Chapter 7: Models and Clocks

* **Distributed System**: asynchronous message-passing system without any shared memory or a global clock.
  + **Absence of a shared clock**: impossible to synchronize the clocks of different processors due to uncertainty in communication delays
  + **Absence of shared memory:** impossible for a singular processor to know the global state of the system.
  + **Absence of accurate failure detection:** impossible to distinguish between a slow and failed processor
* Channels: cannot make assumptions on the ordering of messages received. Any message sent on a channel may experience arbitrary but finite delay, so state of channel is defined to be sequence of messages **sent** along the channel, not received.
* Capturing behavior in a distributed system (**Models**):
  + **Interleaving Model:** global sequence of events. Thus, all events in a run (distributed computation) are **interleaved. Total order** on the set of events.
  + **Happened-Before Model:** in a true distributed system, only a partial order can be determined between events. Thus, all events in a run (distributed computation) is a tuple **(E, ->)** where **E** is the set of events within a **single** **process** and **->** is the partial order on all events in **E** such that all events within a single process are totally ordered.
    - **Happened-Before Relation:** 
      * If **e** happened before **f** in a singular process, then **e -> f.**
      * If **e** is the send event of a message and **f** is the receive event of the same message, then **e -> f**.
      * Transitivity holds: if there exists and event **g** such that **e -> g, g -> f, then e -> f.**
    - **Process-time / Happened-before diagrams:** two events **e** and **f** satisfy the happened-before relation if and only if **there exists a directed path from e to f.**
    - **Concurrent:** since **e->f** is a partial order, two events **cannot** be related by the happened-before relation. Thus, two events **e** and **f** are concurrent if **~(e->f) ^ ~(f->e)**, or there exists no directed path from **e to f** or **f to e**.
* **Clocks:** mechanisms to track relations between events.
  + **Logical clocks:** used for a distributed computation where behavior is a total order, i.e. when there is no possible way to determine the order of events due to absence of accurately synchronized physical clocks. **Develops mechanism which determines a total order which *could* have happened, rather than did happen.**
    - Purpose: **only** to determine an order between events.
    - Definition: **a mapping of E to N, where E is the set of events and N is the set of natural numbers.** 
      * For all events/states e and f**, (e->f) => C(e) < C(f)**.
      * Alternative definition: **s -> t => s.c < t.c**.
    - Algorithm:
      * Var: c = 0; // c is the logical clock for a certain process Pi.
      * Message send (Pi):
        + Send (message, c); // send message along with Pi’s logical clock value
        + c = c + 1; // logical clock for Pi increments.
      * Message receive (Pi) from Pj:
        + c = max(c, d) + 1; // **d** is logical clock value for **Pj** and **c** is logical clock value for **Pi**..
      * Internal events:
        + c = c + 1; // increment logical clock for process which handles the internal event.
    - Recall this algorithm only ensures an order between events. To determine a **total order**, a total order relation **“<”** is defined as:
      * **(s.c, s.p) < (t.c, t.p) == ((s.c < t.c) or (s.c == t.p)) and (s.p < t.p)**
      * where **x.p** is the process which belongs to **state x**.
  + **Vector clocks:** mechanism to provide total order for the **happened-before relation**.
    - Definition: For all states **s, t** : **s -> t ⬄ s.v < t.v**, where **v** is the vector clock belonging to a certain state **x**.
    - Contains a clock vector of size **N**, where **N** is the number of processors in the system, where the index of the processor in the vector is its specific vector clock (i.e. process Pj has a vector clock value of
    - Algorithm:
      * Var: v = array[N] of integers, initially set to all 0’s.
        + For all processes that aren’t the current process **“i”,** v[j] = 0 and v[i] = 1.
      * Message send(Pi): **// \*\* sends message and process’s “v”**.
        + t.v = s.v // update state of vector clock
        + t.v[i] = t.v[i] + 1;
      * Message receive(Pi, sent(u)):
        + for each process **j** in **v**:

t.v[j] = max(s.v[j], u.v[j]) + 1; // take component-wise max of current process’s vector clock for process **j** and process associated with sent message **u**’s vector clock for process **j**.

* + - * + t.v[i] = t.v[i] + 1;
      * Internal event(Pi): // same as message send
  + **Direct-Dependency Clocks**: mechanism which uses only one integer to be sent per message, rather than O(N) messages to be sent per message with a vector clock mechanism.
    - Algorithm:
      * Var: v = array[N] of integers, initially set to all 0’s.
        + For all processes that aren’t the current process index **i,** v[j] = 0 and v[i] = 1.
      * Message send(Pi): **// \*\* sends only message**
        + T.v = s.v
        + T.v[i] = t.v[i] + 1;
      * Message receive(Pi, sent(u)) from Pk:
        + T.v[i] = max(s.v[i], u.v[k]) + 1; // update current process’s vector clock by taking component-wise maximum of **initial state of current process i and vector clock associated with sent message’s process k**.
        + T.v[k] = max(s.v[k], u.v[k]); // update vector clock associated with sent message’s process k by taking max of process **k**’s initial state and **k’**s state of sent message.
      * Internal event(Pi): // same as message send, except doesn’t send message