Today

☐ Finishing up vector clocks and the real RPC deadlock protocol.
☐ An approach to distributed mutual exclusion.
☐ Basics of quorums and coteries
☐ Weighted voting and other ways to construct a quorum
☐ Consistency
☐ Fault tolerant registers with quorums
Locally stable properties

- A *locally stable* property is a stable property in which a process involved in the stable property will stop executing relevant events.
  - Deadlock is locally stable.
  - Lossy token passing, *there are no more than 2 tokens* is not locally stable.
Basic protocol

- Periodically a process $i$ collects $\{B_1, B_2, \ldots, B_n\}$ and determines whether there exists a \textit{maximal consistent subcut} in which $P$ holds in the \textit{latest consistent subcut}.

\[
\{\forall i \ B_i.s: \forall j: B_i.C[i] \geq B_j.C[i]\}\]

- Safety: if $P$ holds in this consistent subcut, it holds now.
- Liveness: from $P$ being locally stable.
Centralized protocol

- When $i$ executes a relevant event, it records into a buffer $B_i$ its state $B_i.s$ and its vector clock $B_i.C$.
- Periodically a process $i$ collects $\{B_1, B_2, \ldots, B_n\}$ and extracts the subset $\{B_i.s: \forall j: B_i.C[i] \geq B_j.C[i]\}$.
- If $P$ holds on this subset, then detect $P$. 
Decentralization

Define a token \( K = \langle D_1, D_2, ..., D_n \rangle \) where \( D_i \) is either a pair \( \langle B_i.s, B_i.C[i] \rangle \) or \( \perp \).

- A process \( i \) generates an empty token \( \langle \perp, \perp, ..., \perp \rangle \), sets \( K.D_i \) to \( \langle B_i.s, B_i.C[i] \rangle \), and passes it to another process.

- When \( j \) receives \( K \):
  - set \( K.D_j \) to \( \langle B_j.s, B_j.C[j] \rangle \)
  - for all \( D_k: D_k \neq \perp \land D_k.C[k] < B_j.C[k] \), set \( K.D_k \) to \( \perp \).
  - If \( P \) holds on non-\( \perp \) values of \( K \), then detect condition else forward to some \( p_m: D_m = \perp \)
    - or, discard it.
RPC Deadlock

State:

- \( rs_i[i \in 1..n] \): number of RPC reply \( i \) sent to \( j \)
- \( rr_i[i \in 1..n] \): number of RPC reply \( i \) received from \( j \)
- \( wf_i \): process \( i \) (if any) is waiting on.

Relevant events are those that change these variables.

- \( i \) waits-for \( j \) when \((wf_i = j) \land (rs_j[i] = rr_i[j])\)

deadlock iff cycle in waits-for graph
RPC Deadlock: Protocol I

When \(wf_i = j\) for unexpected time, \(i\):

- creates new token \(K\)
- sets \(K.D_i\) to \(\langle rr_i[j], B_i.C[i] \rangle\)
- sends \(K\) to \(j\)
  - Note we send only two integers!
    - First to determine if \(i\) waits on \(j\).
    - Second as part of the protocol.
- Still maintaining vector clocks that count relevant events, buffering relevant event state, etc.
RPC Deadlock: Protocol II

When $i$ receives a token $K$ from $j$:

- **if** $K.D_i = \bot$
  - **if** $((w_f_i \neq \emptyset) \land (K.D_j.rr_j[i] = B_i.rs_i[j])$
    - **and** $(\forall k: K.D_k \neq \bot: K.D_k.C[k] \geq B_i.C[k]))$
    - **then** $K.D_i = \langle B_i.rr_i[B_i.wf_j], B_i.C[i] \rangle$;
      pass $K$ to $B_i.wf_j$;
  - **else** drop $K$;
- **else**
  - **if** $(K.D_j.rr_j[i] = B_i.rs_i[j])$ then *detect deadlock*;
  - **else** drop $K$;
RPC Deadlock: Protocol II

When $i$ receives a token $K$ from $j$: 

**if** $K.D_i = \bot$

**if** $((wf_i \neq \emptyset) \land (K.D_j.rr_j[i] = B_i.rs_i[j])$

$\land (\forall k: K.D_k \neq \bot: K.D_k.C[k] \geq B_i.C[k]))$

**then**

$K.D_i = \langle B_i.rr_i[B_i.wf_j], B_i.C[i] \rangle$;

pass $K$ to $B_i.wf_j$;

