The challenges of non-stable predicates

Consider a non-stable predicate $\Phi$ encoding, say, a safety property. We want to determine whether $\Phi$ holds for our program.

Suppose we apply $\Phi$ to $\Sigma^*$

$\Phi$ holding in $\Sigma^*$ does not preclude the possibility that our program violates safety!

Suppose now a different non-stable predicate $\Phi$. We want to determine whether $\Phi$ ever holds during a particular computation.

Suppose we apply $\Phi$ to $\Sigma^*$
The challenges of non-stable predicates

- Consider now a different non-stable predicate $\Phi$. We want to determine whether $\Phi$ ever holds during a particular computation.
- Suppose we apply $\Phi$ to $\Sigma^s$.
- $\Phi$ holding in $\Sigma^s$ does not imply that $\Phi$ ever held during the actual computation!

### Example

Detect whether the following predicates hold:

$\Phi(\Sigma^3, \Sigma^3) \quad \Phi(\Sigma^3, \Sigma^3)$

Assume that initially:

$x = 0; y = 10$

### Possibly

- If $\Sigma^s$ is $\Sigma^{31}$ or $\Sigma^{41}$, $x = y - 2$ is detected, but it may never have occurred.

### Possibly

- If $\Sigma^s$ is $\Sigma^{31}$ or $\Sigma^{41}$, $x = y - 2$ is detected, but it may never have occurred.
- Possibly ($\Psi$)
  - There exists a consistent observation of the computation $\mathcal{O}$ such that $\Phi$ holds in a global state of $\mathcal{O}$.
Definitely

We know that $x = y$ has occurred, but it may not be detected if tested before $\Sigma^{10}$ or after $\Sigma^{54}$.

Definitely

We know that $x = y$ has occurred, but it may not be detected if tested before $\Sigma^{10}$ or after $\Sigma^{54}$.

Definitely ($\Phi$)

For every consistent observation $O$ of the computation, there exists a global state of $O$ in which $\Phi$ holds.

Computing Possibly

Scan lattice, level after level

If $\Phi$ holds in one global state, then Possibly($\Phi$)
Computing Possibly

- Scan lattice, level after level
- If $\Phi$ holds in one global state, then $\text{Possibly}(\Phi)$

$\text{Possibly}(x = y - 2)$

Computing Definitely

- Scan lattice, level after level
- If $\Phi$ holds in one global state, then $\text{Possibly}(\Phi)$
Computing Definitely

Scan lattice, level after level

Given a level, only expand nodes that correspond to states for which \( \neg \Phi \)

If no such state, then \( \text{Definitely}(\Phi) \)

If reached last state \( \Sigma' \), and \( \Phi(\Sigma') \), then \( \neg \text{Definitely}(\Phi) \)

Definitely \( (x = y) \)

Building the lattice: collecting local states

To build the global states in the lattice, \( p_0 \) collects local states from each process.

\( p_0 \) keeps the set of local states received from \( p_i \) in a FIFO queue \( Q_i \)

Key questions:

1. when is it safe for \( p_0 \) to discard a local state \( \sigma_i^k \) of \( p_i \)?
2. Given level \( i \) of the lattice, how does one build level \( i + 1 \)?
Garbage-collecting local states

For each local state $\sigma_i^k$, we need to determine:

- $\Sigma_{min}(\sigma_i^k)$, the earliest consistent state that $\sigma_i^k$ can belong to
- $\Sigma_{max}(\sigma_i^k)$, the latest consistent state that $\sigma_i^k$ can belong to

Defining “earliest” and “latest” Consistent Global State

Defining “earliest” and “latest” Consistent Cut

Defining “earliest” and “latest” Consistent Global State

Defining “earliest” and “latest” Frontier
Computing $\Sigma_{\min}$

$\Sigma_{\min}(\sigma_{k_i}) = (\sigma_{c_1}, \sigma_{c_2}, \ldots, \sigma_{c_n}) : \forall j : c_j = VC(\sigma_{k_i})[j]$ 

Defining “earliest” and “latest”

Consistent Global State 
Consistent Cut 
Frontier 
Vector Clock

Computing $\Sigma_{\min}$

Label $\sigma^k_i$ with $VC(\sigma^k_i)$

$\Sigma_{\min}(\sigma^k_i) = (\sigma^k_1, \sigma^k_2, \ldots, \sigma^k_n) : \forall j : c_j = VC(\sigma^k_i)[j]$ 

$\Sigma_{\min}(\sigma^k_i)$ and $\sigma^k_i$ have the same vector clock!

