Consistency in Distributed Systems

Sami Rollins

Replication

- **In cloud infrastructure services, replication is used ubiquitously to guarantee consistent performance and high availability**
  - Vogel's Eventually Consistent article

- **An early Google paper discussed their philosophy of using cheap/commodity hardware and building reliability in software**
  - Assume failure is the norm and program the system to account for that failure

The C in CAP

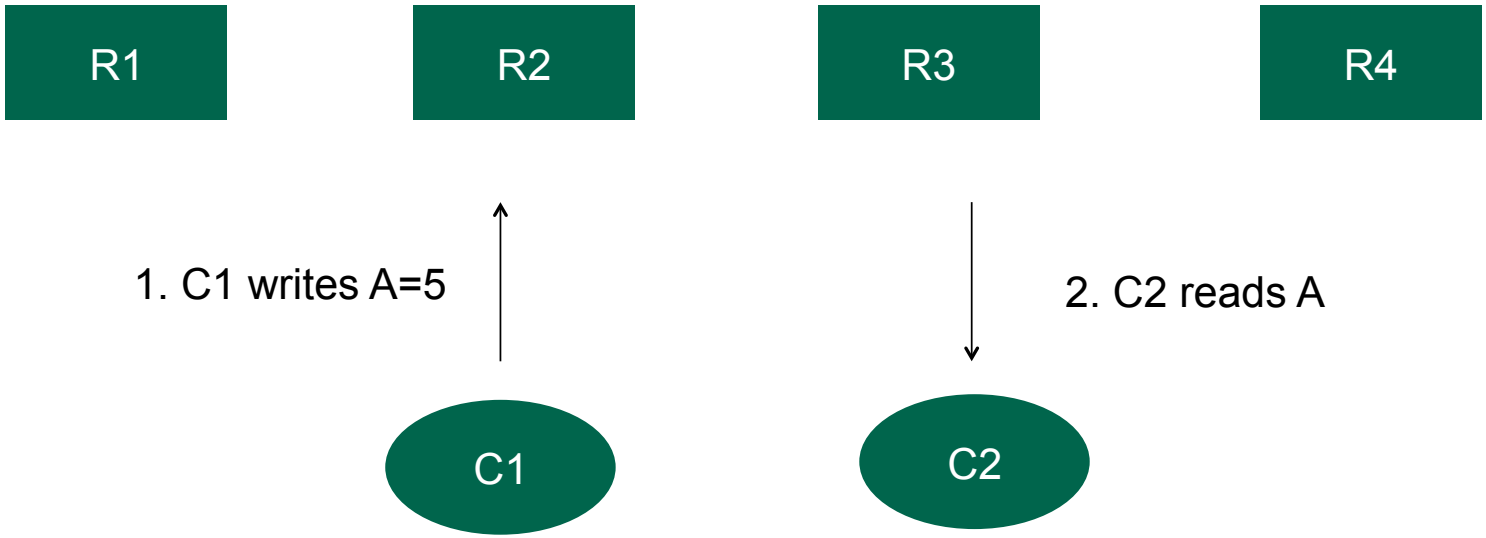
- **Why maintain multiple copies of the data?**