**else** drop $K$;

**else**

**if** $(K.D_j.rr_j[i] = B_i.rs_i[j])$ then *detect deadlock*;

**else** drop $K$;
Why compare timestamps?

- If $K.D_j.rr_j[i] = B_i.rs_i[j]$ then $j$ hasn’t subsequently executed a relevant event.
- Since $j$ hasn’t executed a relevant event, neither has $a$.
- ... and so on along the loop.
  - Don’t need vector clocks.
  - Don’t need separate event buffer.
RPC Deadlock: final protocol

When $\text{wf}_i \neq \emptyset$ for unexpected time

\[ i \text{ sends } (rr_i[\text{wf}_i], i) \text{ to } \text{wf}_i. \]

When $i$ receives $(s, a)$ from $j$:

\begin{align*}
\text{if } (a &\neq i) \\
\text{if } ((\text{wf}_j \neq \emptyset) \land (s = rs_i[j])) &\text{ then send } (rr_i[\text{wf}_i], a) \text{ to } \text{wf}_i; \\
\text{else} &\text{ if } (s = rs_i[j]) \text{ then detect deadlock;}
\end{align*}

Recap

Basics of distributed computing.
- causality
- consistent cuts
- snapshot protocols
- clock condition and logical clocks
- strong clock condition and vector clocks
- the significance of causal delivery order
- RPC deadlock detection
- the value of protocol derivation
Mutual Exclusion (Maekawa)

A more-theoretical-than-practical problem...

- \( N \) processes share a single resource that requires mutually exclusive access
  - Reliable FIFO channels, full connectivity, no process failures
- To obtain mutual exclusion, a process \( i \) asks permission from a quorum: a subset \( Q_i \) of the processes.
  - All processes in \( Q_i \) must grant access for \( i \) to have access
  - For safety, \( \forall 1 \leq i, j \leq N: Q_i \cap Q_j \neq \emptyset \).
  - For performance, \( i \in Q_i \).
  - Minimum quorum size is \( \lceil \sqrt{N} \rceil \)

Maekawa’s Algorithm I

- Requesting mutual exclusion
  - A process $i$ requests mutex by sending a *request* message to each process in $Q_i$.
  - When process $j$ receives a *request* from $i$ for mutex:
    - If it has not yet handed out its vote, $j$ sends *reply* to $i$.
    - Otherwise, $j$ queues *request* from $i$.
  - Process $i$ obtains mutex when it receives a *reply* from all processes in $Q_i$. 
Maekawa’s Algorithm II

- Releasing mutual exclusion
  - When process $i$ releases mutex, it sends a *release* message to each process in $Q_i$.
  - When process $j$ receives *release*, it sends *reply* to a queued request.
Maekawa’s Algorithm III

Deadlock is possible

\[ Q_a = \{a, b, c\} \quad Q_b = \{b, c, d\} \]
\[ Q_c = \{c, d, e\} \quad Q_d = \{a, d, e\} \]
\[ Q_e = \{a, b, e\} \]

- Processes c, d and e simultaneously request mutex
- a and d vote for d
- b and e vote for e
- c votes for c
Maekawa’s Algorithm IV

Avoiding deadlock:

- Attach a *timestamp* to each message.
  - Can be Lamport clock with process ID to break ties.

- A process *recalls its vote* if it granted out of timestamp order
  - if \( j \) receives a request from \( i \) with higher timestamp than the request granted permission, \( j \) sends *fail* to \( i \).
  - If \( j \) receives a request from \( k \) when it has already voted for a request from \( i \) with a higher timestamp, \( j \) sends *inquire* to \( i \).
  - When \( i \) receives *inquire* it replies with *yield* if it did not succeed getting *votes* from all in \( Q_i \).

- When process gets *release*, it sends *reply* to the queued request with the lowest timestamp.
Maekawa’s Algorithm V

- Maekawa was interested in minimizing the number of messages to obtain mutex.
- But, a similar approach can be used to tolerate failures.
  - Quorum selection issues.
  - Protocol issues.
Majority Quorums

A quorum is a *majority of the processes*
Quorums by assigning votes

A quorum is a *subset of processes with a majority of the votes*
Coteries

A coterie $S$ is a set of quorums that satisfy:

- $Q \in S$ implies $Q \neq \emptyset$ and $Q \subseteq N$.
- $\forall Q_i, Q_j \in S: Q_i \cap Q_j \neq \emptyset$
- $\forall Q_i, Q_j \in S: Q_i \nsubseteq Q_j$


Domination

Let $N$ be $\{a, b, c, d\}$.

