-Do you take each other?
  -I do.
  -I do.
  -I now pronounce you atomically committed.
**Properties**

**Property:** a predicate evaluated over a run of the program (also called a trace)

Example:
“every message that is received was previously sent”

Not everything you may want to say about a program is a property:
“the program sends an average of 50 messages in a run”
SAFETY PROPERTIES

- "nothing bad happens"
- only one process can be in the critical section at any time
- messages that are delivered are delivered in causal order
- Windows never crashes
- A safety property is "prefix closed":
  - if it holds in a run, it holds in every prefix
LIVENESS PROPERTIES

- “something good eventually happens”
  - a process that wishes to enter the critical section eventually does so
  - some message is eventually delivered
  - Windows eventually boots
- Every run can be extended to satisfy a liveness property
  - if it doesn’t hold in a run, that doesn’t mean it may not hold eventually
Safely or Liveness?

Whenever process A wants to enter the critical section, then all other processes get to enter at most once before A gets to enter.

This program terminates

If this program eventually sends a message, it will be a well-formed HTTP request.

Safety

Liveness

Safety
A really cool theorem

Every property is a combination of a safety property and a liveness property

(Alpern & Schneider)
Atomic commit: the objective

Preserve data consistency for distributed transactions in the presence of failures
Model

- For each distributed transaction T:
  - one coordinator
  - a set of participants
- Coordinator knows participants; participants don’t necessarily know each other
- Each process has access to a Distributed Transaction Log (DT Log) on stable storage
The setup

- Each process $p_i$ has an input value $vote_i$
  $\quad vote_i \in \{Yes, No\}$

- Each process $p_i$ has an output value $decision_i$
  $\quad decision_i \in \{Commit, Abort\}$
AC SPECIFICATION

AC-1: All processes that reach a decision reach the same one
AC-2: A process cannot reverse its decision after it has reached one
AC-3: The Commit decision can only be reached if all processes vote Yes
AC-4: If there are no failures and all processes vote Yes, then the decision must be Commit
AC-5: If all failures are repaired and there are no more failures, then all processes will eventually decide
AC-1: All processes that reach a decision reach the same one.
AC-2: A process cannot reverse its decision after it has reached one.
AC-3: The **Commit** decision can only be reached if all processes vote **Yes**.
AC-4: If there are no failures and all processes vote **Yes**, then the decision will be **Commit**.
AC-5: If all failures are repaired and there are no more failures, then all processes will eventually decide something.

**AC-1:**
- We do not require all processes to reach a decision.
- We do not even require all correct processes to reach a decision.

**AC-4:**
- Avoids triviality.
- Allows **Abort** even if all processes have voted **Yes**.

**Note:**
- A process that does not vote **Yes** can unilaterally **Abort**.
LIVENESS & UNCERTAINTY

- A process is in uncertainty if it has voted Yes but does not have sufficient information to Commit.
- While uncertain, a process cannot decide unilaterally.
- Uncertainty + communication failures = blocking.
LIVENESS & INDEPENDENT RECOVERY

- Suppose process $p$ fails while running Atomic Commit

- If, during recovery, $p$ can reach a decision without communicating with other processes, we say that $p$ can **independently recover**

- total failure (= all processes fail)
  - independent recovery
  __________________
  blocking
A FEW CHARACTER-BUILDING FACTS

Proposition 1
If communication failures or total failures are possible, then every AC protocol may cause processes to become blocked.

