Distributed System Fundamentals

Xiaohui (Helen) Gu
Agenda

I. Synchronous versus Asynchronous systems
II. Lamport Timestamps
III. Global Snapshots
IV. Impossibility of Consensus proof
Two Different System Models

- **Synchronous Distributed System**
  - Each message is received within bounded time
  - Drift of each process’ local clock has a known bound
  - Each step in a process takes $lb < time < ub$
  - *Ex:* A collection of processors connected by a communication bus, e.g., a Cray supercomputer

- **Asynchronous Distributed System**
  - No bounds on process execution
  - The drift rate of a clock is arbitrary
  - No bounds on message transmission delays
  - *Ex:* The Internet is an asynchronous distributed system

- This is a more powerful model than the synchronous system model. A protocol for an asynchronous system will also work for a synchronous system (though not vice-versa)

- It would be **im**possible to accurately synchronize the clocks of two communicating processes in an asynchronous system
Logic Clocks

• But is accurate (or approximate) clock sync. even required?
• Wouldn’t a **logical ordering** among events at processes suffice?
• Lamport’s **happens-before** ($\rightarrow$) among **events**:
  • On the same process: $a \rightarrow b$, if $time(a) < time(b)$
  • If p1 sends $m$ to p2: $send(m) \rightarrow receive(m)$
  • If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$
• Lamport’s **logical timestamps** preserve causality:
  • All processes use a **local counter** (logical clock) with initial value of zero
  • Just before each **event**, the local counter is incremented by 1 and assigned to the event as its timestamp
  • A $send$ (**message**) event carries its timestamp
  • For a $receive$ (**message**) event, the counter is updated by $\max$($receiver$’s-local-counter, $message$-timestamp) + 1
Lamport Timestamps

- Logical timestamps preserve causality of events, i.e., \( a \rightarrow b \implies TS(a) < TS(b) \)
- Can be used instead of physical timestamps
Spot the Mistake

Physical Time

Host 1
Host 2
Host 3
Host 4

Clock Value

timestamp

Message
Corrected Example: Lamport Logical Time

![Diagram showing Lamport Logical Time with arrows indicating message timestamps and clock values across multiple hosts.]

- Host 1
- Host 2
- Host 3
- Host 4

Clock Value

Timestamp -> Message
Corrected Example: Lamport Logical Time

• $a \rightarrow b \implies TS(a) < TS(b)$ but not the other way around
• Logical time does not account for out-of-band messages
III. Global Snapshot Algorithm

- Can you capture (record) the states of all processes and communication channels at exactly 10:04:50 am?
- Is it necessary to take such an exact snapshot?
- Chandy and Lamport snapshot algorithm: records a logical (or causal) snapshot of the system.

**System Model:**
- No failures, all messages arrive intact, exactly once, eventually
- Communication channels are unidirectional and FIFO-ordered
- There is a communication path between every process pair
Chandy and Lamport Snapshot Algorithm

1. **Marker (token message) sending rule for initiator process $P_0$**
   - After $P_0$ has recorded its state
     - for each outgoing channel $C$, send a marker on $C$

2. **Marker receiving rule for a process $P_k$**:
   
   On receipt of a marker over channel $C$
   - **if this is first marker being received at $P_k$**
     - record $P_k$’s state
     - record the state of $C$ as “empty”
     - turn **on** recording of messages over all other incoming channels
     - for each outgoing channel $C$, send a marker on $C$
   - **else**
     - turn **off** recording messages only on channel $C$, and mark state of $C$ as all the messages recorded over $C$

- Protocol terminates when every process has received a marker from every other process
1- P1 initiates snapshot: records its state (S1); sends Markers to P2 & P3; turns on recording for channels C21 and C31

2- P2 receives Marker over C12, records its state (S2), sets state(C12) = {}; sends Marker to P1 & P3; turns on recording for channel C32

3- P1 receives Marker over C21, sets state(C21) = {a}

4- P3 receives Marker over C13, records its state (S3), sets state(C13) = {}; sends Marker to P1 & P2; turns on recording for channel C23

5- P2 receives Marker over C32, sets state(C32) = {b}

6- P3 receives Marker over C23, sets state(C23) = {}

7- P1 receives Marker over C31, sets state(C31) = {}

**Consistent Cut** = time-cut across processors and channels so no event after the cut “happens-before” an event before the cut
IV. Give it a thought

Have you ever wondered why distributed server vendors always only offer solutions that promise five-9’s reliability, seven-9’s reliability, but never 100% reliable?

