The Consensus Problem

Roger Wattenhofer

a lot of kudos to Maurice Herlihy and Costas Busch for some of their slides

Sequential Computation

memory

object

Concurrent Computation

threads

memory

object

Asynchrony

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

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

Road Map

- 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”

You may ask yourself …

I’m no theory weenie - why all the theorems and proofs?

Fundamentalism

- Distributed & concurrent 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 together

Communications

Red armies send messengers across valley

Communications

Messengers don’t always make it

Your Mission

Design a protocol to ensure that red armies attack simultaneously
Theorem

There is no non-trivial protocol that ensures the red armies attacks simultaneously.

Proof Strategy

• Assume a protocol exists
• Reason about its properties
• Derive a contradiction

Proof

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

Fundamental Limitation

• Need an unbounded number of messages
• Or possible that no attack takes place
You May Find Yourself ...

I want a real-time YAFA compliant Two Generals protocol using UDP datagrams running on our enterprise-level fiber tachyion network ...

You might say

Yes, Ma’am, right away!

Advantage:
• Buys time to find another job
• No one expects software to work anyway

Disadvantage:
• You’re doomed
• Without this course, you may not even know you’re doomed
You might say

I want a real-time YAFE
compliant Two Generals
protocol using UDP datagrams
running on our enterprise-level
fiber tachyon network...

I can't find a fault-tolerant
algorithm, I guess I'm just a
pathetic loser.

You might say

Advantage:
• No need to take course

Disadvantage:
• Boss fires you, hires
University St. Gallen graduate

Using skills honed in course, I
can avert certain disaster!
• Rethink problem spec, or
• Weaken requirements, or
• Build on different platform
**Consensus: Each Thread has a Private Input**

32  
19  
21

**They Communicate**

**They Agree on Some Thread’s Input**

19  
19  
19

**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
You gonna 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

- n processors, with \( n \geq 1 \)
- Processors can atomically *read* or *write* (not both) a shared memory cell

Protocol (Algorithm?)

- 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.

Unexpected Delay

- Swapped out back at...
Heterogeneous Architectures

Fault-Tolerance

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?

Wait-Free Implementation

- Every process (method call) completes in a finite number of steps
- Implies no mutual exclusion
- 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
  \[ r = \text{Read}(c); \]
  \[ \text{if } (r == \text{"?"}) \text{ then} \]
  \[ \text{Write}(c, v_i); \text{decide } v_i; \]
  \[ \text{else} \]
  \[ \text{decide } r; \]

Is the algorithm correct?

Theorem:
No wait-free consensus

Proof Strategy

- Make it simple
  - n = 2, binary input
- Assume that there is a protocol
- Reason about the properties of any such protocol
- Derive a contradiction
Wait-Free Computation

- Either A or B “moves”
- Moving means
  - Register read
  - Register write

The Two-Move Tree

Decision Values

Bivalent: Both Possible
Univalent: Single Value Possible

1-valent: Only 1 Possible

0-valent: Only 0 possible

Summary

- Wait-free computation is a tree
- Bivalent system states
  - Outcome not fixed
- Univalent states
  - Outcome is fixed
  - May not be “known” yet
  - 1-Valent and 0-Valent states
Claim

Some initial system state is bivalent

(The outcome is not always fixed from the start.)

A 0-Valent Initial State

• All executions lead to decision of 0

A 0-Valent Initial State

• Solo execution by A also decides 0

A 1-Valent Initial State

• All executions lead to decision of 1
A 1-Valent Initial State

• Solo execution by B also decides 1

A Univalent Initial State?

• Can all executions lead to the same decision?

State is Bivalent

• Solo execution by A must decide 0
• Solo execution by B must decide 1

Critical States

0-valent

1-valent
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

From a Critical State

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

Model Dependency

• 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
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>?</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>

Reading Registers

- A runs solo, decides 0
- B reads \(x\)
- States look the same to A

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>

Writing Distinct Registers

- A writes \(y\)
- B writes \(x\)
- The song remains the same
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

States look the same to A

A writes x

A runs solo, decides 0

B writes x

A runs solo, decides 1

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

2-element array

FIFO Queue with red and black balls

Coveted red ball  
Dreaded black ball
Protocol: Write Value to Array

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

Protocol: Take Next Item from Queue

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

Why does this 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
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 mem.

