Distributed Systems: Ordering and Consistency

October 11, 2018

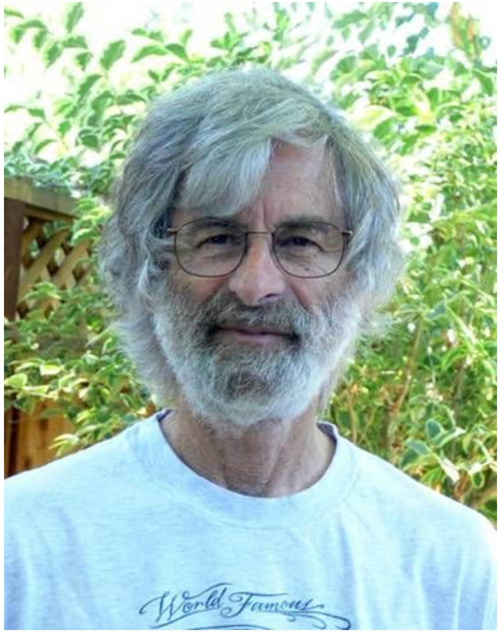
A.F. Cooper

Context and Motivation

- How can we synchronize an asynchronous distributed system?
- How do we make global state consistent?
- Snapshots / checkpoints
- Example: Buying a ticket on Ticketmaster

Leslie Lamport

- MIT / Brandeis
- Industrial researcher
- “Father” of distributed computing
- Paxos
- “Time, Clocks, and the Ordering of Events in a Distributed System” (1978)
  - Test of time award
  - 11,082 citations (Google Scholar)
- Turing Award (2013) for LateX (notably, not for Paxos)
  - Ken Birman was the ACM chair when Paxos paper submitted



Takeaways

- What is time?
- What does time mean in a distributed system?
- In a distributed system, how do we order events such that we can get a consistent snapshot of the entire system state at a point in time?
  - Happened before relation
  - Logical clocks, physical clocks
  - Partial and total ordering of events

Outline

- Model of distributed system
- Happened Before relation and Partial Ordering
- Logical Clocks and The Clock Condition
- Total Ordering
- Mutual Exclusion
- Anomalous Behavior
- Physical Clocks to Remove Anomalous Behavior

Outline

- **Model of distributed system**
- Happened Before relation and Partial Ordering
- Logical Clocks and The Clock Condition
- Total Ordering
- Mutual Exclusion
- Anomalous Behavior
- Physical Clocks to Remove Anomalous Behavior

Model of a Distributed System

Included:

- **Process**: Set of events, a priori total ordering (sequence)
- **Event**: Sending/receiving message
- **Distributed System**: Collection of processes, spatially separated, communicate via messages
  - How do you coordinate between isolated processes?

Not Included:

- Global clock

Outline

- Model of distributed system
- **Happened Before relation and Partial Ordering**
- Logical Clocks and The Clock Condition
- Total Ordering
- Mutual Exclusion
- Anomalous Behavior
- Physical Clocks to Remove Anomalous Behavior

Happened Before and Partial Ordering

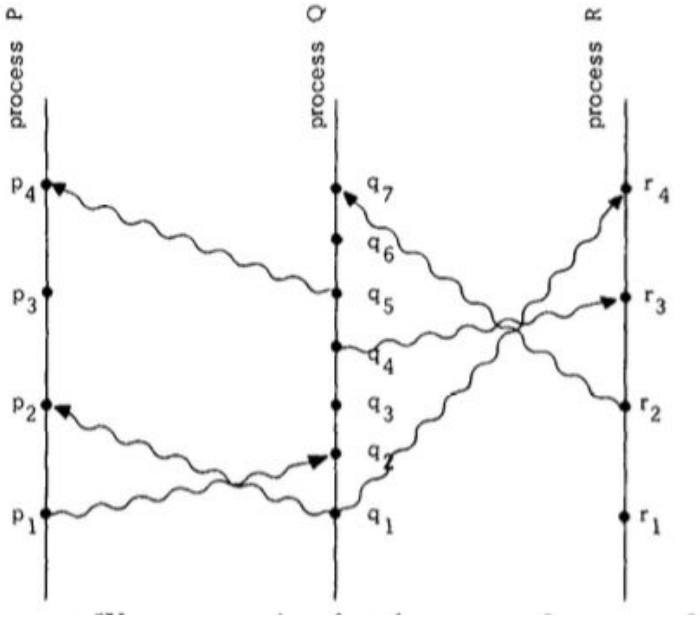
- Used to thinking about global clock time (a total order / timeline)
  - I read a recipe, then I cook dinner (in that order)
- Distributed systems
  - Events in multiple places
    - Everyone in class, each living in a tower
    - Communicate via letter
      - How do we know how letters ordered when sent?
  - Events can be concurrent
  - No global time-keeper
    - We talk about time in terms of “causality”
      - How can we decide we cooked dinner before reading a cookbook?
      - No order unless one event “caused” another
      - I cook dinner, I send a letter suggesting the cookbook I used, which “caused” another person to read the cookbook

