Time and Global States

CMPS 4760/6760: Distributed Systems
Overview

- Physical clocks
- Logical clocks and vector clocks
- Global state
Time and Clock

- **Primary standard of time:** rotation of earth
  - 1 solar second = 1/86,400th of a solar day that the Earth takes to complete one revolution around its axis

- **De facto primary standard of time:** atomic clocks
  - 1 atomic second = 9,192,631,770 orbital transitions of Cesium-133 atom.
  - 86400 atomic sec = 1 solar day – approx. 3 ms (leap second correction each year)

- Coordinated Universal Time (UTC) does the adjustment for leap seconds ± number of hours in your time zone
Global Positioning System: GPS

A system of 32 satellites broadcast accurate spatial coordinates and atomic times

• Location and precise time computed by triangulation

• no leap sec. correction => 18 seconds ahead of UTC (as of 2017),

• Per the theory of relativity, an additional correction is needed. Locally compensated by the receivers.
Terminology

Drift rate \( \frac{d(C(t) - t)}{dt} \)

Clock skew \( \delta \)

Resynchronization interval \( R \)

Max drift rate \( \rho \) implies:

\[
(1 - \rho) \leq \frac{dC(t)}{dt} \leq (1 + \rho)
\]

Drift is unavoidable:

- Ordinary quartz-oscillators clocks: \( 10^{-6} \)
- “High precision” quartz clocks: \( 10^{-8} \)
- Atomic clocks: \( 10^{-13} \)
Physical clock synchronization

Why accurate physical time is important?

• Accurate time keeping: air-traffic control systems

• Accurate timestamps: multi-version objects

• Some security mechanisms depend on the physical times of events, e.g. Kerberos
Physical clock synchronization

- **External Synchronization:** $|C_i(t) - S(t)| < \delta/2$
- **Internal Synchronization:** $|C_i(t) - C_j(t)| < \delta$
- **Challenges:** account for propagation delay, processing delay, and *faulty clocks*
- **Monotonicity:** $t' > t \Rightarrow C(t') > C(t)$
- **Unbounded clocks vs. bounded clocks**
  - Y2K bug
- **Hardware clock vs. Software clock**
  - $C(t) = \alpha H(t) + \beta$
External Synchronization: Cristian’s method

- Client **pulls data** from a **time server** every $R$ unit of time, where $R < \delta/2\rho$ (why?)

- $C_i = T_s + RTT/2$
  - Client estimates round trip time $RTT = T_2 - T_1$
  - if the minimum transmission $min$ is known, the accuracy is $\pm (RTT/2 - min)$

- Improve accuracy and fault tolerance
  - Query multiple times & take minimum RTT
  - Query multiple time servers
Internal Synchronization: Berkeley Algorithm

- The participants elect a master (leader)
- The master coordinates the synchronization

**Step 1.** Salves send their clock values to the master

**Step 2.** Master discards outliers and computes the average

**Step 3.** Master sends the needed adjustment to the slaves
Berkeley Algorithm

To maintain Monotonicity
- **Negative** correction => slowdown
- **Positive** correction => speedup

RTT adjustment as in Cristian’s method (not shown in the figure)
Internal synchronization with byzantine clocks

- Lamport and Melliar-Smith’s algorithm

Assume $N$ clocks, at most $f$ are faulty

Clock $i$ runs the following algorithm:

**Step 1.** Read every clock in the system.
  - $c_i[j]$: clock $i$’s reading of clock $j$’s value

**Step 2.** if $|c_i[j] - c_i[i]| > \delta$, $c_i[j] = c_i[i]$

**Step 3.** Update the clock using the average of these values.

*Synchronization is maintained if* $N > 3f$
Lamport and Melliar-Smith’s algorithm

The maximum difference between the averages computed by two non-faulty nodes is \((3f\delta/N)\)

To keep the clocks synchronized,

\[
\frac{3f\delta}{N} < \delta \iff N > 3f
\]
Overview

- Physical clocks
- Logical clocks and vector clocks
- Global state
Model of a Distributed System

- A distributed program consists of a set of $N$ processes, $\{P_1, P_2, \ldots, P_N\}$, and a set of unidirectional channels.
  - Message passing only, no shared memory, no global clock
  - Channel model: error free, arbitrary but finite delay, no assumptions on ordering
Model of a Distributed System

- A process is defined as a set of states, an initial condition (i.e., a subset of states), and a set of events
- **State of a process**: values of all the variables (including the program counter)
- **State of a channel**: sequence of messages set along the channel but not received
  - A process may record messages sent and received as part of its **local state**
Global State

- The set of global states = the cross product of local states and the states of channels.
- An initial global state is one in which all local states are initial and all the channels are empty.
Model of a Computation

- Each event corresponds to an action and may change the state of a process and the state of at most one channel incident on that process
  - Internal events only change the state of a process
  - External events: sends to/receives from other process

- A set of events \((e^1, e^2, \ldots)\) in a single process is called sequential, and their occurrences can be totally ordered in time using the clock at that process

- A run or a computation of a process \(P_i\) is defined as a sequence of local states and events: \(s_i^0 e_i^0 s_i^1 \ldots e_i^k s_i^k\)
Global State

- “time-based model”: a global state is a set of local states that occur **simultaneously**.

