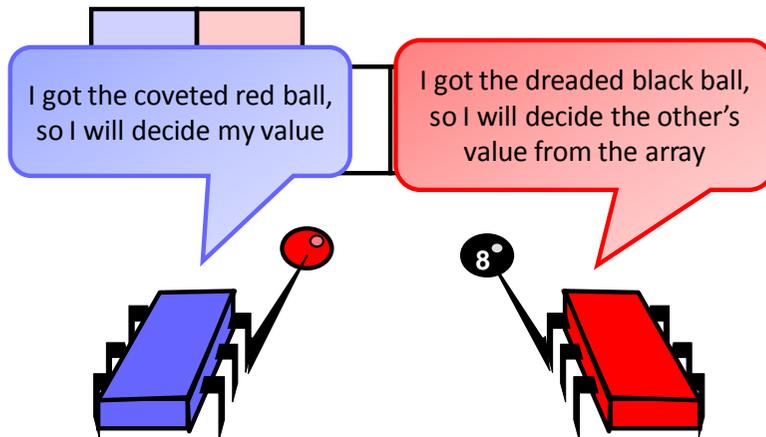## A Consensus Protocol

Why does this work?

- If one thread gets the red ball, then the other gets the black ball
- Winner can take its own value
- Loser can find winner's value in array
  - Because threads write array before dequeuing from queue

Implication

- We can solve 2-thread consensus using only
  - A two-dequeuer queue
  - Atomic registers

Implications

- Assume there exists
  - A queue implementation from atomic registers
- Given
  - A consensus protocol from queue and registers
- Substitution yields
  - A wait-free consensus protocol from atomic registers

contradiction

Corollary

- It is impossible to implement a two-dequeuer wait-free FIFO queue with read/write shared memory.
- This was a proof by reduction; important beyond NP-completeness…

Consensus #3: Read-Modify-Write Shared Memory

- $n > 1$ processors
- Wait-free implementation
- Processors can read *and* write a shared memory cell in one atomic step: the value written can depend on the value read
- We call this a read-modify-write (RMW) register
- Can we solve consensus using a RMW register…?

Consensus Protocol Using a RMW Register

- There is a cell c, initially c="?"
- Every processor i does the following

RMW(c)

```
if (c == "?") then
    write(c, vᵢ); decide vᵢ
else
    decide c;
```

atomic step

Discussion

- Protocol works correctly
  - One processor accesses c first; this processor will determine decision
- Protocol is wait-free
- RMW is quite a strong primitive
  - Can we achieve the same with a weaker primitive?

Read-Modify-Write More Formally

- Method takes 2 arguments:
  - Cell **c**
  - Function **f**
- Method call:
  - Replaces value **x** of cell **c** with **f(x)**
  - Returns value **x** of cell **c**

Read-Modify-Write

```
public class RMW {
    private int value;

    public synchronized int rmw(function f) {
        int prior = this.value;
        this.value = f(this.value);
        return prior;
    }

}
```

Return prior value

Apply function

Read-Modify-Write: Read

```
public class RMW {
    private int value;

    public synchronized int read() {
        int prior = this.value;
        this.value = this.value;
        return prior;
    }

}
```

Identify function

Read-Modify-Write: Test&Set

```
public class RMW {
    private int value;

    public synchronized int TAS() {
        int prior = this.value;
        this.value = 1;
        return prior;
    }

}
```

Constant function

Read-Modify-Write: Fetch&Inc

```java
public class RMW {
  private int value;

  public synchronized int FAI() {
    int prior  = this.value;
    this.value = this.value+1;
    return prior;
  }

}
```

Increment function

Read-Modify-Write: Fetch&Add

```java
public class RMW {
  private int value;

  public synchronized int FAA(int x) {
    int prior  = this.value;
    this.value = this.value+x;
    return prior;
  }

}
```

Addition function

Read-Modify-Write: Swap

```java
public class RMW {
  private int value;

  public synchronized int swap(int x) {
    int prior  = this.value;
    this.value = x;
    return prior;
  }

}
```

Set to x

Read-Modify-Write: Compare&Swap

```java
public class RMW {
  private int value;

  public synchronized int CAS(int old, int new) {
    int prior  = this.value;
    if(this.value == old)
      this.value = new;
    return prior;
  }

}
```

"Complex" function

Definition of Consensus Number

- An object has consensus number n
  - If it can be used
    - Together with atomic read/write registers
  - To implement n-thread consensus, but not $(n+1)$-thread consensus

- Example: Atomic read/write registers have consensus number 1
  - Works with 1 process
  - We have shown impossibility with 2

Consensus Number Theorem

> **Theorem**
>
> If you can implement X from Y
> and X has consensus number c,
> then Y has consensus number at least c

- Consensus numbers are a useful way of measuring synchronization power
- An alternative formulation:
  - If X has consensus number c
  - And Y has consensus number $d < c$
  - Then there is no way to construct a wait-free implementation of X by Y
- This theorem will be very useful
  - Unforeseen practical implications!

Theorem

- A RMW is *non-trivial* if there exists a value v such that $v \neq f(v)$
  - Test&Set, Fetch&Inc, Fetch&Add, Swap, Compare&Swap, general RMW…
  - But not read

> **Theorem**
>
> Any non-trivial RMW object has
> consensus number at least 2

- Implies no wait-free implementation of RMW registers from read/write registers
- Hardware RMW instructions not just a convenience

Proof

- A two-thread consensus protocol using any non-trivial RMW object:

```
public class RMWConsensusFor2 implements Consensus{
  private RMW r;                    Initialized to v

  public Object decide() {
    int i  = Thread.myIndex();
    if(r.rmw(f) == v)                Am I first?
      return this.announce[i];
    else                            Yes, return
      return this.announce[1-i];     my input
  }

}                                   No, return
                                    other's input
```

Interfering RMW

- Let F be a set of functions such that for all f_i and f_j, either
  - They commute: $f_i(f_j(x))=f_j(f_i(x))$
  - They overwrite: $f_i(f_j(x))=f_i(x)$

  > $f_i(x)$ = new value of cell (not return value of f_i)

- Claim: Any such set of RMW objects has consensus number **exactly 2**

Examples:

- Overwrite
  - Test&Set , Swap
- Commute
  - Fetch&Inc, Fetch&Add

Proof

- There are three threads, A, B, and C
- Consider a critical state c:
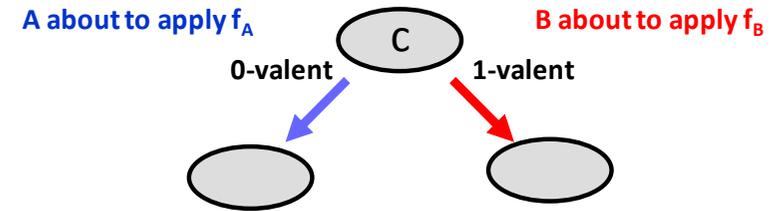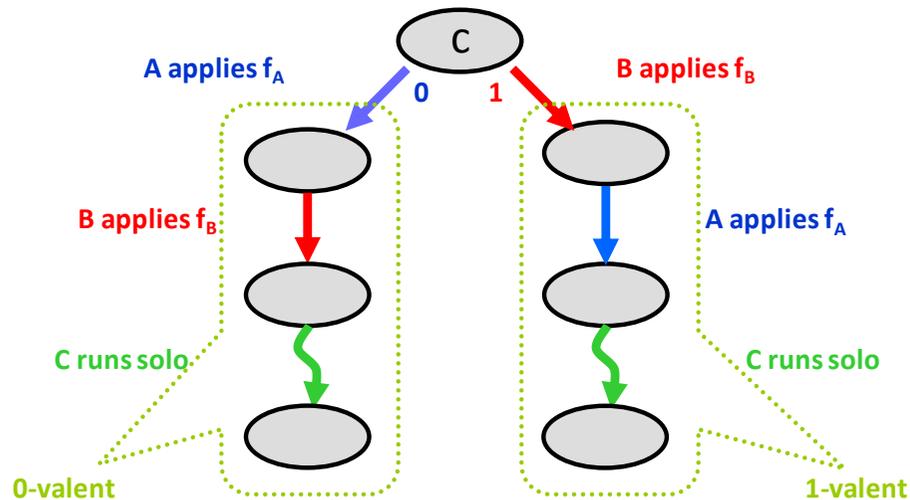
A about to apply f_A C   B about to apply f_B