Happened Before and Partial Ordering

Definition. The relation "$\rightarrow$" on the set of events of a system is the smallest relation satisfying the following three conditions: (1) If a and b are events in the same process, and a comes before b, then $a \rightarrow b$. (2) If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$. (3) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. Two distinct events a and b are said to be *concurrent* if $a \nrightarrow b$ and $b \nrightarrow a$.

Happened Before and Partial Ordering

- Another way to say "a happens before b" is to say that "a causally affects b"
- Concurrent events do not causally affect each other



Outline

- Model of distributed system
- Happened Before relation and Partial Ordering
- **Logical Clocks and The Clock Condition**
- Total Ordering
- Mutual Exclusion
- Anomalous Behavior
- Physical Clocks to Remove Anomalous Behavior

Logical Clocks and the Clock Condition

- We need to assign a sort of "timestamp" to events to order them
- We therefore need a clock (of some kind)
  - Earlier example: What "time" did I eat dinner? What "time" did you read the cookbook?
- A logical clock assigns a "timestamp" (a counter) to events

Logical Clocks and the Clock Condition

- A counter, rather than a real timestamp
- No relation to physical time (for now)

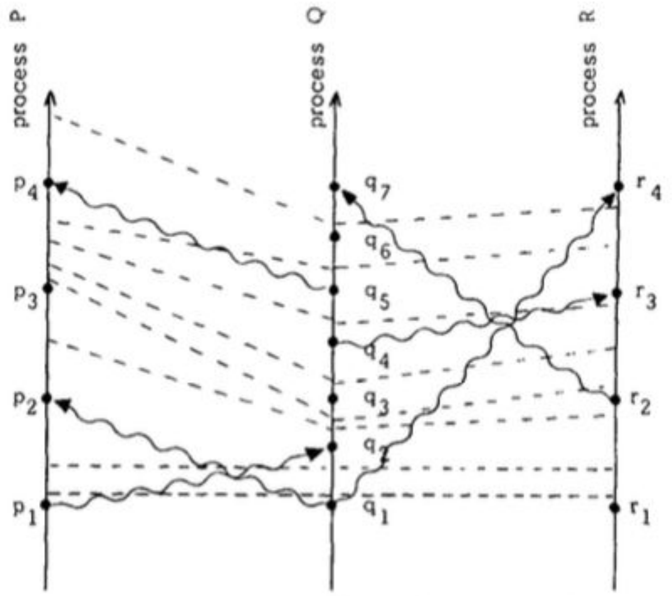
> More precisely, we define a clock C_i for each process P_i to be a function which assigns a number $C_i\langle a \rangle$ to any event a in that process. The entire system of clocks is represented by the function C which assigns to any event b the number $C\langle b \rangle$, where $C\langle b \rangle = C_j\langle b \rangle$ if b is an event in process P_j.

Logical Clocks and the Clock Condition

C1. If a and b are events in process P_i, and a comes before b, then $C_i\langle a \rangle < C_i\langle b \rangle$.

