MADS

Emmanuelle Anceaume

The consensus problem in synchronous environments

http://people.irisa.fr/Emmanuelle.Anceaume/
Content of this lesson

- Consensus problem in synchronous environments
  - algorithm that tolerates crash failures
  - lower bound on the number of rounds
  - algorithm that tolerates Byzantine failures

- Consensus problem in asynchronous environments
  - Impossibility result even in a presence of at most one single crash failure
Consensus
Informal specification

- In this problem processors are trying to reach a consensus on a value.
- Each processor initially proposes a value $v$ taken from a given set of value $V$.
- At the end of the protocol, all processors agree on a single value, called the decided value, or decision.
Distributed systems: a set of $n$ processors that aim at solving a given task

- $\Pi = \{p_1, \ldots, p_n\}$.

Message-passing model: processors communicate and synchronize by exchanging messages

Among them, up to $f$ processors are faulty

- a faulty processor is a processor whose functionality is incorrect
- there are different ways for a processor to be incorrect: from crash failures to Byzantine failures
  - a crash processor stops operating, but does not perform wrong operations (e.g., delivers messages which were not sent or write an erroneous value in a register)
  - a Byzantine processor may behave arbitrarily. We cannot make any assumptions on the way it behaves
- it is not known in advance which processors are faulty
The system is synchronous: an execution is partitioned into rounds.

In each round:

- Each non-faulty processor executes one computational step, sends messages to its neighbors, and delivers all the messages sent to it during the round.
- Each faulty processor may not complete its round in its entirety (e.g., its messages may not have been received by all its neighbors). Once a processor has crashed, it stops forever its execution.
Message-passing synchronous systems

Round $i$  Round $i+1$

$\text{crashed processors do not execute anymore}$
A Byzantine processor behaves arbitrarily.
Each processor $p_i$ has special state components

- $x_i$: the input
- $y_i$: the output (also called decision)

Initially, $x_i$ holds a value, and $y_i$ is undefined. Any assignment to $y_i$ is irreversible.

A solution to the Consensus problem must guarantee the following:

- **Termination**: $y_i$ is eventually assigned a value for every non-faulty processor $p_i$.
- **Agreement**: If $y_i$ and $y_j$ are assigned, then $y_i = y_j$ for all non-faulty processors $p_i$ and $p_j$.
- **Validity**: If for some value $v$, $x_i = v$ for all processors $p_i$, and if $y_i$ is assigned for some non-faulty processor $p_i$ then $y_i = v$. 

Consensus
Specification
A simple consensus algorithm

propose($v_i$) {// algorithm run by process $p_i$
Initially $V = v_i$

1: round $k$, $1 \leq k \leq f + 1$
2: send {$v \in V$: $p_i$ has not already sent $v$} to all processors
3: receive $S_j$ from $p_j$, $0 \leq j \leq n - 1$, $j \neq i$
4: $V := V \cup \bigcup_{j=0}^{n-1} S_j$
5: $y = \min(V)$
6: return decide($y$)
}
A simple consensus algorithm: correctness

```plaintext
propose(v_i) {// algorithm run by process p_i
Initially V = v_i

1: round k, 1 ≤ k ≤ f + 1
2: send \{v ∈ V: p_i has not already sent v\} to all processors
3: receive S_j from p_j, 0 ≤ j ≤ n − 1, j ≠ i
4: V := V ∪ ⋃_{j=0}^{n-1} S_j
5: y = min(V)
6: return decide(y)
}
```

- Termination: Each non-faulty process must eventually decide on a value
  - from the code, the algorithm requires f + 1 rounds
A simple consensus algorithm: correctness

```
propose(v_i) { // algorithm run by process p_i
    Initially V = v_i

    1: round k, 1 ≤ k ≤ f + 1
    2: send {v ∈ V: p_i has not already sent v} to all processors
    3: receive S_j from p_j, 0 ≤ j ≤ n − 1, j ≠ i
    4: V := V ∪ \bigcup_{j=0}^{n-1} S_j
    5: y = \min(V)
    6: return decide(y)
}
```