0-valent   1-valent

Proof: Maybe the Functions Commute

A applies f_A C   B applies f_B

0   1

B applies f_B A applies f_A

C runs solo   C runs solo

0-valent   1-valent

Proof: Maybe the Functions Commute

These states look the same to C

A applies f_A C   B applies f_B

B applies f_B A applies f_A

C runs solo   C runs solo

0-valent   1-valent

Proof: Maybe the Functions Overwrite

A applies f_A

B applies f_B

C

0   1

C runs solo

A applies f_A

C runs solo

0-valent

1-valent

Proof: Maybe the Functions Overwrite

These states look the same to C

A applies f_A

C

B applies f_B

C runs solo

A applies f_A

C runs solo

0-valent

1-valent
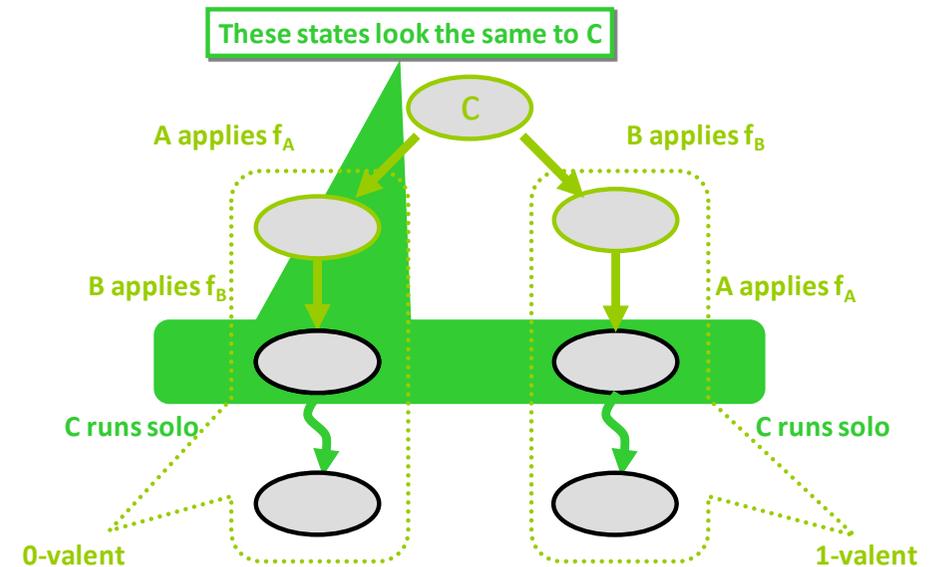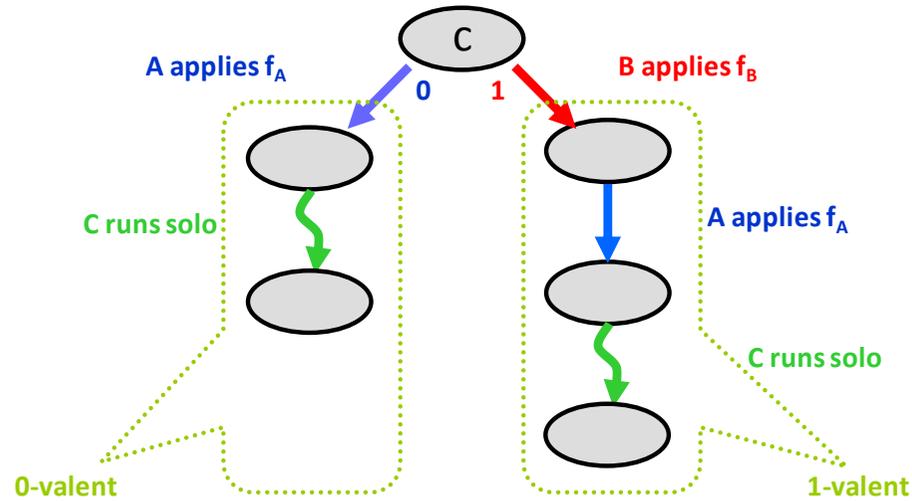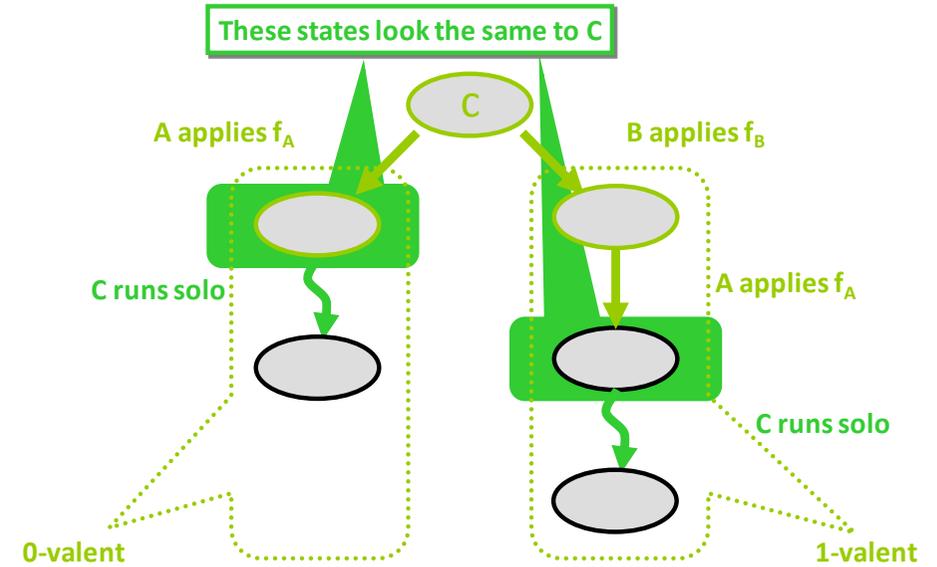
Impact

- Many early machines used these "weak" RMW instructions
  - Test&Set (IBM 360)
  - Fetch&Add (NYU Ultracomputer)
  - Swap

