ADVANCED COURSE

Distributed Systems

True Time Abstractions

Paris Carbone

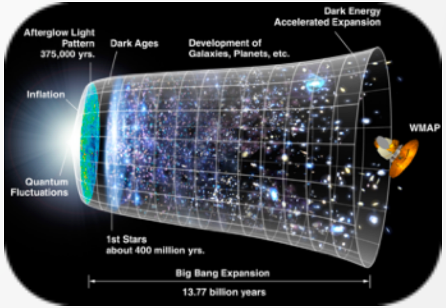
Course Topics

- Intro to Distributed Systems
- Fundamental Abstractions and Failure Detectors
- Reliable and Causal Order Broadcast
- Distributed Shared Memory-CRDTs
- Consensus (Paxos)
- Replicated State Machines (OmniPaxos, Raft, Zab etc.)
- Time Abstractions and Interval Clocks (Spanner etc.)
- Consistent Snapshotting (Stream Data Management)
- Distributed ACID Transactions (Cloud DBs)

LET'S TALK ABOUT TIME

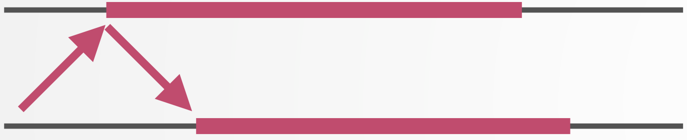
- Aren't clocks always unreliable in D.S.?
- Can we reason based on time measurements?
- Any time invariants we can trust?



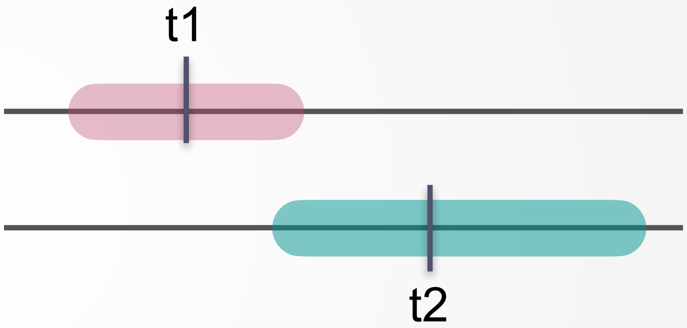
Two Practical Time Abstractions

Time Leases



+ shorter protocols
+ stronger abstractions
+ a standard for client-server comm.
- re-configuration/election slowdowns

Interval Clocks

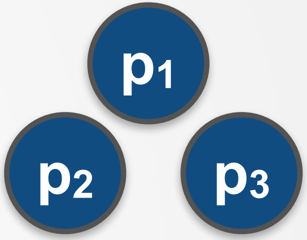


+ linearizable operations
+ practical, conservative time estimates
+ highly in use (Spanner, CockroachDB)
- performance ~ clock quality

Time-Leases

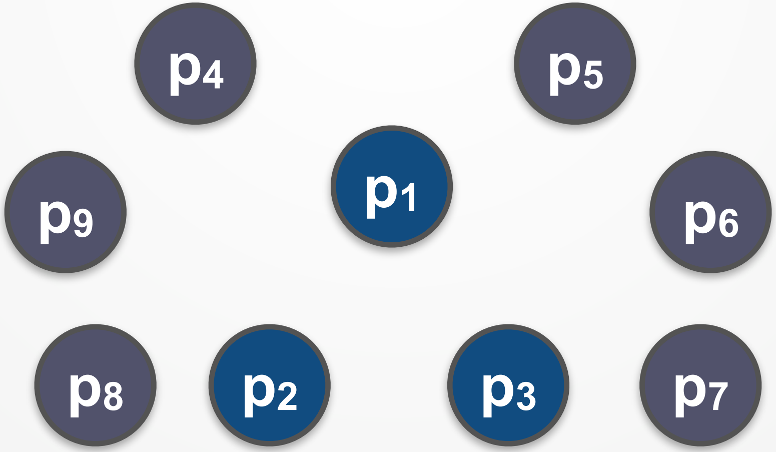
Motivation

- 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 Seq Paxos
  - N=3 replicas, $\Pi_r=\{p_1, p_2, p_3\}$
- Each acting as proposer, acceptor, learner



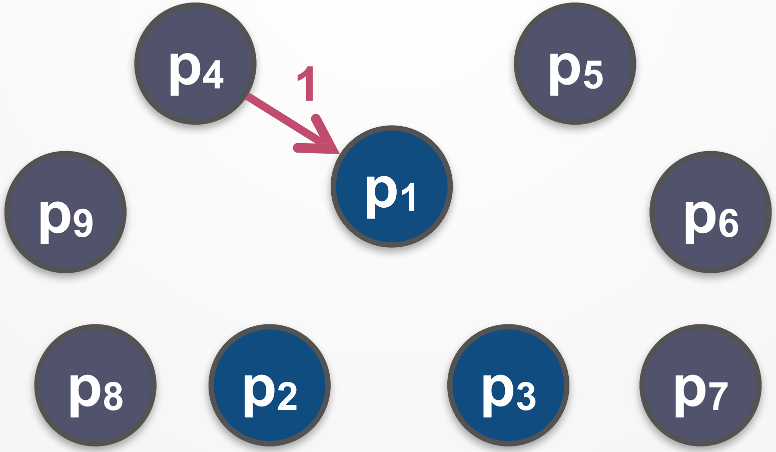
MOTIVATION

- Can have any number of clients $\Pi c=\{p4,\ldots\}$
- Assume network is stable and p1 is leader (has started the highest round)