- **Validity**: If all the non-faulty processes have the same initial value, then the agreed upon value by all the non-faulty processes must be that same value.
  - Processors do not send fictitious values in this failure model. For all i, j, if the initial value v_i is identical to v_j then the only value sent by any processor is the value that has been agreed upon.
A simple consensus algorithm: correctness

- Agreement: If $y_i$ and $y_j$ are assigned, then $y_i = y_j$ for all non-faulty processors $p_i$ and $p_j$.

Lemma

In every execution, at the end of round $f + 1$, $V_i = V_j$ for every nonfaulty $p_i$ and $p_j$.

Proof: It suffices to show that if $x \in V_i$ at the end of round $f + 1$ then $x \in V_j$ at the end of round $f + 1$, for all nonfaulty $p_i$ and $p_j$.

Let $r =$ 1rst round at which $x \in V_i$ for any nonfaulty $p_i$ (L4) (if $x$ is already in $V_i$, $r = 0$)

- if $r \leq f$ then at round $r + 1$ $p_i$ broadcasts $x$ to each $p_j$ (L2), and $p_j$ adds $x$ to $V_j$ if not already done (L4)
Proof (cont’d)

- Otherwise, suppose that $r = f + 1$.
  - let $p_j$ be a nonfaulty such that $p_j$ receives $x$ for the first time in round $f + 1$. There must have a chain of $f + 1$ processors such that $x$ is transferred to $p_j$
    - $p_{i_1}$ sends $x$ to $p_{i_2}$ in round 1
    - ...
    - $p_{i_f}$ sends $x$ to $p_{i_{f+1}}$ in round $f$, and
    - $p_{i_{f+1}}$ sends $x$ to $p_j$ in round $f + 1$
  - by construction each processor sends $m$ only once
  - thus $p_{i_1}, p_{i_2}, \ldots, p_{i_{f+1}}$ are $f + 1$ distinct processors
  - thus there must be at least one correct processor among them
  - thus this processor must have stored $x$ in round $\leq f < r$
  - which contradicts the assumption that $r$ is minimal

- As a consequence, nonfaulty processors have the same set $V$ in L5

- And thus decide on the same value
A simple consensus algorithm: correctness

Illustration of case 2 of Lemma 1. \( f = 3 \)
A simple consensus algorithm: complexity

- There are $f + 1$ rounds, where $f < n$
- The number of messages sent in each round is at most $n^2$
- Hence the total number of messages is $O((f + 1)n^2)$.
We present a lower bound of \( f + 1 \) on the number of rounds required for reaching consensus in the presence of \( f \) crash failures.

We focus on binary consensus - fine for a lower bound.

The intuition behind the lower bound is that if processors decide too early, they cannot distinguish between executions in which they should make different decisions.
We consider a synchronous system consisting of a set of processors $p_1, \ldots, p_n$ and communication channels.

- Each $p_i$ has a one-bit input register $x_{p_i}$, and output register $y_{p_i}$ with values in $\{0, 1, b\}$.
- The state of $p_i$ comprises the value of $x_{p_i}$, the value of $y_{p_i}$ (and its program counter, and its internal storage...).
- Initial state of $p_i$: $x_{p_i} = 0$ or $x_{p_i} = 1$ and $y_{p_i} = b$.
- Decision states: $y_{p_i} = 0$ or $y_{p_i} = 1$.
- Transition function:
  - deterministic.
  - cannot change the decision value ($y_{p_i}$ is writable only once).
Some preliminary definitions

**Definition (View of a processor)**

Let $\alpha$ be an execution and let $p_i$ be a processor. The **view** of $p_i$, denoted by $\alpha|p_i$, is the subsequence of computation and message delivery events that occur in $\alpha$ at $p_i$ together with $p_i$'s initial configuration.