- We now understand their limitations

Consensus with Compare & Swap

```
public class RMWConsensus implements Consensus {
    private RMW r;                          Initialized to -1

    public Object decide() {
        int i = Thread.myIndex();
        int j = r.CAS(-1,i);                Am I first?
        if(j == -1)
            return this.announce[i];        Yes, return
        else                                  my input
            return this.announce[j];
    }                                        No, return
                                             other's input
}
```
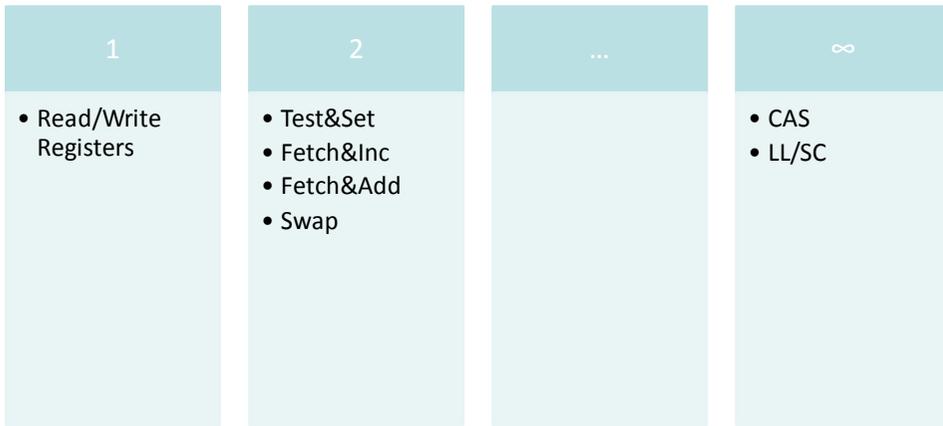
The Consensus Hierarchy

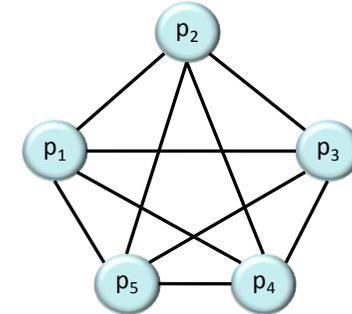| 1	2	...	∞
• Read/Write Registers	• Test&Set • Fetch&Inc • Fetch&Add • Swap		• CAS • LL/SC

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 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
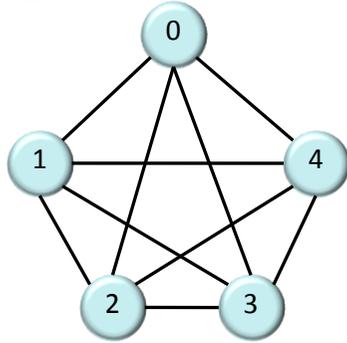 - Some of the messages may be lost, i.e., they are never received

Faulty Processor
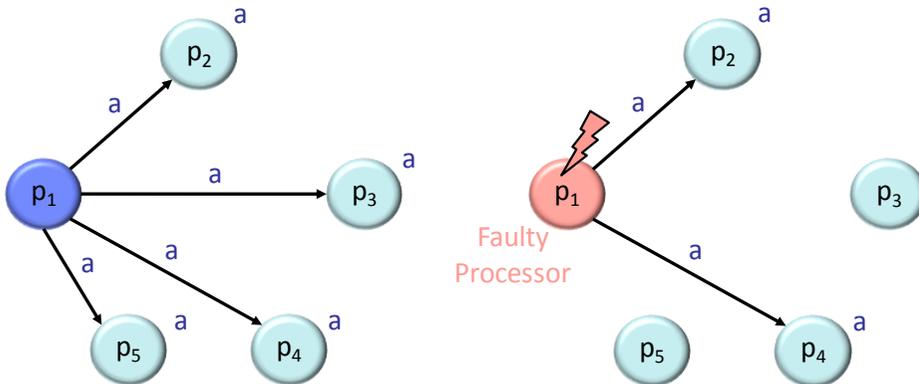
After a Failure, the Process Disappears from the Network

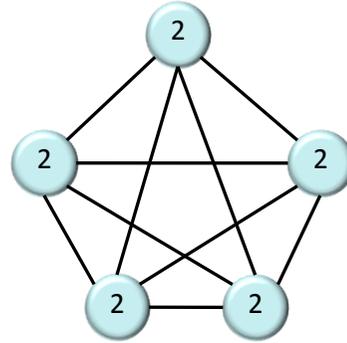Round 1   Round 2   Round 3   Round 4   Round 5

Failure
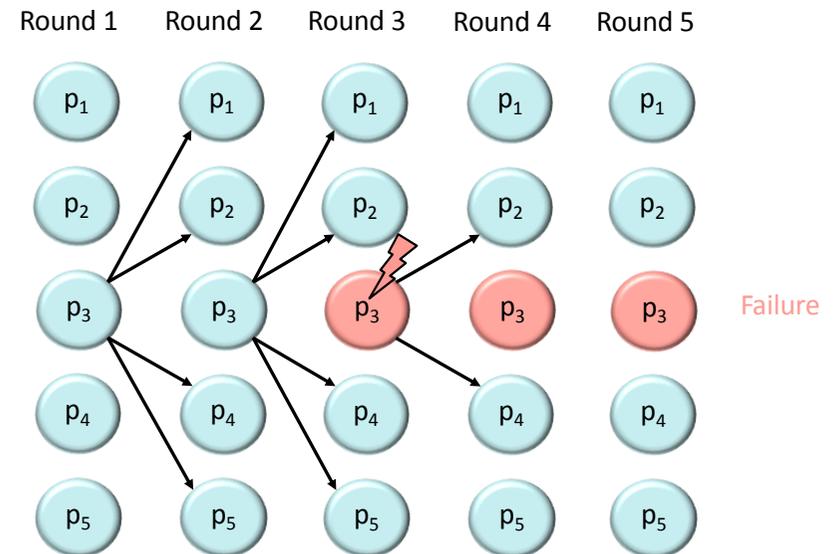
Consensus Repetition

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

Start

Finish

- **Validity conditon**:
  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

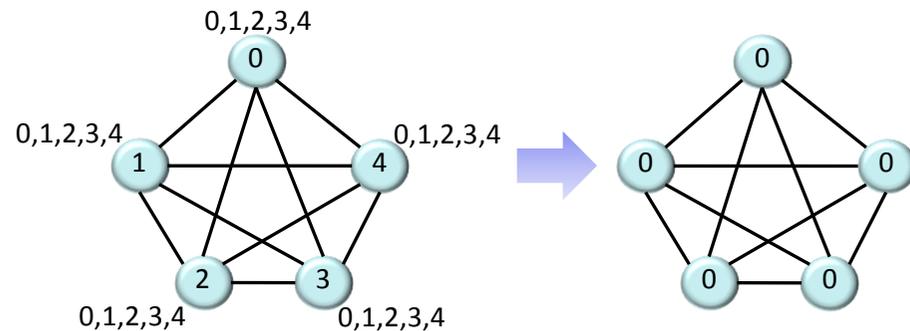Including the own value

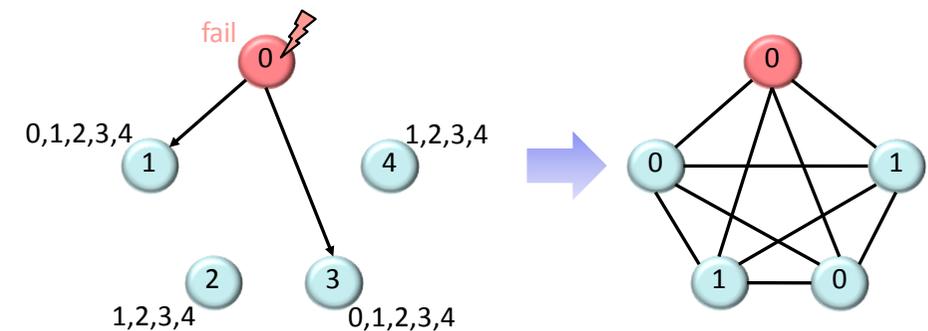Note that only one round is needed!