- “happened-before model”: a global state is a set of local states that are all **concurrent** with each other.

There is nothing called **simultaneous** in the physical world.
Model of a Computation

- Questions:
  - How do we decide if an event happened before another event when the two events are on different processes?
  - How do we decide if two events are concurrent?
  - Can we do these without using physical clocks, since physical clocks are not be perfectly synchronized?
Causality

- The event of sending message must have happened before the event of receiving that message.
Happened Before Model

Notations:

- \( e < f \) iff \( e, f \) are two events in a single process \( P \), and \( e \) proceeds \( f \)
- \( e \preceq f \) iff \( e < f \lor e = f \)
- \( e \sim f \) iff \( e = \text{sending} \) a message, and \( f = \text{receipt} \) of that message

These definitions also apply to states.
The *happened-before* relation (→) is the smallest relation that satisfies

**Rule 1.** if \((e < f) \lor (e \sim f)\) then \(e \rightarrow f\).

**Rule 2.** \((e \rightarrow f) \land (f \rightarrow g) \Rightarrow e \rightarrow g\)

→ defines a partial order on \(E\) (the set of all events)

\(e\) and \(f\) are **concurrent** (denoted by \(e||f\)) if \(\neg(e \rightarrow f) \land \neg(f \rightarrow e)\)

Again, we can similarly define happened-before relation for **states**
Logical Clocks

- A logical clock $LC$ is a map from $E$ to $\mathbb{N}$ with the constraint:

$$\forall e, f \in E, e \rightarrow f \Rightarrow LC(e) < LC(f)$$

- The constraint models:
  - sequential nature of execution at each process
  - Physical requirement that any message transmission requires a nonzero amount of time

Leslie B. Lamport
Lamport timestamps
Implementation

\[ P_i:: \]

\[ \text{var} \]

\[ LC: \text{integer initially 0}; \]

\[ \text{internal event}() : \]

\[ LC = LC + 1; \]

\[ \text{send event} (m) : \]

\[ LC = LC + 1; \]

\[ \text{piggybacks} \ LC \text{ on } m; \]

\[ \text{receive event} (m, t) : \]

\[ LC = \max(LC, t) + 1; \]
A total ordering on events

Let $e$ and $f$ be two events in processes $i$ and $j$, respectively. We can define a total order $\ll$ of events as:

$$ e \ll f \text{ iff either } LC(e) < LC(f) $$

or $LC(e) = LC(f)$ and $i < j$
Vector Clocks

- Logical clocks cannot detect causality
- A vector clock $VC$ is a map from $E$ to $\mathbb{N}^n$ with the constraint:

  $$\forall e, f \in E, e \rightarrow f \iff VC(e) < VC(f)$$

- Given two vectors $x$ and $y$ of dimension $n$, we compare them as follows:

  $$x < y = (\forall k: 0 \leq k \leq n - 1: x[k] \leq y[k]) \land (\exists j: 0 \leq j \leq n - 1: x[j] < y[j])$$
  $$x \leq y = (x < y) \lor (x = y)$$
Implementation

\[ P_i :: \]

\[
\begin{align*}
{\text{var}} & \quad {\text{var}} \\
VC & : \text{array}[1..N] \text{ of integer}; \text{initially } VC[j] = 0 \ \forall j;
\end{align*}
\]

{\text{internal event}} (\cdot):

\[
VC[i] = VC[i] + 1;
\]

{\text{send event}} (m):

\[
VC[i] = VC[i] + 1;
\]

\quad piggybacks \ VC \text{ on } m;\]

{\text{receive event}} (m, t):

\[
VC[i] = VC[i] + 1;
\]

\quad \forall j: \ VC[j] = \max(VC[j], t[j]);\]
Vector timestamps

\[(1,0,0)\quad (2,0,0)\]

\[(2,1,0)\quad (2,2,0)\]

\[(0,0,1)\quad (2,2,2)\]

\[p_1\quad p_2\quad p_3\]

\[a\quad b\quad c\quad d\quad e\quad f\]

\[m_1\quad m_2\]

Physical time
Properties

- Lemma: For any two events $e$ and $f$ in processes $i$ and $j$, respectively:

\[ e \leftrightarrow f \Rightarrow VC(e)[i] > VC(f)[i] \]

- Theorem: $\forall e, f \in E, e \rightarrow f \iff VC(e) < VC(f)$
Overview

- Physical clocks
- Logical clocks and vector clocks
- Global state
Detecting Global Properties

(a) Garbage collection

(b) Deadlock

(c) Termination
Global State

- **Current** global state is hard to get
  - No process has global knowledge

- **Past** global state is often sufficient
  - Failure recovery
  - Monitoring stable properties: has the token been lost?

- A property is called stable if once it is true it stays true forever
Global Snapshot
Global State

- “time-based model”: a global state is a set of local states that occur simultaneously.