• n processors, with n > 1
• Wait-free implementation
• Processors can atomically read \textit{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
  RMW(c), with
    if (c == "?") then
      Write(c, vi); decide vi;
    else
      decide c;

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?

Read-Modify-Write

more formally

• Method takes 2 arguments:
  - Variable x
  - Function f
• Method call:
  - Returns value of x
  - Replaces x with f(x)

Read-Modify-Write

public abstract class RMW {
  private int value;
  public void rmw(function f) {
    int prior  = this.value;
    this.value = f(this.value);
    return prior;
  }
}
Example: Read

class RMW {
    private int value;

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

identity function

Example: test&set

class RMW {
    private int value;

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

constant function

Example: fetch&inc

class RMW {
    private int value;

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

increment function

Example: fetch&add

class RMW {
    private int value;

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

addition function
Example: swap

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

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

Example: compare&swap

```java
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;
    }
}
```

"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 (Herlihy)

- 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

Synchronization Speed Limit

• 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!

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

```java
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];
    }
}
```

Am I first?
Yes, return my input
No, return other's input

Initialized to v

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: \( f_i(f_j(x)) = f_i(x) \)
• 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 $f_A$

B about to apply $f_B$

0-valent 1-valent

Maybe the Functions Commute

A applies $f_A$

B applies $f_B$

C runs solo

0-valent 1-valent

Maybe the Functions Overwrite

This states look the same to C

C runs solo

0-valent 1-valent
Maybe the Functions Overwrite

These states look the same to C

\[ \text{A applies } f_A \quad \text{B applies } f_B \]

C runs solo

\[ \text{C runs solo} \]

0-valent

1-valent

Impact

- 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?

CAS has Unbounded Consensus Number

public class RMWConsensus implements Consensus {
  private RMW r;

  public Object decide() {
    int i = Thread.myIndex();
    int j = r.CAS(-1, i);
    if (j == -1)
      return this.announce[i];
    else
      return this.announce[j];
  }
}

The Consensus Hierarchy

<table>
<thead>
<tr>
<th>1 Read/Write Registers, …</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 T&amp;S, F&amp;I, Swap, …</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>∞ CAS, …</td>
</tr>
</tbody>
</table>
**Consensus #4**

**Synchronous Systems**

- In real systems, one can sometimes tell if a processor had crashed
  - Timeouts
  - Broken TCP connections

- Can one solve consensus at least in synchronous systems?

**Communication Model**

- Complete graph
- Synchronous

Send a message to all processors in one round: Broadcast

At the end of the round: everybody receives a
Broadcast: Two or more processes can broadcast in the same round

At end of round...

Crash Failures

Some of the messages are lost, they are never received
Effect

Faulty processor

After a failure, the process disappears from the network

Consensus:
Everybody has an initial value

Everybody must decide on the same value
Validity condition:
If everybody starts with the same value they must decide on that value

Start

Finish

A simple algorithm

Each processor:

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

(only one round is needed)

Start

Broadcast values
Decide on minimum

This algorithm satisfies the validity condition

Start

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

Finish

Consensus with Crash Failures

The simple algorithm doesn't work

Each processor:

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

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

0

fail

1

0

4

2

3

Broadcasted values

0,1,2,3,4

1

fail

0,1,2,3,4

1,2,3,4

4

1,2,3,4

2

3

0,1,2,3,4

Decide on minimum

0

fail

0,1,2,3,4

1

1,2,3,4

Finish - No Consensus!

0

fail

0

1

1,2,3,4

0

0,1,2,3,4

1

0,1,2,3,4
If an algorithm solves consensus for \( f \) failed processes we say it is an \( f \)-resilient consensus algorithm.

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

\[
\text{Start} \\
1 \\
1 \\
1 \\
1 \\
1 \\
\]

An \( f \)-resilient algorithm

Round 1:
Broadcast my value

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

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

\[
\text{Finish} \\
1 \\
1 \\
1 \\
1 \\
1 \\
\]
Example: $f=1$ failures, $f+1=2$ rounds needed

**Start**

0

1 4

2 3

**Round 1** Broadcast all values to everybody

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

1,2,3,4

(new values)

0,1,2,3,4

**Round 2** Broadcast all new values to everybody

0,1,2,3,4

1,2,3,4

Finish Decide on minimum value

0,1,2,3,4

0 0

0,1,2,3,4

0 0

0,1,2,3,4
Example: $f=2$ failures, $f+1 = 3$ rounds needed

Start Example of execution with 2 failures

**Round 1**

Broadcast all values to everybody

**Round 2**

Broadcast new values to everybody

**Round 3**

Broadcast new values to everybody

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

Finish  
Decide on the minimum value

Failure 1
0, 1, 2, 3, 4

Failure 2
0, 1, 2, 3, 4

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

Round 1 2 3 4 5 6

Example:
5 failures,
6 rounds

At the end of the round with no failure:

- Every (non faulty) process knows about all the values of all the other participating processes
- This knowledge doesn't change until the end of the algorithm

Therefore, at the end of the round with no failure:

Everybody would decide on the same value

However, as we don't know the exact position of this round, we have to let the algorithm execute for $f+1$ rounds
Validity of algorithm:

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

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

Proof sketch:

Assume for contradiction that f or less rounds are enough

Worst case scenario:

There is a process that fails in each round

A Lower Bound

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

Worst case scenario

Round 1

before process $P_i$ fails, it sends its value $a$ to only one process $P_k$
Worst case scenario

Round 1 2

before process $P_k$ fails, it sends value $a$ to only one process $P_m$

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 $f$

Process 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
Consensus with Byzantine Failures

A Byzantine process can behave like a Crashed-failed process

Different processes receive different values

Some messages may be lost

Consensus with Byzantine Failures

f-resilient consensus algorithm solves consensus for f failed processes

After failure the process continues functioning in the network
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

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

Start Finish

Lower bound on number of rounds

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

Proof: follows from the crash failure lower bound

Upper bound on failed processes

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

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

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

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

---

Assume 6 processes are in a ring (just for fun)
Processes think they are in a triangle

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

Conclusion
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

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

Final decision

We reached consensus with 1 failure
Impossible!!!

Conclusion

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

The King Algorithm

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

There are $f+1$ phases
Each phase has two rounds
In each phase there is a different king
Example: 12 processes, 2 faults, 3 kings

initial values

\[ 0 \ 1 \ 1 \ 2 \ 1 \ 0 \ 2 \ 0 \ 1 \ 0 \ 1 \ 0 \]

Faulty

Remark: There is a king that is not faulty

The King algorithm

Each processor \( p_i \) has a preferred value \( v_i \)

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

The King algorithm: Phase k

Round 1, processor \( p_i \):

- Broadcast preferred value \( v_i \)
- Set \( v_i \) to the majority of values received
The King algorithm: **Phase k**

Round 2, **king** $p_k$:
- Broadcast new preferred value $v_k$

Round 2, **process** $p_i$:
- If $v_i$ had majority of less than $\frac{n}{2} + f$
  - then set $v_i$ to $v_k$

Example: 6 processes, 1 fault

Phase 1, Round 1
- Everybody broadcasts
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

Phase 1, Round 2

The king broadcasts

Phase 2, Round 1

Everybody broadcasts
Phase 2, Round 1

Choose the majority

Each majority population is

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

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

Phase 2, Round 2

The king broadcasts

Everybody chooses the king's value

Invariant / Conclusion

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

After that round, the majority will remain value \( v \) with a majority population which is at least

\[ n - f > \frac{n}{2} + f \]
Exponential Algorithm

solves consensus with \( n \) processes and \( f \) failures where \( f < n/3 \) in \( f + 1 \) “phases”

But: uses messages with exponential size

Atomic Broadcast

- One process wants to broadcast message to all other processes
- Either everybody should receive the (same) message, or nobody should receive the message
- Closely related to Consensus: First send the message to all, then agree!

Summary

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

Questions?