The fault lies in the impossibility of consensus.
What is Consensus?

• N processes
• Each process p has
  – input variable $x_p$ : initially either 0 or 1
  – output variable $y_p$ : initially b
• Consensus problem: design a protocol so that either
  – all processes set their output variables to 0
  – Or all processes set their output variables to 1
  – There is at least one initial state that leads to each outcome above
Why is Consensus Important

• Many problems in distributed systems are equivalent to (or harder than) consensus!
  – Agreement (harder than consensus, since it can be used to solve consensus)
  – Leader election (select exactly one leader, and every alive process knows about it)
  – Failure Detection

• Consensus using leader election
  Choose 0 or 1 based on the last bit of the identity of the elected leader.
Let’s Try to Solve Consensus!

• Uh, what’s the model? (assumptions!)

• Synchronous system: bounds on
  – Message delays
  – Max time for each process step
    e.g., multiprocessor (common clock across processors)

• Asynchronous system: no such bounds!
  e.g., The Internet! The Web!

• Processes can fail by stopping (crash-stop failures)
Consensus in a **Synchronous System**

- For a system with at most \( f \) processes crashing
  - All processes are synchronized and operate in “rounds” of time
  - the algorithm proceeds in \( f+1 \) rounds (with timeout), using reliable communication to all members
- \( Values^r_i \): the set of proposed values known to \( P_i \) at the beginning of round \( r \).
- Initially \( Values^0_i = {} \); \( Values^1_i = \{v_j\} \)

  for round = 1 to \( f+1 \) do
    multicast \( (Values^r_i - Values^{r-1}_i) \)
    \( Values^{r+1}_i \leftarrow Values^r_i \)
  for each \( v_j \) received
    \( Values^{r+1}_i = Values^{r+1}_i \cup v_j \)
  end
  end

\( d_i = \text{minimum}(Values^{f+1}_i) \)
Why does the Algorithm Work?

- Proof by contradiction.
- Assume that two non-faulty processes, say \( p_i \) and \( p_j \), differ in their final set of values (i.e., after \( f+1 \) rounds)
- Assume that \( p_i \) possesses a value \( v \) that \( p_j \) does not possess.
  - \( p_i \) must have received \( v \) in the last round (why?)
  - A third process, \( p_k \), sent \( v \) to \( p_i \), and crashed before sending \( v \) to \( p_j \).
  - Similarly, a fourth process sending \( v \) in the last-but-one round must have crashed; otherwise, both \( p_k \) and \( p_j \) should have received \( v \).
  - Proceeding in this way, we infer at least one (unique) crash in each of the preceding rounds.
  - But we have assumed at most \( f \) crashes can occur and there are \( f+1 \) rounds \( \rightarrow \) contradiction.
Consensus in an Asynchronous System

• Impossible to achieve!
  – even a single failed process is enough to avoid the system from reaching agreement

• Proved in a now-famous result by Fischer, Lynch and Patterson, 1983 (FLP)
  – Stopped many distributed system designers dead in their tracks
  – A lot of claims of “reliability” vanished overnight
Recall

• Each process \( p \) has a state
  – program counter, registers, stack, local variables
  – input register \( x_p \) : initially either 0 or 1
  – output register \( y_p \) : initially \( b \)

• Consensus Problem: design a protocol so that either
  – all processes set their output variables to 0
  – Or all processes set their output variables to 1