The C in CAP

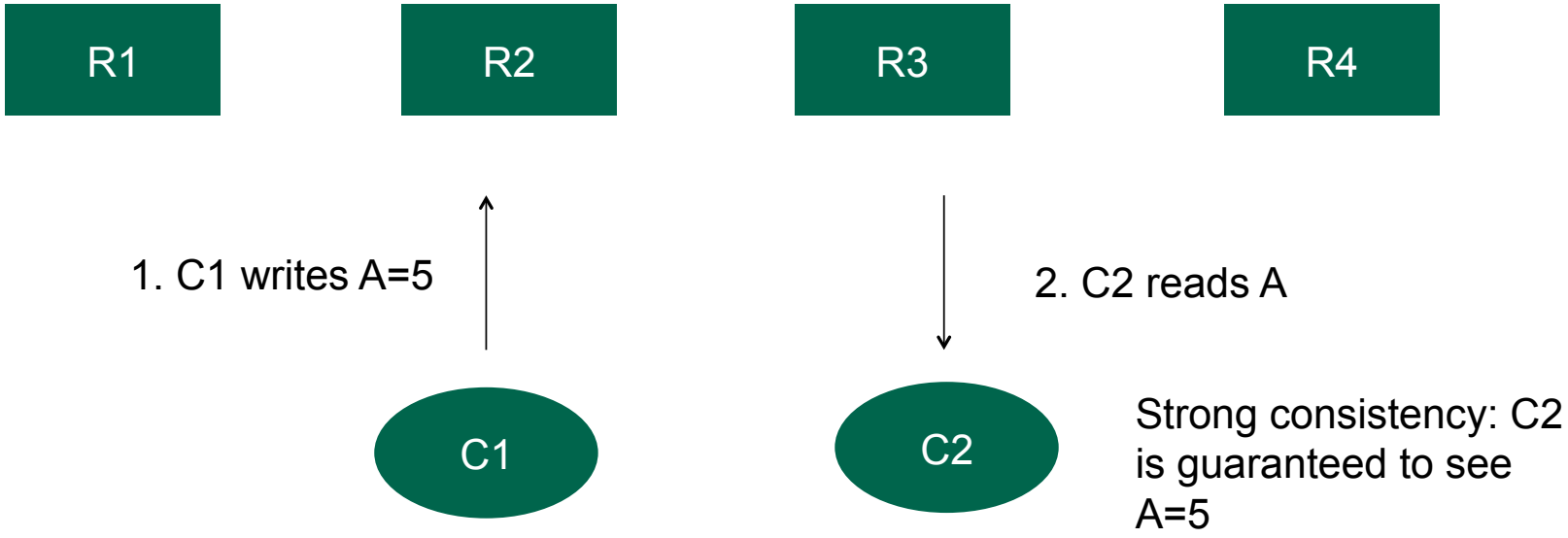
- **Why maintain multiple copies of the data?**
  - Recover from failure
    - For this case, could just have a “backup” copy
  - Decrease latency

- **How do we provide consistency?**

Consistency Models



Strong Consistency



Strong Consistency

- **How would you implement strong consistency?**

Strong Consistency

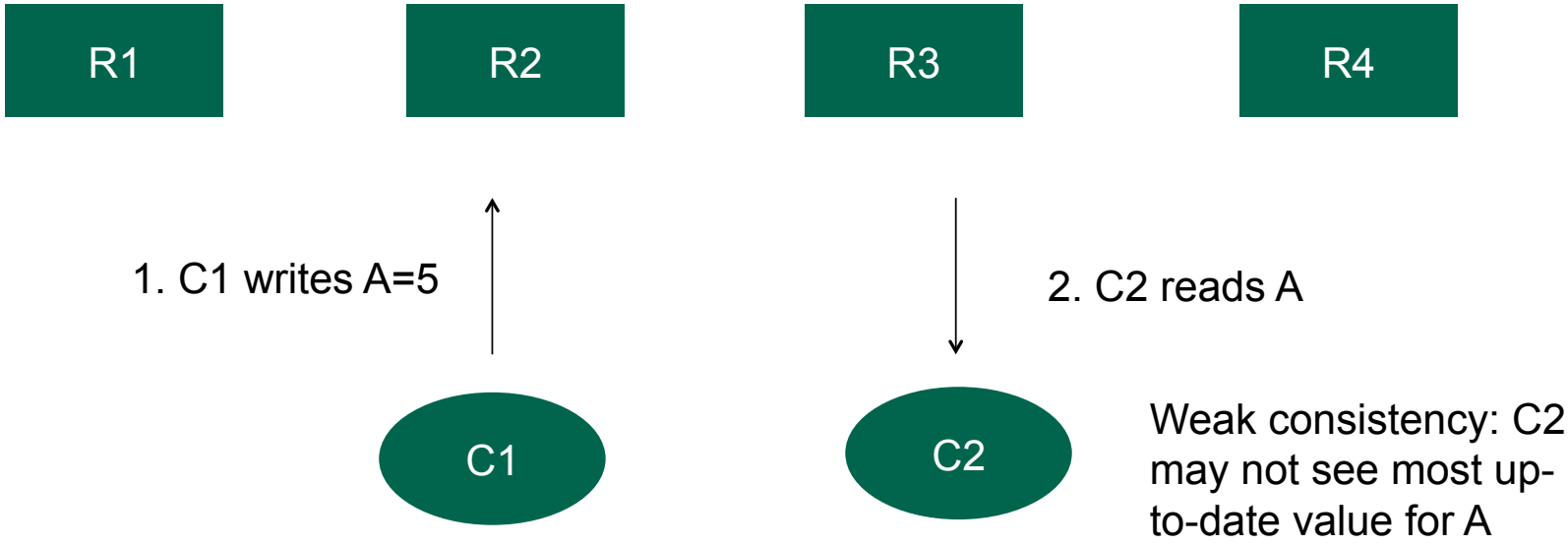
- **How would you implement strong consistency?**

- **Passive replication**
  - All requests go through a single, primary server and the primary copies to all secondaries before replying to the client
- **Active replication**
  - Any replica can accept a write, however it must get all other replicas to reliably apply the write before replying to the client

Strong Consistency

- **Disadvantages?**

Weak Consistency



Weak Consistency

- Good idea?
- Appropriate applications?

