Today

- Reactively detecting properties.
- Generalizing FIFO: Causal delivery.
- Strong clock condition and vector clocks.
  - Some eye-blurring equations.
- Implementing causal delivery.
- Revisiting RPC deadlock.
  - A realistic deadlock detection protocol.
Reactively detecting properties

Rather than periodically constructing a global state to see if a property holds, deconstruct the property into local information, and have a *monitoring process* combine them.

With RPC Deadlock, local information for $p$ is the set of edges incident on $p$ in the *waits-for* graph.

- When executes *RPCsend*.
- When receives a *RPCsend* message.
- When executes *RPCreply*.
Misdetecting deadlock
Misdetecting deadlock
Misdetecting deadlock
Misdetecting deadlock

\[ \neg(a \rightarrow c) \]
Misdetecting deadlock
Misdetecting deadlock
FIFO and Casual Delivery

If a single process were to send information about its state, then non-FIFO delivery of messages could lead to an incorrect deduction.

FIFO: If \( p \) sends \( m_1 \) to \( r \) before \( p \) sends \( m_2 \) to \( r \), then \( r \) does not deliver \( m_2 \) before \( m_1 \).

Inv: \( x \) never decreases
FIFO and Casual Delivery (continued)

Inv: \( x \geq y \)

\( p \) sends \( m_1 \) to \( r \) before \( q \) sends \( m_2 \) to \( r \), then \( r \) does not deliver \( m_2 \) before \( m_1 \).

\( C \) is consistent if, for all events \( e \) in \( C \), all events \( e' \): \( e' \rightarrow e \) are in \( C \).

Winter Quarter 2009

CSE 223A: Vector Clocks
Strong Clock condition

The *Clock condition* is \( e' \rightarrow e \Rightarrow C(e') < C(e) \)

This is the wrong direction for determining the causal ordering of two events.

*Strong Clock condition*: \( e' \rightarrow e \equiv C(e') < C(e) \)

... which would allow determining causal order from the values of the logical clocks.
Strong Clock condition (continued)

The Strong Clock Condition implies

\[ e' \parallel e \equiv (C(e') = C(e)) \]

\[
C(e_1) = C(e_3) \quad C(e_2) = C(e_3)
\]

\[
\begin{align*}
e_1 & \quad \quad \quad \quad e_2 \\
\end{align*}
\]

\[
\begin{align*}
e_3 & \quad \quad \quad \quad C(e_1) = C(e_2) ??
\end{align*}
\]
A Clock that satisfy the Strong Clock condition

Need a clock that can encode an *irreflexive partial order*.

Given $n$ processes,

$$C_i = [K_1, K_2, ..., K_n]$$

where

- $K_i$: the number of events $p_i$ has executed.
- $K_j, j \neq i$: the number of events of $p_j$ in $p_i$'s causal past.
Vector clocks

- $C_i$ is initially $[0, 0, ..., 0]$.
- When $p_i$ executes an event, it increments $C_i[i]$.
- When $p_i$ sends a message $m$ to $p_j$, it piggybacks $C_i$ on $m$.
- When $p_i$ receives a message $m$,
  \[
  \forall j: 1 \leq j \leq n, j \neq i: C_i[j] = \max(C_i[j], m.C[j])
  \]
  \[
  C_i[i] = C_i[i] + 1.
  \]
Vector clocks (continued)

```
0 1 2
0 1 1
0 0 0
3 1 1
2 2 2
4 1 1
5 2 2
```
Vector clock rules

If $e_i$ is of $p_i$ and $V(e)$ is the clock associated with $e$:

- $e_i \rightarrow e_j \equiv V(e_j)[i] \geq V(e_i)[i]$
- $e_i \parallel e_j \equiv (V(e_i)[i] > V(e_j)[i]) \land (V(e_j)[j] > V(e_i)[j])$

Given two events $e$ and $e'$:

- $e' \rightarrow e \equiv \forall i: 1 \leq i \leq n: V(e')[i] \leq V(e)[i]$
  $\land \exists i: 1 \leq i \leq n: V(e')[i] < V(e)[i]$
- $e' \parallel e \equiv \exists i: 1 \leq i \leq n: V(e')[i] < V(e)[i]$
  $\land \exists i: 1 \leq i \leq n: V(e)[i] < V(e')[i]$
Vector clock rules (continued)

Two events $e_i$ and $e_j$ cannot be in the same consistent cut iff

$$(V(e_i)[j] > V(e_j)[j]) \lor (V(e_j)[i] > V(e_i)[i])$$

The tuple of events $\langle e_1, e_2, ..., e_n \rangle$ is the consistent cut iff

$\forall i: 1 \leq i \leq n: V(e_i)[i] = \max(V(e_1)[i], V(e_2)[i], ... V(e_n)[i])$
Implementing Causal Delivery

Causal delivery of messages to a particular destination (say $p_1$):