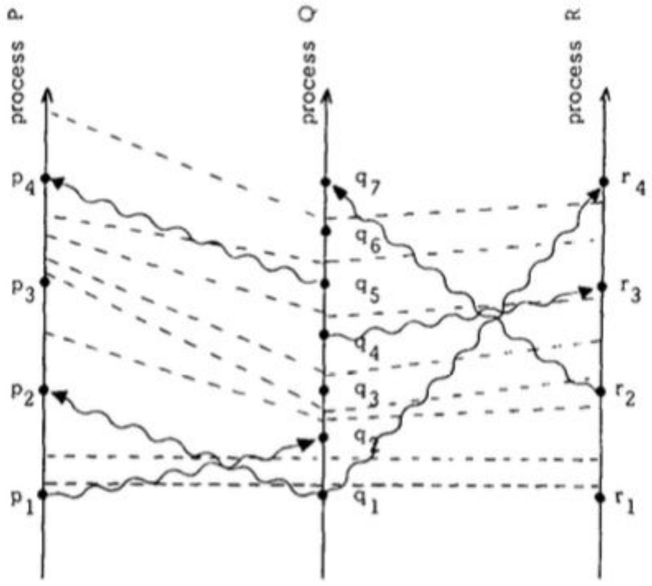
C2. If a is the sending of a message by process P_i and b is the receipt of that message by process P_j, then $C_i\langle a \rangle < C_j\langle b \rangle$.

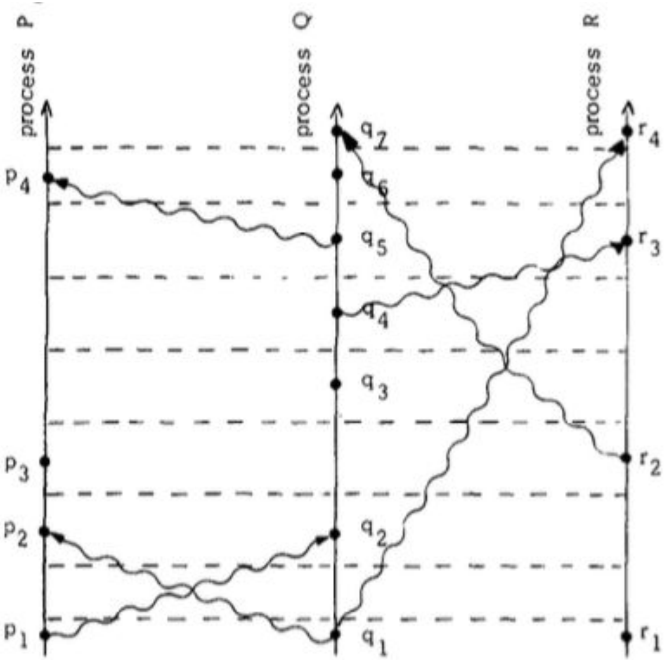
Fig. 2.



Logical Clocks and the Clock Condition

Fig. 2.





Logical Clocks and the Clock Condition

LC1: $\hat{T}_p$ is incremented after each event at p.

LC2: Upon receipt of a message with timestamp τ, process p resets $\hat{T}_p$:

$$\hat{T}_p := \max(\hat{T}_p, \tau) + 1.$$

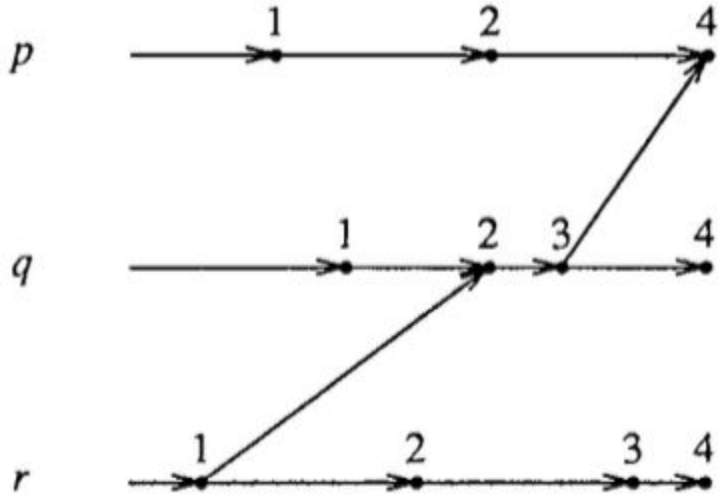


Figure 4. Logical clock example.