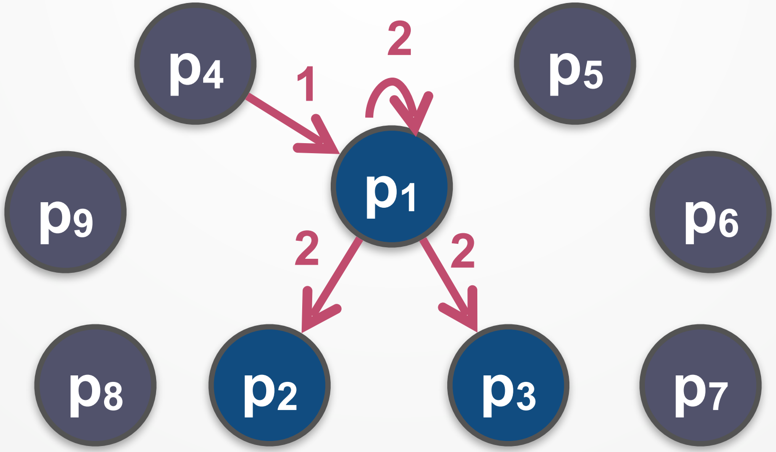
MOTIVATION

- Client p4 that wants to execute a command sends a request (1) to leader p1



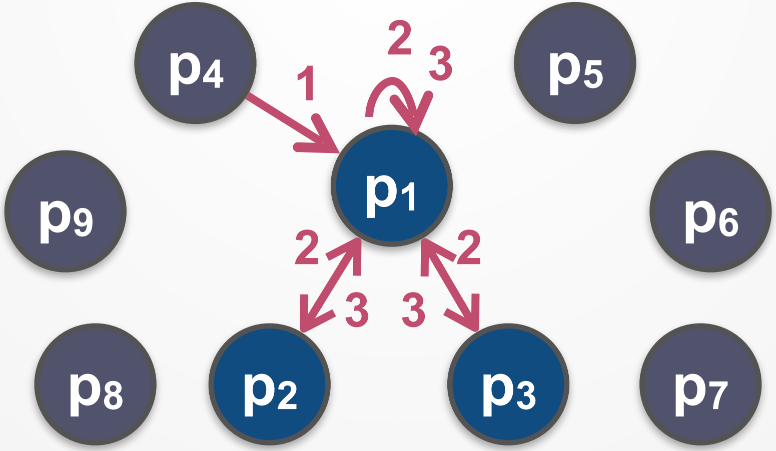
Motivation

- p1 proposes command using Paxos, which sends Accept msgs (2) to replicas (using previously prepared round number)



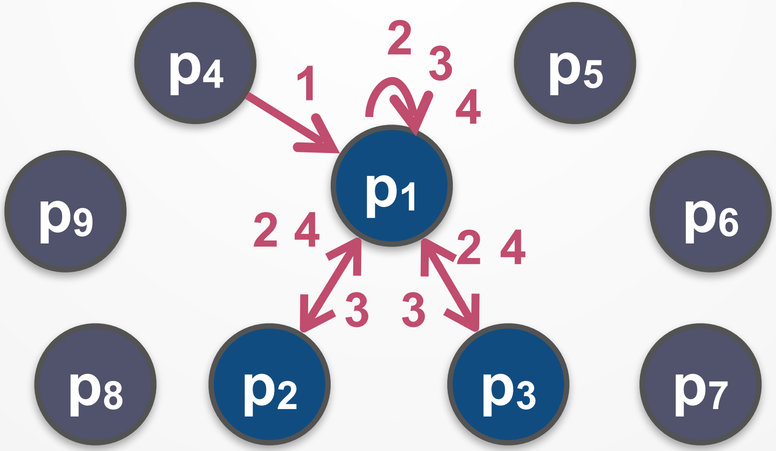
Motivation

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



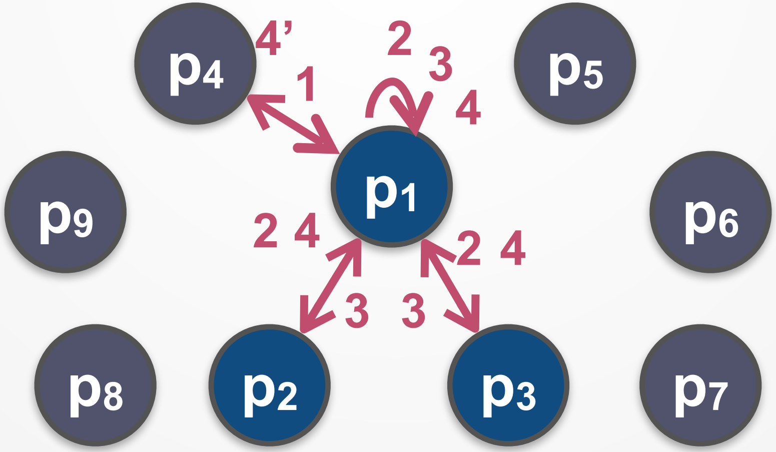
Motivation

- After p1 gets AcceptAck msgs from a majority, the command order is chosen and p1 sends Decide msgs (4)



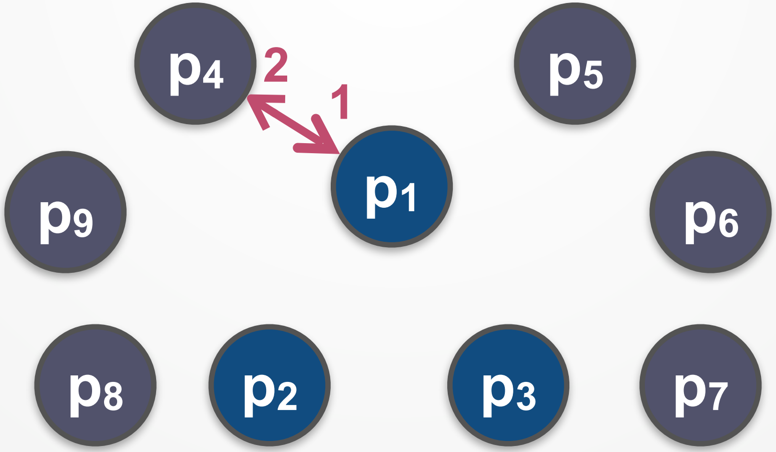
Motivation

- p1 executes the command using the state of the state machine, and sends response (4') with result of the operation to p4



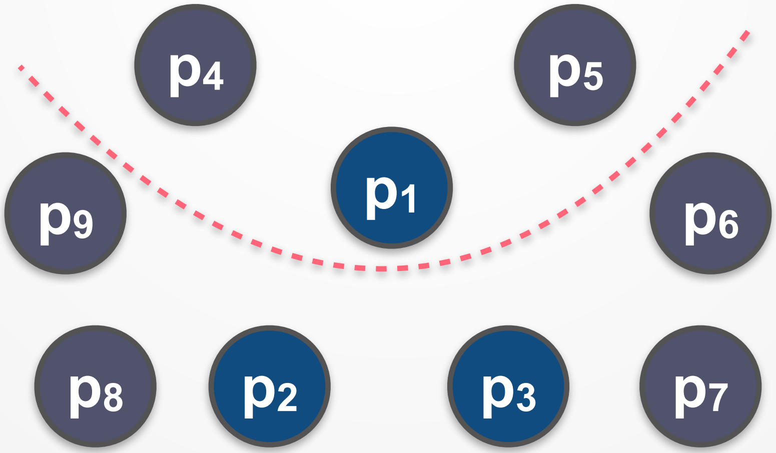
FASTER READS?

