EECS 591
Distributed Systems

Manos Kapritsos
Winter 2018

Slides by: Lorenzo Alvisi
3-Phase Commit

Coordinator $c$

1. sends VOTE-REQ to all participants

Participant $p_i$

2. sends $vote_i$ to Coordinator
   
   if $vote_i = \text{No}$ then
   
   $decision_i := \text{Abort}$
   
   halt

3. if (all votes are \textbf{Yes}) then
   
   send \textbf{Precommit} to all
   
   else
   
   $decision_c := \text{Abort}$
   
   send \textbf{Abort} to all who voted \textbf{Yes}
   
   halt

4. if received \textbf{Precommit} then
   
   send \textbf{Ack}

5. collect \textbf{Ack} from all participants
   
   When all \textbf{Ack}'s have been received:
   
   $decision_c := \text{Commit}$
   
   send \textbf{Commit} to all

6. When $p_i$ receives \textbf{Commit},
   
   sets $decision_i := \text{Commit}$ and halts
A simplifying assumption

No communication failures

Timeout on a process = process failed

A process is operational if it is currently running and participating in the protocol
**Timeout actions**

### Coordinator $c$

- **Step 2:** $p_i$ is waiting for VOTE-REQ from the coordinator
- **Step 3:** Coordinator is waiting for vote from participants
  - Same as in 2PC
- **Step 4:** $p_i$ is waiting for Precommit
  - Run termination protocol
- **Step 5:** Coordinator is waiting for Ack's
  - Coordinator sends Commit
- **Step 6:** $p_i$ is waiting for Commit
  - Run termination protocol

### Participant $p_i$

- **Step 2:** $p_i$ is waiting for VOTE-REQ from the coordinator
- **Step 3:** Coordinator is waiting for vote from participants
  - Same as in 2PC
- **Step 4:** $p_i$ is waiting for Precommit
  - Run termination protocol
- **Step 5:** Coordinator is waiting for Ack's
  - Coordinator sends Commit
- **Step 6:** $p_i$ is waiting for Commit
  - Run termination protocol
Termination protocol

• When \( p_i \) times out, it starts an election protocol to elect a new coordinator.

• The new coordinator sends STATE-REQ to all processes that participated in the election.

• The new coordinator collects the states and follows a set of termination rules.
The new coordinator collects the states and follows a set of termination rules:

TR1: if some process decided **Abort**, then
decide **Abort**
send **Abort** to all
halt

TR2: if some process decided **Commit**, then
decide **Commit**
send **Commit** to all
halt

TR3: if all processes that reported state are uncertain, then
decide **Abort**
send **Abort** to all
halt

TR4: if some process is committable, but none committed, then
send **Precommit** to uncertain processes
wait for **Ack**’s
send **Commit** to all
halt
Termination protocol and failures

Processes can fail while executing the termination protocol

- if \( c \) times out on \( p \), it can just ignore \( p \)

- if \( c \) fails, a new coordinator is elected and the protocol is restarted (election protocol to follow)

- total failures will need special care
Recovering $p$

- If $p$ fails before sending Yes, decide **Abort**
- If $p$ fails after having decided, follow decision
- If $p$ fails after voting Yes, but before receiving decision value
  - $p$ asks other processes for help
  - 3PC is non-blocking: $p$ will receive a response with the decision
- If $p$ has received **Precommit**
  - still needs to ask other processes (cannot just **Commit**)

No need to log **Precommit**!
The election protocol

- Processes agree on linear ordering (e.g. by pid)
- Each process $p$ maintains a set $UP_p$ of all processes that it believes to be operational
- When $p$ detects failure of $c$, it removes $c$ from $UP_p$ and chooses smallest $q$ in $UP_p$ to be the new coordinator
- If $p = q$, then $p$ is the new coordinator
- Otherwise, $p$ sends UR-ELECTED to $q
WHAT IF...?

What if $p'$, which has not detected the failure of $c$, receives a STATE-REQ from $q$?

- it concludes that $c$ must be faulty
- it removes from $UP_{p'}$ every $q' < q$

What if $p'$ receives a STATE-REQ from $q' < q$ after it has changed the coordinator to $q$?

- $p'$ ignores the request
Total failure

Suppose that \( p \) is the first process to recover and that \( p \) is uncertain. Can \( p \) decide Abort?