- “happened-before model”: a global state is a set of local states that are all concurrent with each other.
Cuts

Physical time

Inconsistent cut

Consistent cut

$e_1^0$ $e_1^1$ $e_1^2$ $e_1^3$

$p_1$

$m_1$

$p_2$

$m_2$
Consistent Cuts

- A cut (global state, global snapshot) = a vector of local states containing exactly one state from each process

- A consistent cut (consistent global state) is a cut where all the states are mutually concurrent (in terms of the happened-before relation).

- A linearization (or a consistent run) of a distributed system is a sequence (total ordering) of consistent global states where
  1. the sequence is consistent with the happened-before relation
  2. two adjacent states differ in the state of exactly one process
Chandy and Lamport’s snapshot algorithm

- Assumptions
  - Neither channels nor processes fail
  - Channels are unidirectional and FIFO-ordered (First in First out)
  - The graph of processes and channels is strongly connected (there is a path between any two processes)
  - Any process may initiate a global snapshot at any time
  - The process may continue their execution and send and receive normal messages while the snapshot takes place
Chandy and Lamport’s snapshot algorithm

- **Informal description**
  - Each process is either *white* or *red*. All processes are initially *white*.
  - After recording the local state, a process turns *red*.

- **Two difficulties**
  - Need to ensure that the recorded local states are *mutually concurrent*.
  - Need to capture the *state of the channels*.
Classification of Messages

\[ P \]

\[ Q \]

\( \text{ww} \)

\( \text{rw} \)

\( \text{wr} \)

\( \text{rr} \)
Informal description

- Each process is either white or red. All processes are initially white.
- After recording the local state, a process turns red.
- Once a process turns red, it is required to send a special message called a *marker* along all its outgoing channels before it sends out any message and start recording messages from all incoming channels.
- Once $P_i$ receives a marker from $P_j$:
  - it is required to turn red if it has not already done so.
  - it stops recording messages from $P_j$.
Chandy and Lamport’s snapshot algorithm

\( P_i :: \)

```
var
  color: \{white, red\} initially white;
// assume \( k \) incoming channels
  chan: array[1..k] queues of messages initially null;
  closed: array[1..k] of boolean initially false;

  turn_red() enabled if (color == white):
    save_local_state;
    color = red;
    send (marker) to all neighbors;
```

Upon `receive(marker)` on channel \( j \)
```
  if(color == white) turn_red();
  closed[j] = true;
```

Upon `receive(prog_message)` on channel \( j \)
```
  if(color == red \&\& \neg closed[j])
    chan[j] = append(chan[j],
                      prog_message)
```
Example: Communicating State Machines

\[
\begin{array}{c}
\text{send } M \\
\text{up} \\
\text{send } M' \\
\text{receive } M' \\
\text{down} \\
\text{receive } M' \\
\text{STATE MACHINE } i \\
\text{send } M \\
\text{up} \\
\text{send } M' \\
\text{receive } M \\
\text{down} \\
\text{receive } M \\
\text{STATE MACHINE } j
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{Global state} & \text{Process } i & \text{Process } j \\
\hline
S_0 & \text{down} & \emptyset & \text{down} & \emptyset \\
S_1 & \text{up} & M & \text{down} & \emptyset \\
S_2 & \text{up} & M & \text{up} & M' \\
S_3 & \text{down} & M & \text{up} & \emptyset \\
\hline
\end{array}
\]
Something unusual

- Let machine $i$ start Chandy-Lamport snapshot before it has sent $M$ along $c1$.
- Also, let machine $j$ receive the marker after it sends out $M'$ along $c2$.
- Observe that the snapshot state is
  
  down $\emptyset$ up $M'$

- This state was never reached during the computation!
Understanding snapshot
Understanding snapshot

The *observed state* is a *feasible state* that is reachable from the *initial configuration*. It may not actually be visited during a specific execution.

The *final state* of the original computation is *always reachable* from the *observed state*.
Properties

- **Lemma (Monotonicity of color)** $\forall s, t: s \rightarrow t \Rightarrow s.color \leq t.color$
  
  - Assume white < red

- **Theorem (Safety):** The Chandy–Lamport algorithm records a consistent global state.
  
  - $\forall s, t: s.color = \text{red} \land t.color = \text{red} \Rightarrow rstate(s)||rstate(t)$, where $rstate(s) = \max\{s'|s' \leq s, s'.color = \text{white}\}$
  
  - If both $s$ and $t$ (of $P_i$ and $P_j$, respectively) are red and the channel from $P_i$ to $P_j$ is closed at $t$, then $P_j$ records precisely those messages that are sent before the recorded state in $P_i$ and are received after the recorded state in $P_j$

- **Theorem (Liveness):** Eventually all processes turn red and all channels are closed
  
  - Formally, $\exists s: s.color = \text{red} \Rightarrow \forall j: (\exists t \in P_j: t.color = \text{red} \land t.closed[k] = \text{true} \ \forall \ k)$