EECS 262a
Advanced Topics in Computer Systems
Lecture 14

Lamport Clocks and OCC
March 9th, 2016

John Kubiatowicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~kubitron/cs262

Today's Papers

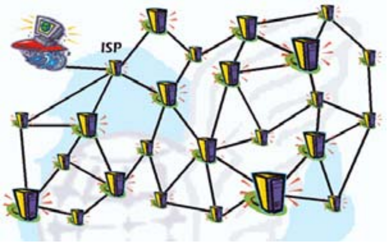
- **Time, Clocks, and the Ordering of Events in a Distributed System**
  Leslie Lamport. Appears in *Communications of the ACM,* Vol 21, No. 7, pp 558-565, July 1978
- **Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks**
  Atul Adya, Robert Gruber, Barbara Liskov, Umesh Maheshwari. Appears in *Proceedings of ACM SIGMOD international conference on Management of Data,*1995

- Thoughts?

Time

- **One dimension. It can not move backward. It can not stop.**
- **It is derived from concept of the order in which events occur.**
- **The concepts "before" and "after" need to be reconsidered in a distributed system.**

Distributed System



- **A distributed system consists of collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages.**
- **It could be a network of interconnected computers, like Internet, or just a single computer with separate processes.**
- **It is sometimes impossible to say that one of two events occurred first in a distributed system. "happened before" is a partial ordering of the events in the system.**

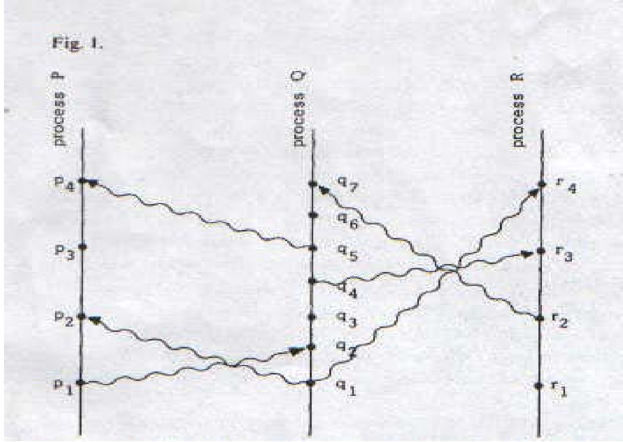
The Partial Ordering

- The system described in paper:
  - System is composed of a collection of processes.
  - Each process consists of a sequence of events.
  - A single process is defined to be a set of events with an a priori total ordering
- Events:
  - Program Events
  - Message Events
- Messages carry dependencies between processes

Definition of “happened before”

- The relation “→” 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→b.
  2. If a is the sending of a message by one process and b is the receipt of the same message by another processes, then a→b.
  3. If a→b and b→c then a→c.
- Two distinct events a and b are said to be concurrent if a —/→b and b —/→ a.
- We assume that a —/→ a for any event a.

Space-time diagram

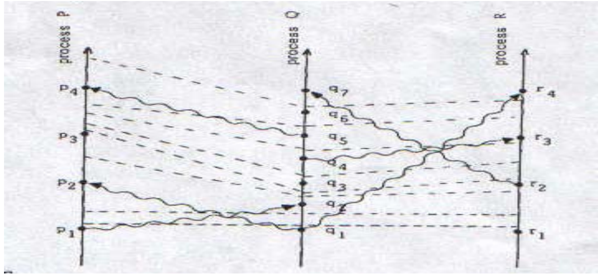


- $p_1 \rightarrow r_4$ since $p_1 \rightarrow q_2$ and $q_2 \rightarrow q_4$ and $q_4 \rightarrow r_3$ and $r_3 \rightarrow r_4$
- p_3 and q_3 are concurrent.