- Assume that the operation requested by p4 is a read operation, C=Read(x)
- p1 stores the entire state, so can p1 read the state variable x and respond immediately?



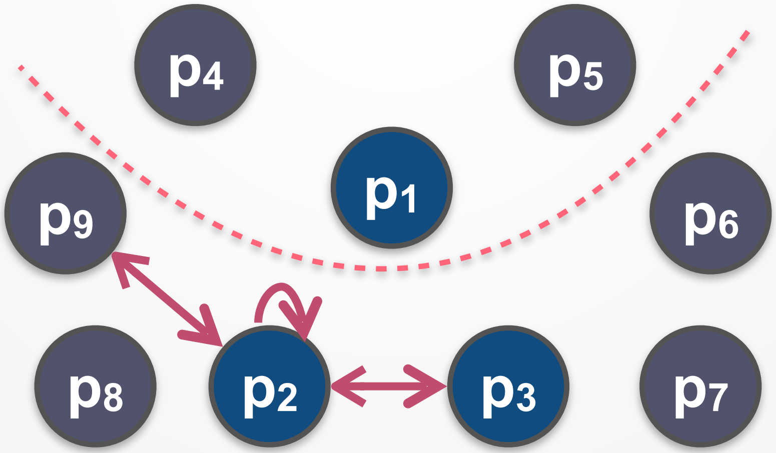
FASTER READS?

- A network split partitions p1 away from p2 and p3
- p2 is elected leader but p1 never hears about this



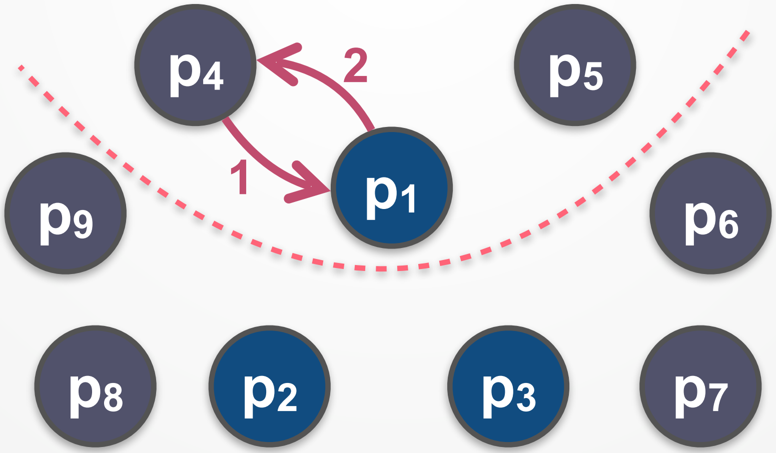
Faster Reads?

- Client p9 sends a Write(x,valnew) request to p2, p2 communicates with p3 and commits the write operation



FASTER READS?

- After this, p1 gets Read(x) request from p4
- p1 is unaware of the split and the write operation, and responds to p4 with the old value of x
- Linearizability is **violated**!



The Problem

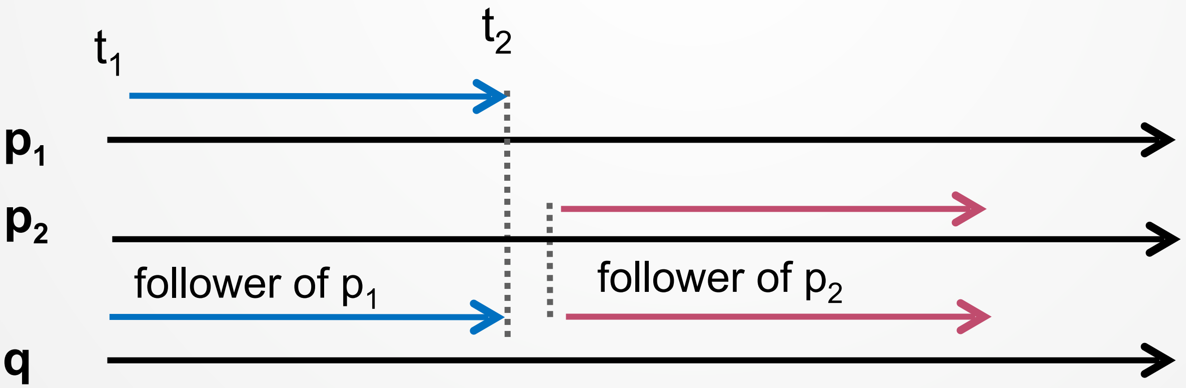
- The reason p1 can't respond with its current state because some other replica may have assumed leadership and modified the state without p1 knowing about it
- Is there some way to avoid this?

