Distributed Systems: Replicated State Machines

Where we are:
- Last week:
  - Consistency models
    - Linearizability
    - Sequential consistency
    - Strict Serializability
  - Ended by talking about ways to build fault tolerant consistent ADTs
- Today: RSMs: Building fault tolerant systems
- Next week/laboratory: Putting it into practice

Reordering things a bit compared to last semester, going to talk about a few ideas today that we will revisit later.

What we want: fault tolerance

Informally, users (and the nature of the system due to
any changes in a system due to failures

Generally requires assumptions about

\[
\text{FAILURE MODEL} \begin{cases} 
- \text{\# of processes that have failed} \\
- \text{Types of failures}
\end{cases}
\]

Types of Failures
- Fail stop

- Fail Recover

- Byzantine

For most of the class, going to assume Fail-stop (see Chapter 10, 13, 14, and 15).
What we want: fault tolerance for an ADI/service

→ Assuming failures fit within model, want to ensure ADI behaves as if no failures

Our goal: A general framework for building fault tolerant ADIs:

- Replicated State Machines

State Machine

```
State
```

State & output determined by sequence of commands

Observe, definition assumes total order of commands

- Queue

- R. Counter

  get():
  
  release count

  maybe-inc():
  
  if rand() % 2 == 0:
count ++

- R. Count -2

\begin{verbatim}
get()
    \textbf{new} count
    maybe_inc(n)
    \textbf{if} n \% 2 == 0
        count ++
\end{verbatim}

\underline{Deterministic WRT Commands}

\rightarrow \textbf{Common Assumption For Fault Tolerance}

\underline{Replicated State Machines}

\begin{itemize}
    \item C1
    \item C2
    \item C3
\end{itemize}

Agreement & Ordering: All non-faulty replicas receive the same commands in the same order. Eventually
Core requirement: Agree on an ordered log

Slot/Log entry/Index

P₀

P₁

P₂

Look at one slot [i]

Requirement:

(IC1) Agreement. If a correct process \textbf{Decides}

slot \(i\) contains command \(c\), eventually
all correct processes \textbf{Decide} \(i\)
contains \(c\).

Validity. If a correct process \textbf{decides} slot \(i\) contains \(c\), then some client
must have issued \(c\).
General Problem: AGREEMENT/CONSENSUS

- \( l_0 \) → \( l_1 \) → \( l_2 \)
- \( P_0 \) → \( P_1 \) → \( P_2 \)

- \( P_0 \) → \( P_1 \) → \( P_2 \) \( \rightarrow \) eventually \( P_0 \) \( \downarrow \) \( P_1 \) \( \downarrow \) \( P_2 \) \( \downarrow \)

```
Fischer, Lynch, Paterson 1985 (FLP)/ Impossibility of Consensus

No deterministic protocol that works in the asynchronous setting can solve consensus
\( \rightarrow \) tolerate 1 (or more) failures.

Show that any safe algorithm might not terminate.

Circumventing This

- Relax asynchrony assumptions:
  - Partial Synchrony (Dwork & Lynch '88)
  - OR
```
Synchronous

- Randomized algorithms
  Note: Using randomness for the algorithm, not in the state machine

- Failure detectors/Oracles
  (Add components to assume it away)

Partial Synchrony

Implications: Safety, Agreement & Validity

Liveness: Termination

Agreement from the paper

IC1: All nonfaulty processors agree on the
IC2: If the transmitter is nonfaulty, then all nonfaulty processors use its value as the one on which they agree.

- Causal Ordering
- Other approaches

  \[ \rightarrow \] Decouple replication & ordering

- Own (as in this classes) toolkit for Building RSM

  1. Assume partial synchrony

  2. Use the same protocol for ordering

    - Raft (next class, lab)

But, important to remember other options are available.