Execution Without 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)

Execution With Failures

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

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:

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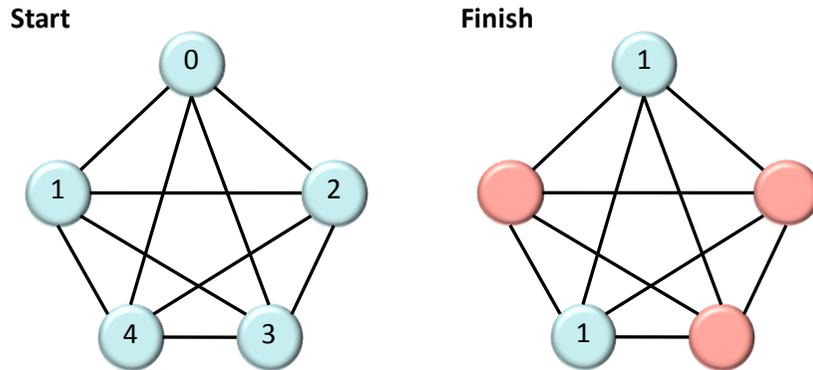
- **Refined validity condition**:
  All processes decide on a value that is available initially

An *f*-resilient Consensus Algorithm

Each process:

Round 1:
Broadcast own value

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

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

Failure 1

1,2,3,4   1,2,3,4

0

1,2,3,4   0,1,2,3,4
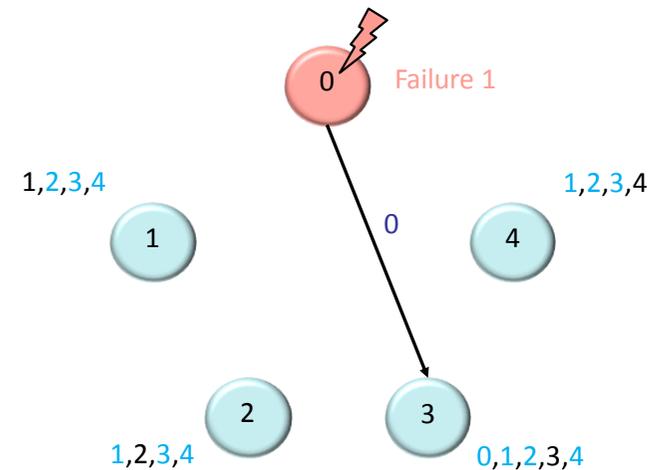
An *f*-resilient Consensus Algorithm

- Round 2: Broadcast all new values to everybody

Failure 1

0,1,2,3,4   1   4   1,2,3,4

0

2   3   Failure 2

1,2,3,4   0,1,2,3,4
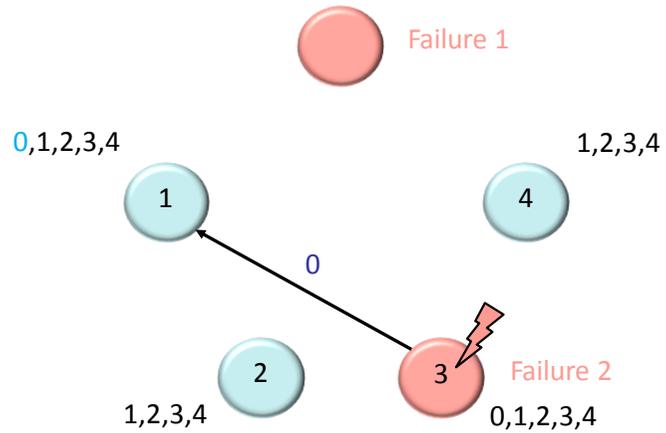
An *f*-resilient Consensus Algorithm

- Round 3: Broadcast all new values to everybody

Failure 1

0,1,2,3,4   1   0   4   0,1,2,3,4

0

2   Failure 2

0,1,2,3,4
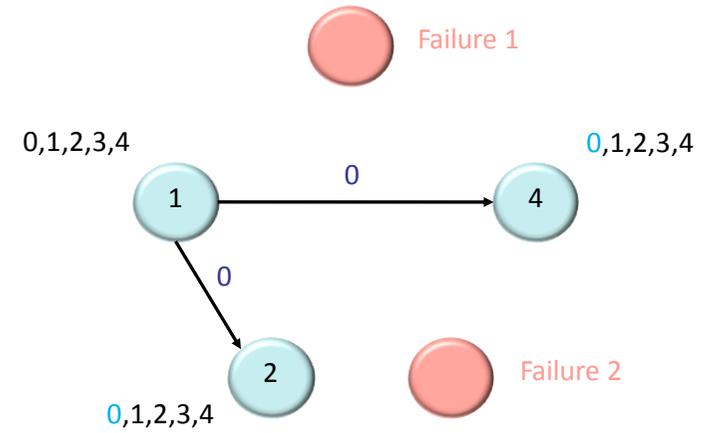
An *f*-resilient Consensus Algorithm

- Decide on minimum → Consensus!

Failure 1

0,1,2,3,4   0   0   0   0,1,2,3,4

0

0

Failure 2

0,1,2,3,4
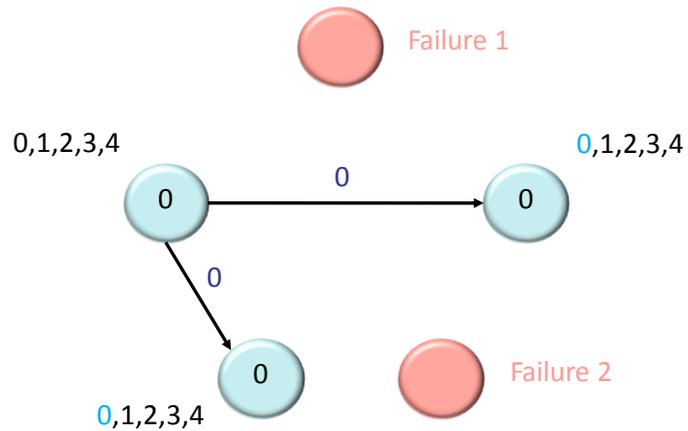
Analysis

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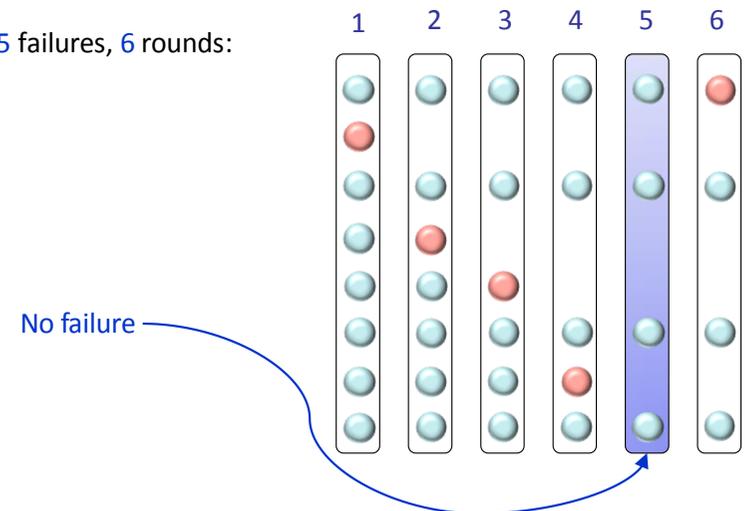
No failure