- **Strawman attempt:**

- p2 must communicate with p1 before p2 can become leader
- But this can't work since p1 may be dead

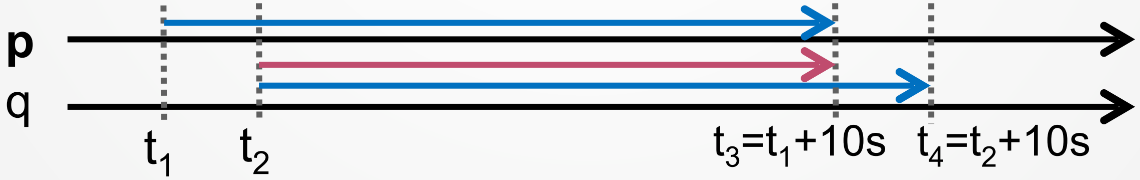
Leader-Leases

- 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



LEADER-LEASES

- A proposer 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



Leader-Leases

- 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 to not 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

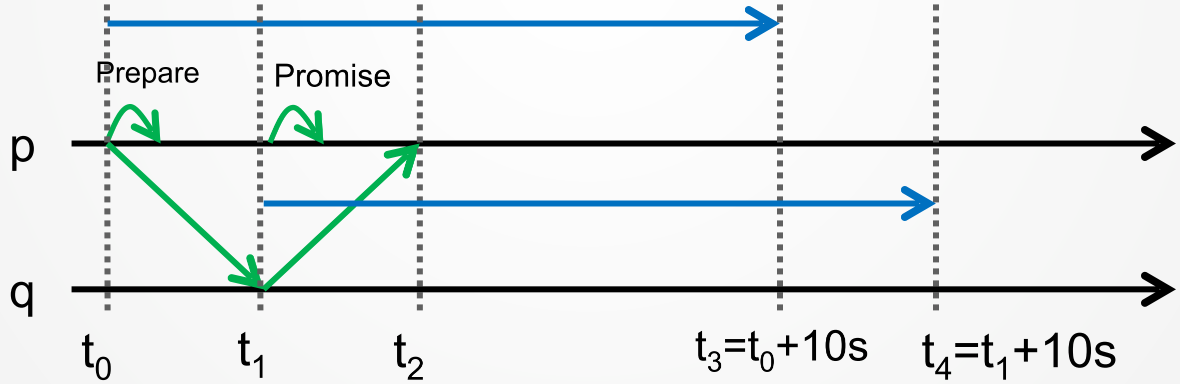
ARE WE THERE YET?

- Notice that we are only talking about physical time intervals and not about absolute clock values
- We have to take two issues into account:

  - Network Asynchrony
  - Clock drift

NETWORK ASYNCHRONY

- p can't know at what exact time q sent the Promise, only that t0≤t1≤t2
  - p has to be conservative and assume that t1=t0
  - p holds lease until t3=t0+10s



Clock Drift

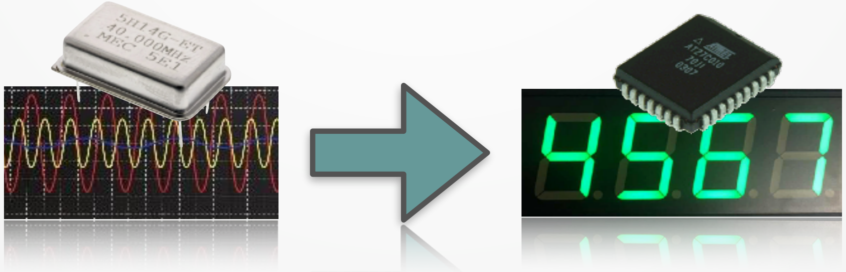
- To understand the clock drift issue, we have to describe clocks and time more formally and in more detail
- A clock at a process pi is a monotonically increasing function from real-time to some real value

THE CLOCK

- Each process pi has an associated clock Ci
- Ci(.) 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 Internals

- 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 ageing
  - The oscillator's manufacturer specifies a nominal frequency and an error bound



CLOCK INTERNALS

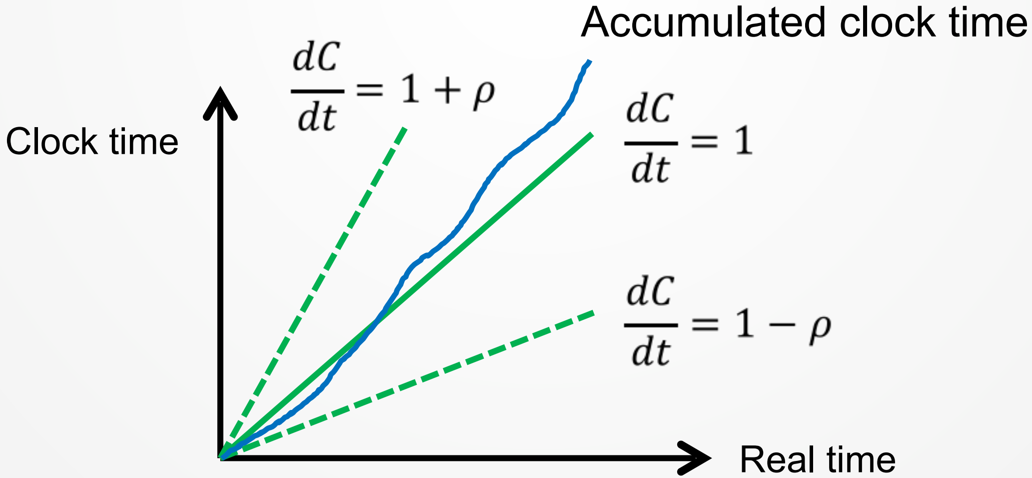
- The clock rate specifies how much the clock is increment each second of real time.
- For example: The counter increment by nominally 1,000,000 ticks per second, with an error bounded to ±100 ticks per second.
- From here on we normalise the clock rate such that 1.0 is the nominal rate and the error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

$$1 - \rho = \frac{1-\rho^2}{1+\rho} \approx \frac{1}{1+\rho} \leq \frac{dC}{dt} \leq 1 + \rho$$

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

CLOCK DRIFT

- Clock Drift is the accumulated effect of a clock rate that deviates from real time.
- Ideally, $\dfrac{dC}{dt} = 1$



Proposer Logic

- 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**

Proposer Logic

- Proposer must compensate by assuming its clock is running as slowly as possible ($\frac{dC}{dt} = 1 - \rho_p$), and compensate

  - $dt \leq 10$ at most 10 seconds real time
  - $dC = dt(1 - \rho_p) \leq 10(1 - \rho_p)$

Acceptor Logic

- 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.

