Goals:
- Asynchronous computation $\rightarrow$ distributed (multi-agent) computation
- Introduce logical and vector clocks to keep track of event ordering

Reading:
- P. Sivilotti, *Introduction to Distributed Algorithms*, Chapter 5
Where We Are In the Course

Weeks 1-3: UNITY programs
- Predicate calculus, equivalence, quantification [HW #1]
- Program execution (UNITY semantics) [HW #2]
- Stability properties (next, stable, invariant, unless) [HW #2]
- Progress properties (transient, ensures, leadsto) [HW #3]
- Induction (metrics) and proofs of correctness [HW #3, 4]

Week 4: Intro multi-agent systems
- Logical clocks and vector clocks [HW #4]
- Diffusing computations [HW #4]

Week 5: Mutual exclusion (not covered on the midterm)
- Restrict access to a resource to a single process
- User processes + control protocols (composition)
Distributed (Multi-Agent) Systems

Models for distributed systems
• Fixed set of agents (processes) + fixed set of directed channels
• Represent by a directed graph: vertex = agent, edge = (directed) channel

Agents and channels
• Agent: a message-passing automaton
• Channel: sequence of messages, initially empty
• State transitions:
  - Change state without sending or receiving a message
  - Change agent state and send a message on a channel
  - Receive a message on a channel and change agent state
• Channel can be FIFO, TCP-like (out of sequence), or UDP-like (dropped messages)

Challenges:
• How do we write programs that require “synchronization” (agreement on time/order)?
• How do we reason about correctness of a program with channels?
System State for a Multi-Agent System

Definition of system state: $\Sigma_1 \times \cdots \times \Sigma_N \times Q_{c_1} \times \cdots \times Q_{c_M}$

- Global state = state of each agent + state of each channel (current messages in queue)

Possible actions on this state

- Local change of state: agent $i$ updates its state = assignment action
- Sending a message: agent $i$ sends message to agent $j$:
  - Local agent updates its state (eg, msg pending $\rightarrow$ msg sent)
  - Channel adds msg to queue
- Receiving a message: agent $j$ receives message from agent $i$:
  - Channel removes msg from queue
  - Receiving agent updates local state (eg, store message info)

Execution semantics

- Any enabled action can be executed at any time
- Receive action is enabled whenever a channel message queue is not empty

"Space-time diagram"
Causality in Distributed Communications (Lamport, ‘78)

Partial ordering: \(A \rightarrow B\) (“happened before”)
- If \(A\) and \(B\) are events in the same process, then \(A \rightarrow B\) if \(A\) occurs first
- If \(A\) is the sending of a message by one process and \(B\) is the receipt of the same message by another process, then \(A \rightarrow B\)
- Some events cannot be ordered

Logical clocks (Lamport notation)
- Let \(C_i(A)\) be a clock for process \(P_i\) that assigns a number to an event \(A\)
- Define \(C(B) = C_j(B)\) if \(B\) is event in \(P_j\)
- *Clock condition:* for any two events \(A, B\):
  - if \(A \rightarrow B\) then \(C(A) < C(B)\)

Remarks
- Events are partially ordered: can compare some events but not all events
- Example from diagram: \(p_1 \rightarrow q_3\) but \(p_3\) and \(q_3\) are not related
- Clocks are not unique (can choose any set of integers with appropriate relations)
- If \(A \rightarrow B\) and \(B \rightarrow C\) then \(A \rightarrow C\); interpret \(A \rightarrow B\) as “\(A\) can causally effect \(B\)"
Basic channel model: FIFO queue (messages delivered in order sent)

Safety properties
- Basic invariant: messages in a channel are delivered \textit{in the order they were sent}
- Sequence interpretation: let \( c.\text{sent} \) and \( c.\text{received} \) be the sequence of messages sent and received (respectively) on a channel
- Property 1: \( c.\text{received} \) is an initial prefix of \( c.\text{sent} \)
  \[
  \text{invariant} \ (c.\text{received} \preceq c.\text{sent})
  \]
- Property 2: Messages are never lost:
  \[
  \text{stable} \ (\text{sequence} \preceq c.\text{sent})
  \]
- Example: \( c.\text{sent} = [10, 20, 30, 40] \) and \( c.\text{received} = [10, 20] \)

Liveness
- All messages eventually get sent:
  \[
  (\text{sequence} \preceq c.\text{sent}) \rightsquigarrow (\text{sequence} \preceq c.\text{received})
  \]

Comments
- Message-passing systems versus shared variables: once a message is sent, it cannot be deleted
- Channel model assumes TCP-like protocol; UDP is also possible (how?)

https://en.wikipedia.org/wiki/File:Mm1_queue.svg
Logical Clocks

Basic idea: “synchronize” clocks between processes

- Make use of the fact that a message arrives after it was sent
- Attach local clock to each message => get information about other processes

Space-time diagram (= “timeline”)

- Graph must be acyclic (why: _________)

Logical clocks

- Increment local clock on each action (including send and receive)
- Timestamp each message with the local clock time of the sender
- Receiver ensures its local clock is greater than the timestamp of any received msg

Properties

- If A → B then clock(A) < clock (B)
- If A and B are not causally related, then clocks cannot be compared
**Algorithm for Implementing Logical Clock**