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

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

Round   1   2
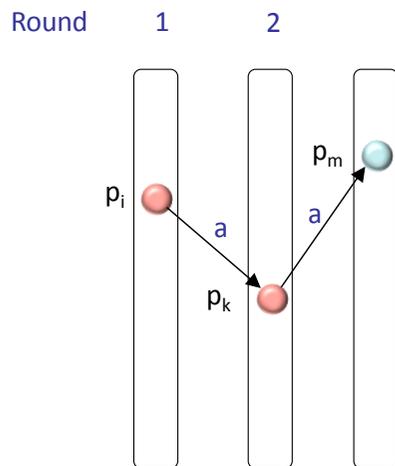
- Before process p_i fails, it sends its value a only to one process p_k
- Before process p_k fails, it sends its value a to only one process p_m

Worst-case Scenario

Round   1   2   3 f
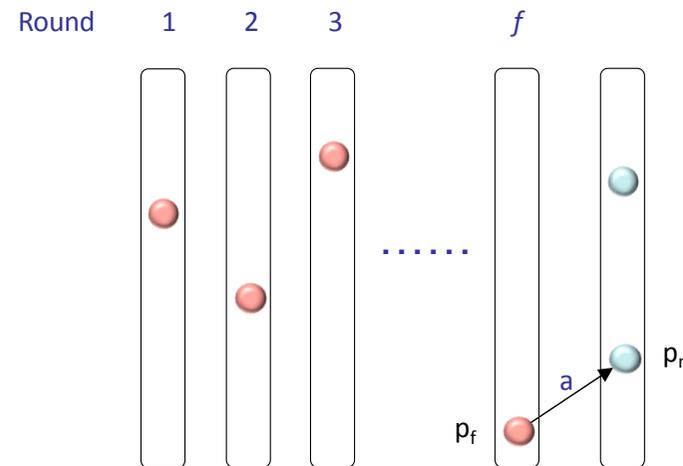
- At the end of round f only one process p_n knows about value a

Worst-case Scenario

Round 1 2 3 f decide
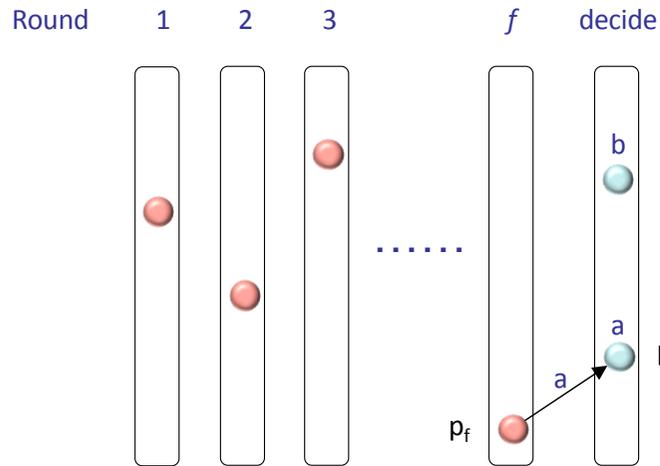
- Process p_n may decide on a and all other processes may decide on another value b
- Therefore f rounds are not enough → At least $f+1$ rounds are needed

Arbitrary Behavior

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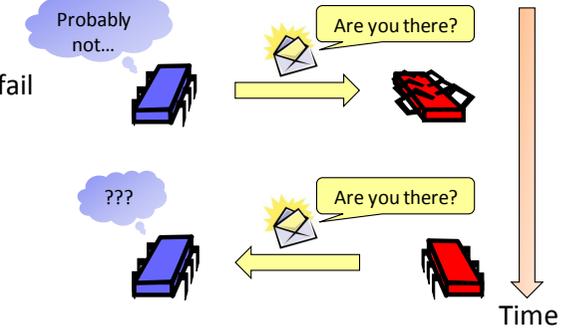
- Maybe the processes fail and recover:

- Maybe the processes are damaged:
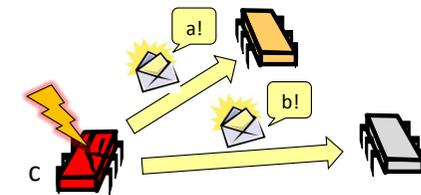
Consensus #5: Byzantine Failures

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

Faulty processor
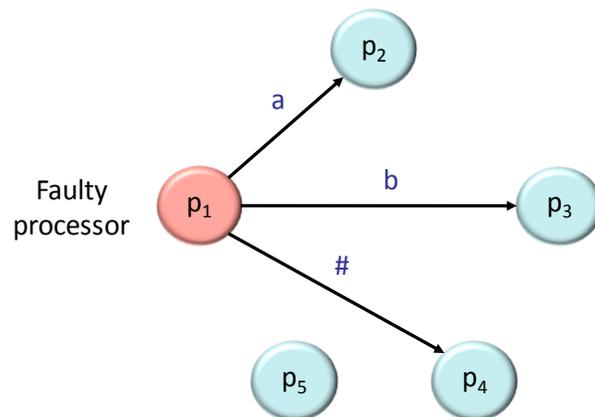
After a Failure, the Process Remains in the Network

Round 1 Round 2 Round 3 Round 4 Round 5
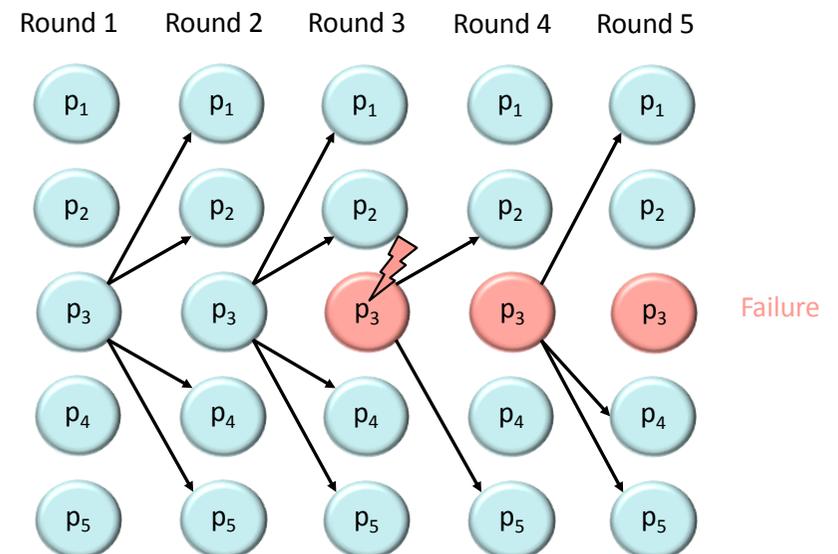
Failure

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
  - 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)?

Theorem

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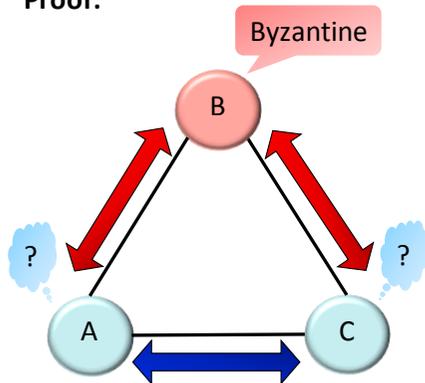
- Intuition:
- Process A may also receive information from C about B's messages to C
- Process A may receive conflicting information about B from C and about C from B (the same for C!)
- It is impossible for A and C to decide which information to base their decision on!