Logical Clocks

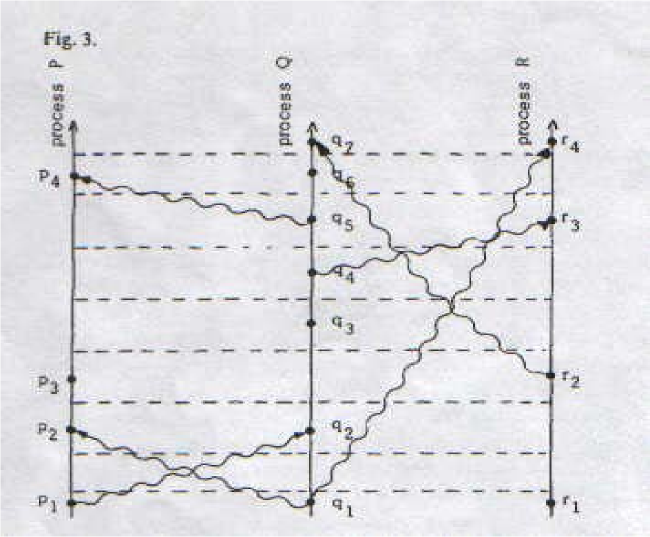
- A clock is just a way of assigning a number to an event
  - Monotonically increasing except when reset
  - Not necessarily related to “real time” in any particular frame
- Definition of logical clocks:
  - A clock C_i for each process P_i is a function which assigns a number C_i<a> to any event a in that process.
  - The entire system of clocks is represented by the function C which assigns to any event b the number C<b> = C_j<b> if b is an event b in process P_j.

Clock Condition



- For any events a, b: if a→ b then C<a> < C<b>
- Clock Condition is satisfied if
  - C1: If a and b are events in P_i, and a comes before b, then C_i<a> < C_i<b>
  - C2: If a is the sending of a message by process Pi and b is the receipt of that message by process P_j, then C_i<a> < C_j<b>
- C1 means that there must be a tick line between any two events on a process line
- C2 means that every message line must cross a tick line.

Redraw Previous Figure

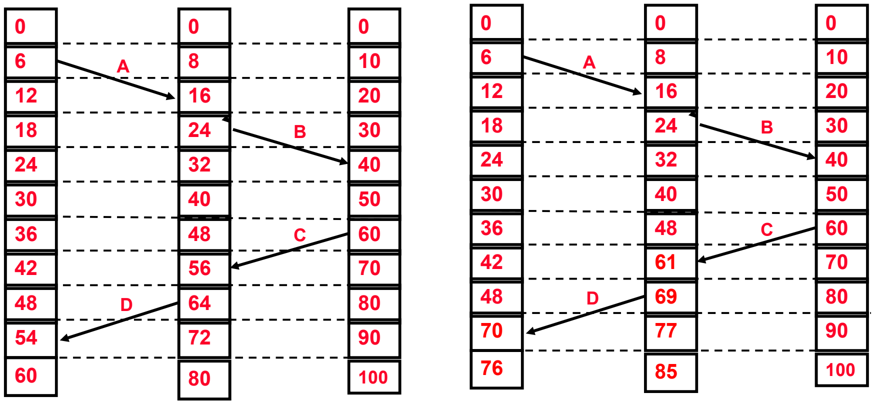


Clock Condition

Now assume that the processes are algorithms, and the events represent certain actions during their execution. Process P_i's clock is represented by a register C_i, so that C_i<a> is the value contained by C_i during the event a.

- To meet condition C1 and C2, the processes need to obey the following rules:
  - IR1: Each process P_i increments C_i between any two successive events.
  - IR2: (a) If event a is the sending of a message m by process P_i, then the message m contains a timestamp T_m=C_i<a>. (b) Upon receiving a message m, process P_j sets C_j greater than or equal to its present value and greater than T_m.

Partial Ordering: Unregulated Clocks



0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50
36	48	60
42	56	70
48	64	80
54	72	90
60	80	100

0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50
36	48	60
42	61	70
48	69	80
70	77	90
76	85	100

- Version on Left has message "D" appearing to take negative time!
- With Clock adjustment clause (IR2b), fixed on right

Definition of total ordering “⇒”

- We can use a system of clocks satisfying the Clock Condition to place a total ordering on the set of all events:
  - We simply order the events by the times at which they occur
  - To break ties, we use any arbitrary total ordering ~< of the processes.
- Definition of total ordering “⇒”
  - If a is an event in process P_i and b is an event in process P_j, then a⇒b if and only if either
    (i) $C_i(a)<C_j(b)$ or (ii) $C_i(a) = C_j(b)$ and P_i ~< P_j.
  - Clock Condition implies that if a → b then a⇒b.
    In other words, the relation ⇒ is a way of completing the “happened before” partial ordering to a total ordering
- The ordering “⇒” depends upon the clock systems and is not unique!
  - Example: If we have $C_i(a) = C_j(b)$ and choose P_i ~< P_j, then a ⇒ b. If we choose P_j ~< P_i, then b ⇒ a
- The partial ordering “→” is uniquely determined by the system of events.