Consistent Global State 
Consistent Cut 
Frontier 
Vector Clock

Defining “earliest” and “latest”

Associate a vector clock with each consistent global state

$\Sigma_{\min}(\sigma^k_i)$ is the consistent global state with the lowest vector clock that has $\sigma^k_i$ on its frontier

$\Sigma_{\max}(\sigma^k_i)$ is the one with the highest

Computing $\Sigma_{\max}$

$\Sigma_{\max}(\sigma^k_i) = (\sigma^k_1, \sigma^k_2, \ldots, \sigma^k_n)$:

$\forall j : VC(\sigma^k_j)[j] \leq VC(\sigma^k_i)[j]$ 

$\forall j : (\sigma^k_j = \sigma^k_i) \vee VC(\sigma^k_j+1)[j] > VC(\sigma^k_i)[j])$
Computing $\Sigma_{max}$

$\Sigma_{max}(\sigma^k_i) = (\sigma^i_1, \sigma^i_2, \ldots, \sigma^i_n) :$

$\forall j : VC(\sigma^i_j)[i] \leq VC(\sigma^k_i)[i]$

$\land((\sigma^i_j = \sigma^i_{j'}) \lor VC(\sigma^i_{j+1})[i] > VC(\sigma^k_i)[i]))$

set of local states one for each process, s.t.

all local states are pairwise consistent with $\sigma^k_i$

and they are the last such state

Assembling the levels

To build level $l$

- wait until each $Q_i$ contains a local state for whose vector clock:

  $\sum_{i=1}^{n} VC[i] \geq l$

To build level $l + 1$

- For each global state $\sum$ on level $l$, build

  $\sum_{i} + \sum_{i+1} + \sum_{i+2} + \ldots + \sum_{i+\ldots+n}$

- Using VC's, check whether these global states are consistent
Liveness properties

Property: a predicate that is evaluated over a run of the program
“nothing bad happens”
“every message that is received was previously sent”

A process that wishes to enter the critical section eventually does so if it does not hold in a prefix of a run, it does not mean it may not hold eventually

Properties

Property: a predicate that is evaluated over a run of the program
“every message that is received was previously sent”
“the program sends an average of 50 messages in a run”

Safety properties

“nothing bad happens”
o no more than k processes are simultaneously in the critical section
□ 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

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 $vote_i$:
  \[ vote_i \in \{Yes, No\} \]
- Each process $p_i$ has output value $decision_i$:
  \[ decision_i \in \{Commit, Abort\} \]
**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 will be Commit.
AC-5: If all failures are repaired and there are no more failures, then all processes will eventually decide.

**Comments**

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 reported and there are no more failures, then all processes will eventually decide.

**Liveness & Uncertainty**

A process is uncertain 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 AC.

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

Total failure (i.e. 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

Notes on 2PC
- Satisfies AC-1 to AC-4
- But not AC-5 (at least “as is”)
  i. A process may be waiting for a message that may never arrive
     Use Timeout Actions
  ii. No guarantee that a recovered process will reach a decision consistent with that of other processes
     Processes save protocol state in DT-Log

2-Phase Commit
I. Coordinator $c$ sends VOTE-REQ to all participants.
II. When participant $p_i$ receives a VOTE-REQ, it responds by sending a vote to the coordinator.
   - if \( \text{vote}_i = \text{NO} \), then \( \text{decide}_i := \text{ABORT} \) and \( p_i \) halts.
III. $c$ collects votes from all.
   - if all votes are yes, then \( \text{decide}_c := \text{COMMIT} \), sends COMMIT to all
     - else \( \text{decide}_c := \text{ABORT} \), sends ABORT to all who voted YES
     \( c \) halts.
IV. if participant $p_i$ receives COMMIT then \( \text{decide}_i := \text{COMMIT} \)
    - else \( \text{decide}_i := \text{ABORT} \)
    \( p_i \) halts.