Proof

- 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
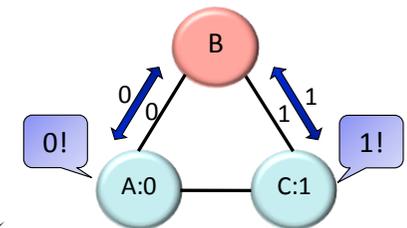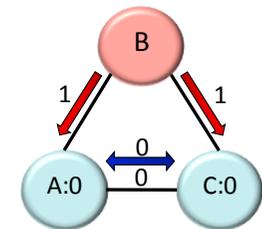
The General Case

- Assume for contradiction that there is an f-resilient algorithm A for n 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
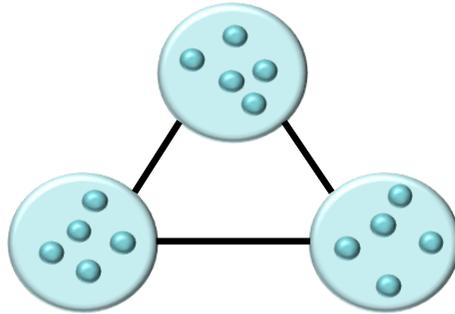
The General Case

- 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!

contradiction
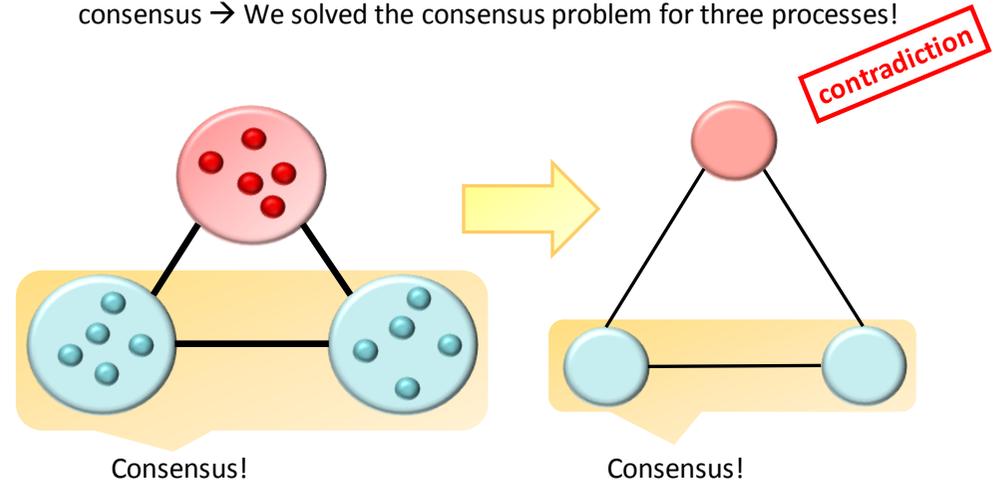
Consensus!   Consensus!

Consensus #6: A Simple Algorithm for Byzantine Agreement

- Can the processes reach consensus if $n > 3f$?
- A simpler question: Can the processes reach consensus if $n=4$ and $f=1$?
- The answer is yes. It takes two rounds:
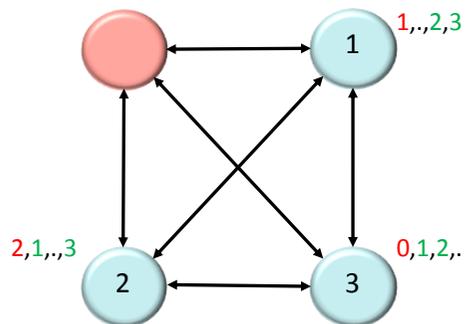
Round 1: Exchange all values   Round 2: Exchange the received info
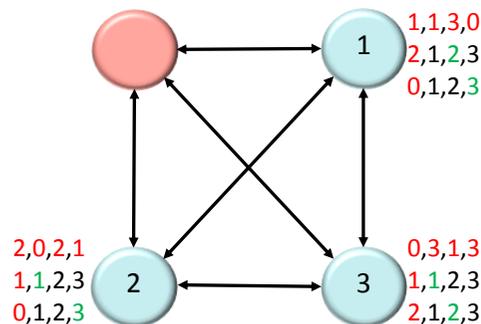
[matrix: one column for each original value, one row for each neighbor]

A Simple Algorithm for Byzantine Agreement

- After the second round each node has received 12 values, 3 for each of the 4 input values (columns). If at least 2 of 3 values of a column 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

Consensus!
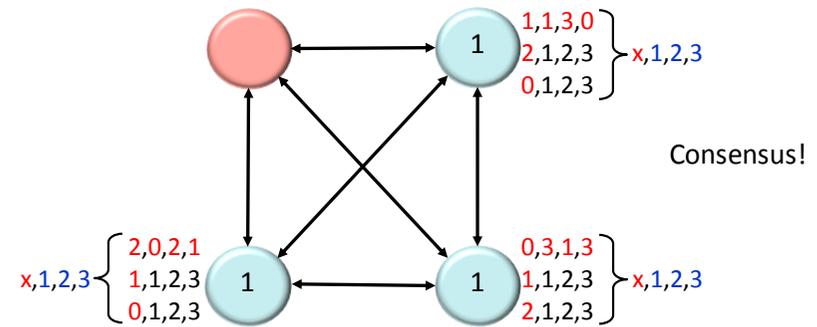
A Simple Algorithm for Byzantine Agreement

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

Round 1: Exchange all values   Round 2: Exchange the received info
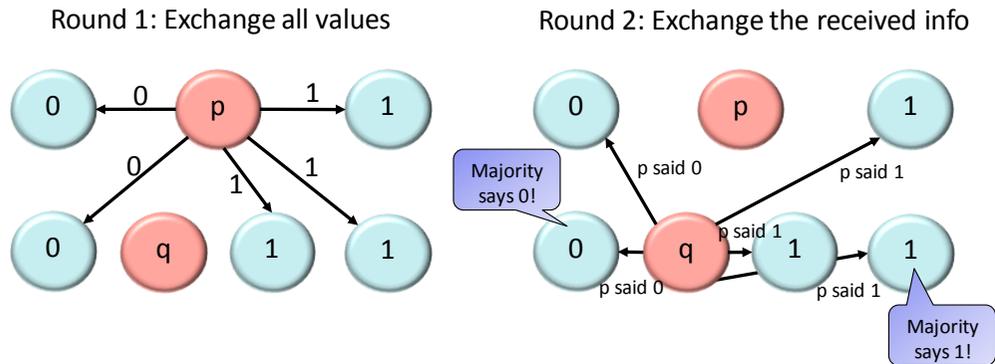
- The problem is that q can say different things about what p sent to q!
- What is the solution to this problem?