Some process could have decided Commit after \( p \) crashed!

\( p \) is blocked until some process \( q \) recovers such that either

- \( q \) can recover independently
- \( q \) is the last process to fail: then \( q \) can simply invoke the termination protocol
Determining the last process to fail

Suppose a set $R$ of processes has recovered

Does $R$ contain the last process to fail?

- the last process to fail is in the $UP$ set of every process
- so the last process to fail must be in

$$
\bigcap_{p \in R} UP_p
$$

$R$ contains the last process to fail if:

$$
\bigcap_{p \in R} UP_p \subseteq R
$$
**Administrivia**

- Homework #1 handed out today after class
  - Due Monday, Feb 5, before class (prefer hardcopy)
  - Individual work only
    - No collaboration with classmates
    - No looking up solutions online
  - No handwritten answers
- Research project
  - Declare your team and topic by Feb 7
  - Not sure what to do? Come talk to me.
Consensus and Reliable Broadcast
Broadcast

If a process sends a message $m$, then every process eventually delivers $m$.

How can we adapt the spec for an environment where processes may fail?
RELIABLE BROADCAST

Validity: If the sender is correct and broadcasts a message $m$, then all correct processes eventually deliver $m$.

Agreement: If a correct process delivers a message $m$, then all correct processes eventually deliver $m$.

Integrity: Every correct process delivers at most one message, and if it delivers $m \neq SF$, then some process must have broadcast $m$. 
Terminating Reliable broadcast

Validity
If the sender is correct and broadcasts a message \( m \), then all correct processes eventually deliver \( m \)

Agreement
If a correct process delivers a message \( m \), then all correct processes eventually deliver \( m \)

Integrity
Every correct process delivers at most one message, and if it delivers \( m \neq SF \), then some process must have broadcast \( m \)

Termination
Every correct process eventually delivers some message
**Consensus**

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
Properties of \textit{send}(m) and \textit{receive}(m)

Benign failures:

\textbf{Validity} \quad \text{If } p \text{ sends } m \text{ to } q, \text{ and } p, q \text{ and the link between them are correct, then } q \text{ eventually receives } m

\textbf{Uniform* integrity} \quad \text{For every message } m, q \text{ receives } m \text{ at most once from } p, \text{ and only if } p \text{ sent } m \text{ to } q

\* A property is called uniform if it applies to both correct and faulty processes
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 $\min(V)$
AN EXECUTION

\[ p_1 \quad p_2 \quad p_3 \quad p_4 \]
AN EXECUTION

What should $p_3$ decide at the end of the round?
What should $p_3$ decide at the end of the round?
A process that receives a proposal in round 1, relays it to others during round 2.

Suppose $p_3$ hasn’t heard from $p_2$ at the end of round 2. Can $p_3$ decide?
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 $\min(V)$
To execute \texttt{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. \( V := V \cup S_j \)

\textbf{decide}( ) occurs as follows:

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

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

Every correct process

\begin{itemize}
  \item Reaches round \( f + 1 \)
  \item Decides \( \min(V) \), which is well defined
\end{itemize}
To execute `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. \( V := V \cup S_j \)

`decide()` occurs as follows:

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

**At most one value:**
One `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 $\text{propose}(v_i)$:

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. $\text{decide } \min(V)$

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, $V = \{v^*\}$

$\min(V) = v^*$

$\text{decide}(v^*)$
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 \( p_{k-1} \) sends \( v \) and \( p_k \) receives it.

Proof

By induction on the length of the sequence
To execute \( \text{propose}(v_i) \):
\[
\begin{align*}
\text{round} \quad k, 1 \leq k \leq f + 1 \\
1. \quad \text{Send } \{ v \in V : p_i \text{ has not already sent } v \} \text{ to all} \\
2. \quad \text{for all } j, 0 \leq j \leq n + 1, j \neq i, \text{ do} \\
3. \quad \text{receive } S_j \text{ from } p_j \\
4. \quad V := V \cup S_j
\end{align*}
\]
\( \text{decide( )} \) occurs as follows:
\[
\begin{align*}
5. \quad \text{if } k = f + 1 \\
6. \quad \text{decide } \min(V)
\end{align*}
\]

**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.

**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 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!