Timeout actions
Processes are waiting on steps 2, 3, and 4
- Step 2 \( p_i \) is waiting for VOTE-REQ from coordinator
- Step 3 Coordinator is waiting for vote from participants
- Step 4 \( p_i \) (who voted YES) is waiting for COMMIT or ABORT
Timeout actions

Processes are waiting on steps 2, 3, and 4

<table>
<thead>
<tr>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\pi_i) is waiting for VOTE-REQ from coordinator. Since it has not cast its vote yet, (\pi_i) can decide ABORT and halt.</td>
<td>Coordinator is waiting for vote from participants. Coordinator can decide ABORT, send ABORT to all participants which voted YES, and halt.</td>
<td>(\pi_i) (who voted YES) is waiting for COMMIT or ABORT. (\pi_i) cannot decide: it must run a termination protocol.</td>
</tr>
</tbody>
</table>

Termination protocols

I. Wait for coordinator to recover
   - It always works, since the coordinator is never uncertain
   - May block recovering process unnecessarily

II. Ask other participants
**Cooperative Termination**

- $c$ appends list of participants to VOTE-REQ
- when an uncertain process $p$ times out, it sends a DECISION-REQ message to every other participant $q$
- if $q$ has decided, then it sends its decision value to $p$, which decides accordingly
- if $q$ has not yet voted, then it decides ABORT, and sends ABORT to $p$
- what if $q$ is uncertain?

**Logging actions**

1. When $c$ sends VOTE-REQ, it writes START-2PC to its DT Log
2. When $p_i$ is ready to vote YES, 
   i. $p_i$ writes YES to DT Log  
   ii. $p_i$ sends YES to $c$ ($p_i$ writes also list of participants) 
3. When $p_i$ is ready to vote NO, it writes ABORT to DT Log  
4. When $c$ is ready to decide COMMIT, it writes COMMIT to DT Log before sending COMMIT to participants  
5. When $c$ is ready to decide ABORT, it writes ABORT to DT Log  
6. After $p_i$ receives decision value, it writes it to DT Log

**$p$ recovers**

1. When coordinator sends VOTE-REQ, it writes START-2PC to its DT Log  
2. When participant is ready to vote YES, it writes YES to DT Log before sending YES to coordinator (writes also list of participants)  
3. When participant is ready to vote NO, it writes ABORT to DT Log  
4. After participant receives decision value, it writes it to DT Log

**$p$ recovers**

1. When coordinator sends VOTE-REQ, it writes START-2PC to its DT Log  
2. When participant is ready to vote YES, it writes YES to DT Log before sending YES to coordinator (writes also list of participants)  
3. When participant is ready to vote NO, it writes ABORT to DT Log  
4. After participant receives decision value, it writes it to DT Log

- if DT Log contains START-2PC, then $p = c$:
  - if DT Log contains a decision value, then decide accordingly
  - else decide ABORT
3-Phase Commit

Two approaches:

1. Focus only on site failures
   - Non-blocking, unless all sites fail
   - Communication failures can cause blocking

2. Tolerate both site and communication failures
   - Partial failures can still cause blocking, but less often than in 2PC

$p$ recovers

- If DT Log contains START-2PC, then $p = c$
- If DT Log contains a decision value, then decide accordingly
- Else decide ABORT
- Otherwise, $p$ is a participant:
  - If DT Log contains a decision value, then decide accordingly
  - Else, if it does not contain a Yes vote, decide ABORT
  - Else (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 no communication failures among operating processes!

Blocking and uncertainty

Why does uncertainty lead to blocking?
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
2PC Revisited

In state PC
a process knows that it will commit unless it fails

3PC: The Protocol

Dale Skeen (1982)

I. \( c \) sends VOTE-REQ to all participants.
II. When \( p_i \) receives a VOTE-REQ, it responds by sending a vote to \( c \)
    if \( \text{vote}_i = \text{No} \), then \( \text{decide}_i := \text{ABORT} \) and \( p_i \) halts.
III. \( c \) collects votes from all.
     if all votes are Yes, then \( c \) sends PRECOMMIT to all
     else \( \text{decide}_i := \text{ABORT} \); sends ABORT to all who voted Yes
     \( c \) halts
IV. if \( p_i \) receives PRECOMMIT then it sends ACK to \( c \)
V. \( c \) collects ACKs from all.
   When all ACKs have been received, \( \text{decide}_i := \text{COMMIT} \); \( c \) sends COMMIT to all
VI. When \( p_i \) receives COMMIT, \( p_i \) sets \( \text{decide}_i := \text{COMMIT} \) and halts.
Wait a minute!

Messages are known to the receiver before they are sent... so, why are they sent?

1. c sends VOTE-REQ to all participants
2. When participant \( p_i \) receives a VOTE-REQ, it responds by sending a vote to c.
   - If vote = No, then decide = ABORT and \( p_i \) halts
3. c collects vote from all
   - If all votes are Yes, then c sends PRECOMMIT to all
   - Else, decide = ABORT, c sends ABORT to all who voted Yes
   - c halts
4. If \( p_i \) receives PRECOMMIT then it sends ACK to c
5. c collects ACKs from all
   - When all ACKs have been received, decide = COMMIT
   - c sends COMMIT to all
6. When \( p_i \) receives COMMIT, \( p_i \) sets decide = COMMIT
   - \( p_i \) halts

Wait a minute!