Outline

- Model of distributed system
- Happened Before relation and Partial Ordering
- Logical Clocks and The Clock Condition
- **Total Ordering**
- Mutual Exclusion
- Anomalous Behavior
- Physical Clocks to Remove Anomalous Behavior

Total Ordering

- Need a total order that everyone can agree on
  - May not reflect “reality”
  - I ate first or second, you read cookbook first or second, or concurrently
- Order events by the time at which they occur
- Break ties semi-arbitrarily (by process id -- establish a priority among processes)
- Not unique; depends on system of clocks

To break ties, we use any arbitrary total ordering $\prec$ of the processes. More precisely, we define a relation $\Rightarrow$ as follows: if a is an event in process P_i and b is an event in process P_j, then $a \Rightarrow b$ if and only if either (i) $C_i\langle a \rangle < C_j\langle b \rangle$ or (ii) $C_i\langle a \rangle = C_j\langle b \rangle$ and $P_i \prec P_j$. It is easy to see that this defines a total ordering, and that the Clock Condition implies that if $a \rightarrow b$ then $a \Rightarrow b$. In other words, the relation $\Rightarrow$ is a way of completing the "happened before" partial ordering to a total ordering.[3]

Outline

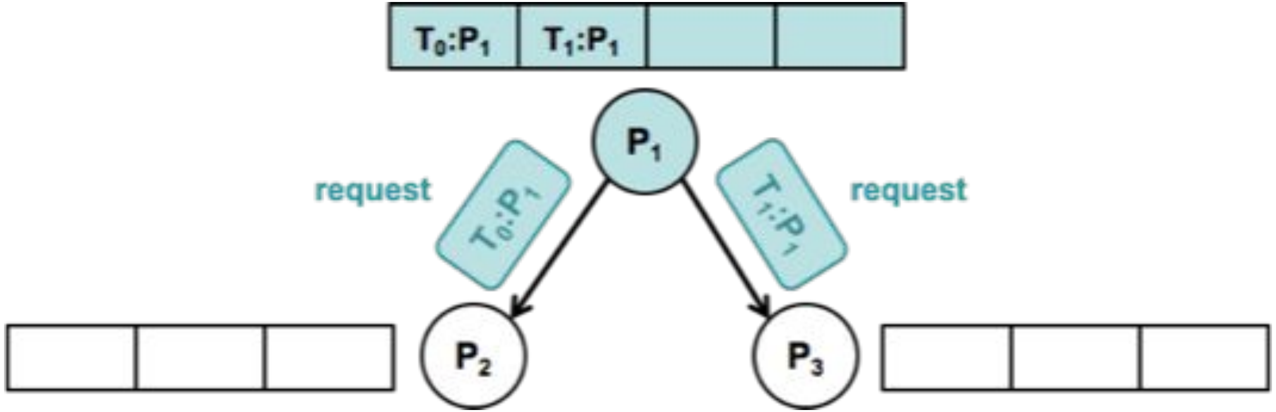
- Model of distributed system
- Happened Before relation and Partial Ordering
- Logical Clocks and The Clock Condition
- Total Ordering
- **Mutual Exclusion**
- Anomalous Behavior
- Physical Clocks to Remove Anomalous Behavior

Mutual Exclusion

- Single resource, many processes
- Only one process can access resource at a time
  - E.g., only one process can send to a printer at a time
- Synchronize access
- FIFO granting / releasing of access to resource
- If every process granted the resource *eventually* releases it, then every request is *eventually* granted (we'll come back to this "*eventually*")