$S_1 = \{\{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{b, c, d\}\}$

$S_2 = \{\{a, b\}, \{a, c\}, \{a, d\}, \{b, c, d\}\}$

Note that for each $Q_1 \in S_1 \exists Q_2 \in S_2: Q_2 \subseteq Q_1$

$S_2$ dominates $S_1$. 
Coteries that have no vote assignments

Coterie over seven processes \{a, ..., g\}: \{a, b\},
\{a, c, d\}, \{a, c, e\}, \{a, d, f\}, \{a, e, f\}, \{b, c, f\},
\{b, d, e\}

\[ v(a) + v(b) \geq \text{maj} \]
\[ v(a) + v(c) + v(d) \geq \text{maj} \]

\[ v(b) + v(c) + v(f) \geq \text{maj} \]
\{b, c, f\} is a quorum

\[ v(a) + v(c) + v(f) < \text{maj} \]
\{a, c, f\} is not a quorum

\[ v(b) > v(a) \]

\[ v(a) + v(c) + v(d) \geq \text{maj} \]
\{a, c, d\} is a quorum

\[ v(b) + v(c) + v(d) \geq \text{maj} \]
\{b, c, d\} is a quorum
Maekawa grid quorums (dominated)
Maekawa grid quorums (nondominated)
Better grid quorums
B-Grid Quorums

d columns, h bands,
r rows/band
Common metrics

- **Load** $L(S)$
  - An access strategy $W$ gives the probability that a given quorum $Q \in S$ is accessed.
  - $\ell_w(p)$: the load *induced on* $p$ *by* $W$.
  - $L_w(S)$: *load induced by* $W$: maximal load induced by $W$ on any server.
  - $L(S)$: *system load*: minimum over $W$ of $L_w(S)$

- **Resilience** $R(S)$: largest $f$: for all subsets $F$: $|F| = f$, there is at least one quorum that does not intersect $F$.

- **Failure probability** $F_p(S)$: probability that at least one server of every quorum fails.
## Comparison

<table>
<thead>
<tr>
<th>$S$</th>
<th>$L(S)$</th>
<th>$R(S)$</th>
<th>$F_p(S)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Singleton</td>
<td>1</td>
<td>0</td>
<td>$p$</td>
</tr>
<tr>
<td>Majority</td>
<td>$\frac{1}{2}$</td>
<td>$\lfloor (n-1)/2 \rfloor$</td>
<td>$e^{-\Omega(n)}$</td>
</tr>
<tr>
<td>Grid</td>
<td>$O\left(1/\sqrt{n}\right)$</td>
<td>$\sqrt{n} - 1$</td>
<td>$\approx 1$ $^*$</td>
</tr>
<tr>
<td>B-Grid $^{**}$</td>
<td>$O\left(1/\sqrt{n}\right)$</td>
<td>$O\left(\sqrt{n}\right)$</td>
<td>$O(e^{n^{1/4}/2})$</td>
</tr>
</tbody>
</table>

* for large $n$.  

$^{**}$ for $d^2 = n$, $r = \ln d$, and $0 \leq p \leq 1/3$
Weighted voting to make reading faster

The quorum used for a write needs to intersect any quorum used for a read or used for a write.

The quorums chosen by two read operations don't need to intersect.

... basis of weighted voting: Given total votes in all, choose read quorum size $rv$ and write quorum size $wv$:

$$2 \times wv \geq total \text{ and } rv + wv \geq total$$

Weighted voting: Example

\[ n = 7 \]
\[ rv = 4 \]
\[ wv = 4 \]
Weighted voting: Example

\[ n = 7 \]
\[ rv = 2 \]
\[ wv = 6 \]

- Sites can crash
- Processes can crash

- Majority
  - Quorum: 5 processes
  - In some step, no quorum can be formed

- Using $S_p$ as quorums
  - In every step, at least one quorum can be formed
Shared register semantics

- Shared data in a *data store* (registers, file system, database).
- Clients access data store through *read* and *write* operations.
  - These operations take time and so can overlapp.
- Consistency semantics: states what clients can expect to obtain when accessing the data store.

Lamport’s model

- **Safe**: a read not concurrent with any write returns the most recently written value.

- **Regular**: safe and a read overlapping a write obtains either the old or the new value.

- **Atomic**: reads and writes are totally ordered so that values returned by reads are the same as if the operations had been performed with no overlapping.
Lamport’s model

- **Safe**: a read not concurrent with any write returns the most recently written value.
- **Regular**: *safe* and a read overlapping a write obtains either the old or the new value.
- **Atomic**: reads and writes are totally ordered so that values returned by reads are the same as if the operations had been performed with no overlapping.

\[
\begin{align*}
\text{r}_1 & \quad \text{r}_2 & \quad \text{r}_3 \\
\text{w}(5) & \quad & \text{w}(6)
\end{align*}
\]
Lamport’s model

- **Safe**: a read not concurrent with any write returns the most recently written value.