Acceptor Logic

- Acceptor must assume its clock is running as fast as possible $(\frac{dC}{dt} = 1 + \rho_a)$, and compensate
    - $dt \geq 10$ for 10 seconds real time
    - $dC = dt(1 + \rho_a) \geq 10(1 + \rho_a)$

Acceptor Leases

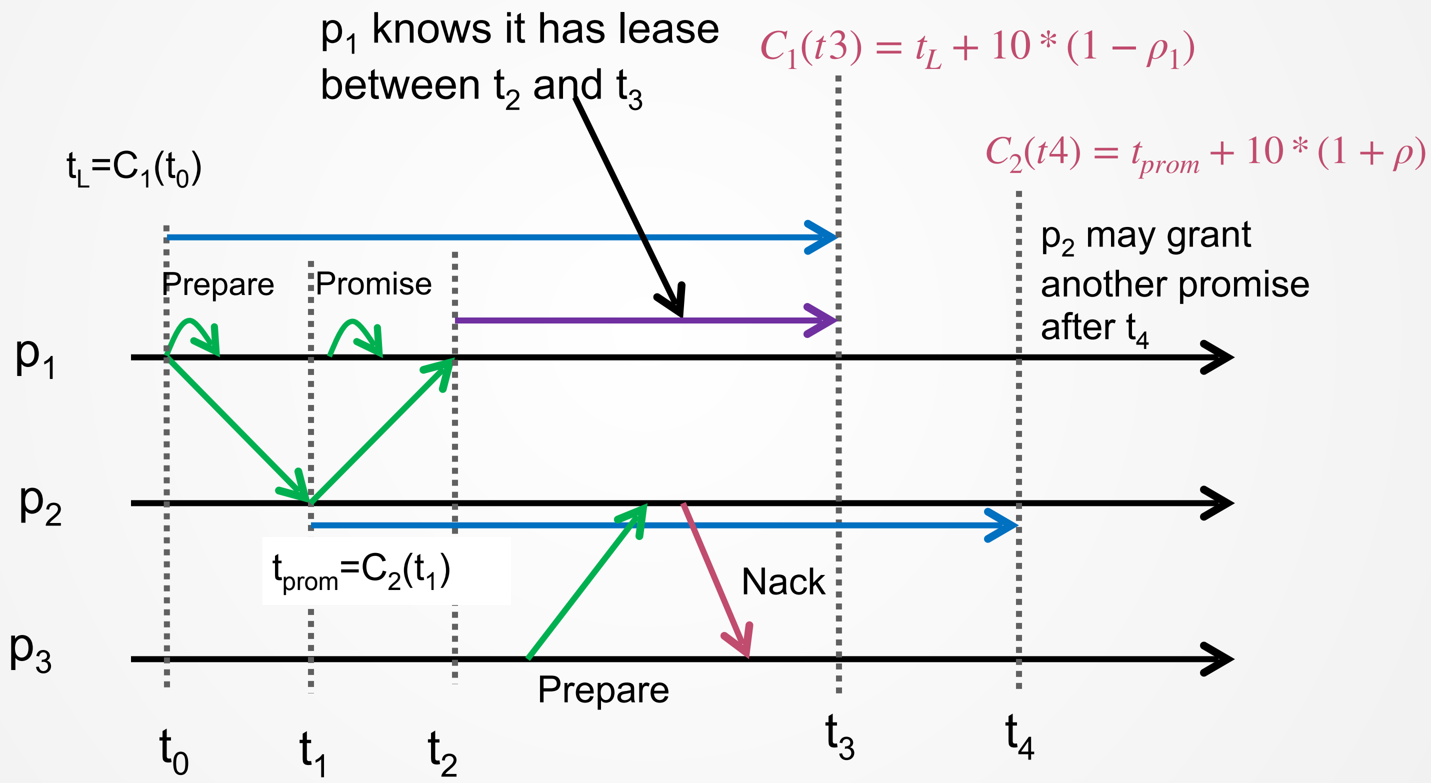
- Acceptors have new state variable
    - t_{prom} : Clock time when last promise was given
- If acceptor p_j gets Prepare(n) at time T and
    - $n > n_{prom}$ and $C_j(T) - t_{prom} > 10(1 + \rho_j)$
    - then give promise to reject rounds lower than n, and not give new promises within the next 10s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

Proposer Leases

- Proposer has a new state variable t_L
  - t_L : Clock time before "Prepare" was broadcasted
- If p_i gets promises from a majority, p_i knows that no other process can become leader until 10s after t_L

- Therefore, p_i can serve local state reads if:
  - Prepare phase is successful (quorum-received prepare)
  - $C_i(T) - t_L < 10(1 - \rho_i)$

Time Diagram



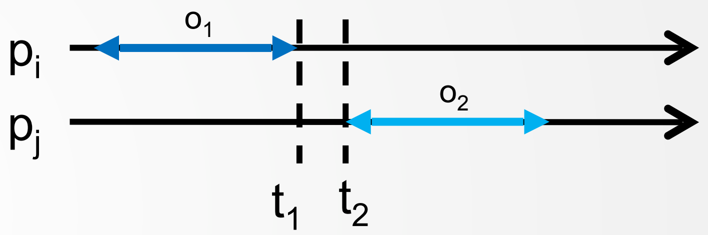
LEASE EXTENSIONS

- As long as p_i is alive and well it should remain the leader
- p_i can ask for lease extension in the meantime
  - 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