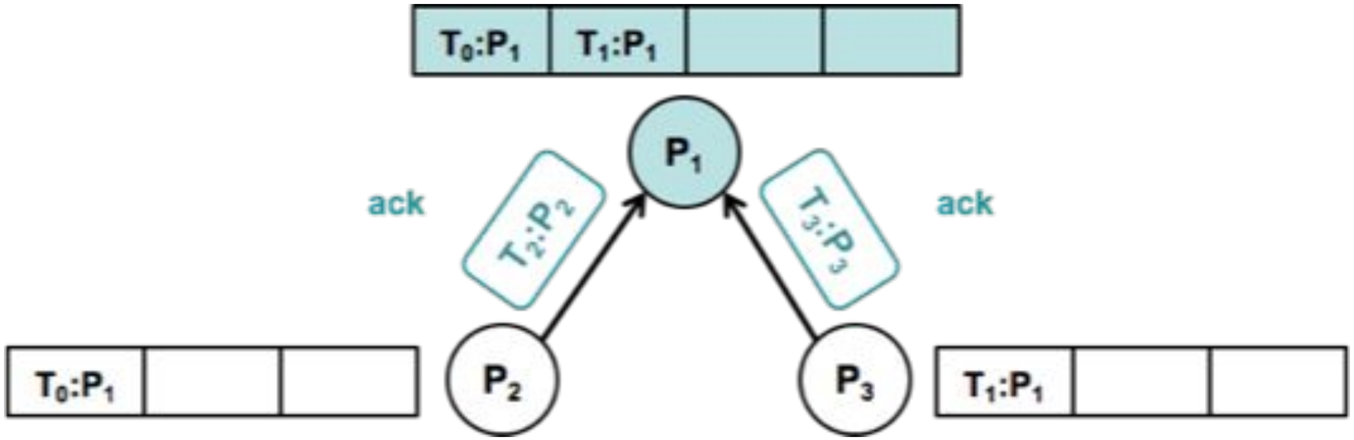
Mutual Exclusion

1. To request the resource, process P_i sends the message T_m:P_i *requests resource* to every other process, and puts that message on its request queue, where T_m is the timestamp of the message.



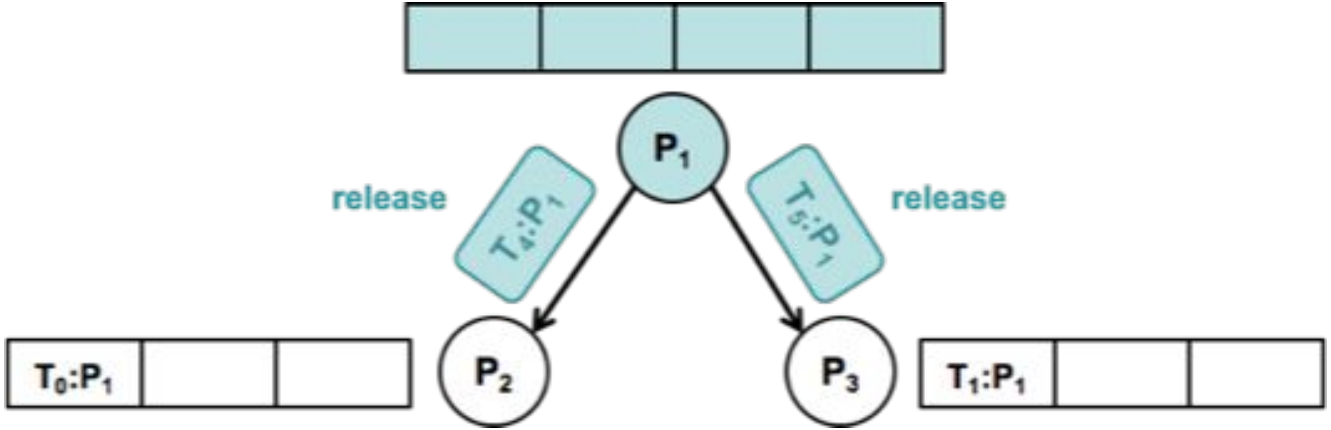
Mutual Exclusion

2. When process P_j receives the message $T_m{:}P_i$ *requests resource*, it places it on its request queue and sends a (timestamped) acknowledgment message to P_i.[5]