- **Regular**: safe and a read overlapping a write obtains either the old or the new value.

- **Atomic**: reads and writes are totally ordered so that values returned by reads are the same as if the operations had been performed with no overlapping.
Lamport’s model

- **Safe**: a read not concurrent with any write returns the most recently written value.

- **Regular**: safe and a read overlapping a write obtains either the old or the new value.

- **Atomic**: reads and writes are totally ordered so that values returned by reads are the same as if the operations had been performed with no overlapping.
Lamport’s model

- Can characterize a register by its *domain* (binary or multivalued), its *semantics* (safe, regular, atomic) and its *access* (SRSW, MRSW, MRMW).

- Most are equally powerful... eg, can implement multi-valued MRMW atomic register from binary SRSW safe register.
Example

- Wish to implement a multi-valued MRSW register $B$ from $k$ binary MRSW registers $b_0, b_1, \ldots, b_{k-1}$.
  - Domain of $B$ is $[0, \ldots, k-1]$.
  - Implementation uses unary encoding.
  - Initially $b_0 = 1$ and $b_1, b_2, \ldots, b_{k-1} = 0$. 
Example

Write($B, v$):
\[\text{for } i \text{ in } [0.. k - 1] \text{ do } \{
\text{if } (i == v) \text{ Write} (b_i, 1);
\text{else Write} (b_i, 0);
\}\]
\text{return ok;}

Read($B$):
\[\text{for } i \text{ in } [0.. k - 1] \text{ do } \{
\text{if } (\text{Read} (b_i) == 1)
\text{return } i;
\text{return } 0;
\}\]

$B$ has safe semantics.
Example

Write\((B, v)\):
\[
\text{Write}(b_v, 1);
\text{for } i \text{ in } [v - 1 .. 0] \text{ do}
\quad \text{Write}(b_i, 0);
\]
\text{return ok;}

Read\((B)\):
\[
\text{for } i \text{ in } [0.. k - 1] \text{ do }
\quad \text{if } (\text{Read}(b_i) == 1)
\quad \text{return } i;
\]
\text{return 0;}

\(B\) has same semantics as \(b_i\).

MRSW regular register with quorums I

- $N$ storage servers that can fail by silently crashing.
  - The servers do not communicate with each other.
  - Clients read and write data to storage servers by sending reliable FIFO asynchronous messages.
  - Clients do not fail.

- *Wait-free*: a client completes every operation independently from server failures and independently from speed of other clients.

- $S$ is a coterie over $N$. 
MRSW regular register with quorums II

Writer maintains a timestamp $t$.
Each server stores local value $v_i$ and timestamp $t_i$.

- **Write $v$:**
  - **client does:**
    - Choose any quorum $Q \in S$.
    - Send $(v, t)$ to each $p \in Q$.
    - until receive *ack* from all in $Q$. If takes too much time, retry.
  - **server $i$ does:**
    - upon receiving $(v, t)$, set $(v_i, t_i)$ to $(v, t)$ and return *ack*. 
MRSW regular register with quorums III

☐ Read:

■ client does:
  ■ Choose any quorum $Q \in S$.
  ■ Send read to each $p \in Q$.
  ■ until receive $(v_i, t_i)$ from all $i \in Q$. If takes too much time, retry.
  ■ return value with maximum timestamp.

■ server $i$ does:
  ■ upon receiving read, return $(v_i, t_i)$.
MRSW atomic register with quorums

**write**(v):
\[ t = t + 1 \]
send \((v, t)\) to all servers
wait for \textit{ack} from all servers in some quorum \(Q \in S\)
return \textit{ok}

**read**:send \textit{read} to all servers
wait for \((v_i, t_i)\) from all servers \(i\) in some quorum \(Q \in S\)
let \((v, t)\) be the received \((v_i, t_i)\) with largest \(t_i\)
send \((v, t)\) to all server
wait for \textit{ack} from all servers in some quorum \(Q \in S\)
return \(v\)

\textbf{when} receive \((v, t)\) from \(c\): if \((t > t_i)\) \((v_i, t_i) = (v, t)\); send \textit{ack} to \(c\)

\textbf{when receive} \textit{read} from \(c\): send \((v_i, t_i)\) to \(c\)