Messages are known to the receiver before they are sent... so, why are they sent?

1. c sends VOTE-REQ to all participants
2. When participant \( p_i \) receives a VOTE-REQ, it responds by sending a vote to c.
   - If vote = No, then decide = ABORT and \( p_i \) halts
3. c collects vote from all
   - If all votes are Yes, then c sends PRECOMMIT to all
   - Else, decide = ABORT, c sends ABORT to all who voted Yes
   - c halts
4. If \( p_i \) receives PRECOMMIT then it sends ACK to c
5. c collects ACKs from all
   - When all ACKs have been received, decide = COMMIT
   - c sends COMMIT to all
6. When \( p_i \) receives COMMIT, \( p_i \) sets decide = COMMIT
   - \( p_i \) halts

They inform the recipient of the protocol's progress!

When \( c \) receives ACK from \( p_i \), it knows \( p_i \) is not uncertain

When \( p_i \) receives COMMIT, it knows no participant is uncertain, so it can commit

Timeout Actions

Processes are waiting on steps 2, 3, 4, 5, and 6

- Step 2 \( p_i \) is waiting for VOTE-REQ from coordinator
- Step 3 Coordinator is waiting for vote from participants
- Step 4 \( p_i \) waits for PRECOMMIT
- Step 5 Coordinator waits for ACKs
- Step 6 \( p_i \) waits for COMMIT

Timeout Actions

Processes are waiting on steps 2, 3, 4, 5, and 6

- Step 2 \( p_i \) is waiting for VOTE-REQ from coordinator
- Step 3 Coordinator is waiting for vote from participants
- Step 4 \( p_i \) waits for PRECOMMIT
- Step 5 Coordinator waits for ACKs
- Step 6 \( p_i \) waits for COMMIT

Exactly as in 2PC
Timeout Actions

Processes are waiting on steps 2, 3, 4, 5, and 6

- **Step 2**: $p_i$ is waiting for VOTE-REQ from coordinator
  - Exactly as in 2PC

- **Step 4**: $p_i$ waits for PRECOMMIT
  - Run some Termination protocol

- **Step 6**: $p_i$ waits for COMMIT

- **Step 3**: Coordinator is waiting for vote from participants
  - Exactly as in 2PC

- **Step 5**: Coordinator waits for ACKs
  - Coordinator sends COMMIT

Participant knows what is going to receive...
Process states

At any time while running 3 PC, each participant can be in exactly one of these 4 states:

- **Aborted**: Not voted, voted NO, received ABORT
- **Uncertain**: Voted YES, not received PRECOMMIT
- **Committable**: Received PRECOMMIT, not COMMIT
- **Committed**: Received COMMIT

Not all states are compatible

<table>
<thead>
<tr>
<th></th>
<th>Aborted</th>
<th>Uncertain</th>
<th>Committable</th>
<th>Committed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aborted</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Uncertain</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Committable</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Committed</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Termination protocol

- When 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 termination rule.

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 ACKs 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
  - ask 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)

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 $p$ maintains set $UP_p$ of all processes that $p$ 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 new coordinator
- If $q = p$, then $p$ is new coordinator
- Otherwise, $p$ sends UR-ELECTED to $q$

A few observations

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

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 \( U_P \) every \( q' < q \)

What if \( p' \) receives a STATE-REQ from \( c \) after it has changed the coordinator to \( q' \)?

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 \( U_P \) every \( q' < q \)

What if \( p' \) receives a STATE-REQ from \( c \) after it has changed the coordinator to \( q' \)?

A few observations

Suppose \( p \) is the first process to recover, and that \( p \) is uncertain

Can \( p \) decide ABORT?

Some processes could have decided COMMIT after \( p \) crashed!

\( p \) is blocked until some \( q \) recovers s.t. either

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

Total failure
Total failure

- Suppose $p$ is the first process to recover, and that $p$ is uncertain
- Can $p$ decide ABORT?
  - Some processes could have decided COMMIT after $p$ crashed!

Determining the last process to fail

- Suppose a set $R$ of processes has recovered
- Does $R$ contain the last process to fail?

Total failure

- Suppose $p$ is the first process to recover, and that $p$ is uncertain
- Can $p$ decide ABORT?
  - Some processes could have decided COMMIT after $p$ crashed!
- $p$ is blocked until some $q$ recovers s.t. 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 $\bigcap_{p \in R} UP_p$
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

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