**Definition (Similar executions)**

Let $\alpha_1$ and $\alpha_2$ be two executions and let $p_i$ be a processor that is nonfaulty in both $\alpha_1$ and $\alpha_2$. $\alpha_1$ is similar to $\alpha_2$ with respect to $p_i$, denoted by $\alpha_1 \sim^{p_i} \alpha_2$, if $\alpha_1|p_i = \alpha_2|p_i$. 
Some preliminary definitions

A key notion is the set of decisions that can reached from a particular configuration.

**Definition (Valence of a configuration)**

The valence of a configuration $C$ is the set of all values that are decided by nonfaulty processors in a configuration reachable from $C$.

- $C$ is univalent if this set contains a single value.
  - 0-valent if the value is 0
  - 1-valent if the value is 1
- bivalent if the set contains both 0 and 1.
Proof of the lower bound

Theorem

Any consensus algorithm $A$ for $n$ processors, resilient to $f$ crash failures, requires at least $f + 1$ rounds in some execution for all $n \geq f + 2$

Proof. Let $A$ be any consensus algorithm for $n$ processors and $f$ crash failures

1. we first show that there exists an $(f - 1)$-round execution of $A$ in which the configuration at the end is undecided
   - by induction on the round number
2. we then show that with just one more round it is not possible for the processors to decide explicitly
3. thus at least $f + 1$ rounds are required for deciding
Lemma (1): Bivalent initial configuration(s)

Lemma (1)

Algorithm $A$ has at least one bivalent initial configuration.
Proof of Lemma (1)

Proof: By contradiction. Suppose that all the initial configurations are univalent (i.e. are completely determined by the set of initial values). By the validity property,

- initial configurations such that 0 is decided
- initial configurations such that 1 is decided

We can order initial configurations in a chain of configurations, where two configurations are next to each other if they differ by only one value.

→ the difference between two adjacent configurations is the starting value of a one processor

\[
\begin{align*}
0 &\quad [0,0,\ldots,0] \\
1 &\quad [1,0,\ldots,0] \\
2 &\quad [1,1,\ldots,0] \\
\vdots &\quad [1,\ldots,1,0,0,\ldots,0] \\
i &\quad [1,\ldots,1,0,0,\ldots,0] \\
i+1 &\quad [1,\ldots,1,1,0,\ldots,0] \\
n &\quad [1,1,\ldots,1]
\end{align*}
\]
Proof of Lemma (1)
Proof of Lemma (1)
Proof of Lemma (1)
Proof of Lemma (1)
Proof of Lemma (1)

In σ processor $p_i$ takes no step
So its initial value cannot be observed by someone else

All processors must eventually decide
(failure tolerant protocol)

Since $C_0$ is 0-valent the decision state is 0
Proof of Lemma (1)

In $\sigma$ processor $p_i$ fails initially (i.e., does not receive nor send messages).

So its initial value cannot be observed.

All processors must eventually decide (1-failure tolerant protocol)

Since $C_0$ is 0-valent the decision state is 0.

Run $\sigma$ can be made from $C_1$ too since no processor has ever heard about $p_i$.

Thus all the processors (except $p_i$) should reach the "0" deciding state.

This is a contradiction since by assumption $C_1$ is a "1"-valent configuration.
Proof of Lemma (1)

- So this result contradicts the fact that the outcome of the consensus algorithm is uniquely predetermined by the initial configurations.
- $C_0$ can lead to a "0" decision state or to a "1"-decision state, depending on the pattern of failures.