A Simple Algorithm for Byzantine Agreement

- The solution is simple: Again exchange all information!
- This way, the processes can learn 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 $n > 3f$

A Simple Algorithm for 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 some considerable disadvantages:
- ''Ignoring all processes that provided inconsistent information'' is not easy to formalize
- The size of the messages increases exponentially! This is a severe problem
  It is 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 processes 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 processes choose the same value and that in future rounds the processes do not change their values anymore

The Queen Algorithm

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

> 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 own value appears > $n/2+f$ times
    support this value
else
    do not support any value

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

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

All received values

0,0,**1**,**1**,**1**,2

0,0,0,**1**,**1**,2

0,0,0,**1**,**1**,2

0,0,0,**1**,**1**,2

0,0,**1**,**1**,**1**,2

> Majority value
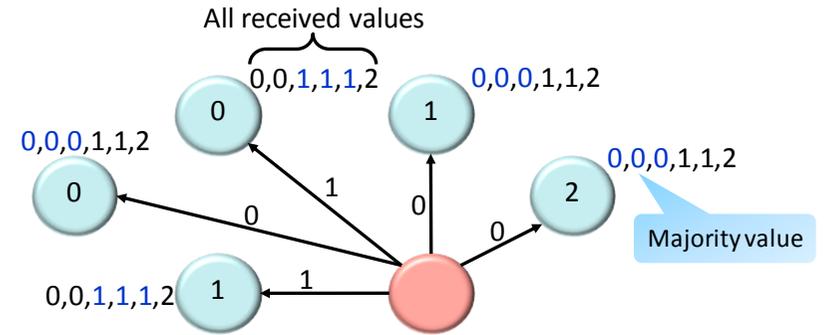
The Queen Algorithm: Example

- Phase 1, round 2 (Queen broadcasts):

> All processes choose the queen's value
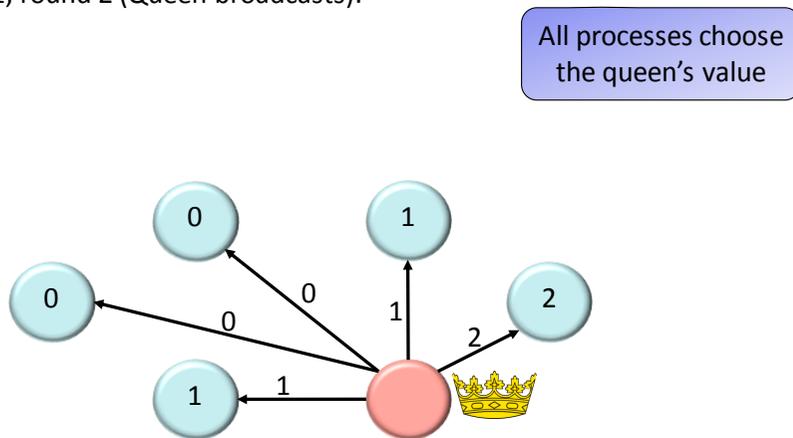
The Queen Algorithm: Example

- Phase 2, round 1 (All broadcast)

> No process supports a value

0,0,**1**,**1**,**1**,2

0,0,0,**1**,**1**,2

0,0,0,**1**,**1**,2

0,0,0,**1**,**1**,2

0,0,**1**,**1**,**1**,2
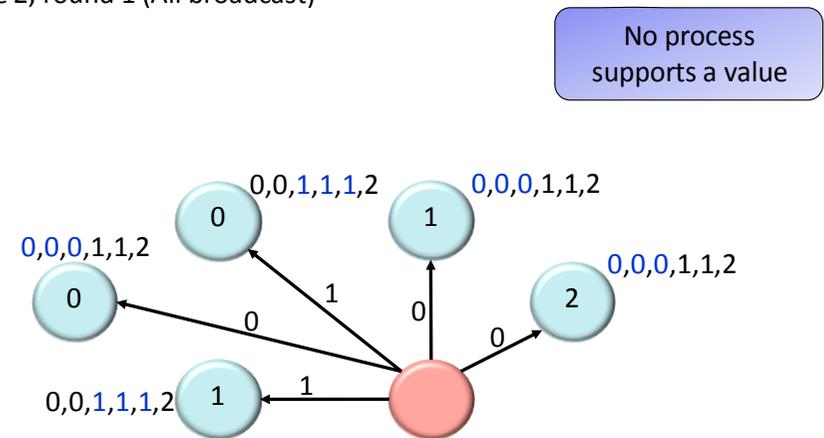
The Queen Algorithm: Example

- Phase 2, round 2 (Queen broadcasts):

All processes choose the queen's value

Consensus!
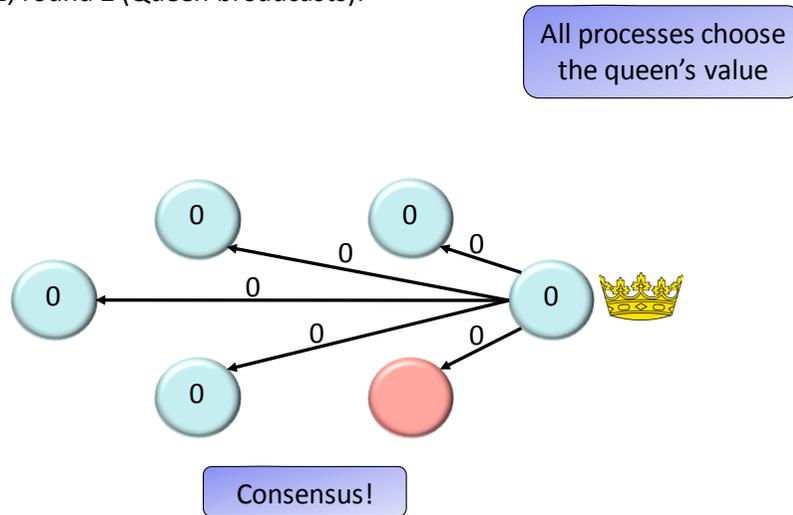
The Queen Algorithm: Analysis

- 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 $n-f > n/2$ 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 $n-f > n/2+f$ 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 $i \in 1...f+1$:

(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

1/125

The King Algorithm: Example

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

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

All processes choose the king' value

"Propose 1"

0,0,1,1   0,1,1,1   2 propose 1   2 propose 1

1 proposal each

Round 1   Round 2   Round 3

1/126

The King Algorithm: Example

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

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

Consensus!

"Propose 1"
2 propose 1   2 propose 1
Set to 1!

I take the king's value!

0,0,1,1   0,1,1,1

3 propose 1

I keep my own value!

