Network Algorithms

More Leader Elections: Consensus with Failures!

Slides: Maurice Herlihy, Costas Busch, Roger Wattenhofer
Leader election solved!
So we can solve any network problem!? Game/Lecture over?
Recap (2)

• Leader Election and Consensus
  – When possible, when not? It depends!

• Leader Election in Networks
  – Not possible in ring without IDs
  – With IDs, in $O(n)$ time and $O(n \log n)$ messages, even if $n$ unknown and transmissions asynchronous
  – This is optimal: in asynchronous, need $> n \log n$ message is ring!
  – Synchronous: can do with $n$ messages, treat messages with time!

\[ M(n/2) \]

\[ \geq n/4 \text{ msgs when closed} \]
Recap (3)

• Introducing failures: Byzantine Generals consensus impossible under message loss
  – Harder than NP-hard 😊

• Make it simpler: consensus in shared memory (no network!)
  – Consensus possible if processes do not die! Your algo: write value in my register, decide on minimum!
  – What about failures, does it work too?! If failures are fail-stop? If behavior is “malicious”? If behavior is Byzantine (arbitrary)?
Sequential Computation

memory

object

thread

object
Often simpler than thinking about networks! “Higher-level language”, can focus on fundamental distributed system aspects!
Asynchrony

- Sudden unpredictable delays
  - Cache misses (*short*)
  - Page faults (*long*)
  - Scheduling quantum used up (*really long*)
Model

- Multiple *threads*
  - Sometimes called *processes*
- Single shared *memory*
- *Objects* live in memory
- Unpredictable asynchronous delays
Two Generals Problem

Red army wins if both sides attack together

Stefan Schmid @ T-Labs Berlin, 2013/4
Red armies send messengers across valley
Communications

Messengers don’t always make it
There is no non-trivial protocol that ensures the red armies attack simultaneously!
• Assume a protocol exists
• Reason about its properties
• Derive a contradiction
1. Consider the protocol that sends fewest messages
2. It still works if last message lost
3. So just don’t send it
   – Messengers’ union happy
4. But now we have a shorter protocol!
5. Contradicting #1
Consensus: Start...

32

19

21
... Communicate...
... Agree on Someone’s Input!
Why Consensus?

• 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
What you will learn...

• In some models, consensus is possible
• In some other models, it is not

• Goal of this and next lecture: to learn whether for a given model consensus is possible or not … and prove it!
Consensus #1: Shared Memory

Problem:

- $n$ processors, with $n > 1$
- Processors can atomically read or write (not both) a shared memory cell
- Must decide on one of the input values

Idea:

- 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 $j$ ($j$ not 1) reads $c$ until $j$ reads something else than “?”, and then decides on that.
Problems: Unexpected Delay ...
Problems: ... Heterogenous Resources ...
Problems: ... Fault-Tolerance?

Keeps lock on objects…
E.g., your algorithm: when to decide on minimum? And minimum of which subset of processes? Asynchronous! Has other process died?? Need to be “wait-free”!

Keeps lock on objects…
Consensus #2: Wait-Free Shared Memory

- n processors, with n > 1
- Processors can atomically read or write (not both) a shared memory cell
- Processors might crash (halt)
- **Wait-free** implementation… huh?
Model

• **Wait-free** = every process (method call) completes in a finite number of its own steps
  – if scheduled sufficiently frequently, we’re fine

• **Register**: object that supports read/write
  – not punctual, takes time!

• **We assume that we have wait-free atomic register implementations**
  – that is, it seems that reads and writes to same register do not overlap, and the real-time order is respected
  – real-time: response of previous operation precedes the invocation of the next operations
  – “linearizable”
A Correct, Wait-Free Algorithm?

• There is a cell $c$, initially $c=\text{“?”}$
• Every processor $i$ does the following
  
  ```
  r = \text{Read}(c);
  \text{if} \ (r == \text{“?”}) \ \text{then}
      \text{Write}(c, v_i); \ \text{decide} \ v_i;
  \text{else}
      \text{decide} \ r;
  ```
Correct?
Consensus #2 Impossible!

Theorem: Consensus #2 is impossible!

Proof Strategy:

• Make it simple
  – \( n = 2 \), binary input
  – one or more r/w registers
• Assume that there is a protocol (choose yours!)
• Reason about the properties of any such protocol
• Derive a contradiction: choose “bad schedule”, i.e., who is next (asynchronous), who dies, …
Wait-Free Computation

- Either A or B “moves” (atomic read/write!)
  - Asynchronous: “scheduler can choose!”

