Every process has a value $v_i$ to propose. After running a consensus algorithm, all processes should deliver the same value.
**Consensus**

**Validity**
If all processes that propose a value propose $v$, then all correct processes eventually decide $v$

**Agreement**
If a correct process decides $v$, then all correct processes eventually decide $v$

**Integrity**
Every correct process decides at most one value, and if it decides $v$, then some process must have proposed $v$

**Termination**
Every correct process eventually decides some value
MODEL

- **Synchronous** message passing
  - Execution is a sequence of rounds
  - In each round every process takes a step
    - sends messages to neighbors
    - receives messages send in that round
    - changes its state
- Network is fully connected
- **No communication failures**
A SIMPLE CONSENSUS ALGORITHM

Process \( p_i \): 
Initially \( V = \{v_i\} \)

To execute \( \text{propose}(v_i) \): 
1. Send \( \{v_i\} \) to all 

\( \text{decide}(\ ) \) occurs as follows: 
2. for all \( j, 0 \leq j \leq n + 1, j \neq i \), do 
3. receive \( S_j \) from \( p_j \) 
4. \( V := V \cup S_j \) 
5. decide \( \min(V) \)
What is going on

A correct process $p$ has not received all proposals by the end of round $i$. Can $p$ decide?

Another process may have received the missing proposal at the end of round $i$ and be ready to relay it in round $i + 1$.
DANGEROUS CHAINS

 Dangerous chain

 The last process in the chain is correct, all others faulty
LIVING DANGEROUSLY

How many rounds can a dangerous chain span?

- $f$ faulty processes
- At most $f + 1$ nodes in the chain
- Spans at most $f$ rounds

It is safe to decide by the end of round $f + 1$!
THE ALGORITHM

Process $p_i$:
Initially $V = \{v_i\}$

To execute $\text{propose}(v_i)$:

round $k, 1 \leq k \leq f + 1$

1. Send $\{v \in V: p_i \text{ has not already sent } v\}$ to all
2. for all $j, 0 \leq j \leq n + 1, j \neq i$, do
3. receive $S_j$ from $p_j$
4. $V := V \cup S_j$

$\text{decide()}$ occurs as follows:
5. if $k = f + 1$
6. decide $\text{min}(V)$
**Proving termination**

To execute $\text{propose}(v_i)$:

1. Send \( \{v \in V : p_i \text{ has not already sent } v \} \) to all

2. for all \( j, 0 \leq j \leq n + 1, j \neq i \), do

3. receive \( S_j \) from \( p_j \)

4. \( V := V \cup S_j \)

\( \text{decide( )} \) occurs as follows:

5. if \( k = f + 1 \)

6. decide \( \min(V) \)

Every correct process

• Reaches round \( f + 1 \)

• Decides \( \min(V) \), which is well defined
**Proving Integrity**

To execute \( \text{propose}(v_i) \):

1. Send \( \{ v \in V : p_i \text{ has not already sent } v \} \) to all
2. for all \( j, 0 \leq j \leq n + 1, j \neq i \), do
3. receive \( S_j \) from \( p_j \)
4. \( V := V \cup S_j \)

\( \text{decide}() \) occurs as follows:

5. if \( k = f + 1 \)
6. decide \( \min(V) \)

**At most one value:**
One \( \text{decide}() \) and \( \min(V) \) is unique

**Only if it was proposed:**

- To be decided, must be in \( V \) in round \( f + 1 \)
- If value = \( v_i \), then it is proposed in round \( 1 \)
- else, suppose it was received in round \( k \)
  - By induction:
    - \( k = 1 \)
      - By Uniform Integrity of underlying send and receive, it must have been sent in round \( 1 \)
      - By the protocol, and because we only have benign failures, it must have been proposed
    - Induction hypothesis: all values received up to round \( k = j \) have been proposed
    - \( k = j + 1 \)
      - Sent in round \( j + 1 \) (Uniform Integrity of send and synchronous model)
      - Must have been part of \( V \) of sender at end of round \( j \)
      - By the protocol, must have been received by sender by the end of round \( j \)
      - By induction hypothesis, must have been proposed
To execute \textit{propose}(v_i):

1. Send \( \{v \in V: p_i \text{ has not already sent } v\} \) to all
2. for all \( j, 0 \leq j \leq n + 1, j \neq i \), do
3. receive \( S_j \) from \( p_j \)
4. \( V := V \cup S_j \)

\textit{decide( )} occurs as follows:

5. if \( k = f + 1 \)
6. decide \( \min(V) \)

Suppose every process proposes \( v^* \)

Since we only deal with crash failures, only \( v^* \) can be sent

By Uniform Integrity of send and receive, only \( v^* \) can be received

By the protocol, \( V = \{v^*\} \)

\( \min(V) = v^* \)

decide\( (v^*) \)
To execute \( \text{propose}(v_i) \):