Eventual Consistency

- **Eventually, all replicas will have the most up-to-date information**

- **Often uses *lazy* replication**
  - Any replica accepts the write
  - Writes are propagated asynchronously
  - A read may not see the most recent version of the data, but will eventually

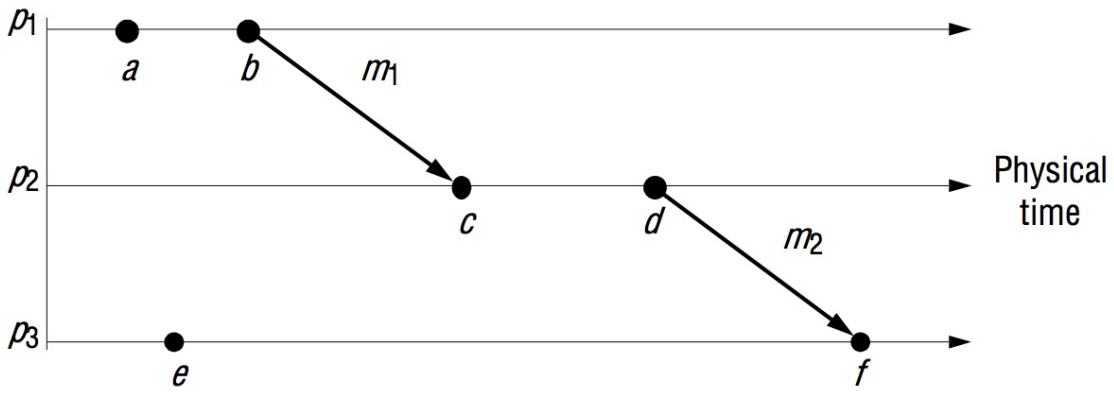
Causal Consistency

- **Ensure that any read sees all writes that *happened before***

Logical Time

- **For many applications, the *causal ordering* of events is most important. The happened-before relation (→) is denoted as follows:**
  - If two events occurred at the same process p_i then they occurred in the order in which p_i observes them.
  - For any message m send(m) →receive(m)
  - If e→e' and e' →e'' then e→e''
- **Events that are not ordered by → are concurrent – a || e**

Logical Time

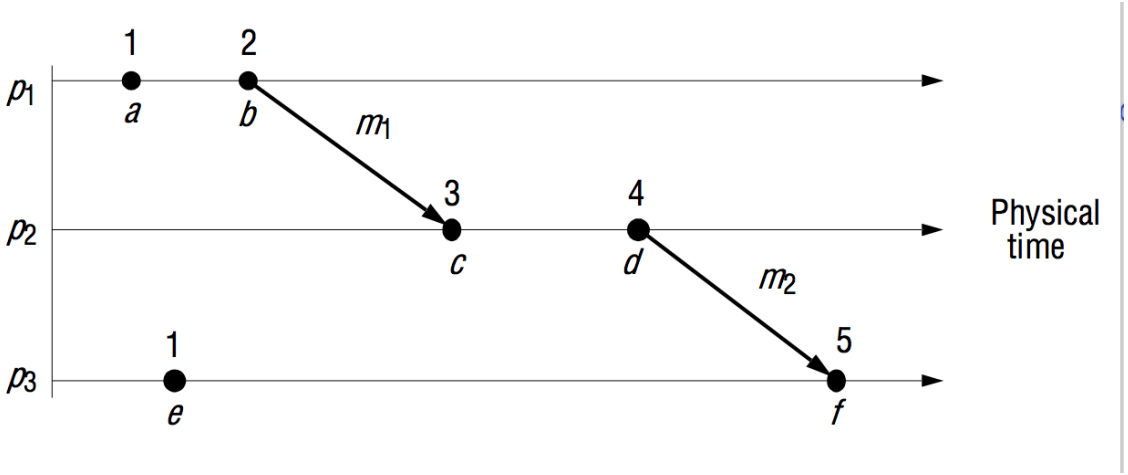


- b→c, c→d, a||e
- Any other → or || relationships?