Interval Clocks

SHARED MEMORY REFRESHER

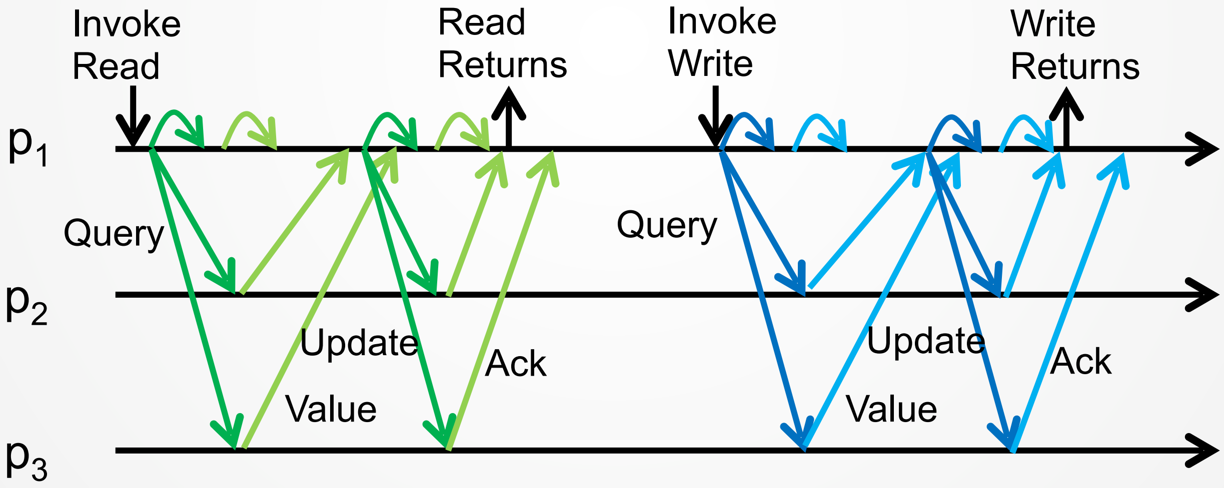
- A set of *atomic registers*
- Two operations:
  - **Write**(v): update register's value to v
  - **Read**(): return the register's value
- Correctness: **Linearizability**
  - If operation o_1 returns before operation o_2 is invoked, then o_1 must be ordered before o_2 (the linearization point of o_1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

OPTIMIZING READ OPERATION

- If in the query phase all processes in a majority set respond with the same timestamp-value pair (t_s , 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 clock 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

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
- Perfectly synchronized clocks :if each pair of clocks has δ = 0-synchronized
- Loosely synchronized clocks attempt to be as closely synchronized as possible, but give no guarantees
  - In practice, can be arbitrarily out of sync…(no guarantee)

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 DB (1.0) used loosely synchronised clocks in this way, and could therefore not guarantee linearizability. It later adopted a replicated log-based approach.**

Correctness – Logical clocks

If clocks are logical clocks (Lamport clocks) then the shared memory doesn't satisfy linearizability

Instead, it can only satisfy sequential consistency

PROBLEM SOLVED?

Using perfectly synchronized clocks (PSCs) guarantees linearizability, so just use PSCs and everything is good?

No, since PSCs are **impossible** to implement

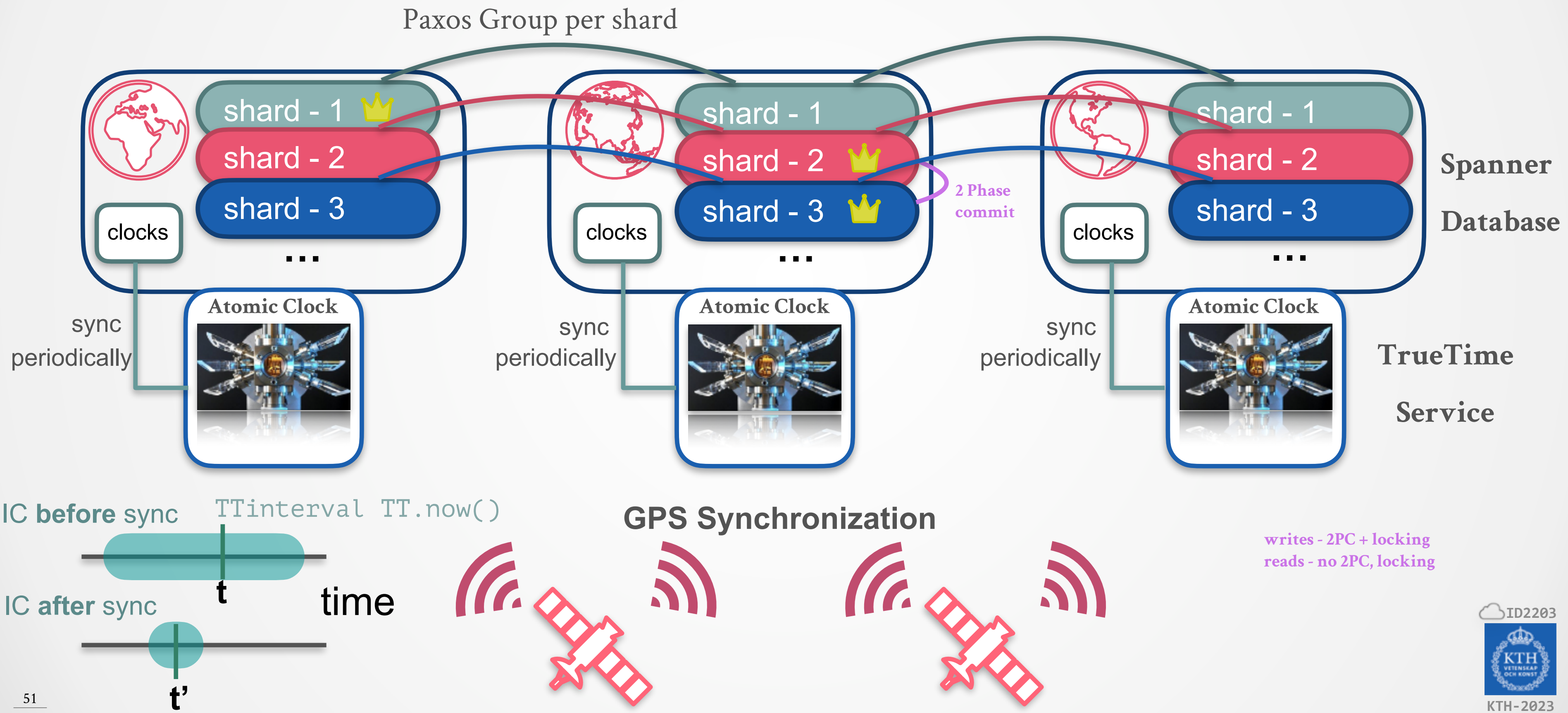
- Any measurement contains some uncertainty
- Synchronizing clocks across an asynchronous network adds more uncertainty