Program running on each agent

Program

```
LogicalClock j
```

var

```
j, k : processes,
ch.j.k : channel from j to k,
clock.j : logical time of j,
time.A : logical time of A,
```

initially

```
clock.j = 0
```

assign

```
local event A →
   clock.j := clock.j + 1
   time.A := clock.j

send event A →
   clock.j := clock.j + 1
   time.A, time.m := clock.j, clock.j
   ch.j.k := ch.j.k | m

rcv event A →
   clock.j := max (time.m, clock.j) + 1
   time.A, ch.j.k := clock.j, tail(ch.j.k)
```

Invariant:

```
(∀ A, j : A occurs at j : time.A ≤ clock.j )
∧ (∀ m, j, k : m ∈ ch.j.k : ( ∃ A : A occurs at j : time.A = time.m ) )
∧ (∀ A, B :: A → B ⇒ time.A < time.B )
```
**Vector Clocks**

**Goal: try to find a protocol such that** \( \text{time}(A) < \text{time}(B) \implies A \rightarrow B \)

- Do this by creating a *partial order*: won’t be possible to order *every* pair of events

**Basic idea: each process keeps track of the last time it heard from other processes**

- Each clock maintains vector of timestamps for each process in the system
- Increment local timestamp for every event (including send and receive)
- Local update: just update local (logical) clock
- Received message: take max between local clock and timestamp; update local clock entry

**Properties of vector clocks**

- Can define partial order within each process

\[
\begin{align*}
\text{vtime}.A &= \text{vtime}.B & \equiv & \quad (\forall i :: \text{vtime}.A.i = \text{vtime}.B.i ) \\
\text{vtime}.A &\leq \text{vtime}.B & \equiv & \quad (\forall i :: \text{vtime}.A.i \leq \text{vtime}.B.i ) \\
\text{vtime}.A &< \text{vtime}.B & \equiv & \quad \text{vtime}.A \leq \text{vtime}.B \land \text{vtime}.A \neq \text{vtime}.B
\end{align*}
\]

Note: P1 and P4 not shown
Algorithm for Implementing a Vector Clock

Program $\text{VectorClock}$

```
var
  j, k : processes,
  ch.j.k : channel from j to k,
  m : message,
  A : event,
  vclock.j : vector time of j,
  vtime.A : vector time of A,

initially
  vclock.j.j = 1
  \land (\forall k : k \neq j : vclock.j.k = 0)

assign
  local event A \rightarrow
    vclock.j.j := clock.j.j + 1
    ; vtime.A := vclock.j

  \] send event A \rightarrow
    vclock.j.j := vclock.j.j + 1
    ; vtime.A, vtime.m := vclock.j, vclock.j
    ; ch.j.k := ch.j.k \mid m

  \] rcv event A \rightarrow
    vclock.j := \text{max} (vtime.m, vclock.j)
    ; vclock.j.j := vclock.j.j + 1
    ; vtime.A, ch.j.k := vclock.j, tail(ch.j.k)
```

$v\text{clock}.i = v\text{clock}[i] = \text{i th processes vector clock}$

$v\text{clock}.i.j = (v\text{clock}[i])[j] = \text{jth entry of ith processes vector clock}$
Invariant: the following sets of properties are always true

\[
\begin{align*}
( \forall j, k :: & \ vsClock.k.j \leq vsClock.j.j ) \\
\land & \ ( \forall j, k, m_j :: \ vTime.m_j.k \leq vsClock.j.k ) \\
\land & \ ( \forall A_j, B_k :: \ A_j \rightarrow B_k \equiv vTime.A_j < vTime.B_k ) \\
\land & \ ( \forall A_j, B_k :: \ A_j \rightarrow B_k \leftarrow vTime.A_j.j \leq vTime.B_k.j )
\end{align*}
\]

local clock for process is always bigger than remote estimates of clock

\[m_j = \text{message from jth process}\]
Summary: Time, Clocks, and Synchronization

Channel model: FIFO, lossless, directed

Events, system timelines and logical time
- Can’t assume process clocks agree
- Make use of “logical time”
  \[ A \rightarrow B \Rightarrow \text{time}.A < \text{time}.B \]

Algorithm for setting logical time

- local event \( A \rightarrow \)
  \[
  \begin{align*}
  &\text{clock}.j := \text{clock}.j + 1 \\
  &\text{time}.A := \text{clock}.j \\
  
  \end{align*}
  \]

- send event \( A \rightarrow \) (to \( k \))
  \[
  \begin{align*}
  &\text{clock}.j := \text{clock}.j + 1 \\
  &\text{time}.A, \text{time}.m := \text{clock}.j, \text{clock}.j \\
  &\text{ch}.j.k := \text{ch}.j.k | m
  \end{align*}
  \]

- rcv event \( A \rightarrow \) (\( m \) from \( k \))
  \[
  \begin{align*}
  &\text{clock}.j := \max(\text{time}.m, \text{clock}.j) + 1 \\
  &\text{time}.A, \text{ch}.j.k := \text{clock}.j, \text{tail}(\text{ch}.j.k)
  \end{align*}
  \]

Properties

\[
(\forall A, j : A \text{ occurs at } j : \text{time}.A \leq \text{clock}.j )
\]
\[
\wedge (\forall m, j, k : m \in \text{ch}.j.k : ( \exists A : A \text{ occurs at } j : \text{time}.A = \text{time}.m ) )
\]
\[
\wedge (\forall A, B :: A \rightarrow B \Rightarrow \text{time}.A < \text{time}.B )
\]

Vector clocks: \( A \rightarrow B \equiv \text{vtime}.A < \text{vtime}.B \)