- Each process increments its local counter of a vector clock when it sends a message to $p_1$.
- $p_1$ maintains an array $\text{delivered}[1..n]$ where $\text{delivered}[i]$ is the number of message $p_1$ has delivered from $p_i$.
- When $p_1$ receives a message $m$, it adds the message to a set $\text{received}$.
- $p_1$ delivers message $m \in \text{received}$ sent by $p_i$ when:
  - $m.C[i] = \text{delivered}[i] + 1$
  - $\forall j: 1 \leq j \leq n, j \neq i: m.C[j] = \text{delivered}[j]$
Causal delivery

210: $a \rightarrow c$

010: $b \rightarrow a$

110: $b \rightarrow a$

211: $a \rightarrow c$

212: $\neg(a \rightarrow c)$

213: $c \rightarrow b$

223: $c \rightarrow b$

000
Causal delivery

210: $a \rightarrow c$

110: $b \rightarrow a$

010

223: $c \rightarrow b$

211: $a \rightarrow c$

212: $\neg(a \rightarrow c)$

213: $c \rightarrow b$
Causal delivery

\[\begin{align*}
210: & \ a \to c \\
211: & \ a \to c \\
212: & \neg (a \to c) \\
213: & \ c \to b \\
223: & \ c \to b \\
\end{align*}\]
Causal delivery

210

223: $c \rightarrow b$

211: $a \rightarrow c$

212: $\neg(a \rightarrow c)$

213: $c \rightarrow b$
Causal delivery

211

223: \(c \rightarrow b\)

212: \(\neg(a \rightarrow c)\)

213: \(c \rightarrow b\)
Causal delivery
Causal delivery
Causal delivery
Causal Delivery (continued)

- Implementing causal delivery order for all messages sent among processes requires a vector clock for every destination: $O(n^2)$ information.
Revisiting RPC deadlock detection

Neither of these protocol for RPC deadlock detection are very efficient.
A class of stable properties are easier to detect.
Let a relevant event be an event that is used in formulating the stable property.

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.
A bit more formally...

Given a stable property $P$:

- $S$: a global state.
- $S_P$: values of variables in $S$ that are in the formulation of $P$.
- $S | A$: subset of $S$ that consists of the states of processes in $A$ (where channel states are encoded in process states).

$P$ is locally stable if, for any $S$ that satisfies $P$:

Let $A$ be the set of processes that execute no relevant events in an execution starting with $S$.

$S$ can be determined by considering only the variables in $S_P | A$. 
Basic protocol I

- When $i$ executes a relevant events, it records into a buffer $B_i$ its state $B_i.s$ and its vector clock $B_i.C$.
  - We have the vector clock only count relevant events.
  - Since $P$ is locally stable, once $P$ holds, there will be a consistent subcut - i.e., a subset of $\{B_1, B_2, \ldots, B_n\}$ whose timestamps indicate they are pairwise consistent - that establishes $P$.
  - If $A$ is a consistent subcut, then $A' \subseteq A$ is also a consistent subcut.
Basic protocol II

- Periodically a process $i$ collects $\{B_1, B_2, \ldots, B_n\}$ and determines whether there exists a maximal consistent subcut in which $P$ holds.
  - **Safety**: if $P$ holds in this consistent subcut, it holds now.
  - **Liveness**: from $P$ being locally stable.
Maximal subcuts

\[ C(e_2)[1] > C(e_1)[1] \]

\[ C(e_4)[3] > C(e_3)[3] \]

\[ C(e_n)[n-1] > C(e_{n-1})[n-1] \]

\[ 2^{n/2} \text{ maximal subcuts} \]
Latest subcut

\[
\begin{align*}
C(e_2)[1] & > C(e_1)[1] \\
C(e_4)[3] & > C(e_3)[3] \\
& \vdots \\
C(e_n)[n-1] & > C(e_{n-1})[n-1]
\end{align*}
\]

\[\{ \forall \ i \ B_i.s: \forall \ j: B_i.C[i] \geq B_j.C[i] \}\]
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 \{ $B_1$, $B_2$, ... $B_n$ \} 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 $K = \langle D_1, D_2, \ldots, D_n \rangle$ 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 $\langle \perp, \perp, \ldots, \perp \rangle$, 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** $((wf_i \neq \emptyset) \land (K.D_j.rr_j[i] = B_i.rs_i[j]))$
    - **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 \(j\):

\[
\text{if } K.D_i = \perp \\
\quad \text{if } ((wf_i \neq \emptyset) \land (K.D_j.rr_j[i] = B_i.rs_i[j]) \\
\quad \quad \land (\forall k: K.D_k \neq \perp: K.D_k.C[k] \geq B_i.C[k]))
\]

then

\[
K.D_i = \langle B_i.rr_i[B_i.wf_j], B_i.C[i] \rangle; \\
\text{pass } K \text{ to } B_i.wf_j;
\]

else drop \(K\);

else

\[
\text{if } (K.D_j.rr_j[i] = B_i.rs_i[j]) \text{ then } \text{detect deadlock};
\]

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 $wf_i \neq \emptyset$ for unexpected time
$i$ sends $(rr_i[wf_i], i)$ to $wf_i$.

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

- if ($a \neq i$)
  - if ($(wf_j \neq \emptyset) \land (s = rs_i[j])$)
    - then send $(rr_i[wf_i], a)$ to $wf_i$;
  - else
    - if ($s = rs_i[j]$) then detect deadlock;