Proposition 2
No AC protocol can guarantee independent recovery of failed processes.
2-Phase Commit

Coordinator \( c \)

1. sends VOTE-REQ to all participants

Participant \( p_i \)

2. sends \( v_{\text{vote}_i} \) to Coordinator

\[
\text{if } v_{\text{vote}_i} = \text{No} \text{ then}
\]

\[
\text{decision}_i := \text{Abort}
\]

\[
\text{halt}
\]

3. if (all votes are Yes) then

\[
\text{decision}_c := \text{Commit}
\]

\[
\text{send Commit to all}
\]

else

\[
\text{decision}_c := \text{Abort}
\]

\[
\text{send Abort to all who voted Yes}
\]

\[
\text{halt}
\]

4. if received Commit then

\[
\text{decision}_i := \text{Commit}
\]

else

\[
\text{decision}_i := \text{Abort}
\]

\[
\text{halt}
\]
Notes on 2PC

- Satisfies AC-1 to AC-4
- But not AC-5 (at least “as is”)
  - A process may be waiting for a message that may never arrive
    - Use Timeout Actions
  - No guarantee that a recovered process will reach a decision consistent with that of other processes
    - Processes save protocol state in DT-Log

AC-5: If all failures are repaired and there are no more failures, then all processes will eventually decide
**Timeout actions**

**Coordinator c**

**Participant p_i**

**Step 2:** $p_i$ is waiting for VOTE-REQ from Coordinator.

Since it has not cast its vote yet, $p_i$ can decide **Abort** and halt.

**Step 3:** Coordinator is waiting for vote from participants.

Coordinator can decide **Abort**, send **Abort** to all participants who voted **Yes**, and halt.

**Step 4:** $p_i$ (who voted **Yes**) is waiting for **Commit** or **Abort**.

$p_i$ cannot decide: it must run a termination protocol.
Termination protocols

A. Wait for coordinator to recover
   - it always works, since the coordinator is never uncertain
   - may block recovering process unnecessarily

B. Ask other participants
Cooperative termination

- Coordinator appends list of participants to VOTE-REQ
- When an uncertain process $p$ times out, it sends a DECISION-REQ message to every other participant
- if $q$ has decided, it sends its decision to $p$, which acts accordingly
- if $q$ has not yet voted, it decides **Abort** and sends **Abort** to $p$
- What if $q$ is uncertain?
Logging actions

- When $c$ sends VOTE-REQ, it writes START-2PC to its DT Log
- When $p_i$ is ready to vote Yes,
  - $p_i$ writes Yes to DT Log, along with a list of participants
  - $p_i$ sends Yes to $c$
- When $p_i$ is ready to vote No, it writes Abort to its DT Log
- When $c$ is ready to Commit, it writes Commit to its DT Log
  - before sending Commit to participants
- When $c$ is ready to decide Abort, it writes Abort to its DT Log
- After $p_i$ receives a decision value, it writes it to its DT Log
\( p \) recovers

- When coordinator sends VOTE-REQ, it writes START-2PC to its DT Log
- When participant is ready to vote \textbf{Yes}, it writes \textbf{Yes} to DT Log, along with a list of participants
  When participant is ready to vote \textbf{No}, it writes \textbf{Abort} to DT Log
- When coordinator is ready to \textbf{Commit}, it writes \textbf{Commit} to DT Log \textbf{before} sending \textbf{Commit} to participants
  When coordinator is ready to decide \textbf{Abort}, it writes \textbf{Abort} to DT Log
- After participant receives a decision value, it writes it to DT Log

- if DT Log contains START-2PC, then \( p = c \)
  - if DT Log contains a decision value, decide accordingly
  - else, decide \textbf{Abort}

- otherwise, \( p \) is a participant
  - if DT Log contains a decision value, decide accordingly
  - else if it does not contain a \textbf{Yes} vote, decide \textbf{Abort}
  - else (\textbf{Yes} but no decision) run a termination protocol
2PC AND BLOCKING

- Blocking occurs whenever the progress of a process depends on the repairing of failures.
- No AC protocol is non-blocking in the presence of communication or total failures.
- But 2PC can block even with non-total failures and with no communication failures among operating processes!

Enter 3PC!
Blocking and uncertainty

Why does uncertainty lead to blocking?

An uncertain process does not know whether it can safely decide Commit or Abort, because some of the processes it cannot reach could have decided either

Non-blocking property
If any operational process is uncertain, then no process has decided Commit