Logical Clocks

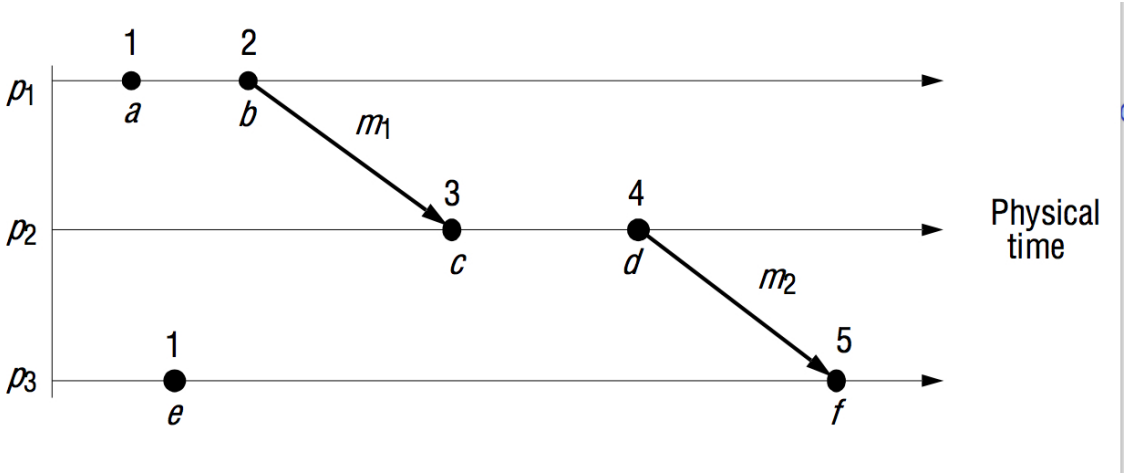
- **Lamport timestamps**
- Monotonically increasing counter maintained at each process and updated according to the following rules:
  - LC1: L_i is incremented before each event is issued at process p
  - LC2:
    - When a process sends a message *m* it piggybacks its value of $t = L_i$
    - On receiving (*m, t*), a process computes $L_j := \max(L_j, t)$ and applies LC1 before timestamping the event receive(m)

Logical Clocks



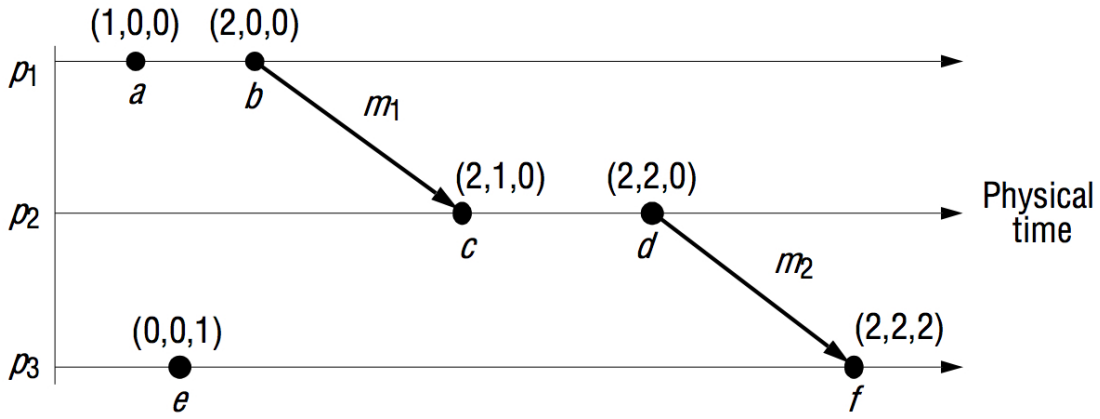
- If e→e' then L(e) < L(e')
- If L(e) < L(e') then can we say e→e'?

Logical Clocks



- **If e→e' then L(e) < L(e')**
- **If L(e) < L(e') then can we say e→e'?**
  - No! 1 < 3 but we can't say anything about the relationship between e and c in the example above.

Vector Clocks



Causal Consistency

- ***Client* specifies the last timestamp seen**
- **Replica provides data at timestamp that is concurrent with or happened after the last timestamp seen**

- **Read-my-write consistency**
  - Similar to causal consistency, but guarantees that the client sees its own writes