- Moving means
  - Register read
  - Register write
The 2-Move Tree

Initial state

Final states

Initial state
Decision Values
Bivalent: Both Possible
Univalent: Only One Possible

Univalent

Stefan Schmid @ T-Labs Berlin, 2013/4
Univalent: Only One Possible
Summary

• Wait-free computation is a tree
• Bivalent system states
  – Outcome not fixed, even given the process inputs (but not the execution / who dies)
• Univalent states
  – Outcome is fixed
  – Maybe not “known” yet
  – 1-Valent and 0-Valent states
Exists bivalent system state.

(The outcome is not always fixed from the start, even if process values are given.)
0-Valent Initial State

All executions lead to decision of 0
All executions lead to decision of 0
Solo execution by A also decides 0
1-Valent Initial State

All executions lead to decision of 1
Solo execution by B also decides 1
Can all executions lead to the same decision? No, must depend on execution!
Bivalent Initial State

Solo execution by A must decide 0

Solo execution by B must decide 1
Definition: Critical State
Critical States

- Starting from a bivalent initial state
- The protocol can reach a critical state
  - Otherwise we could stay bivalent forever
  - And the protocol is not wait-free
  - Note: if only left 0-valent, right still bivalent: not critical yet, take c lower

If A goes first, protocol decides 0
If B goes first, protocol decides 1
Critical States

- Starting from a bivalent initial state
- The protocol can reach a critical state
  - Otherwise we could stay bivalent forever
  - And the protocol is not wait-free
  - Note: if only left 0-valent, right still bivalent: not critical yet, take c lower

We will show that processes cannot distinguish who goes first: Contradiction to critical state!

If A goes first, protocol decides 0
If B goes first, protocol decides 1
• So far, memory-independent!
• True for
  – Registers
  – Message-passing
  – Carrier pigeons
  – Any kind of asynchronous computation
What Are the Threads Doing?

- Reads and/or writes
- To same/different registers (one or more registers!)

Possible Interactions after critical state:

<table>
<thead>
<tr>
<th></th>
<th>x.read()</th>
<th>y.read()</th>
<th>x.write()</th>
<th>y.write()</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.read()</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>y.read()</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>x.write()</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>y.write()</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
A runs solo, decides 0

States look the same to A

A runs solo, decides 1

B reads x

A cannot learn that B read x…
## Possible Interactions

<table>
<thead>
<tr>
<th></th>
<th>x.read()</th>
<th>y.read()</th>
<th>x.write()</th>
<th>y.write()</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.read()</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>y.read()</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>x.write()</td>
<td>no</td>
<td>no</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>y.write()</td>
<td>no</td>
<td>no</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
A writes y

B writes x

A and B cannot distinguish: register x=0, register y=1 anyway!
Contradiction to critical state, remains bivalent

Stefan Schmid @ T-Labs Berlin, 2013/4
## Possible Interactions

<table>
<thead>
<tr>
<th></th>
<th>x.read()</th>
<th>y.read()</th>
<th>x.write()</th>
<th>y.write()</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.read()</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>y.read()</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>x.write()</td>
<td>no</td>
<td>no</td>
<td>?</td>
<td>no</td>
</tr>
<tr>
<td>y.write()</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>?</td>
</tr>
</tbody>
</table>
Writing Same Registers: Write-Write Same Register

States look the same to A

A runs solo, decides 0

A writes x

A runs solo, decides 1

B writes x

A writes x
That’s all, folks!

<table>
<thead>
<tr>
<th></th>
<th>x.read()</th>
<th>y.read()</th>
<th>x.write()</th>
<th>y.write()</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.read()</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>y.read()</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>x.write()</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>y.write()</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Theorem

- It is impossible to solve consensus using read/write atomic registers
  - Assume protocol exists
  - It has a bivalent initial state
  - Must be able to reach a critical state
  - Case analysis of interactions
    - Reads vs others
    - Writes vs writes
What does Consensus have to do with Distributed Systems?
We want to build a concurrent FIFO queue...
... with multiple dequeuers.
A Consensus Protocol based on 2 FIFO Queues & Atomic Registers

2-element array

FIFO Queue with red and black balls

Coveted red ball (winner)

Dreaded black ball (looser)
Rough Protocol Idea

1. Put my value to “my” register
2. Dequeue initialized queue: am I winner or loser?
Write Value to Array
Protocol: Take Next Item from the Queue
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
Why does it work?

• If one thread gets the red ball
• Then the other gets the black ball
• Winner can take her own value
• Loser can find winner’s value in array
  – Because threads write array before dequeuing from queue
1. Put my value to “my” register
2. Dequeue initialized queue: am I winner or loser?