We can instead introduce a new practical kind of clock…

Interval Clocks

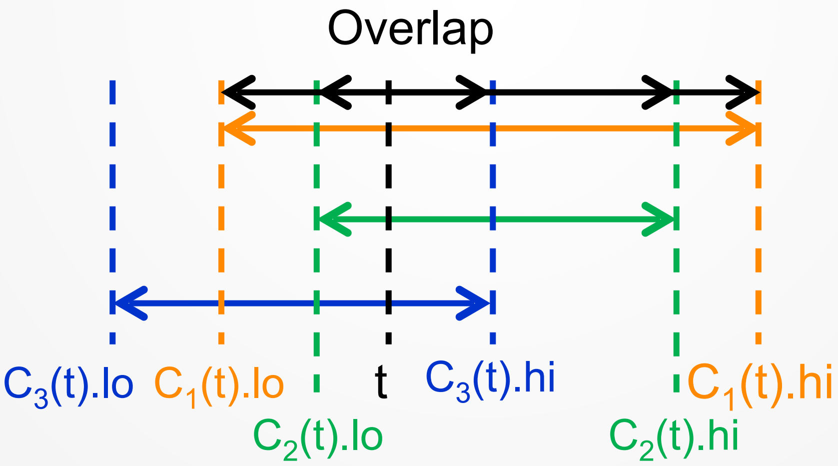
- An interval clock (IC) at process pi read at time t returns a pair Ci(t)=[lo, hi]
- Represents an interval $[C_i(t).lo \dots C_i(t).hi]$
- The correct time t is guaranteed to be in interval
    - $[C_i(t).lo \leq t \leq C_i(t).hi]$
- Synchronisation uncertainty is exposed in **width of interval**
- This is the strongest guarantee that can be implemented in practice
    - Too wide interval - can only hurt protocol performance
    - Too small interval - can hurt correctness

Clock Synchronization at Google



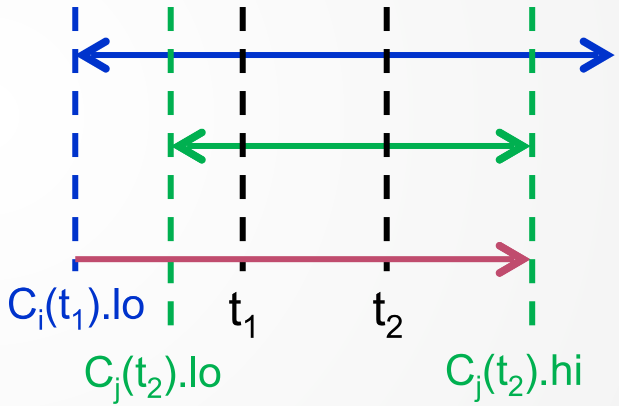
OVERLAPPING INTERVALS

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



Interval Clock Measurements

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



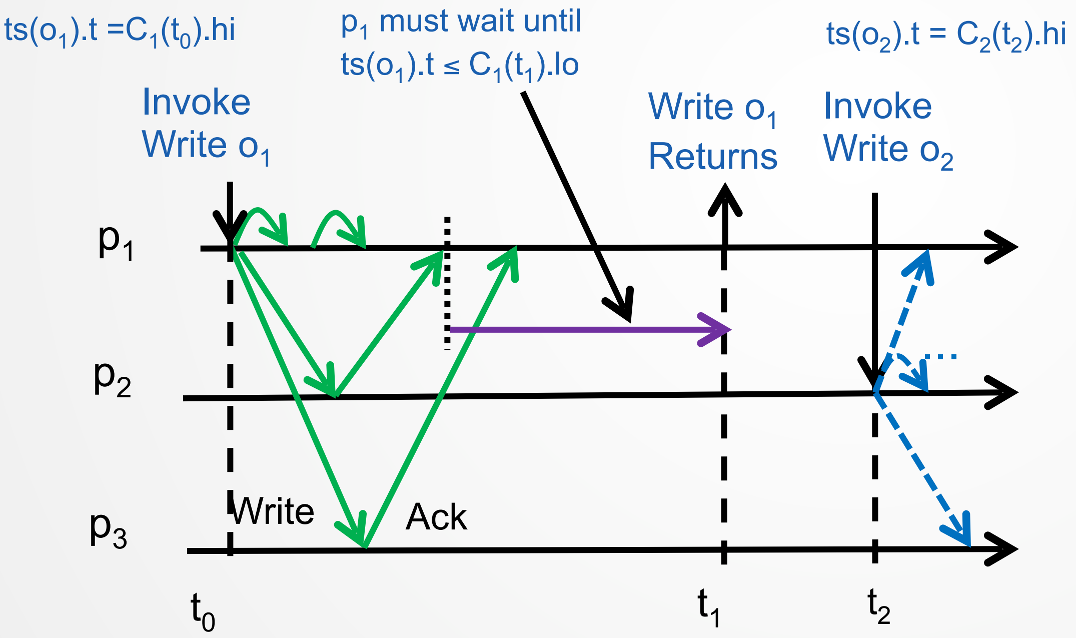
- $C_i(t_1).lo \leq t_1 < t_2 \leq C_j(t_2).hi$

BYPASSING THE QUERY PHASE

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

Intuition Behind Waiting

- o1 is allowed to return when ICs guarantee that later write will pick a higher timestamp



IC guarantee:

If $t_1 < t_2$ then

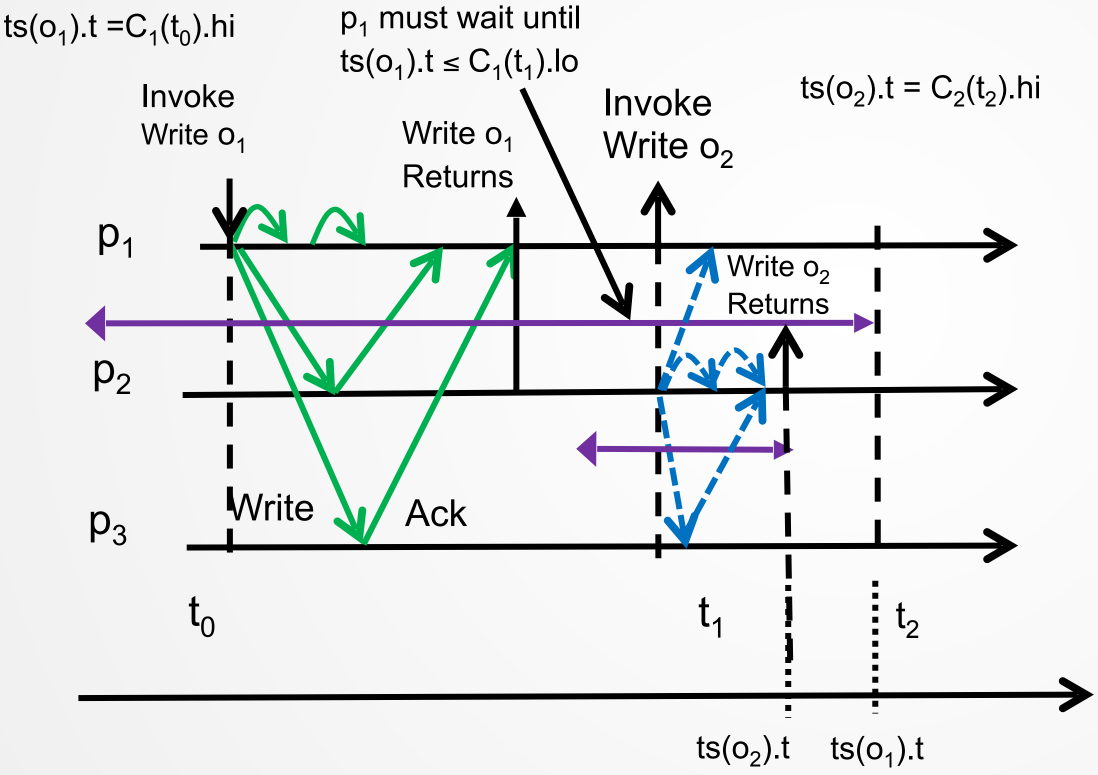
$C_1(t_1).lo < C_2(t_2).hi$

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 BEHIND WAITING



- If o1 is **completed** before o2 is invoked, then o1 must be **ordered** before o2

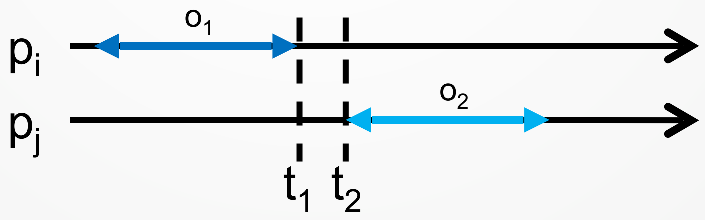
- **Case**: o1 does not wait
- o1 completes before o2 is issued: **no guarantee that o1 before o2** (ts(o1).t > ts(o2).t)

ID2203

KTH-2023

CORRECTNESS

- Updated algorithm satisfies 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 pi, and o_2 is invoked at t2 by pj
    - t_1<t_2 implies that $ts(o_1).t \leq Ci(t_1).lo < Cj(t_2).hi = ts(o_2).t$
    - Since $ts(o_1) < ts(o_2)$, the value in o1 is overwritten by the value of o_2

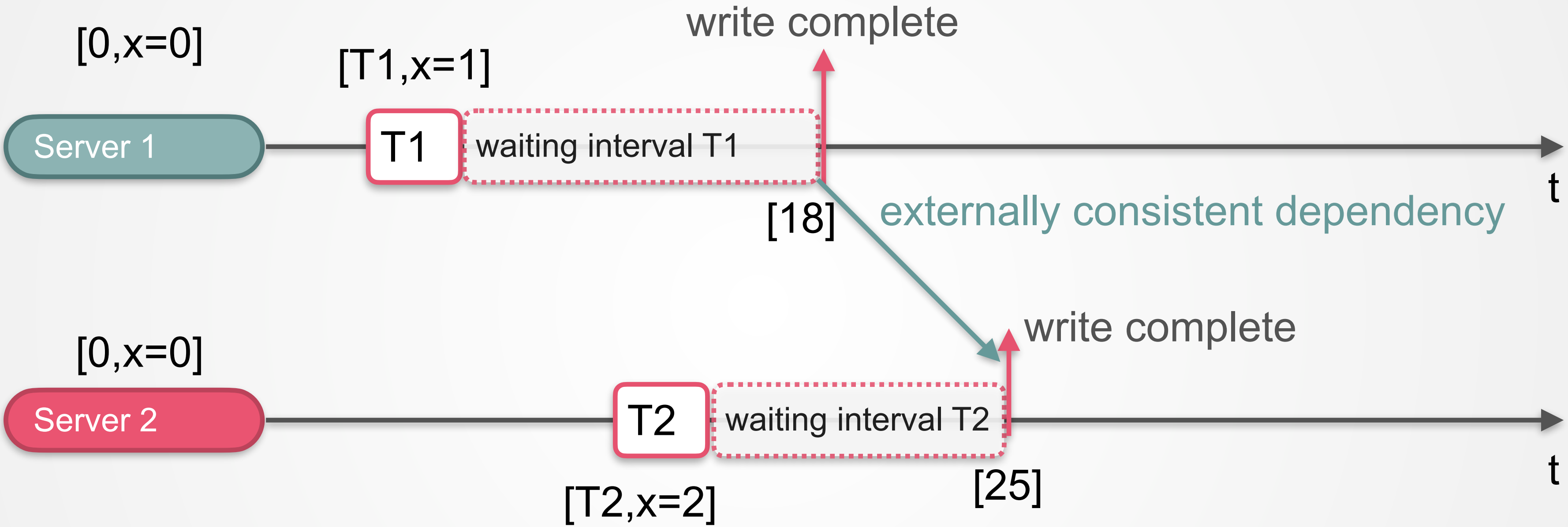


FINAL ALGORITHM

- **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**

Google Spanner - Fast Reads and MVCC



[0,x=0]	[0,y=0]	[0,z=0]
[18,x=1]	[17,y=9]	[5,z=12]
[25,x=2]	[19,y=6]	[37,z=3]
…	…	…

MVCC (Multi-Version Concurrency Control)

Reads Can Be consistently issued on a time stamp (e.g., at t = 18)