**round** \( k, 1 \leq k \leq f + 1 \)

1. Send \( \{v \in V: p_i \text{ has not already sent } v\} \) to all
2. for all \( j, 0 \leq j \leq n + 1, j \neq i \), do
3. receive \( S_j \) from \( p_j \)
4. \( V := V \cup S_j \)

\( \text{decide}() \) occurs as follows:

5. if \( k = f + 1 \)
6. decide \( \min(V) \)

**Lemma 1**

For any \( r \geq 1 \), if a process \( p \) receives a value \( v \) in round \( r \), there exists a sequence of distinct processes \( p_0, p_1, \ldots, p_r \) such that \( p_r = p \), \( p_0 \) is \( v \)'s proponent and in each round \( k \), \( p_{k-1} \) sends \( v \) and \( p_k \) receives it.

**Proof**

By induction on the length of the sequence
To execute \(\text{propose}(v_i)\):

1. Send \(\{v \in V: p_i \text{ has not already sent } v\}\) to all
2. for all \(j, 0 \leq j \leq n + 1, j \neq i\), do
3. receive \(S_j\) from \(p_j\)
4. \(V := V \cup S_j\)

\(\text{decide}( )\) occurs as follows:

5. if \(k = f + 1\)
6. decide \(\min(V)\)

**Proof**

- Show that if a correct \(p\) has \(x\) in its \(V\) at the end of round \(f + 1\) then every correct process has \(x\) in its \(V\) at the end of round \(f + 1\)
- Let \(r\) be the earliest round \(x\) is added to the \(V\) of a correct process. Let that process be \(p^*\)
- If \(r \leq f\), then \(p^*\) sends \(x\) in round \(r + 1 \leq f + 1\)
  Every correct process receives \(x\) and adds it to its \(V\) in round \(r + 1\)
- **What if \(r = f + 1\)?**
  - By Lemma 1, there exists a sequence of distinct processes \(p_0, \ldots, p_{f+1} = p^*\)
  - Consider processes \(p_0, \ldots, p_f\)
  - \(f + 1\) processes; only \(f\) can be faulty
  - One of \(p_0, \ldots, p_f\) is correct and adds \(x\) to its \(V\) before \(p^*\) does it in round \(r\)
  - Contradiction!

**Lemma 2**

In every execution, at the end of round \(f + 1\), \(V_i = V_j\) for every correct process \(p_i\) and \(p_j\)

Agreement follows from Lemma 2, since \(\min\) is a deterministic function
ADMINISTRIVIA

- Research project
  - Declare your team by Oct 1st (by email to me)
  - Declare your topic by Oct 8 (by email to me)
  - Not sure what to do? Come talk to me.
MODELING FAULTS

- Mean Time To Failure/Mean Time To Recover
  - used mostly for disks
  - of questionable value in expressing reliability

- Threshold: $f$ out of $n$
  - makes condition for correct operation explicit
  - measures fault-tolerance of the architecture, not of individual components

- Enumerate failure scenarios
A Hierarchy of Failure Models

- Fail-stop
- Crash
- Send omission
- Receive omission
- General omission
- Arbitrary (Byzantine) failures

○ = benign failures
A hierarchy of failure models
FAULT TOLERANCE: THE PROBLEM

Clients

Server

Solution: replicate the server
**Replication in time**

- When a server fails, restart it or replace it
- Failures are **detected**, not masked
- Lower maintenance, lower availability
- Tolerates only benign failures
**Replication in space**

- Run multiple copies of a server (replicas)
- Vote on replica output
- Failures are **masked**
- High availability and can tolerate arbitrary failures
  - but at high cost
The enemy: non-determinism

An event is non-deterministic if its output is not uniquely determined by its input.

The problem with non-determinism:

- Replication in time: must reproduce the original outcome of all non-deterministic events.
- Replication in space: each replica must handle non-deterministic events identically.
The solution: state machines

Design the server as a deterministic state machine

Diagram:

1 - a - 3 - b - 2
   |     |      |
   v     v      v
4 - f - 1 - e - 4
   |     |      |
   v     v      v
2 - d - 3 - c - 2
The solution: state machines

State machine example: a switch
Ingredients: a server

1. Make server deterministic (state machine)
2. Replicate server
3. Ensure that all replicas go through the same sequence of state transitions
4. Vote on replica outputs
STATE MACHINE REPLICATION

Ingredients: a server

1. Make server deterministic (state machine)

2. Replicate server

3. Ensure that all replicas go through the same sequence of state transitions

4. Vote on replica outputs

All state machines receive all commands in the same order
Ingredients: a server

1. Make server deterministic (state machine)
2. Replicate server
3. Ensure that all replicas go through the same sequence of state transitions
4. Vote on replica outputs
4. Vote on replica outputs

When in trouble, cheat!

Voter and client share fate!