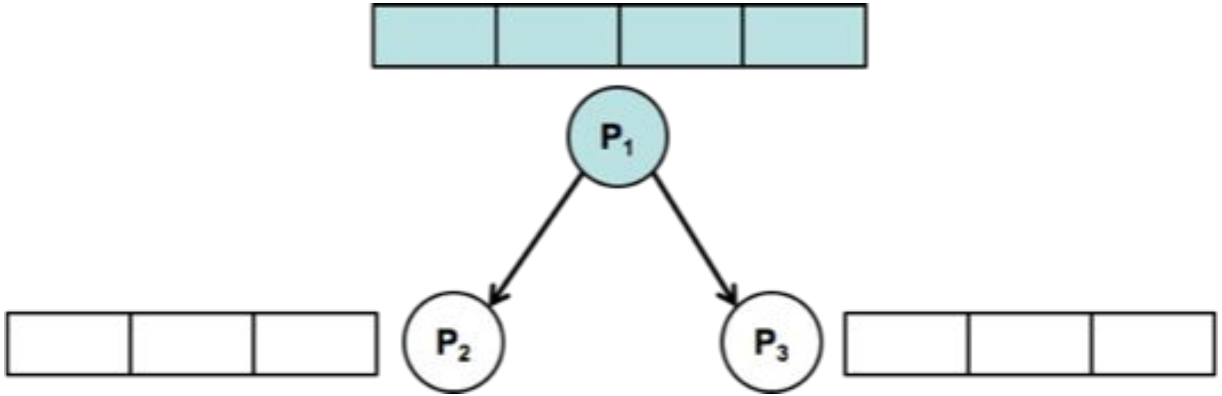
Mutual Exclusion

3. To release the resource, process P_i removes any T_m:P_i *requests resource* message from its request queue and sends a (timestamped) P_i *releases resource* message to every other process.



Mutual Exclusion

4. When process P_j receives a P_i *releases resource* message, it removes any T_m:P_i *requests resource* message from its request queue.



Mutual Exclusion

- Distributed algorithm
  - No centralized synchronization
- State Machine specification
  - Set of commands (C), set of states (S)
  - Relation that executes on a command and a state, returns a new state
    - Prior example:
      - Commands: Request resource, release resource
      - States: Queue of waiting request and release commands
- Synchronization because of total order according to timestamps
- Failure not considered

Outline

- Model of distributed system
- Happened Before relation and Partial Ordering
- Logical Clocks and The Clock Condition
- Total Ordering
- Mutual Exclusion
- **Anomalous Behavior**
- Physical Clocks to Remove Anomalous Behavior

Anomalous Behavior

- Imagine a game of telephone
  - Person A -- issues request on computer (A)
  - Person A telephones person B (in another city)
  - Person A tells Person B to issue a different request on computer (B)
- Anomalous result
  - Person B's request can have a lower timestamp than A
  - B can be ordered before A
  - A preceded B, but the system has no way to know this
- Precedence information is based on messages external to system

Strong Clock Condition

Strong Clock Condition. For any events a, b in $\mathcal{S}$:

$$\text{if } a \boldsymbol{\rightarrow} b \text{ then } C\langle a\rangle < C\langle b\rangle.$$

This is stronger than the ordinary Clock Condition because $\boldsymbol{\rightarrow}$ is a stronger relation than $\rightarrow$. It is not in general satisfied by our logical clocks.