• For impossibility proof, OK to consider (i) more restrictive system model, and (ii) easier problem
Global Message Buffer

send(p', m)

receive(p')

may return null

“Network”
- State of a process
- **Configuration**=global state. Collection of states, one for each process; and state of the global buffer.
- Each **Event** (different from Lamport events)
  - receipt of a message by a process (say p)
  - processing of message (may change recipient’s state)
  - sending out of all necessary messages by p
- **Schedule**: sequence of events
Configuration C

Event $e' = (p', m')$

Schedule $s = (e', e'')$

Event $e'' = (p'', m'')$

Equivalent
Lemma 1

Disjoint schedules are commutative

s1 and s2 involve disjoint sets of receiving processes

Schedule s1

Schedule s2
• Let config. C have a set of decision values V reachable from it
  – If |V| = 2, config. C is bivalent
  – If |V| = 1, config. C is 0-valent or 1-valent, as is the case

• Bivalent means outcome is unpredictable
What the FLP Proof Shows

1. There exists an initial configuration that is bivalent
2. Starting from a bivalent config., there is always another bivalent config. that is reachable
Lemma 2

Some initial configuration is bivalent

• Suppose all initial configurations were either 0-valent or 1-valent.
• If there are N processes, there are $2^N$ possible initial configurations.
• Place all configurations side-by-side (in a lattice), where adjacent configurations differ in initial $x_p$ value for exactly one process.

• There has to be some adjacent pair of 1-valent and 0-valent configs.
Lemma 2

Some initial configuration is bivalent

• There has to be some adjacent pair of 1-valent and 0-valent configs.
• Let the process $p$ that has a different state across these two configs. be the process that has crashed (silent throughout)

Both initial configs. will lead to the same config. for the same sequence of events

Therefore, at least one of these initial configs. are bivalent when there is such a failure
What we’ll Show

1. There exists an initial configuration that is bivalent

2. Starting from a bivalent config., there is always another bivalent config. that is reachable
Lemma 3

Starting from a bivalent config., there is always another bivalent config. that is reachable
Lemma 3

A bivalent initial config.

let e=(p,m) be an applicable event to the initial config.

Let $C$ be the set of configs. reachable without applying e
Lemma 3

A bivalent initial config.

let e=(p,m) be an applicable event to the initial config.

Let $C$ be the set of configs. reachable without applying $e$

Let $D$ be the set of configs. obtained by applying $e$ to some config. in $C$
Lemma 3

bivalent

[don’t apply event e=(p,m)]
**Claim.** $D$ contains a bivalent config.

**Proof.** By contradiction.

1. $D$ contains both 0- and 1-valent configurations (why?)
2. There are states $C_0$ and $C_1$ in $C$ such that $C_1 = C_0$ followed by some event $e' = (p', m')$ and
   - $D_0$ is 0-valent, $D_1$ is 1-valent
   - $D_0 = C_0$ foll. by $e = (p, m)$
   - $D_1 = C_1$ foll. by $e = (p, m)$
Proof. (contd.)

- Case I: \( p' \) is not \( p \)
- Case II: \( p' \) same as \( p \)

Why? (Lemma 1)
But \( D_0 \) is then bivalent!
Proof. (contd.)

- Case I: $p'$ is not $p$
- Case II: $p'$ same as $p$

But $A$ is then bivalent!
Lemma 3

Starting from a bivalent config., there is always another bivalent config. that is reachable
Putting it all Together

- Lemma 2: There exists an initial configuration that is bivalent
- Lemma 3: Starting from a bivalent config., there is always another bivalent config. that is reachable

- Theorem (Impossibility of Consensus): There is always a run of events in an asynchronous distributed system such that the group of processes never reach consensus (i.e., stays bivalent all the time)
Summary

• **Consensus Problem**
  – agreement in distributed systems
  – Solution exists in synchronous system model (e.g., supercomputer)
  – Impossible to solve in an asynchronous system (e.g., Internet, Web)
    • Key idea: with even one (adversarial) crash-stop process failure, there are always sequences of events for the system to decide any which way
      • Whatever algorithm you choose!
    – FLP impossibility proof