**Initial bivalent configuration**

Any synchronous consensus algorithm that is resilient to $f$ crashes has at least one bivalent initial configuration.
Lemma (2)

For each \( k, \ 0 \leq k \leq f - 1 \), there is a \( k \)-round execution of \( A \) that ends in a bivalent configuration

Proof: by induction on the round number

- The base case, \( k = 0 \), follows from Lemma (1)

- Assume that the lemma is true for rounds \( 0 \leq k' \leq f - 2 \), and show that it is true for \( 0 \leq k \leq f - 1 \)

- Let \( \alpha_{k-1} \) be the \((k - 1)\)-round execution ending in a bivalent configuration (exists by the inductive assumption)

- Assume by contradiction that all one-round extensions of \( \alpha_{k-1} \) with at most one crash failure end in 0-valent and 1-valent configurations
Suppose that $p_i$ fails to sends to the subset of processors $p_1, \ldots, p_m$ with $1 \leq m \leq n$. 

1-valent $\beta_k = \alpha^0_k$

1-valent $\alpha^i_k$

0-valent $\alpha^{j+1}_k$

0-valent $\gamma_k = \alpha^m_k$
Switch from a 1-valent configuration to a 0-valent configuration

The only difference between both $\alpha_k^j$ and $\alpha_k^{j+1}$ is that $p_i$ sends a value to $p_{j+1}$ in $\alpha_k^j$ but not in $\alpha_k^{j+1}$.
Proof of Lemma (2) (cont’d)

- the number of crashes in $\alpha^j_k$ and in $\alpha^{j+1}_k$ is less than $f$ (since at most $k - 1 < f - 1$ processors crash in $\alpha_{k-1}$ and $p_i$ crashes in round $k$)
- thus there is still one more processor that can crash without violating the bound $f$
- consider both extensions
  - $\delta^j_{k+1}$ of $\alpha^j_k$
  - $\delta^{j+1}_{k+1}$ of $\alpha^{j+1}_k$
- in both extensions, $p_{j+1}$ fails at the beginning of round $k + 1$
- $p_{j+1}$ did not get the opportunity of revealing whether or not it received a message from $p_i$ at the beginning of round $k + 1$
- both extensions are similar with respect to every nonfaulty processor (since the only difference between them is that $p_i$ sends its information to $p_{j+1}$ in $\delta^j_k$ but not in $\delta^{j+1}_k$, but $p_{j+1}$ failed)
Bivalent configuration

\[ \alpha_{k-1} < f - 1 \text{ crash failures} \]

1 crash failure

rounds 1 to k-1

Contradiction!
Thus there must exist a one-round extension of $\alpha_{k-1}$ ending in a bivalent configuration.
we have shown that we necessarily have a \((f - 1)\) round execution ending in a bivalent configuration

we now show that the \(f\)-th round execution does not necessarily preserve bivalence, but nonfaulty processors cannot determine yet what decision to make, and thus an additional round is necessary
Lemma (3)

Lemma

Let $\alpha_{f-1}$ be an $(f - 1)$-round execution of $A$ that ends in a bivalent configuration, then there exists a one round extension of $\alpha_{f-1}$ in which some non-faulty processor has not decided.

Proof
Lemma (3)

Proof

- Let $\beta_f$ be the one-round extension of $\alpha_{f-1}$ in which no failure occurs in round $f$
- If $\beta_f$ ends in a bivalent configuration we are done
- Suppose that $\beta_f$ ends in a univalent configuration (say 1-valent)
- Since the configuration at the end of $\alpha_{f-1}$ is bivalent, some other one-round extension of $\alpha_{f-1}$ must be 0-valent. Let $\gamma_f$ be this extension
- Let $p_i$ be some faulty processor
Lemma (3)

$\beta_f$ and $\delta_f$ are similar with respect to $p_k$. Thus $p_k$ is either undecided or decide 1 at the end of round $f$.

$\gamma_f$ and $\delta_f$ are similar with respect to $p_j$. Thus $p_j$ is either undecided or decide 0 at the end of round $f$.

Since algorithm A satisfies the agreement Property, it cannot be the case that in $\delta_f$ both $p_k$ and $p_j$ have decided.
Thus Lemmas 2 and 3 together imply the existence of an $f$-round execution in which some nonfaulty processor has not decided. Thus $f + 1$ rounds are required for termination.
Synchronous systems with Byzantine failures

Byzantine generals

Several divisions of the Byzantine army are waiting outside an enemy city. Each division is commanded by a general. Generals can communicate with reliable messengers.