Outline

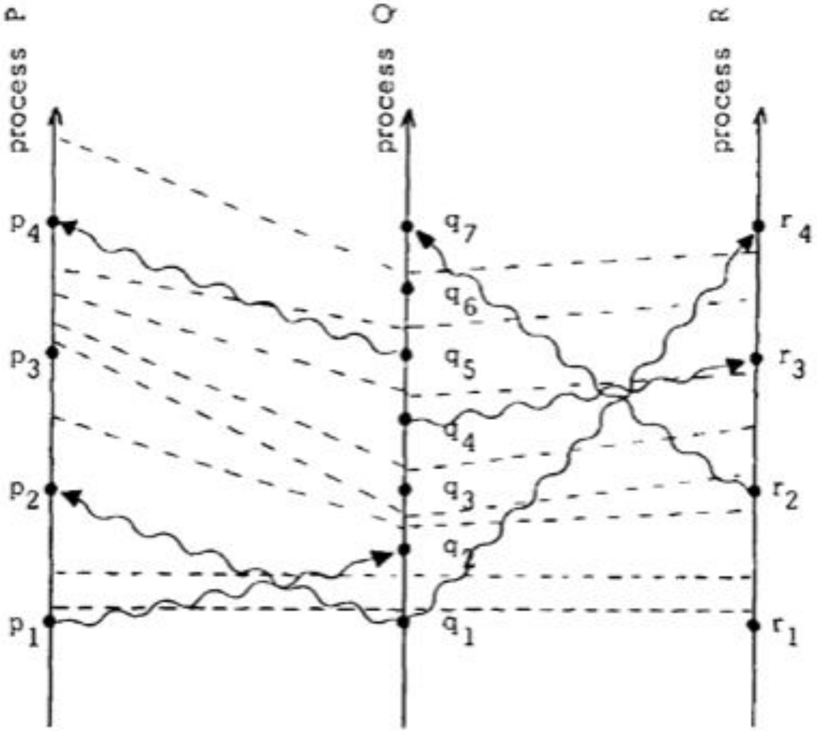
- Model of distributed system
- Happened Before relation and Partial Ordering
- Logical Clocks and The Clock Condition
- Total Ordering
- Mutual Exclusion
- Anomalous Behavior
- **Physical Clocks to Remove Anomalous Behavior**

Physical Clocks

- Introduce physical time to our clocks
- Needs to run at approximately correct rate
  - Clocks can’t get too out-of-synch
- We put bounds on how out-of-synch clocks relative to each other

Physical Clocks

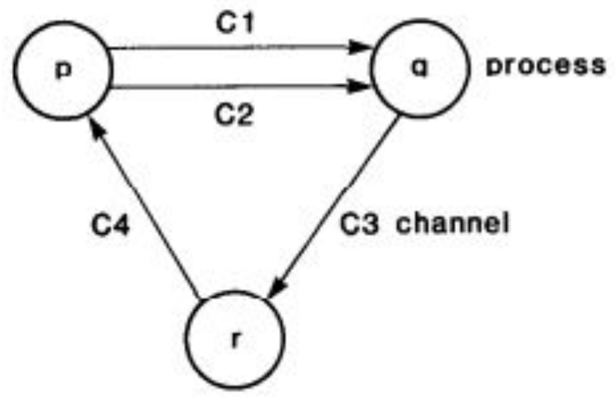
Fig. 2.



Impact: Global State Intuition

Global State Detection and Stable Properties

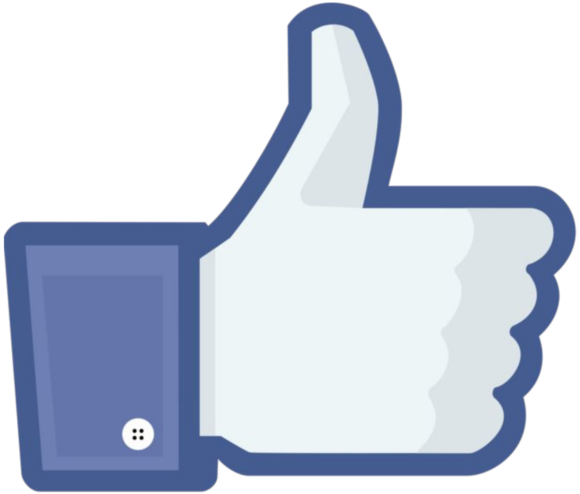
- Must not affect underlying computation
- Stable property detection
  - Computation terminated
  - System deadlocked
- Consistent cuts
  - Checkpoint / facilitating error recovery
- Algorithm components
  - Cooperation of processes
  - Token passing



Drawbacks -- “Eventually”

- CAP
  - Consistency
  - Availability
  - Partition Tolerance
- COPS
  - Clusters of Order-Preserving Services
  - Don’t settle for eventual
  - Causal+ consistency
  - ALPS
    - Availability
    - (Low) Latency
    - Partition Tolerance
    - Scalability

> If every process which is granted the resource eventually releases it, then every request is eventually granted.