Solving Mutual Exclusion

- Mutual exclusion: Only one process can use the resource at a time, the other processes will be excluded from doing the same thing
- Requirements:
  1. A process which has been granted the resource must release it before it can be granted to another process.
  2. Different requests for the resource must be granted in the order in which they are made.
  3. If every process which is granted the resource eventually release it, then every request is eventually granted.
- How to do it with clocks: Implement Clocks as above, define “⇒”
- Assumptions:
  1. For any two processes Pi and Pj, the messages sent from Pi to Pj are received in the same order as they are sent.
  2. Every message is eventually received.
  3. A process can send messages directly to every other process.
  4. Each process maintains its own request queue which is never seen by any other process. The request queues initially contain the single message T0:P0 requests resource.

Mutual Exclusion Algorithm

- 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
- 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.
- To release the resource, process P_i removes any T_m:P_i request resource message from its request queue and sends a (timestamped)
  “P_i releases resource”
  messages to every other process.

Mutual Exclusion Algorithm (Con’t)

- When process P_j receives a
  “P_i release resource”
  message, it removes any T_m:P_i requests resource message from its request queue.
- Process P_i is granted the resource when the following two conditions are satisfied:
  1. There is a T_m:P_i request resource message in its request queue which is ordered before any other request in its queue by the relation ⇒.
  2. P_i has received a message from every other process time-stamped later than T_m.

Anomalous Behavior (External Channels)

- Consider a nationwide system of interconnected computers. Suppose a person issues a request A on a computer A, and then telephones a friend in another city to have him issue a request B on a different computer B. It is quite possible for a request B to receive a lower timestamp and be ordered before request A.
- Relevant external events may influence the ordering of system events!

- Two possible solutions:
  1. The user makes sure that the timestamp TB is later than TA
  2. Construct a system of clocks which satisfies the Strong Clock Condition: For any event a, b in φ: if a→b then C<a> < C<b>

Physical Clocks

- We can construct a system of physical clocks which, running quite independently of one another, will satisfy the Strong Clock Condition. Then we can use physical clocks to eliminate anomalous behavior:

  $C_i(t+\mu) – C_j(t) > 0$, with $\mu <$ shortest transmission time
- Above condition translates into strong clock condition, since we know that it takes longer than μ to send message, if a → b in physical time means that C<a> < C<b>
- Properties of clocks:
  1. Clock runs continuously.
  2. Clock runs at approximately the correct rate. i.e. dCi(t)/dt ≈1 for all t.
  3. Clocks must be synchronized so that $C_i(t) \approx C_j(t)$ for all i, j, and t.
- Two new conditions for physical clocks:
  – PC1. There exists a constant κ<<1 such that for all i: | dCi(t)/dt -1| < κ
  – PC2. For all i, j: | Ci(t) - Cj(t) | < ε (ε is a sufficiently small constant)
- Clock Synchronization algorithm:
  – Send messages so that clocks stay in sync (and always move forward)

General Ideas from Paper

- Using Virtual Clocks to order events in distributed system
- Using Resulting Ordering to build distributed state machines
- Clock Synchronization with no backtracking

Is this a good paper?

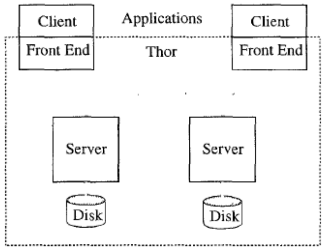
- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?

Break

OCC with Loosely Synchronized Clocks

- Basic Idea: Use Loosely Synchronized Clocks to pick ordering of transactions
  - Ultimately, this is the Serializable order
- Slight Twist: Want consistency with real world's view of transaction order
- Two desired consistency properties:
  - *Serializability*: The committed transactions can be placed in a total order, called the serialization order, such that the actual effect of running the transactions is the same as running them one at a time in that order
  - *External consistency:* The serialization order is such that, if transaction S committed before T began (in real time), S is ordered before T

OCC with Clocks: Basic Sketch



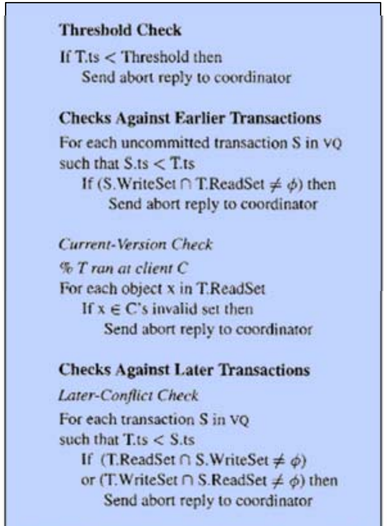
- Clients perform transactions Locally on Cached Pages
  - Reads and Writes done locally
  - When ready to commit, send request to one server which will interact with all servers which have data in read and write set of transaction
- Receiving server timestamps transaction with local clock, then performs 2-phase commit of transaction
  - Prepare phase: Ask each participating server if it is ok to commit
    - Response: server either says "yes" or "no"
    - If all servers say "yes", then transaction is committed
  - Commit phase: If all servers say "yes"
    - Note commit in stable log, notify everyone that it is time to commit
    - Can be done in background – client can go on immediately