But how to initialize queue?
1. Need two shared queues! And two registers and two flags…
2. Enqueue winner then loser in “my” queue
3. Set shared register flag for me true
4. If flag of process i set => look at i’s queue =>
   dequeue i’s
5. Otherwise i+1 mod 2’s
6. See “Skript”
Implications

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

- 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
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 processors, with n > 1
- Wait-free implementation
- Processors can atomically read and write a shared memory cell in one atomic step: the value written can depend on the value read
- We call this a RMW register
Protocol

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

\[ \text{RMW}(c), \text{ with} \]
\[
\text{if } (c == "?") \text{ then} \\
\hspace{1cm} \text{write}(c, v_i); \text{ decide } v_i; \\
\text{else} \\
\hspace{1cm} \text{decide } c; \\
\]

atomic step
Discussion

- Protocol works correctly
  - One processor accesses c as the 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?
RMW: More Formally

• Method takes 2 arguments:
  – Variable \( x \)
  – Function \( f \)

• Method call:
  – Returns value of \( x \)
  – Replaces \( x \) with \( f(x) \)
public abstract class RMW {
  private int value:

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

Return prior value

Apply function
public abstract class RMW {
    private int value;

    public void read() {
        int prior = this.value;
        this.value = this.value;
        return prior;
    }
}
public abstract class RMW {
    private int value;

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

constant function
public abstract class RMW {
    private int value;

    public void fai() {
        int prior = this.value;
        this.value = this.value + 1;
        return prior;
    }
}

increment function
public abstract class RMW {
    private int value;

    public void faa(int x) {
        int prior = this.value;
        this.value = this.value + x;
        return prior;
    }
}

addition function
public abstract class RMW {
    private int value;

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

constant function
public abstract class RMW {
    private int value;

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

complex function
Non-Trivial RMW

- Not simply read
- But
  - test&set, fetch&inc, fetch&add, swap, compare&swap, general RMW
- Definition: A RMW is non-trivial if there exists a value $v$ such that $v \neq f(v)$
Consensus Numbers

- 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
Consensus Numbers

• Theorem
  – Atomic read/write registers have consensus number 1

• Proof
  – Works with 1 process
  – We have shown impossibility with 2
Consensus Numbers

• Consensus numbers are a useful way of measuring synchronization power

• Theorem
  – If you can implement X from Y
  – And X has consensus number c
  – Then Y has consensus number at least c
Conversely
  - 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!
• 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

```java
public class RMWConsensusFor2 implements Consensus {
    private RMW r;

    public Object decide() {
        int i = Thread.myIndex();
        if (r.rmw(f) == v)
            return this.announce[i];
        else
            return this.announce[1-i];
    }
}
```

**Initialized to v**

Am I first?

Yes, return my input

No, return other’s input
• We have displayed
  – A two-thread consensus protocol
  – Using any non-trivial RMW object
• 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: $f_i(f_j(x)) = f_i(x)$

• Claim: Any such set of RMW objects has consensus number exactly 2
Examples

• Test-and-Set
  – Overwrite
• Swap
  – Overwrite
• Fetch-and-inc
  – Commute
Meanwhile Back at the Critical State

A about to apply $f_A$

0-valent

C

1-valent

B about to apply $f_B$
Maybe the Functions Commute

A applies $f_A$

B applies $f_B$

C runs solo

0-valent

B applies $f_B$

A applies $f_A$

C runs solo

1-valent

0

1
Maybe the Functions Commute

These states look the same to C

A applies $f_A$

B applies $f_B$

C runs solo

B applies $f_B$

A applies $f_A$

C runs solo

0-valent

1-valent

C

0

1
Maybe the Functions Overwrite

A applies $f_A$

C runs solo

0-valent

C runs solo

1-valent

B applies $f_B$

A applies $f_A$
Maybe the Functions Overwrite

These states look the same to C

C runs solo

A applies $f_A$

B applies $f_B$

0-valent

0

C runs solo

1-valent

1

Stefan Schmid @ T-Labs Berlin, 2013/4
• Many early machines used these “weak” RMW instructions
  – Test-and-set (IBM 360)
  – Fetch-and-add (NYU Ultracomputer)
  – Swap
• We now understand their limitations
  – But why do we want consensus anyway?
<table>
<thead>
<tr>
<th>Level</th>
<th>Consensus Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Read/Write Registers, ...</td>
</tr>
<tr>
<td>2</td>
<td>T&amp;S, F&amp;I, Swap, ...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>∞</td>
<td>CAS, ...</td>
</tr>
</tbody>
</table>
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?
• Complete graph
• Synchronous
Broadcast

Diagram:

- Node $p_1$ is connected to $p_2$, $p_3$, $p_5$, and $p_4$.
- Arrows indicate the direction of communication:
  - $p_1$ to $p_2$ and $p_3$.
  - $p_1$ to $p_5$ and $p_4$.
  - $p_2$ to $p_3$ and $p_5$.
  - $p_3$ to $p_5$.
  - $p_4$ to $p_5$.

