# What is a Distributed System?

- multiple computers
  - CPU+memory+stable storage+I/O
- interconnected
  - …or not very interesting
- sharing state
  - maintain global invariants through coordinate operations

# All the usual problems...

- coordination
- synchronization
- abstractions
- efficiency

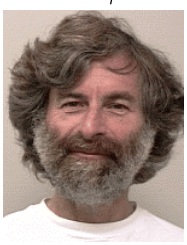
# …only harder!

- independent failures
- unreliable communication
- insecure communication
- costly communication

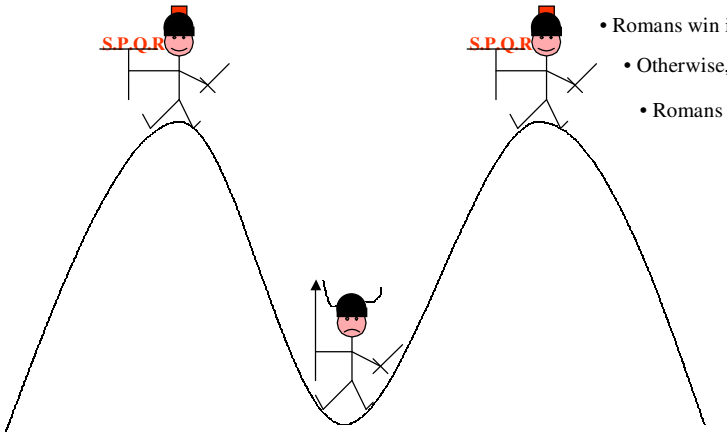
# To put it differently…

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

**Leslie Lamport**

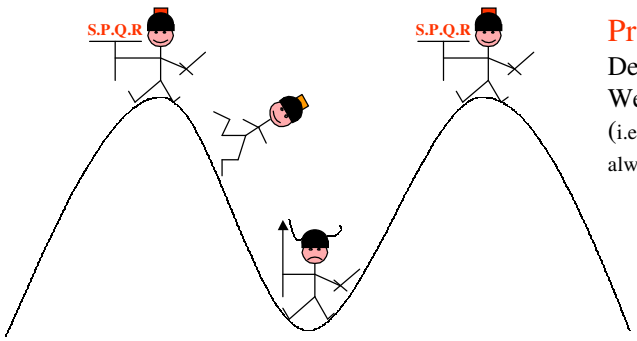


# Two Generals' Problem




- Romans win if attack simultaneously
- Otherwise, Barbarians win
- Romans must coordinate attack

# Two Generals' Problem




Problem:
Design a protocol to save Western Civilization
(i.e. one that ensures that Romans always attack simultaneously)

- only communication is by messenger
- messengers must sneak through the valley
- they don't always make it

## Two Generals' Problem

**Claim:** There is no non-trivial protocol that guarantees that the Romans will always attack simultaneously

**Proof:** Consider the shortest such protocol.

- consider last message
- protocol must work if last message never arrives
- so don't send it
- but now we have a shorter protocol!

**Fundamental Limitation:** Solution needs

- either an unbounded number of messages
- or guaranteed message delivery

Otherwise, possibility no attack takes place!

## 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, but 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 $vote_i$:
  $vote_i \in \{\text{Yes, No}\}$
- Each process $p_i$ has an output value $decision_i$:
  $decision_i \in \{\text{Commit, Abort}\}$

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

- AC1:
  - We do not require that all processes reach a decision
  - We do not require that all *correct* processes reach a decision (with link failures, this is equivalent to the "2 Generals Problem)
- AC4:
  - avoids triviality
  - allows Abort even if all processes have voted yes

NOTE: A process that does not vote Yes can unilaterally abort

# To be or not to be…

*Uncertainty:* A process is uncertain if it has voted Yes but does not have sufficient information to commit

What if a process becomes isolated while uncertain?

Uncertainty + Failures can cause *blocking*:

a process is blocked if it must wait for the recovery of another process before deciding

## Independent Recovery?

*Independent Recovery*: the ability of a recovering process to reach a decision without communicating with other processes.

No independent recovery + total failures ⇒ 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

Nico Gazardo, IBM for the Italian Social Security System (1971)
Rediscovered and published: Lampson, Sturgis (1976)

1. Coordinator $c$ sends VOTE-REQ to all participants.
2. When participant $p_i$ receives a VOTE-REQ, it responds by sending a vote to the coordinator.
   **if** $vote_i$ = NO, **then** $decide_i$ := ABORT and $p_i$ halts.
3. Coordinator $c$ collects votes from all.
   **if** all votes are yes, **then** $decide_c$ := COMMIT; **sends** COMMIT **to** all
   **else** $decide_c$ := ABORT; **sends** ABORT **to** all participants who voted YES
   $c$ halts
4. **if** participant $p_i$ receives COMMIT **then** $decide_i$ := COMMIT
   **else** $decide_i$ := ABORT
   $p_i$ halts.

# Notes on 2PC

- Satisfies AC-1 to AC-4
- But not AC-5…
  - 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
    - Allow processes to write data to their DT-Log

# Timeout Actions

Processes are waiting at the beginning of steps 2,3, and 4

**Step 2** A participant is waiting for VOTE-REQ from coordinator

Since this happens before vote, participant can decide ABORT and halt.

**Step 3** Coordinator is waiting for vote form participants

Coordinator can decide ABORT, send ABORT to all participants which voted YES, and halt.

**Step 4** Participant (who has voted YES) waits for COMMIT or ABORT

Participant cannot decide: it must run a *Termination Protocol*

# Termination Protocols

| Simple TP | Cooperative TP |
|---|---|
| Wait for coordinator to recover | Ask other participants |
| + Works, because coordinator is never uncertain<br>– Process may be blocked unnecessarily<br>• Use Cooperative TP! | Need to know identity of other participants<br>• coordinator attaches list of participants to VOTE-REQ |

# Cooperative Termination Protocol

- When an uncertain process times out, it sends a DECISION-REQ message to all other participants
- Let *p* be the *initiator*, *q* be a *responder*

1. **if** q has decided, **then**
   *q* sends its decision value to *p*
   *p* decides accordingly
2. **if** *q* has not yet voted **then**
   $decide_q$ := ABORT
   *q* sends ABORT to *p*
3. **if** *q* is uncertain **then**
   *q* cannot help *p*

# Logging Actions

Specify which information should be written to DT Log

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

# What happens during recovery?

1. When coordinator sends VOTE-REQ, it writes START-2PC to its DT Log.
2. When participant is ready to vote YES, writes YES to DT Log before sending YES to coordinator (writes also list of participants).
   When participant is ready to vote NO, it writes ABORT to DT Log .
3. When coordinator is ready to decide COMMIT, it writes COMMIT to DT Log before sending COMMIT to participants.
   When coordinator is ready to decide ABORT, 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 process is coordinator:
  - if it also contains ABORT or COMMIT, no problem
  - if it contains nothing else, ABORT
- otherwise, process is participant:
  - if DT Log contains COMMIT or ABORT, no problem
  - if it contains no YES vote, unilaterally decide ABORT
  - if it contains YES but no decision value, use termination protocol