Theoretical Foundations for Distributed Systems

Manas Saksena
University of Pittsburgh

Distributed Systems

- Opportunities
  - Performance through Parallelism
  - Resource Sharing (Cost Effectiveness)
  - Reliability and Availability
    - Avoid single points of failure
  - Expandability
- Limitations
  - No Global Clock
  - No Shared Memory

Distributed Algorithms

- Data may be scattered on different machines
  - No global memory
- Decisions based on local information
  - but, information can be exchanged
- Avoid Single Point of Failure
  - to improve reliability, availability
- Lack of a precise global time source
  - no common clock

Absence of a Global Clock

- On a single machine
  - A clock can be used to determine ordering of events in concurrent processes
- Example Use
  - make program
- Problem Scenario

Absence of Shared Memory

- How much money is there in the two accounts together?
  - X = Query(P1:A)
  - Y = Query(P2:B)
  - Amount = X + Y
- What happens when money is in transit?

Lamport’s Logical Clocks

- Clock
  - ordering by time (happened before)
  - time interval
  - absolute time
- Ordering by time
  - a counter is sufficient
- Time Interval
  - clock must tick at the same rate as real-time
- Absolute Time
  - clock must be “in-sync” with absolute time
Logical Clocks

- Happened Before Relation $\rightarrow$
  - $a \rightarrow b$, if $a$ and $b$ are in the same process, and $a$ occurred before $b$
  - $a \rightarrow b$, $a$ is send event, $b$ is corresponding receive event
  - $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$ (transitive)
- Causally Related
  - Event $a$ causally effects Event $b$ if $a \rightarrow b$
- Concurrent
  - if not ($a \rightarrow b$) and not ($b \rightarrow a$)

Logical Clocks

- Associate a Clock $C_i$ with each Process $P_i$
- Clock
  - associated a timestamp to each event
- How to implement clock to realize the happened before relation?
- Two Conditions:
  - $C_i(a) < C_i(b)$ if $a$ occurs before $b$ in $P_i$
    - IR1: $C_i := C_i + d$ ($d > 0$)
  - $P_i$ to $P_j$:
    - IR2: $C_j := \max(C_j, t_m + d)$ ($d > 0$)

Logical Clocks

- Total Ordering from Lamport's Clocks
  - Use process id to break ties
  - $a \rightarrow b$, if
    - $C_i(a) < C_i(b)$, or
    - $C_i(a) < C_j(b)$ and $P_i < P_j$
- Limitation
  - if $a \rightarrow b$ then $C(a) < C(b)$, but
    - the reverse is not true
- How do we implement clocks for which this is true?

Vector Clocks

- Clock:
  - Vector of length $N$
  - $C[i] = \text{logical clock of } P_i$
  - $C[i] = \text{time of occurrence of last event at } P_i \text{ which happened before the current time in } P_i$
- Implementation Rules:
  - IR1: $C[i] = C[i] + d$ (on local events)
  - IR2: $C[k] = \max(C[k], t_m[k])$

Vector Clocks: Example

P1: 0,0,0 $\rightarrow$ 1,0,0 $\rightarrow$ 2,0,0
P2: 0,0,0 $\rightarrow$ 1,1,0 $\rightarrow$ 2,2,1
P3: 0,0,0 $\rightarrow$ 2,0,1

- It is possible to compare clocks to see if
  - they are equal or not
    - all are equal (one is not)
    - less than or equal (partial order)
      - all are less than or equal
      - less than
        - less than or equal and not equal
        - not less than
          - not less than or equal and not equal
          - concurrent
            - both not less than
Vector Clocks

♦ Causally Related
  ➢ if \( t_a < t_b \) or \( t_b < t_u \), otherwise concurrent
  ➢ \( a \rightarrow b \) iff \( t_a < t_b \)

Causal Message Ordering

♦ What?
  ➢ If \( \text{send}(m_1) \rightarrow \text{send}(m_2) \) then
    ➢ \( \text{receive}(m_1) \rightarrow \text{receive}(m_2) \)

♦ Why?
  ➢ Example: Replicas
    ➢ Ensures replica consistency

Causal Broadcast

♦ Message can be only delivered when all causally earlier messages have been delivered
♦ Use Vector Clocks with Messages
  ➢ Each process assigns a sequence number to the messages it sends
  ➢ When a message is sent, the current vector at the sending node is sent as the timestamp