Each edge labeled with 'a' signifies a communication action.
Broadcast

$p_1$ $p_2$ $p_3$ $p_4$ $p_5$
Multiple Processes Can Broadcast in Same Round
At the End of the Round

\[ p_2 \]

\[ p_1 \]

\[ p_3 \]

\[ p_4 \]

\[ p_5 \]
Faulty processor

Crash Failures
Crash Failures

Faulty processor

Diagram showing a network with nodes labeled $p_1$ to $p_4$ and an edge labeled 'a' connecting $p_1$ to $p_2$. Additionally, nodes $p_5$ and $p_3$ are connected to $p_1$ via the edge 'a'.
Faulty processor

Crash Failures

Faulty processor $p_1$, $p_2$, $p_3$, $p_4$, $p_5$
After Failure, Process Disappears

Round 1
- $p_1$
- $p_2$
- $p_3$
- $p_4$
- $p_5$

Round 2
- $p_1$
- $p_2$
- $p_3$
- $p_4$
- $p_5$

Round 3
- $p_1$
- $p_2$
- $p_3$
- $p_4$
- $p_5$

Round 4
- $p_1$
- $p_2$
- $p_3$
- $p_4$
- $p_5$

Round 5
- $p_1$
- $p_2$
- $p_3$
- $p_4$
- $p_5$

Failure
Consensus: Start With Own Values

Start

Diagram: A network of nodes labeled 0, 1, 2, 3, 4 with lines connecting them.
Consensus: Decide on One

Finish

- 3
- 3
- 3
- 3
- 3

Stefan Schmid @ T-Labs Berlin, 2013/4
Validity condition:
If everybody starts with the same value they must decide on that value
Each processor:

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

(only one round is needed)
Start
Broadcast Values

0,1,2,3,4

0,1,2,3,4

0,1,2,3,4

0,1,2,3,4

0,1,2,3,4

0,1,2,3,4
Decide Minimum
Finish
If everybody starts with the same initial value, everybody sticks to that value (minimum)
The simple algorithm doesn’t work

Each processor:

1. Broadcasts value to all processors

2. Decides on the minimum
Broadcasted Values

![Diagram showing a network with nodes labeled 1, 2, 3, 4, and 0. Node 0 has an arrow pointing to itself labeled 'fail'. Connections are shown between nodes 1 and 0, 2 and 0, 3 and 0, and 0 and 4.](image-url)
Broadcasted Values

0,1,2,3,4
1

fail

1,2,3,4
4

1,2,3,4
2

0,1,2,3,4
3
Decide on Minimum

- 0, 1, 2, 3, 4
- 1, 2, 3, 4
- 0
- 1
- Fail
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
New validity condition:
all non-faulty processes decide on a value that is available initially.
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

0

1

2

3

4
Example: $f=1$ failures, $f+1 = 2$ rounds needed

Round 1  Broadcast all values to everybody

(new values)
Example: $f=1$ failures, $f+1 = 2$ rounds needed

Round 2  Broadcast all new values to everybody

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

1  4

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

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

Finish  Decide on minimum value

0,1,2,3,4

0

0

0,1,2,3,4

0

0

0,1,2,3,4

0

0

0
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

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

0  4

1

2

3

0

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

Round 2  Broadcast new values to everybody

Failure 1

Failure 2

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

Round 3  Broadcast new values to everybody

0,1,2,3,4

Failure 1

0

1

4

0,1,2,3,4

0,1,2,3,4

Failure 2

2

3

Stefan Schmid @ T-Labs Berlin, 2013/4
Example: $f=2$ failures, $f+1 = 3$ rounds needed

Finish

Decide on the minimum value

Failure 1

$0, 1, 2, 3, 4$

$0$

$0$

Failure 2

$0, 1, 2, 3, 4$

$0$

$0$

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

Finish

Decide on the minimum value

Failure 1

$0, 1, 2, 3, 4$

$0$

$0$

Failure 2

$0, 1, 2, 3, 4$

$0$

$0$

Stefan Schmid @ T-Labs Berlin, 2013/4
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
• 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
Round 1

before process $p_i$ fails, it sends its value $a$ to only one process $p_k$
Round 1 2

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 knows about value $a$. 