Validation during Prepare Phase

- This is where the optimism comes in to play
  - Set of rules to look at log of previously validated transactions to see if any of them conflict with incoming commit request
  - If rules violated, then "abort". If rules not violated, the "accept"
- Since no locking, it is possible that ongoing transactions will conflict and need to be aborted
  - Conflicting transactions proceed together – OCC will eventually abort one of them
  - Insufficient information may occasionally cause aborts when unnecessary
  - Validation algorithm is *conservative* in allowing commit to proceed
- Key property: Serializability
  - In picking serializable order, must make sure that values written by earlier transactions picked up by later transactions
- Key property: External Consistency
  - If transaction S committed before T (in real time), then T should not appear before S in final order

Full Algorithm

- **Information Flow:**
  - **If using value from uncommitted transaction (because have later timestamp), must fail**
  - **If using stale value, must fail**
- **External Consistency:**
  - **Make sure that transactions with earlier timestamps that commit later can be reordered to match external appearances**
- **Threshold Truncation**
  - **If validation depends on truncated part of log, simply abort**

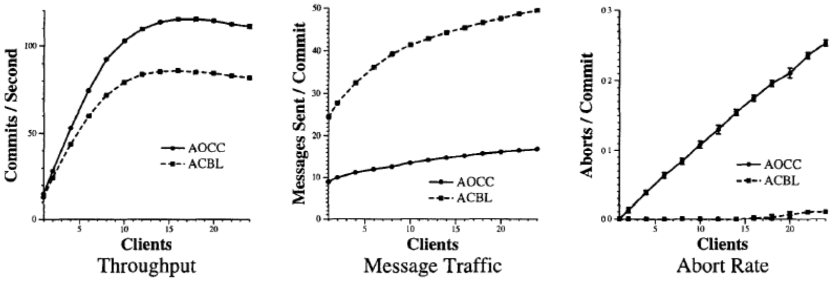


```
Threshold Check
If $T.ts < Threshold$ then
    Send abort reply to coordinator

Checks Against Earlier Transactions
For each uncommitted transaction S in VQ
such that $S.ts < T.ts$
    If ($S.WriteSet \cap T.ReadSet \neq \phi$) then
        Send abort reply to coordinator

Current-Version Check
% T ran at client C
For each object x in T.ReadSet
    If $x \in C$'s invalid set then
        Send abort reply to coordinator

Checks Against Later Transactions
Later-Conflict Check
For each transaction S in VQ
such that $T.ts < S.ts$
    If ($T.ReadSet \cap S.WriteSet \neq \phi$)
    or ($T.WriteSet \cap S.ReadSet \neq \phi$) then
        Send abort reply to coordinator
```

Figure 2: Validation Checks for Transaction T

Simulation Results

- **Comparison with Locking Discipline**



Throughput

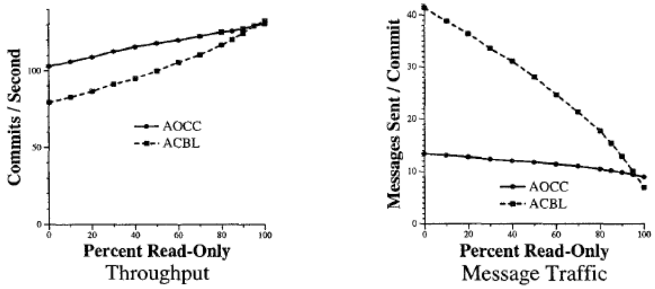
Message Traffic

Abort Rate

- **Overhead of locking involves multiple round-trips, while overhead of OCC involves Abort and Retry**
- **Which is better?**

Read Only Transactions

- **What about high-percentage of read-only transactions?**
  - **ACBL does not need to lock – simply use local state**
- **AOCC Still better for most transaction mixes:**



Throughput

Message Traffic

Is this a good paper?

- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?