Distributed Systems

- Are everywhere
  - share resources
  - communicate
  - increase performance (replication, speed, fault tolerance)

- Are characterized by
  - independent activities (concurrency)
  - loosely coupled parallelism (heterogeneity)
  - inherent uncertainty
  - need for synchronization
Example I: Coordinated Clubbing

Coordinate meeting in a club by texting

– Only one club & one time to go

It is absolutely bad if only one party shows up

Theorem: If message delivery is not guaranteed, then coordinated clubbing cannot be achieved

Example I: Coordinated Clubbing

Ping-pong execution w/o message loss

k smallest number of messages s.t.

some participant, e.g., \( p_0 \), decides go

Agreement \( \Rightarrow p_1 \) also decides go

Remove last message, from \( p_1 \) to \( p_0 \)

\( p_1 \) still decides go

\( \Rightarrow \) Execution with k-1 messages!

Theorem: If message delivery is not guaranteed, then coordinated clubbing cannot be achieved
Uncertainty in Distributed Systems

- differing process speeds
- varying communication delays
- (partial) failures

To ensure that a system is still correct
- identify fundamental problems and state them precisely
- design algorithms to solve these problems and prove the correctness of these algorithms
- analyze their complexity (e.g., time, space, messages)
- prove impossibility results and lower bounds

Applications of Distributed Computing

Classic problems come from:
- multi-threaded operating systems
- communication networks
- multicore processors
- replicated servers
- (distributed) database systems
- software fault-tolerance
Example II: Online Accounts

A (single) account on a multi-processor

Two operations:

**Deposit** (one $), **Withdraw** (one $)

Simple implementation by reads and writes does not work

```
lval = read(balance)
lval++
write(balance,lval)
```

process A

```
lval = read(balance)
lval++
write(balance,lval)
```

process B

```
lval = read(balance)
lval++
write(balance,lval)
```

One $ added despite two deposits

Need stronger primitives

```
lval = read(balance)
lval++
write(balance,lval)
```
**Example II: Single-Owner Account?**

Only process $p_0$ can withdraw (one $)

Other processes just deposit (one $)

\[
\text{Balance()} \\
\text{lval}[1..n] = \text{read}(M[1..n]) \\
\text{return } \sum_{j=1}^{n} \text{lval}[j]
\]

\[
\text{Withdraw}_0() \\
\text{if } \text{balance()} > 0 \\
\text{lval}_0 = \text{read}(M[0]) \\
\text{lval}_0-- \\
\text{write}(M[0], \text{lval}_0)
\]

\[
\text{Deposit}_i() \\
\text{lval}_i = \text{read}(M[i]) \\
\text{lval}_i++ \\
\text{write}(M[i], \text{lval}_i)
\]

---

**Course Overview: Models**

Two basic communication models:

- message passing
- shared memory

and two basic timing models:

- synchronous
- asynchronous

<table>
<thead>
<tr>
<th></th>
<th>Message passing</th>
<th>Shared memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>synchronous</td>
<td></td>
<td>PRAM</td>
</tr>
<tr>
<td>asynchronous</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Topics Covered

- mutual exclusion
- fault-tolerant consensus
- concurrent data structures
- causality and time

Failure models:
- crash: faulty process just stops.
- Byzantine (arbitrary): conservative assumption, fits when failure model is unknown or malicious.