Round 1 2 3 4 5

$n$ $n$ $n$ $p_f$ $p_n$
Worst Case Scenario

Round 1  2  3  f  decide

Process may decide on a, and all other processes may decide on another value (b)

\[ p_n \]
**Worst Case Scenario**

<table>
<thead>
<tr>
<th>Round</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>f</th>
<th>decide</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Therefore $f$ rounds are not enough.

At least $f + 1$ rounds are needed.
Consensus #5
Byzantine Failures

Faulty processor

Different processes receive different values

Stefan Schmid @ T-Labs Berlin, 2013/4
Some messages may be lost

A Byzantine process can behave like a Crashed-failed process
After failure the process continues functioning in the network.
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.

Start

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

Proof: Follows from the crash failure lower bound
There is no $f$-resilient algorithm for $n$ processes, where $f \geq n/3$

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)
faulty
faulty

(validity condition)
Stefan Schmid @ T-Labs Berlin, 2013/4
Impossibility

\[\begin{array}{c}
0 \\
p_2 \\
p_1 \\
p_0 \\
1
\end{array}\]

faulty
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

\begin{align*}
\text{start} & : 0 & 1 & 1 & 2 & 1 & 0 & 2 & 0 & 1 & 0 & 1 \\
& \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
& p_1 & p_2 & \cdots \\
\text{failures} & \\
\text{finish} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \cdots p_n \\
& \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
& p_1 & p_2 & \cdots & \cdots p_n
\end{align*}
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

algorithm A tolerates $n/3$ failures

all decide $k$
Final decision

We reached consensus with 1 failure

Impossible!!!
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

Faulty

0  1  1  2  1  0  2  0  1  0  1  0
Example: 12 processes, 2 faults, 3 kings

initial values

King 1  King 2  King 3

Remark: There is a king that is not faulty
Each processor $p_i$ has a preferred value $v_i$. In the beginning, the preferred value is set to the initial value.
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 $v_k$
End of Phase $f+1$:  
Each process decides on preferred value
Example: 6 processes, 1 fault
Phase 1, Round 1

2,1,1,1,0,0

2,1,1,0,0,0

2,1,1,0,0,0

2,1,1,0,0,0

0

1

0

1

0

2

0

0

1

king 1

Everybody broadcasts

Stefan Schmid @ T-Labs Berlin, 2013/4
Phase 1, Round 1

Choose the majority

Each majority population was

$$3 \leq \frac{n}{2} + f = 4$$

On round 2, everybody will choose the king’s value
The king broadcasts
Phase 1, Round 2

Everybody chooses the king’s value

king 1
Phase 2, Round 1

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

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

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

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

king 2

Everybody broadcasts
Phase 2, Round 1

Choose the majority

Each majority population is

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

\[ 3 \leq \frac{n}{2} + f = 4 \]
Phase 2, Round 2

The king broadcasts
Phase 2, Round 2

Everybody chooses the king’s value

Final decision

Stefan Schmid @ T-Labs Berlin, 2013/4
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 safely 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 $\text{coins}_i$, and memorize $c_j$. 

* Stefan Schmid @ T-Labs Berlin, 2013/4
Shared Coin Algorithm

4. If you have seen exactly n-f local coins then copy the set coins\textsubscript{i} into the set seen\textsubscript{i} (but do not stop extending coins\textsubscript{i} if you see new coins)

5. Use reliable broadcast to tell all processes about your set seen\textsubscript{i}. 
6. If you have seen at least $n-f$ seen\(_j\) which satisfy seen\(_j \subseteq \text{coins}_i\), 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) 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 $\text{seen}_j$ (columns) and local coins $c_k$ (rows) process $i$ has seen.
- We draw an “X” in the matrix if and only if set $\text{seen}_i$ includes coin $c_k$. 
Proof: Matrix \((f=2, n=7, n-f=5)\)

<table>
<thead>
<tr>
<th></th>
<th>seen_1</th>
<th>seen_3</th>
<th>seen_5</th>
<th>seen_6</th>
<th>seen_7</th>
</tr>
</thead>
<tbody>
<tr>
<td>coin_1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>coin_2</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>coin_3</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>coin_5</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>coin_6</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>coin_7</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

- 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 \( f \cdot (n-f) + (n-f) \cdot f \), ...
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.
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.
• 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.
• 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.