Round 1   Round 2   Round 3

1/127
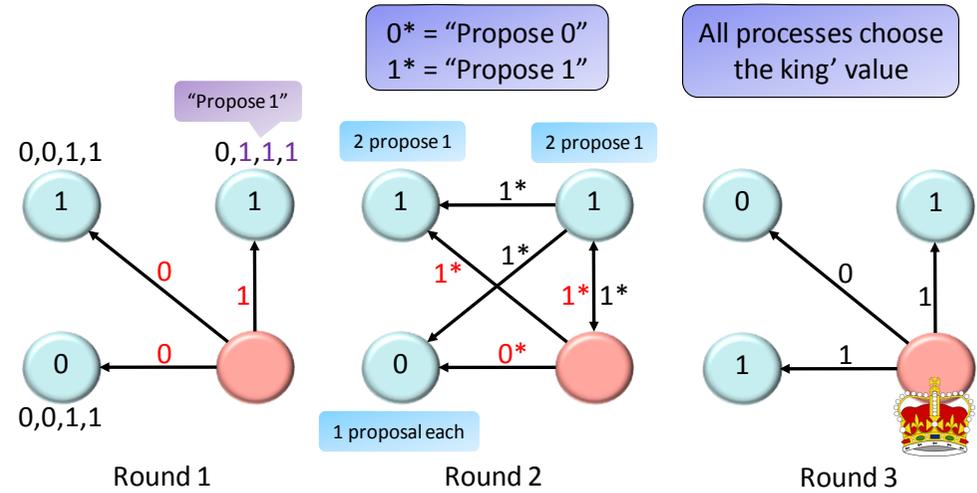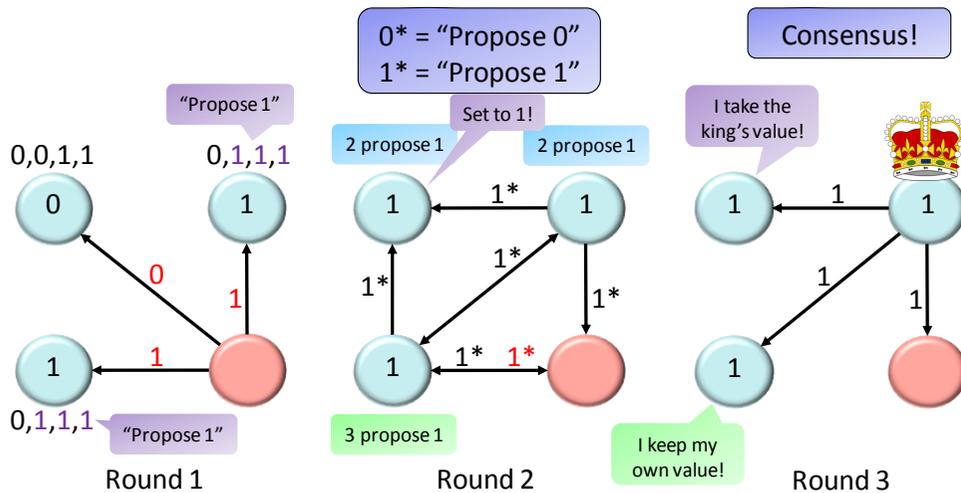
The King Algorithm: Analysis

- 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, i.e., both processes received their value from $\geq n-2f$ distinct correct processes
  - In total, there must be $\geq 2(n-2f) + f > n$ processes, a contradiction!

  (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 $\geq n-f$ times proposals, i.e., no correct process will ever change its value to the king's value

1/128

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 ≥ n-f times → ≥ n-$2f$ correct processes broadcast this proposal and all correct processes receive it ≥ 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

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

Consensus #9: Byzantine Agreement Using Authentication

- A simple way to reach consensus is to use authenticated messages

- Unforgeability condition: If a process never sends a message m, then no correct process ever accepts m
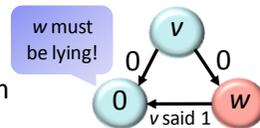
- Why is this condition helpful?
  - A Byzantine process cannot convince a correct process that some other correct processes voted for a certain value if they did not!

- **Idea:**

- There is a designated process P. The goal is to decide on P's value
- For the sake of simplicity, we assume a binary input. The default value is 0, i.e., if P cannot convince the processes that P's input is 1, everybody chooses 0

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 ∈ 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
```

After f+1 rounds:

Decide value

In total r+1 authenticated "P has 1" messages

Byzantine Agreement Using Authentication: Analysis

- Assume that 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
- Assume that P is Byzantine
  - P tries to convince some correct processes that its input is 1
  - Assume that a correct process p sets its value to 1 in a 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!

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 occurs 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 modify the algorithm so that it also works with an arbitrary input?
  - Yes! In fact, the algorithm does not have to be changed much
  - We won't discuss this modification in class
- Can we get rid of the authentication?
  - Yes! Use *consistent-broadcast*. This technique is not discussed either
  - 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 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 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

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: Analysis

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

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 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

- 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 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 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 processes know the outcome of the shared coin toss after each execution of the subroutine

Shared Coin Algorithm

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

Shared Coin: Analysis

- Termination (of the subroutine)
  - 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
- 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 *3f+1 = n*, therefore *f+1 > n/3*
  - Since these 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 $2f < n-f$

Here we use 3*f* < *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

> **Theorem**
> All processes decide 0 with constant probability, and all processes decide 1 with constant probability

Proof:

- With probability $(1-1/n)^n \approx 1/e \approx 0.37$ all processes choose 1. Thus, all correct processes return 1

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

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 $n-4f$ 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
- What about Byzantine failures?

Byzantine AND Asynchronous?

- Are there algorithms that can solve Byzantine agreement in an asynchronous environment?

+ Yes, there are.

− However, the solution comes with a cost. The fraction of nodes that can be Byzantine decreases to $1/35715$. Also, the expected message complexity of the state of the art algorithm is $O(n^3)$.

+ Message sizes remain polynomial.

+ The algorithm does not use a global coin, but tries to detect (biased) local coinflips from Byzantine processes.

Message Complexity

- In all of the previously discussed message passing algorithms, each (good) node broadcasts. Message complexity is $\Omega(n^2)$.

- It was believed for long that the communication overhead of Byzantine Agreement is inherently large.

- Can the message complexity be improved?
+ Yes!
+ Byzantine Agreement can be solved in a synchronous environment with $\tilde{O}(n\sqrt{n})$ messages.
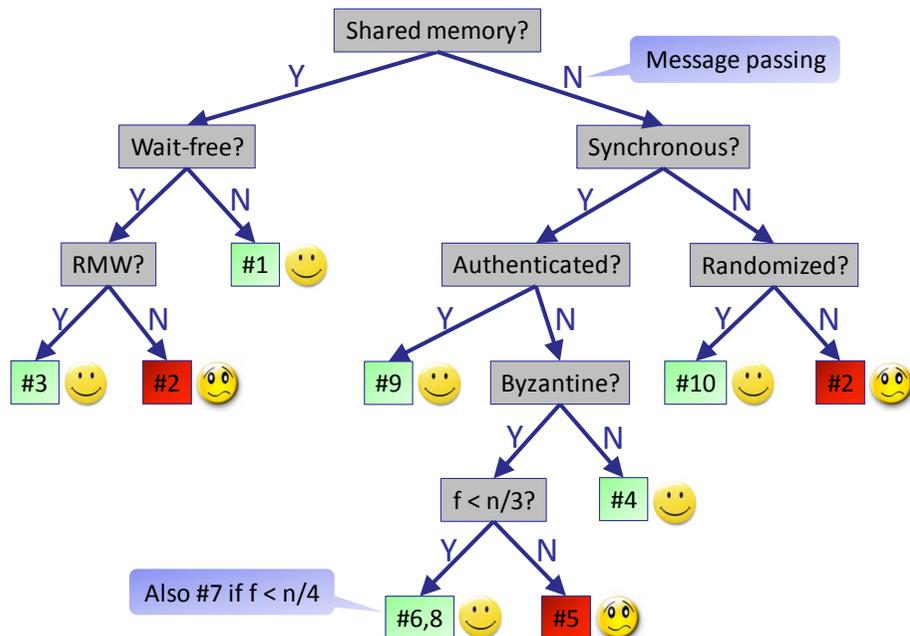− Requires polylogarithmic time.

Summary

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

- In the next part, we will discuss fault-tolerance in practice

Consensus: Decision Tree

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 agreement problem (#5) and the simple algorithm (#6) are from Lamport, Shostak, Pease, 1980ff., and others
- The Queen algorithm (#7) and the King algorithm (#8) are from Berman, Garay, and Perry, 1989.
- The algorithm using authentication (#9) is due to Dolev and Strong, 1982.
- The first randomized algorithm (#10) is from Ben-Or, 1983.
- The concept of a shared coin was introduced by Bracha, 1984.
- Byzantine Agreement with less than n^2 messages is from King and Saia 2011.
- Byzantine Agreement in an asynchronous setting is from King and Saia 2013.

That's all, folks!

Questions & Comments?

Roger Wattenhofer