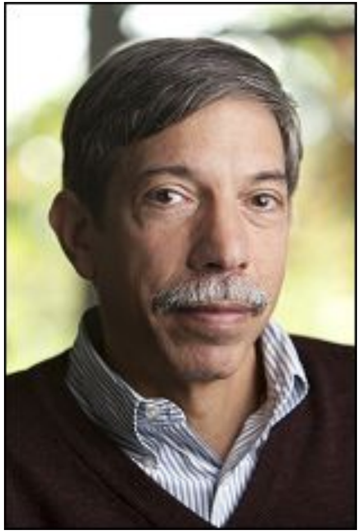


Drawbacks -- Handling Failures

- Byzantine generals problem
- How do reliable computer systems handle failing components?
  - Particularly, components giving conflicting information
- Majority voting
  - “Commander” - input generator
  - “Generals” - processors (loyal ones are non-faulty)

Drawbacks -- Handling Failures

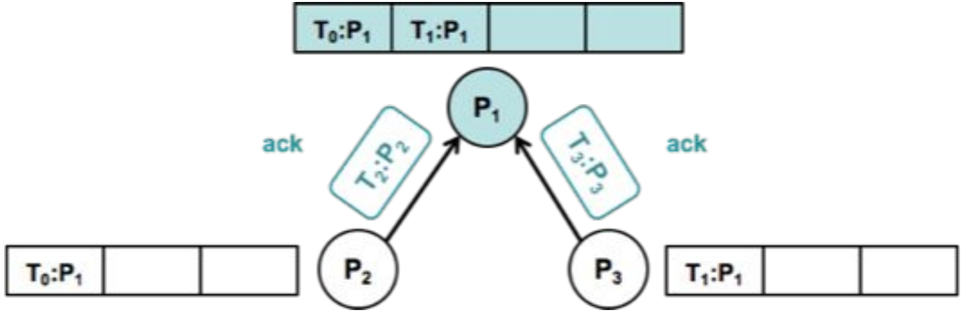
- Implementing fault-tolerant services using the State Machine Approach
- Byzantine failure and fail-stop
- Service only as tolerant as processor executing →
  - Replicas (multiple servers that fail independently)
  - Coordination between replicas
- State machine
  - State variables
  - Commands



Fred Schneider

Drawbacks -- Every Process

- Process must communicate with all other processes
- Schneider deals with this
    - Replica-generated identifier approach
        - Next class
        - Nutshell: Communication only between processors running the client and SM replicas



Drawbacks -- Implementation

- Theory only
  - Useful for reasoning about distributed systems
  - But, gap between theory and practice
- Modern distributed systems require more
  - Physical time
  - Network Time Protocol (NTP) syncing

Other Types of Clocks

- 1988: Vector clocks (DynamoDB)
- 2012: TrueTime (Spanner)
- 2014: Hybrid Logical Clocks (CockroachDB)
- 2018: Sync NIC clocks (Huygens)

Referenced Works

- Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, Volume 21, Number 7, 1978.
- K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, Volume 3, Number 1, 1985.
- K. Mani Chandy and Jayadev Misra. How Processes Learning. *ACM*, 1985.
- Leslie Lamport, et. al. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, Volume 4, Number 3, 1982.
- Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, Volume 22, Number 4, 1990.
- Sandeep S. Kulkarni, et. al. Logical Physical Clocks. M. *Principles of Distributed Systems*, 2014
- Wyatt Lloyd, et. al. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. *SOSP*, 2011.
- Yilong Geng, et. al. Exploiting a Natural Network Effect for Scalable Fine-grained Clock Synchronization. *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*, 2018.

Questions?

- How can we conceive of synchronization in modern, heterogeneous data centers?
- How can we achieve synchronization using commodity hardware
- What does “consistency” even mean as we move toward real-time computing?