Each general should eventually decide (Termination) on a plan, and this plan should be common: attack the city or not (Agreement), and if the generals are unanimous in their opinion, then this opinion should be the decision (Validity). Some of the generals may be traitors, and may try to prevent the loyal generals from agreeing.

To do so, traitors send conflicting messages to different generals, falsely report on what they have heard from other generals, and even conspire and form a coalition.
The phase-king algorithm (proposed by Berman and Garay) solves the consensus problem in a synchronous model.

- \( n > 4f \) processors, \( f \) Byzantine processors
- the algorithm requires \( f + 1 \) phases
- at each phase, a unique processor plays an asymmetrical role (leader)
Consensus problem with Byzantine failures

- **Agreement**: All non-faulty processors must agree on the same single value.
- **Validity**: If all the non-faulty processors have the same initial value, then the agreed upon value by all the non-faulty processors must be that same value.
- **Termination**: Each non-faulty processor must eventually decide on a value.
The phase-king algorithm

Round 1 :

- Each processor sends its estimate of the consensus value to all processors
- Each processor waits for the values of all other processors
- At the end of the round, it counts \#0 and \#1
  - \( \text{maj} \leftarrow 0 \) if greater than \( n/2 \), and \( \text{mult} \leftarrow \#0 \) or
  - \( \text{maj} \leftarrow 1 \) if greater than \( n/2 \), and \( \text{mult} \leftarrow \#1 \)
  - \( \text{maj} \leftarrow \) default value (0 or 1), and \( \text{mult} \leftarrow \#0 \) or \#1

happens if faulty processors do not respond and nonfaulty ones are split
The phase-king algorithm

Round 2:

- the phase-king initiates processing
- sends to all its majority value \( maj \) - serves as a tie-breaker value for \( p_j \) if \( p_j \) has received less than \( n/2 + f \) votes
- the reason for this is that among \( p_j \)’s received votes, \( f \) can be bogus and hence \( p_j \) has not received a clear majority of votes (i.e. \( >n/2 \)) from a majority of non-malicious processors
- otherwise (\( mult > n/2 + f \)) \( p_j \) can safely updates its estimate of the consensus variable \( v \), irrespective of the value sent by the phase-king

At the end of phase \( f + 1 \), it is guaranteed that the estimate \( v \) of all the processors is the correct consensus value
propose($v_i$) {// algorithm run by process p_i
boolean: $v \leftarrow v_i$
integer: $f \leftarrow$ maximum number of malicious processors, $f < [n/4]$

1: Each processor executes the following $f + 1$ phases
2: for phase = 1 to $f + 1$ do
3: // Execute the following round 1 actions
4: send $v$ to all processors
5: wait value $v_j$ from each processor $p_j$
6: $maj \leftarrow$ the value among the $v_j$ that occurs $> n/2$ times
   ($maj \leftarrow$ default_value if no majority)
7: $mult \leftarrow$ number of times $maj$ occurs
8: // Execute the following round 2 actions
9: if $i = phase$ then
10: send $maj$ to all processors
11: receive $tiebreaker$ from $p_{phase}$
   ($tiebreaker \leftarrow$ default_value if nothing is received)
12: if $mult > n/2 + f$ then
13: $v \leftarrow maj$
14: else $v \leftarrow tiebreaker$
15: if $phase = f + 1$ then
16: return decide($v$)
}
Message pattern for the phase-king algorithm

The diagram illustrates the message pattern for the phase-king algorithm, showing how messages are exchanged between different phases and rounds. Each phase is represented by a horizontal line, and each round is represented by a vertical line. The arrows indicate the direction of message transmission.

- **Phase 1**: Messages are exchanged between the first and second rounds.
- **Phase f+1**: Messages are exchanged between the first and second rounds.