Causal Ordering

♦ Message Reception: Delivered if
  ➢ if message received from \( P_i \) then local sequence number for \( P_i \) must be 1 less than what’s in the message
    ➢ Implies message is in sequence
  ➢ For all other processors, the local sequence number must be no less than what’s in the message
    ➢ Sender has not seen any message that the receiver has missed
♦ Else: buffer for later delivery

Distributed Algorithms Model

♦ A Set of Processes (or Processors)
♦ Each process has local memory
  ➢ State = values of variables
♦ Each process can have local clock(s)
  ➢ Logical Clocks
    ➢ e.g., Timestamps, Counters, Sequence_nums
    ➢ No relationship to absolute time
  ➢ Physical Clock (approximation of absolute time)
    ➢ Clock synchronization
    ➢ Real-time systems

Distributed Algorithms Model

♦ Communication
  ➢ One-Way Message Passing (no shared memory)
  ➢ Point-to-Point Channels
  ➢ Multicast Channels
    ➢ Single sender, multiple receivers
  ➢ Broadcast Channels
♦ Channel Properties
  ➢ Reliable
    ➢ No Losses, No Duplicates, No errors, FIFO
  ➢ Unreliable
Distributed Algorithms Model

- Failures
  - Channels
    - Lost/Duplicate messages
    - Message Errors
    - Crash (no messages after failure)
  - Processors
    - Crash
    - Byzantine
- Non-Instantaneous Communication
  - What you see depends on where you are!!!

Consistent Global States

- Process State:
  - encapsulates messages sent/received
  - all messages sent/received prior to recording state are part of the state
- Channel State:
  - Suppose we have recorded states for Pi and Pj
  - Then, the channel state (for a channel from Pi to Pj) should be:
    - all messages m, such that sent(m) is in the recorded state of Pi, and recv(m) is not in recorded state of Pj

Global Consistent States

- A global state is inconsistent if
  - there is a message m, from Pi to Pj, such that
    - sent(m) is not part of local state of Pi, but
    - recv(m) is part of local state of Pj
- Recording Channel States
  - done by receiver side of channel
  - Sender side sends a marker after recording its own state
  - Receiver side records channel state upon receiving the marker

Recording Channel State

- On receipt of marker on receiver side
  - Case 1: if receiver state is not recorded
    - record receiver state
    - mark channel state as empty
      - all recorded sends on the sender side have already been received by the receiver side
      - NOTE: Assumes FIFO delivery
  - Case 2: if receiver state is already recorded
    - mark channel state as all messages received from the time it recorded its own state until the receipt of the marker

Example: System Model

- Two Processes
  - P1 & P2
- Two One-way Point-to-Point Channels
  - c & c'
Example: Process Behavior

- Behavior of each process is represented using a finite state machine
- Process States are represented abstractly (in this example)
- Channel states are the messages in transit on the channel

![Diagram](image)

Example: Possible System Evolution

<table>
<thead>
<tr>
<th>Event</th>
<th>P1</th>
<th>P2</th>
<th>C</th>
<th>C'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>A</td>
<td>C</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>e1: Send M</td>
<td>B</td>
<td>C</td>
<td>M</td>
<td>-</td>
</tr>
<tr>
<td>e2: Send M'</td>
<td>B</td>
<td>D</td>
<td>M</td>
<td>M'</td>
</tr>
<tr>
<td>e3: Recv M'</td>
<td>A</td>
<td>D</td>
<td>M</td>
<td>-</td>
</tr>
</tbody>
</table>

- Three events: e1, e2, e3
- Assume that they occurred in this order
- Each row is a global state of the system
- The table shows the evolution of the system
- Note: we know that e2 must occur before e3 (e2→e3)

Example

- Process P1
  - Record State (as A)
  - Send Marker on C
  - Send M (e1)
  - Receive M' (e3)
  - Receive Marker on C'
  - Record C' state (as M')
  - Final Recorded State: P1 (A), P2 (D), C (-), C' (M')
- Process P2
  - Send M' (e2)
  - Receive Marker on C
  - Record own state (as D)
  - Record C state (as -)
  - Send Marker on C'
  - Record C' state (as M')
- Note: this corresponds to event sequence: e2, e1, e3
  - Possible & consistent sequence, but may not have happened