# Distributed Software Development

## Consensus and Agreement

Chris Brooks

Department of Computer Science

University of San Francisco

### 6-2: Previously on CS 682

- Time is a big challenge in systems that don't share a clock.
- Insight: we often don't need to know the exact time that events occur.
- Instead, we need to know the order in which they happened.

### 6-3: Cause and Effect

- Cause and effect can be used to produce a partial ordering.
- Local events are ordered by identifier.
- Send and receive events are ordered.
  - If $p_1$ sends a message $m_1$ to $p_2$, $send(m_1)$ must occur before $receive(m_1)$.
  - Assume that messages are uniquely identified.
- If two events do not influence each other, even indirectly, we won't worry about their order.

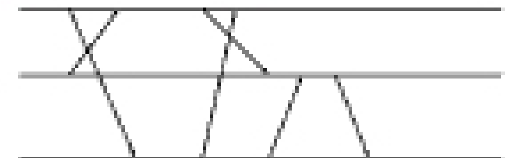
### 6-4: Happens before

- The **happens before** relation is denoted $\rightarrow$.
- Happens before is defined:
  - If $e_i^k, e_i^l$ and $k < l$, then $e_i^k \rightarrow e_i^l$
  - (sequentially ordered events in the same process)
  - If $e_i = send(m)$ and $e_j = receive(m)$, then $e_i \rightarrow e_j$
  - (send must come before receive)
  - If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$
  - (transitivity)
- If $e \not\rightarrow e'$ and $e' \not\rightarrow e$, then we say that $e$ and $e'$ are concurrent. ($e||e'$)
- These events are unrelated, and could occur in either order.

### 6-5: Happens before

- Happens before provides a partial ordering over the global history. $(H, \rightarrow)$
- We call this a distributed computation.
- A distributed computation can be represented with a space-time diagram.

### 6-6: Space-time diagram



## 6-7: Space-time diagram

- Arrows indicate messages sent between processes.
- Causal relation between events is easy to detect
- Is there a directed path between events?
- $e_1^1 \to e_3^3$
- $e_1^2 \| e_2^3$

## 6-8: Monitoring a distributed computation

- Recall that we want to know what the global state of the system is at some point in time.
- Active monitoring won't work
  - Updates from different processes may arrive out of order.
- We need to restrict our monitor to looking at **consistent cuts**
- A cut is consistent if, for all events $e$ and $e'$
  - $(e \in C \text{ and } e' \to e) \Rightarrow e' \in C$
- In other words, we retain causal ordering and preserve the 'happens before' relation.

## 6-9: Synchronous communication

- How could we solve this problem with synchronous communication and a global clock?
- Assume FIFO delivery, delays are bounded by $\delta$
  - $\textbf{send}(i) \to \textbf{send}(j) \Rightarrow \textbf{deliver}(i) \to \textbf{deliver}(j)$
  - Receiver must buffer out-of-order messages.
- Each event $e$ is stamped with the global clock: $RC(e)$.
- When a process notifies $p_0$ of event $e$, it includes $RC(e)$ as a timestamp.
- At time $t$, $p_0$ can process all messages with timestamps up to $t - \delta$ in increasing order.
- No earlier message can arrive after this point.

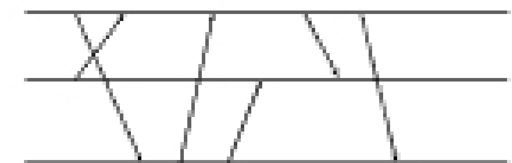
## 6-10: Why does this work?

- If we assume a delay of $\delta$, at time $t$, all messages sent before $t - \delta$ have arrived.
- By processing them in increasing order, causality is preserved.
- $e \to e' \Rightarrow RC(e) < RC(e')$
- But we don't **have** a global clock!!

## 6-11: Logical clocks

- Each process maintains a logical clock. ($LC$).
- Maps events to natural numbers. (0,1,2,3,...).
- In the initial state, all LCs are 0.
- Each message $m$ contains a timestamp indicating the logical clock of the sending process.
- After each event, the logical clock for a process is updated as follows:
  - $LC(e) = LC + 1$ if $e$ is a local or send event.
  - $LC(e) = \max(LC, TS(m)) + 1$ if $e = \textbf{receive}(m)$.
- The LC is updated to be greater than both the previous clock and the timestamp.

## 6-12: Logical clock example



## 6-13: Logical clocks

- Notice that logical clock values are increasing with respect to causal precedence.
  - Whenever $e \rightarrow e'$, $LC(e) < LC(e')$
- The monitor can process messages according to their logical clocks to always have a consistent global state.
- Are we done?
  - Not quite: this delivery rule lacks **liveness**.
  - Without a bound on message delay, we can never be sure that we won't have another message with a lower logical clock.
  - We can't detect gaps in the clock sequence.
  - Example: we'll wait forever for the nonexistent message for $e_8$ from $p_3$.

## 6-14: Adding liveness

- We can get liveness by:
  - Delivering messages in FIFO order, buffering out-of-order messages.
  - Deliver messages in increasing timestamp order.
  - Deliver a message $m$ from process $p_i$ after at least one message from all other processes having a greater timestamp has been buffered.

## 6-15: Causal delivery

- Recall that FIFO only refers to messages sent by the same process.
- **causal delivery** is a more general property which says that if $send(m_1) \rightarrow send(m_2)$, then $deliver(m_1) \rightarrow (m_2)$ when different processes are sending $m_1$ and $m_2$.
- Logical clocks aren't enough to give us causal delivery.

## 6-16: Vector clocks

- Solution: keep a "logical clock" for each process.
- these are stored in a vector $VC$.
  - Assumes number of processes in known and fixed.
- Update rule:
  - $VC(e)[i] = VC[i] + 1$ for send and internal.
  - $VC(e) = max(VC, TS(m))$ for receive; then $VC(e)[i] = VC[i] + 1$
- On receive, the vector clock takes the max on a component-by-component basis, then updates the local clock.

## 6-17: Vector Clock example