The pattern repeats for different phases, with arrows indicating the flow of messages.
Proof of correctness of the phase-king consensus algorithm

The proof is achieved in 3 steps:

**Step 1**: among the \( f + 1 \) phases, the phase king \( p_k \) of some phase \( k \) is non-malicious

**Step 2**: all non-malicious processors have the same estimate \( v \) at the end of phase \( k \). Indeed, any two nonfaulty \( p_i \) and \( p_j \) set their estimate in 3 ways:

**Step 2.1** both \( p_i \) and \( p_j \) use their own \( maj \) values.
- let \( x \) be \( p_i \)'s \( maj \) value.
- we have \( \text{mult} > \frac{n}{2} + f \) (L12).
- at least \( \frac{n}{2} \) voters are nonfaulty.
- thus \( p_j \) must also have received \( \geq \frac{n}{2} \) votes for the value \( x \), thus \( p_j \)'s majority value is \( x \).
The proof is achieved in 3 steps:

**Step 1**: among the \(f + 1\) phases, the phase king \(p_k\) of some phase \(k\) is non-malicious

**Step 2**: all non-malicious processors have the same estimate \(v\) at the end of phase \(k\). Indeed, any two nonfaulty \(p_i\) and \(p_j\) set their estimate in 3 ways:

- **Step 2.2** both \(p_i\) and \(p_j\) use the phase king’s tie-breaker value.
  - \(p_k\) is non-malicious (by assumption)
  - thus \(p_k\) must have sent the same tie-breaker value to both \(p_i\) and \(p_j\).
Proof of correctness of the phase-king consensus algorithm

The proof is achieved in 3 steps:

**Step 1**: among the \( f + 1 \) phases, the phase king \( p_k \) of some phase \( k \) is non-malicious.

**Step 2**: all non-malicious processors have the same estimate \( v \) at the end of phase \( k \). Indeed, any two nonfaulty \( p_i \) and \( p_j \) set their estimate in 3 ways:

1. \( p_i \) uses its own \( \text{maj} \) value and \( p_j \) uses the phase king’s tie-breaker as their new estimate.
2. let \( x \) be \( p_i \)'s majority value.
3. we have \( \text{mult}_i > n/2 + f \).
4. at least \( n/2 \) voters are nonfaulty.
5. thus \( p_k \) must also have received \( \geq n/2 \) votes for \( x \).
6. thus \( p_k \)'s majority value is \( x \), and sends \( x \) as tie-breaker.
Proof of correctness of the phase-king consensus algorithm

The proof is achieved in 3 steps:

**Step 1**: among the \( f + 1 \) phases, the phase king \( p_k \) of some phase \( k \) is non-malicious

**Step 2**: all non-malicious processors have the same estimate \( v \) at the end of phase \( k \). Indeed, any two nonfaulty \( p_i \) and \( p_j \) set their estimate in 3 ways:

**Conclusion of step 2** For all three possibilities, any two non-malicious processes \( p_i \) and \( p_j \) agree on the consensus estimate at the end of phase \( k \), where the phase king \( p_k \) is non-malicious.
Step 3: all the non-malicious processors have the same consensus estimate at the start of phase \( k + 1 \). By the algorithm, they keep it up to the end of phase \( k + 1 \). This is self-evident because:

- we have \( n > 4f \)
- each nonfaulty processor receives at least \( n - f > n/2 + f \) votes for \( x \) from the nonfaulty processors in round 1 of phase \( k + 1 \)
- hence, each nonfaulty keeps its own estimate \( v = x \) of the consensus value at the end of phase \( k + 1 \)

The same logic holds for all subsequent phases. Hence the consensus value is correct.

Complexity: the algorithm requires \( f + 1 \) phases made of 2 rounds in which the \( n \) processors send their value to all the other ones. Thus \((f + 1)n^2 \) messages.
Bibliography

- N. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers