Time and Clocks in Distributed Systems

Seif Haridi
KTH

Outline

- Motivation for using physical clocks

- Two algorithms:
  - Time-based leader leases

  - Shared memory using clocks

Motivation

- Consider a slightly stronger system model:
  - Computation
    - No bounds on time to take a step
  - Communication
    - No bounds on latency
    - So far, this is the asynchronous system model
  - Clocks
    - Lower and upper bounds on clock rate

Motivation

- This is a fairly weak model in practice
  - ***"Our machine statistics show that bad CPUs are 6 times more likely than bad clocks. That is, clock issues are extremely infrequent, relative to much more serious hardware problems***." – Google

Motivation

- Why consider algorithms that use clocks?
  - By making stronger assumptions about the system we can get better efficiency/performance
  - In this slightly stronger model we cannot still solve problems that are harder than what can be solved in the asynchronous model
    - i.e. the FLP impossibility still holds
  - But we can define some abstractions will better properties

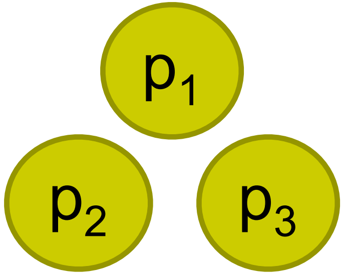
Time-based Leader Leases

Outline – Leader Leases

- The optimization opportunity by using clocks
- The proposed algorithm
- An argument why correctness is maintained

Background

- We implement a key-value store using RSM
- Supporting the following commands:
  - Read(k), Write(k, v), CAS(k, v_{exp}, v_{new})
    - CAS:
      - writes v_{new} if old value is v_{exp}; returns old value
    - Needs RSM to do CAS (Shared Mem. is too weak)
- Service runs on leader-based Sequence Paxos
  - N=3 replicas, $\Pi_r=\{p_1, p_2, p_3\}$
- Each acting as proposer, acceptor, learner



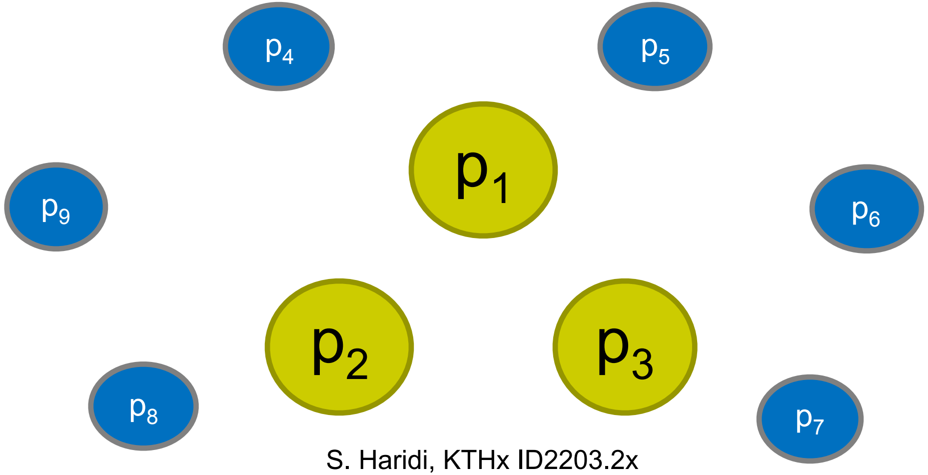
Command ordering

- Paxos guarantees that all replicas execute commands in same order

Old state	Command	Result	New state
{}	Write(x,1)	OK	{x=1}
{x=1}	Write(y,0)	OK	{x=1,y=0}
{x=1,y=0}	Read(x)	1	{x=1,y=0}
{x=1,y=0}	CAS(y,0,1)	0	{x=1,y=1}
{x=1,y=1}	CAS(y,0,1)	1	{x=1,y=1}
{x=1,y=1}	Read(y)	1	{x=1,y=1}
{x=1,y=1}	Write(y,0)	OK	{x=1,y=0}
...	...	...	...

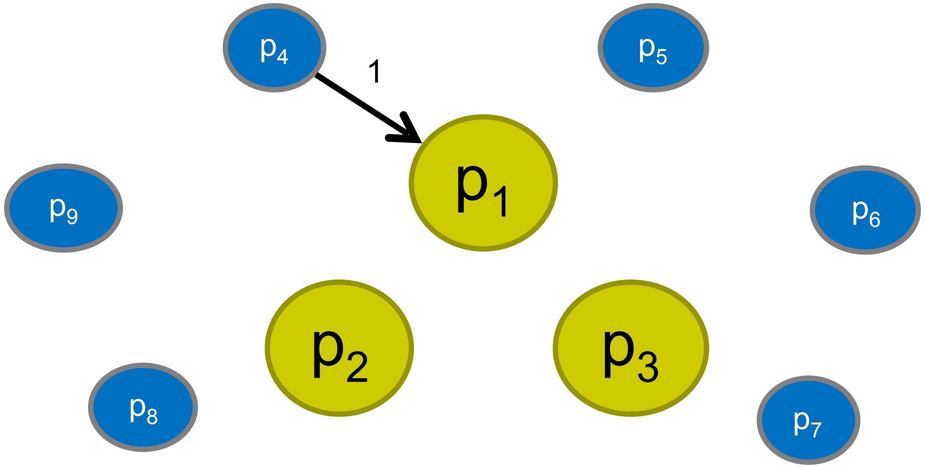
Clients and Leader

- Can have any number of clients $\Pi_c=\{p_4,\ldots\}$
- Assume network is stable and p_1 is leader (has started the highest round)



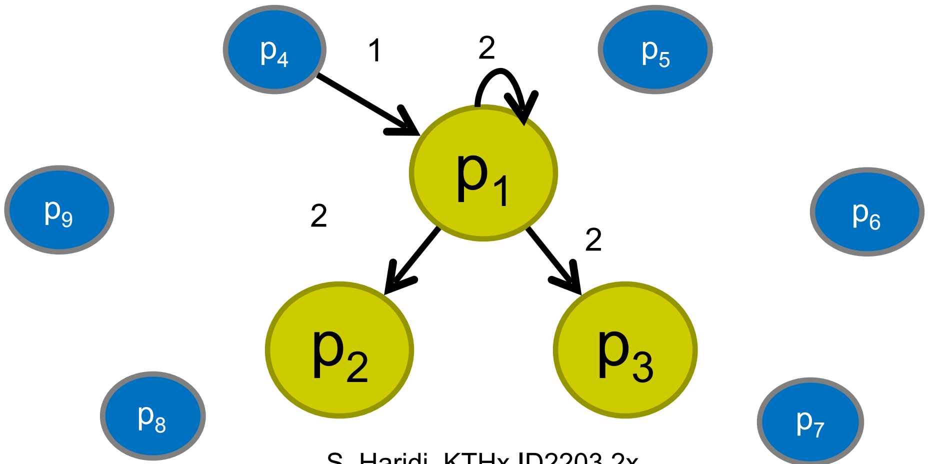
Executing a Command

- Client p_4 that wants to execute a command sends a request (1) to leader p_1



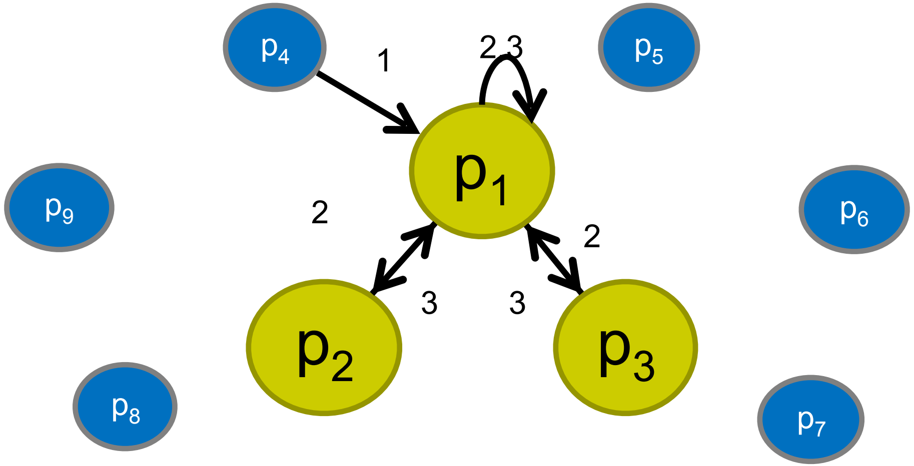
Executing a Command

- p_1 proposes command using Paxos, which sends Accept msgs (2) to replicas (using previously prepared round number)



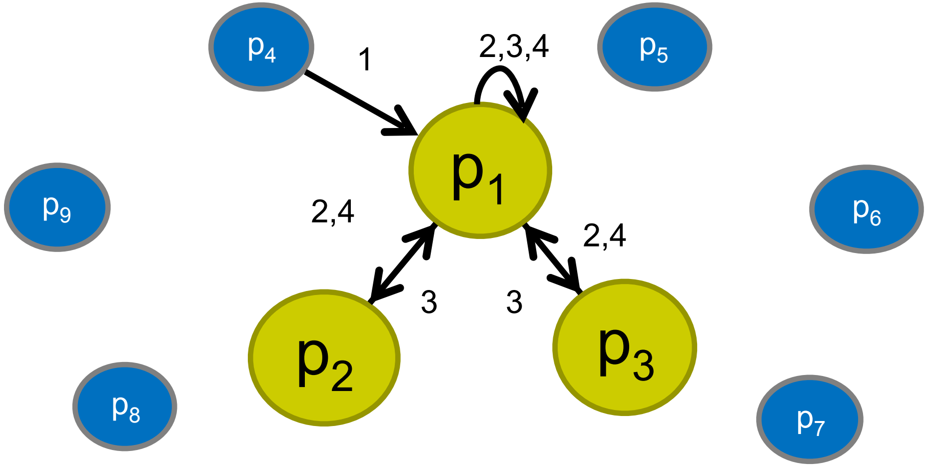
Executing a Command

- The replicas accept and respond with AcceptAck (Accepted) messages (3)



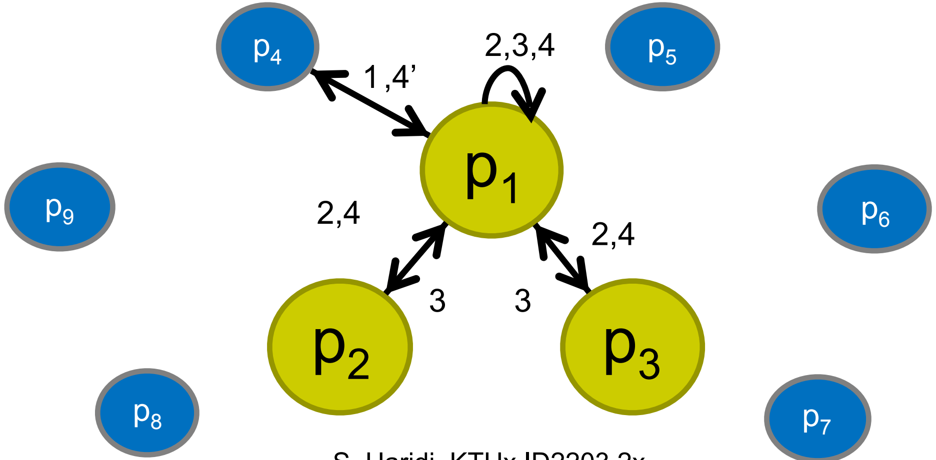
Executing a Command

- After p_1 gets AcceptAck msgs from a majority, the command order is chosen and p_1 sends Decide msgs (4)



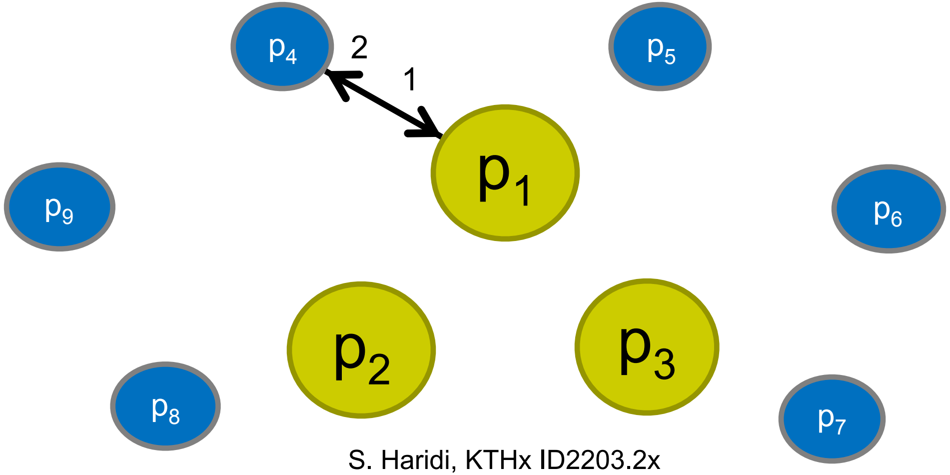
Executing a Command

- p_1 executes the command using the state of the state machine, and sends response (4') with result of the operation to p_4



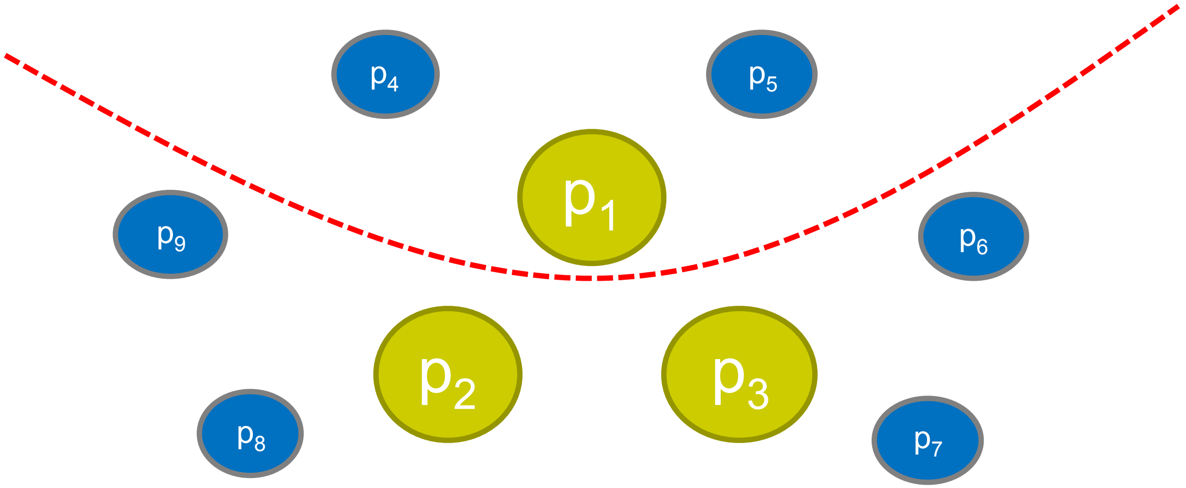
Opportunity: Faster Reads

- Assume that the operation requested by p_4 is a read operation, C=Read(x)
- p_1 stores the entire state, so can p_1 read the state variable x and respond immediately?



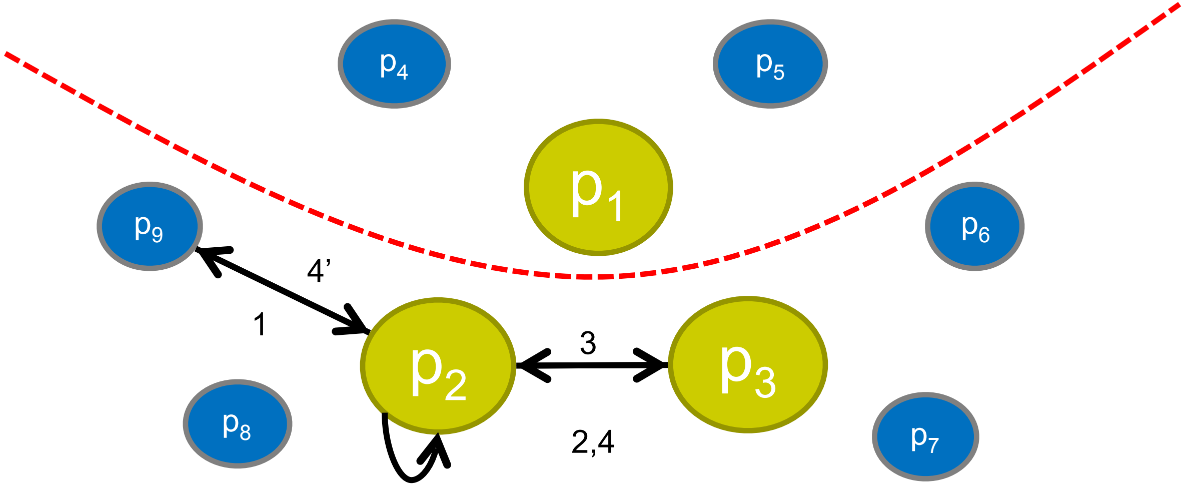
What could go wrong?

- A network split partitions p_1 away from p_2 and p_3
- p_2 is elected leader but p_1 never hears about this



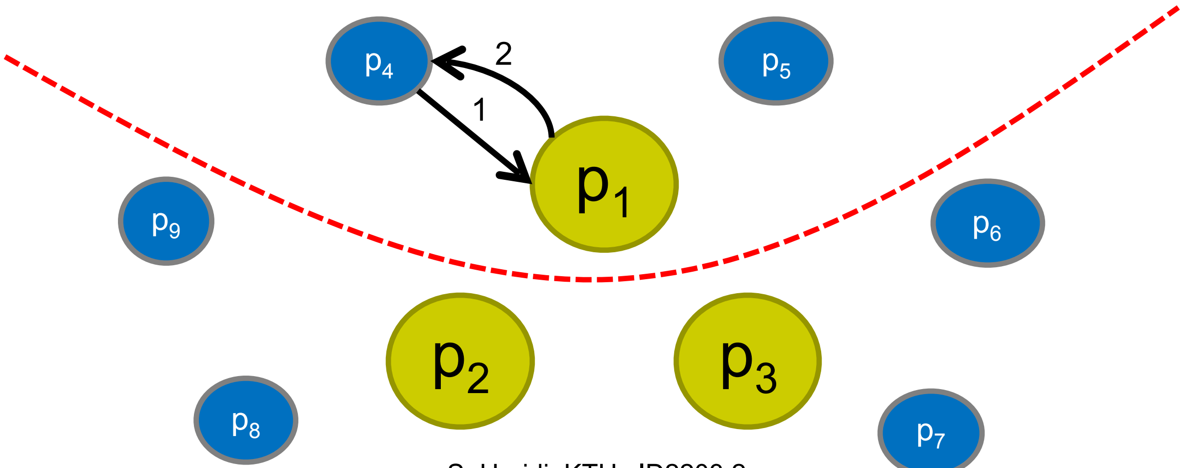
What could go wrong?

- Client p_9 sends a Write(x,val_{new}) request to p_2, p_2 communicates with p_3 and then executes the write operation



What could go wrong?

- After this, p_1 gets Read(x) request from p_4
- p_1 is unaware of the split and the write operation, and responds to p_4 with the old value of x
- **Linearizability is violated!**



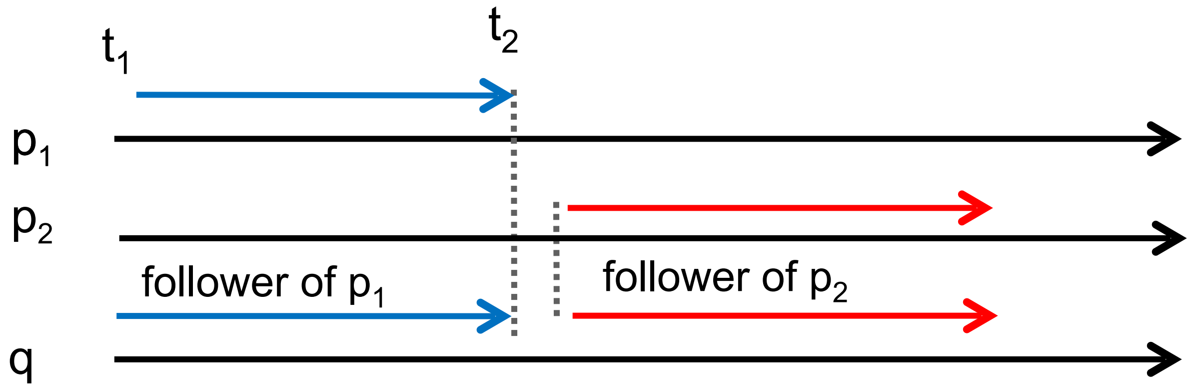
Problem summarized

- The reason p_1 can't respond with its current state because some other replica may have assumed leadership and modified the state without p_1 knowing about it
- Is there some way to avoid this?
- False attempt:
  - p_2 must communicate with p_1 before p_2 can become leader
  - But this can't work since p_1 may be dead

Time Leases

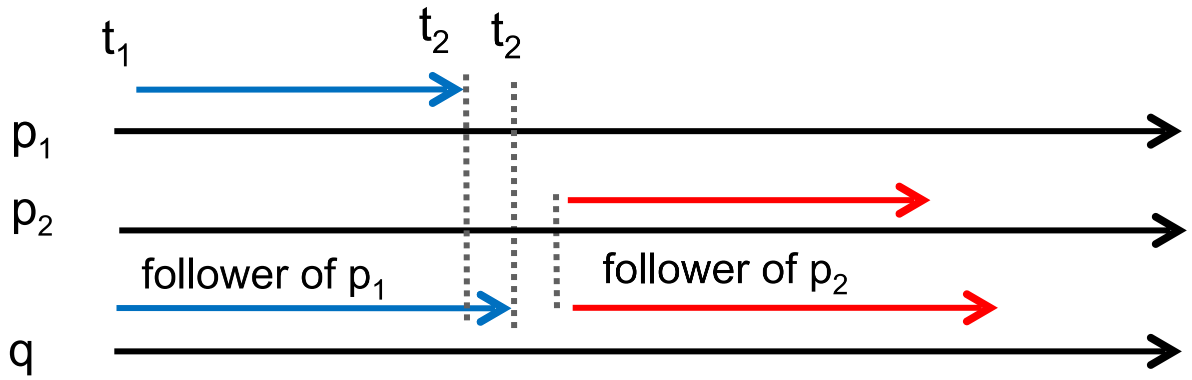
Solution: time-based leader lease

- We would like leaders to be disjoint in time
- Think of this as a Paxos group
  - Only one leader at an given point of time t
  - If q is a follower of p at time t then no other no other process can be a leader at t



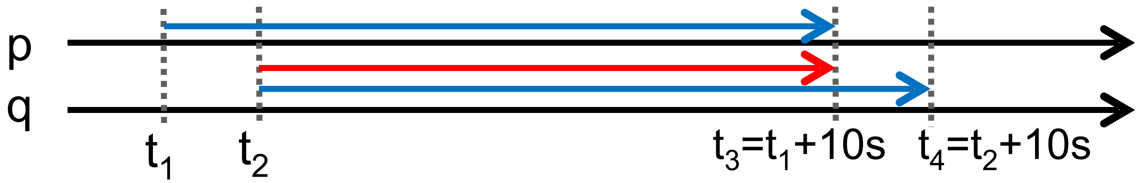
Solution: time-based leader lease

- We would like leaders to be disjoint in time
- Think of this as a Paxos group
  - Only one leader at an given point of time t
  - If q is a follower of p at time t then no other no other process can be a leader at t



Solution: time-based leader lease

- A propose p to become leader: sends a request (prepare) to acceptors
  - An acceptor gives a time-based leader lease to p , lasting for 10 seconds
  - If a proposer gets leases from a majority of acceptors, then proposer holds lease on group and becomes a leader
  - In the time until the first acceptor lease expires, the proposer knows that no other proposer can hold the lease on the group
    - During this time, the leader can safely respond to reads from local state



Solution: time-based leader lease

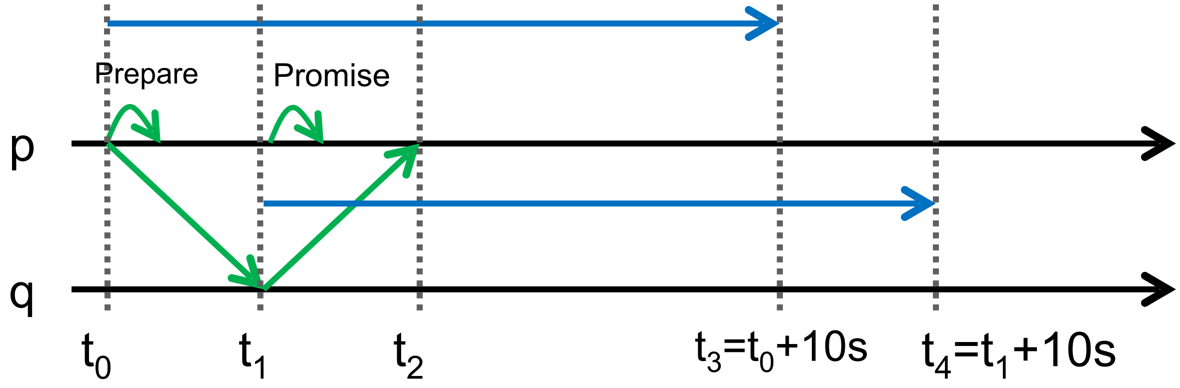
- Can be integrated with Paxos messages:
  - **As before** acceptor q joins round n by sending a Promise in response to a Prepare(n), and promises *to not accept proposals in lower rounds*
  - **In addition**, we require that if q joins round n at time t then q promises *not to join a higher round until after time t+10s*
  - If proposer p gets promises from a majority then p knows that no other proposer can get a majority of promises during next 10 seconds

Issues

- Notice that we are only taking about physical time intervals and not about absolute clock values
- We have to take two issues into account:
    - Network is asynchronous
    - Clocks drift

Issue 1: asynchronous network

- p can't know at what exact time q sent the Promise, only that $t_0 \le t_1 \le t_2$
  - p has to be conservative and assume that $t_1 = t_0$
  - p holds lease until $t_3 = t_0 + 10s$



Clock Drift

Issue 2: clock drift

- To understand the clock drift issue, we have to describe clocks and time more formally and in more detail
- A clock at a process p_i is a monotonically increasing function from real-time to some real value

Introduction to clocks

- Each process p_i has an associated clock C_i
- $C_i(.)$ is modelled as a function from real times to clock times
  - Real time is defined by some time standard, such as Coordinated Universal Time (UTC)
  - The unit of time in UTC is the SI second, whose definition states that:
    - "***The second is the duration of 9 192 631 770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the caesium 133 atom***."

Clock implementation

- A clock is implemented as an oscillator and a counter register that is incremented for each period of the oscillator
  - The oscillator frequency is not completely stable, varying depending on environmental conditions such as temperature, and aging
  - The oscillator's manufacturer specifies a nominal frequency and **an error bound**

Clock rate

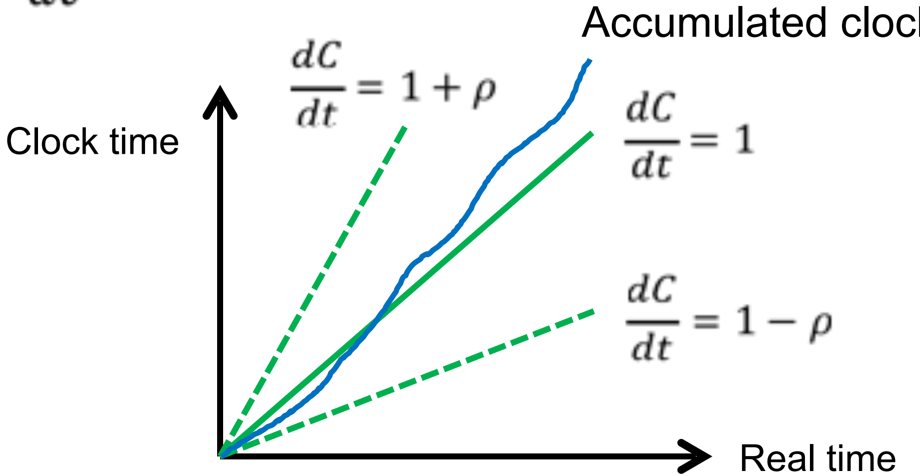
- The clock rate specifies how much the clock is incremented each second of real time
  - For example: the counter increments by nominally 1,000,000 ticks per second, with an error bounded to ±100 ticks per second
- From here on we normalize the clock rate so that 1.0 is the nominal rate, and the error is given by ρ such that

$$\frac{1}{1+\rho} \approx 1-\rho \leq \frac{dC}{dt} \leq 1+\rho$$

  - In our example ρ = 100/1,000,000 = 100ppm

Clock drift

- Clock drift is the accumulated effect of a clock rate that differs from real time
  - Ideally, $\frac{dC}{dt} = 1$



Issue 2: clock drift at proposer

- Reason about what happens if proposer uses clock time instead of real time without any compensation?
  - Clock runs faster than real time: safety **cannot be violated** as proposer believes that its lease expired sooner than it actually did
  - Clock runs slower than real time: proposer believes it holds lease even after lease has expired, and proposer may respond to read, and **violate safety**

Issue 2: clock drift at proposer

- Proposer must compensate by assuming its clock is running as slowly as possible, $\frac{dC}{dt} = 1 - \rho$, and compensate
  - $\Delta t \leq 10$, at most 10 seconds real time
  - $\Delta C = \Delta t \times (1 - \rho) \leq 10 \times (1 - \rho)$

Issue 2: clock drift at acceptor

- What happens if acceptor uses clock time instead of real time without compensation?
  - Clock runs faster than real time: acceptor believes its promise expired too soon, and may give new lease early, **violating safety**
  - Clock runs slower than real time: safety cannot be violated if acceptor waits longer than necessary to give new promise

Issue 2: clock drift at acceptor

- Acceptor must assume its clock is running as fast as possible, $\frac{dC}{dt} = 1 + \rho$, and compensate
  - $\Delta t \geq 10$, at least 10 seconds real time
  - $\Delta C = \Delta t \times (1 + \rho) \geq 10 \times (1 + \rho)$

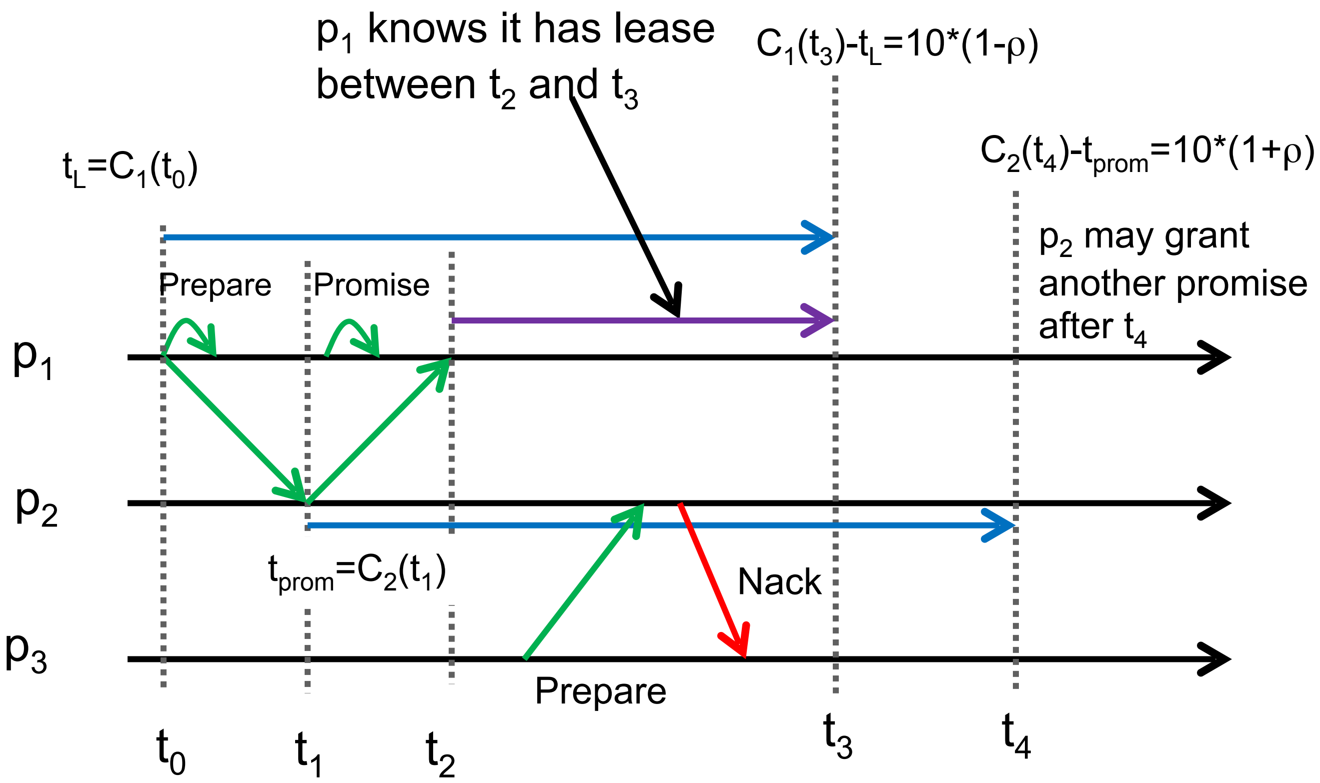
Leases at acceptor

- Acceptors have new state variable, t_{prom}
  - The clock time when gave last promise
- If acceptor p_j gets Prepare(n) at time **T** and
  - $n > n_{prom}$ and $C_j(\mathbf{T}) - t_{prom} > 10*(1+\rho)$
  - then give promise to reject rounds lower than n, and not give new promises within the next 10s (set $t_p = C_j(\mathbf{T})$)
  - Otherwise respond with Nack

Leases at proposer

- Proposer has new state variable t_L
- Before proposer p_i sends Prepare(n) at time **T** messages it sets variable $t_L = C_i(\mathbf{T})$
- If p_i gets promises from a majority, p_i knows that no other process can become leader until 10s after t_l
- As long as $C_i(\mathbf{T}) - t_L < 10*(1-\rho)$, p_i can respond to reads from its local state

Time diagram



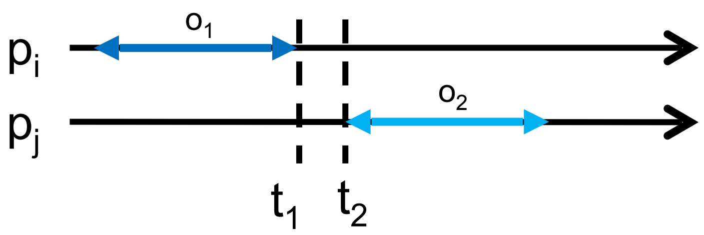
Extending a lease

- As long as p_i is alive and well it should remain the leader
- To not loose the lease, p_i can ask for an extension of the lease
  - I.e. a few seconds before the lease expires, p_i records the current clock time t and asks for an extension
  - If an extension is granted by a majority of replicas then p_i holds the lease until 10s after t
  - Each acceptor adjust its t_{prom} accordingly

Shared Memory Using Clocks

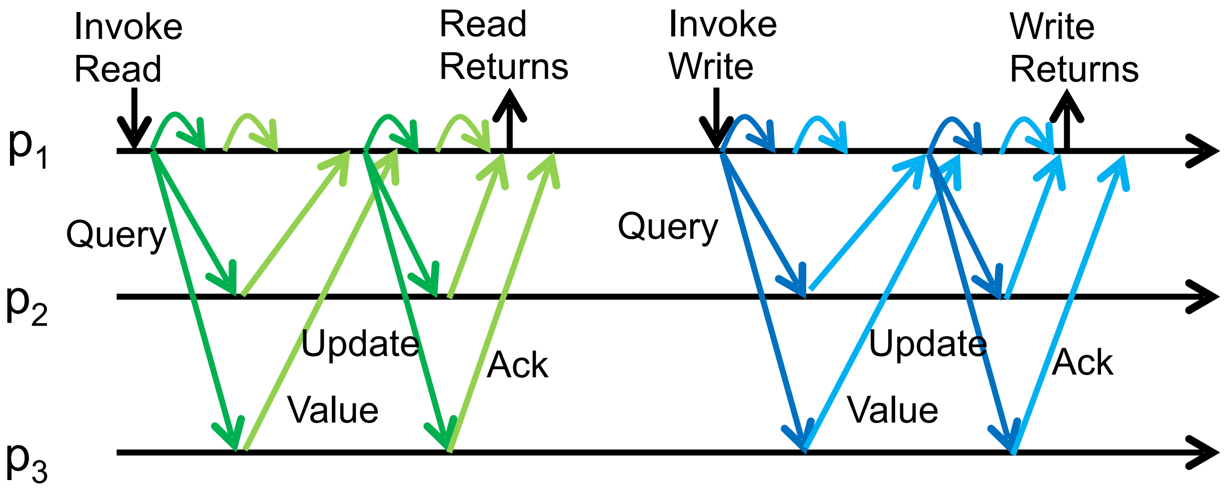
Review of shared memory

- A set of *atomic registers*
- Two operations:
  - Write(v): update register's value to v
  - Read(): return the register's value
- Correctness: Linearizability
  - If operation o_1 returns before operation o_2 is invoked, then o_1 must be ordered before o_2 (the linearization point of o_1 is before the linearization point of o_2)



Algorithm in course: RIWCM

- The **R**ead-**I**mpose **W**rite-**C**onsult-**M**ajority algorithm does 2 round-trips to a majority of processes for both reads and writes



Phases

- A *phase* is one round-trip of communication to a majority of replicas
- Refer to the first phase as the *query phase* and the second phase as the *update phase*

Read operation

- Process p_i invokes read operation o_r
- In the query phase, each process responds with the highest *timestamp-value pair* received
- p_i picks the highest timestamp-value pair received in the query phase, denoted (ts, v)
- Before returning value v, p_i performs an update phase using the pair
  - This way, any operation invoked after o_r is completed is guaranteed to see a timestamp greater than or equal to ts

Optimizing read operation

- If in the query phase all processes in a majority set respond with the same timestamp-value pair (ts, v), then the update phase can be skipped
  - This works since a majority of the processes already store a timestamp-value pair with a timestamp greater than or equal to ts
- In good conditions (network is stable, low contention) this is likely to be the case, and reads can complete in a single round-trip

Write operation

- Process p_i invokes write operation o_w
- In the query phase, each process responds with the highest timestamp-value pair received
- After the query phase, p_i picks a unique timestamp higher than all timestamps received and pairs it with the value to write
- In the update phase, each process stores this timestamp-value pair if the pair is greater the timestamp than the previously stored pair's timestamp

Optimizing write operation

- If processes have access to clocks then it is possible to skip the query phase
- Process p_i invoking a write instead picks a timestamp by reading the current time and forms a timestamp $ts=(C_i, i)$
    - Timestamps are time-pid pairs; (t, pid)
- How well clocks are *synchronized* will determine if the atomicity property of the Atomic Register abstraction is satisfied

Synchronized Clocks

Optimizing write operation

- If processes have access to clocks then it is possible to skip the query phase
- Process p_i invoking a write instead picks a timestamp by reading the current time and forms a timestamp $ts=(C_i, i)$
  - Timestamps are time-pid pairs; (t, pid)
- How well clocks are *synchronized* will determine if the atomicity property of the Atomic Register abstraction is satisfied

Clock synchronization

- Clocks C_i and C_j are δ-*synchronized* if, for all times t, $|C_i(t)-C_j(t)| \leq \delta$
  - Saying that C_i and C_j are synchronized to within 10ms means that δ=10ms
- A set of clocks are *perfectly synchronized* if each pair of clocks is δ = 0-synchronized
- *Loosely synchronized clocks* attempts to be as closely synchronized as possible, but give no guarantees
  - In practice, can be arbitrarily out of synch

Correctness of write optimization

- If clocks are perfectly synchronized then registers satisfy linearizability
  - o_1 is read or write, o_2 is read: by the same argument as before, o_1 is ordered before o_2
  - o_1 is write, o_2 is write: as o_1 is completed before o_2 is invoked, $ts(o_1)<ts(o_2)$, and value written by o_1 is overwritten by value of o_2
  - o_1 is read, o_2 is write: exists a write o_0 that was invoked before o_1 completed, $ts(o_0)=ts(o_1)<ts(o_2)$
- Writes (and often reads) take one round-trip, and correctness is guaranteed

Correctness of write optimization

- If clocks are loosely synchronized then registers don't satisfy linearizability
  - If write o_1 is complete before write o_2 is invoked then the timestamp picked by o_1 may still be greater than the timestamp picked by o_2
- Important to remember in practice
  - **Cassandra uses loosely synchronized clocks in this way, and can therefore not guarantee linearizability**

Correctness – Logical clocks

- If clocks are logical clocks (Lamport clocks) then the shared memory doesn’t satisfy linearizability
- Instead, the memory satisfies sequential consistency
  - We have seen the proof in part 1 of the course

Problem solved?

- Using perfectly synchronized clocks (PSCs) guarantees linearizability, so just use PSCs and everything is good?
- No, since PSCs are **impossible** to implement
  - Any measurement contains some uncertainty
  - Synchronizing clocks across an asynchronous network adds more uncertainty
- We introduce a new kind of clocks…

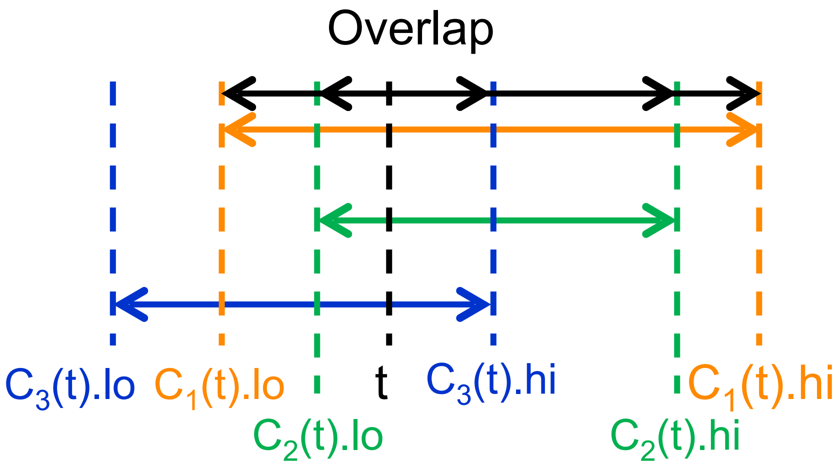
Interval Clocks

Interval clocks

- An interval clock (IC) at process p_i read at time t returns a pair $C_i(t)=(lo, hi)$
- Represents an interval $[C_i(t).lo \,..\, C_i(t).hi]$
  - The correct time t is guaranteed to be in interval
    - $C_i(t).lo \leq t \leq C_i(t).hi$
- Synchronization uncertainty is exposed in width of interval
- This is the strongest guarantee that can be implemented in practice
  - Wide interval may hurt performance of algorithm using ICs, but does not affect correctness

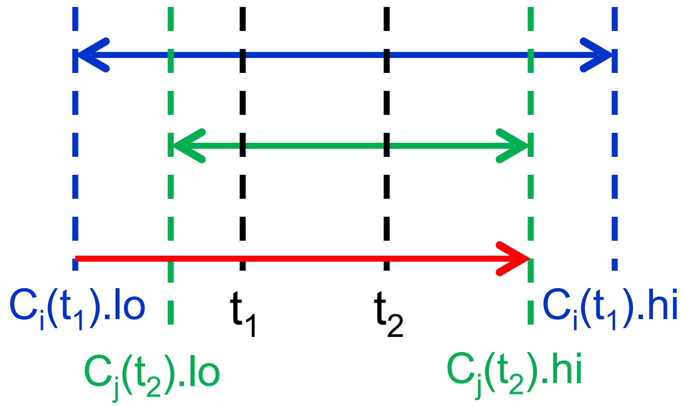
Overlapping intervals

- The interval values of a set of clocks read at the same time t are guaranteed to overlap in the correct time



Clocks read at different times

- C_i read at t_1, C_j read at t_2, and $t_1 < t_2$
  - $C_i(t_1).lo \le t_1 \le C_i(t_1).hi$
  - $C_j(t_2).lo \le t_2 \le C_j(t_2).hi$
  - Implies: $C_i(t_1).lo < C_j(t_2).hi$



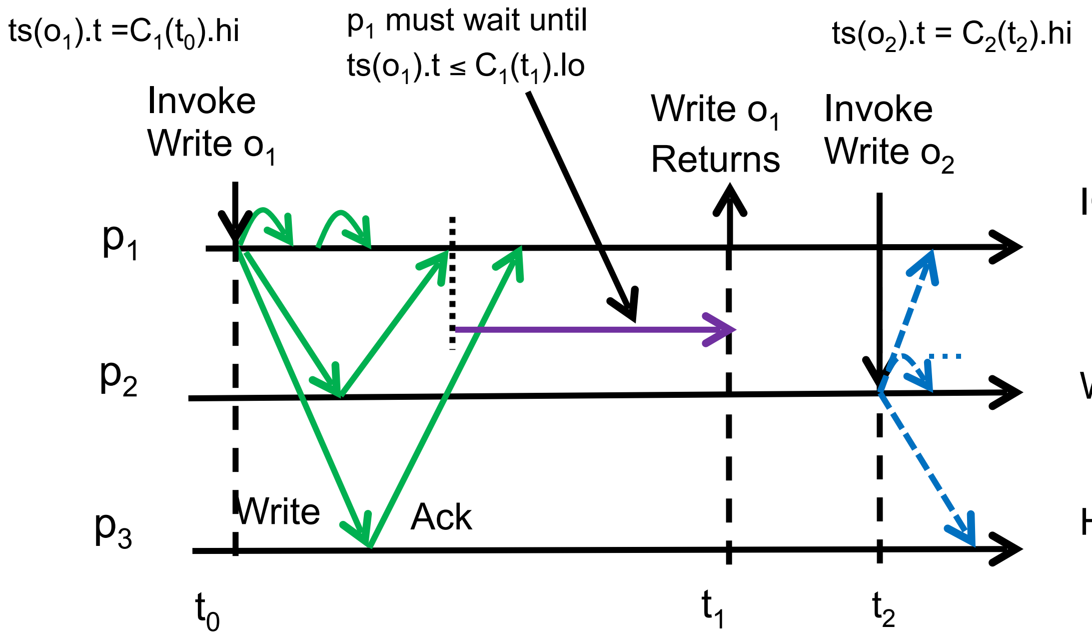
- $C_i(t_1).lo \le t_1 < t_2 \le C_j(t_2).hi$

Using ICs to remove query phase in write operations

- Two changes:
  - In process p_i that is invoking a write operation, use timestamp ts = $(C_i.hi, i)$
  - Before an operation o (a read or a write) executed by process p_i can return it has to wait until $ts(o).t < C_i.lo$
    - ts(o) is the timestamp associated with the value that is read or written by operation o

Intuition why waiting is needed

- o_1 is allowed to return when ICs guarantee that later write will pick a higher timestamp



IC guarantee:

If $t_1 < t_2$ then

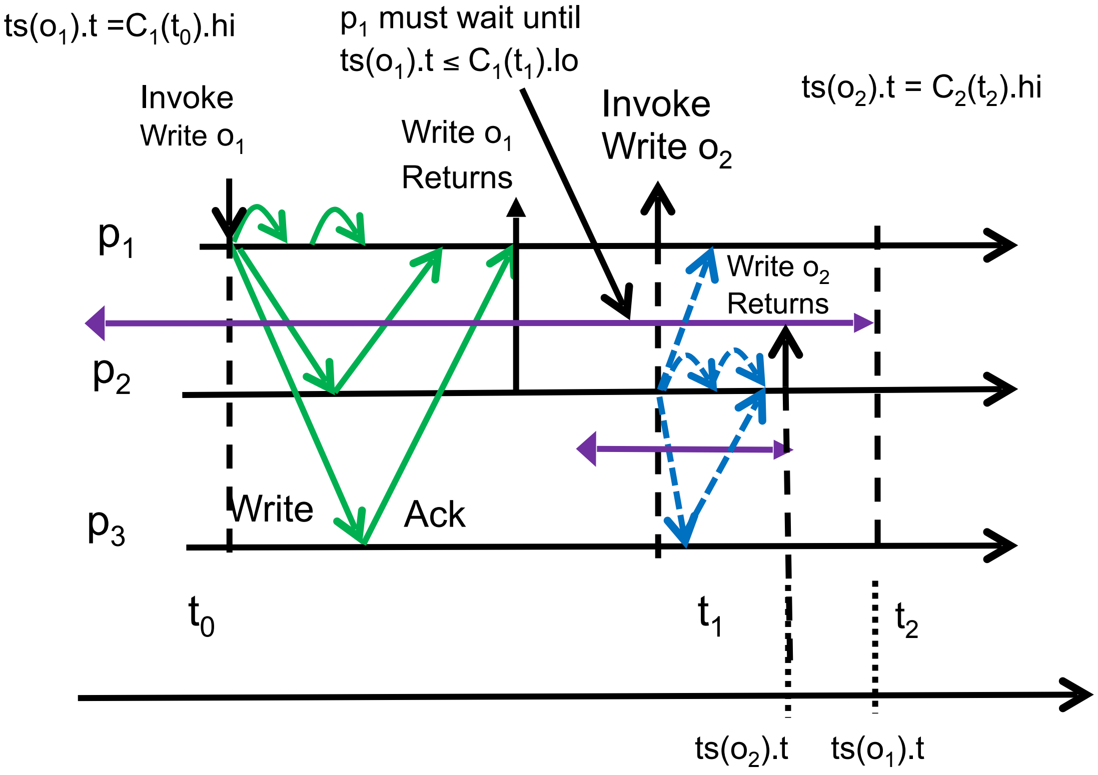
$C_1(t_1).lo < C_2(t_2).hi$

We have:

$ts(o_1).t \leq C_1(t_1).lo < C_2(t_2).hi = ts(o_2).t$

Hence: $ts(o_1) < ts(o_2)$

Intuition why waiting is needed

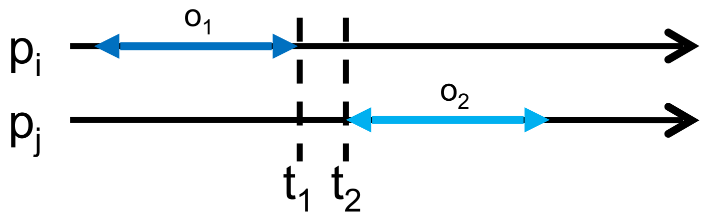


If o_1 is completed before o_2 is invoked, then o_1 must be ordered before o_2

Case: o1 does not wait
o_1 completes before o_2 is issued: no guarantee that o_1 before o_2 ($ts(o_1).t > ts(o_2).t$)

Correctness

- Algorithm with ICs satisfy linearizability:
  - o_1 is read or write, o_2 is read: by the same argument as before, o_1 is ordered before o_2
  - o_1 is read or write, o_2 is write:
    - o_1 is completed at t_1 by p_i, and o_2 is invoked at t_2 by p_j
    - $t_1 < t_2$ implies that $ts(o_1).t \leq C_i(t_1).lo < C_j(t_2).hi = ts(o_2).t$
    - Since $ts(o_1) < ts(o_2)$, the value in o_1 is overwritten by the value of o_2



- **On Init:**
  - ts := (0, 0)
  - v := 0

- **On ReadInvoke**:
  - reading := true
  - readlist := $[\perp]^N$
  - **send** ⟨**Read**⟩ **to** Π

- **On** ⟨**Read**⟩ **from** p_i:
  - **send** ⟨**Value**, ts, v⟩ **to** p_i

- **On** ⟨**Value**, ts', v'⟩ **from** q:
  - readlist[q] := (ts', v')
  - **if** #(readlist) > N/2:
  -   (rts, rv) = max(readlist)
  -   **if** all pairs in readlist are equal:
  -     **DoReturn()**
  -   **else**:
  -     acks := 0
  -     **send** ⟨**Write**, rts, rv⟩ **to** Π

- **On WriteInvoke**(v):
  - reading := false
  - rts := (C_i.hi, i)
  - acks := 0
  - **send** ⟨**Write**, rts, v⟩ **to** Π

- **On** ⟨**Write**, ts', v'⟩ **from** p_i:
  - **if** ts' > ts:
  -   ts := ts'
  -   v := v'
  - **send** ⟨**Ack**⟩ **to** p_i

- **On** ⟨**Ack**⟩:
  - acks := acks + 1
  - **if** acks > N/2:
  -   **DoReturn()**

- **fun DoReturn**():
  - **wait until** rts.t < C_i.lo
  - **if** reading: **trigger ReadReturn(rv)**
  - **else**: **trigger WriteReturn**