Asymmetric Linearizable Local Reads

Myles Thiessen Guy Khazma Sam Toueg Eyal de Lara

Department of Computer Science
University of Toronto

Linearizability

Linearizability

- Provides the *illusion* that a distributed database comprised of many replicas is a single replica that performs operations one at a time.

Linearizability

- Provides the *illusion* that a distributed database comprised of many replicas is a single replica that performs operations one at a time.

$\Longrightarrow$ If someone reads version V of the database, no one will later read a version $< V$.

Linearizability

- Provides the *illusion* that a distributed database comprised of many replicas is a single replica that performs operations one at a time.

  $\Longrightarrow$ If someone reads version V of the database, no one will later read a version $< V$.

- Leads to more *reliable* and *maintainable* software.

Linearizability

- Provides the *illusion* that a distributed database comprised of many replicas is a single replica that performs operations one at a time.

  $\Longrightarrow$ If someone reads version V of the database, no one will later read a version $< V$.

- Leads to more *reliable* and *maintainable* software.

State Machine Replication (SMR)

State Machine Replication (SMR)

- Databases use an SMR algorithm to provide linearizability and fault tolerance.

  e.g. Paxos, Raft, etc...

State Machine Replication (SMR)

- Databases use an SMR algorithm to provide linearizability and fault tolerance.

  e.g. Paxos, Raft, etc...

- SMR algorithms:

  1. Order all operations by electing a distinguished *leader* replica ().
  2. Provide fault tolerance by applying operations on *follower* replicas.

State Machine Replication (SMR)

- Databases use an SMR algorithm to provide linearizability and fault tolerance.

  e.g. Paxos, Raft, etc...

- SMR algorithms:

  1. Order all operations by electing a distinguished *leader* replica (👑).
  2. Provide fault tolerance by applying operations on *follower* replicas.

State Machine Replication (SMR)

- Databases use an SMR algorithm to provide linearizability and fault tolerance.

  e.g. Paxos, Raft, etc...

- SMR algorithms:

  1. Order all operations by electing a distinguished *leader* replica (👑).
  2. Provide fault tolerance by applying operations on *follower* replicas.

State Machine Replication (SMR)

- Databases use an SMR algorithm to provide linearizability and fault tolerance.

  e.g. Paxos, Raft, etc...

- SMR algorithms:

  1. Order all operations by electing a distinguished *leader* replica (👑).
  2. Provide fault tolerance by applying operations on *follower* replicas.

State Machine Replication (SMR)

- Databases use an SMR algorithm to provide linearizability and fault tolerance.

  e.g. Paxos, Raft, etc...

- SMR algorithms:

  1. Order all operations by electing a distinguished *leader* replica ().
  2. Provide fault tolerance by applying operations on *follower* replicas.

State Machine Replication (SMR)

- Databases use an SMR algorithm to provide linearizability and fault tolerance.

  e.g. Paxos, Raft, etc...

- SMR algorithms:

  1. Order all operations by electing a distinguished *leader* replica (👑).
  2. Provide fault tolerance by applying operations on *follower* replicas.

State Machine Replication (SMR)

- Databases use an SMR algorithm to provide linearizability and fault tolerance.

  e.g. Paxos, Raft, etc...

- SMR algorithms:

  1. Order all operations by electing a distinguished *leader* replica ().
  2. Provide fault tolerance by applying operations on *follower* replicas.

State Machine Replication (SMR)

- Databases use an SMR algorithm to provide linearizability and fault tolerance.

  e.g. Paxos, Raft, etc...

- SMR algorithms:

  1. Order all operations by electing a distinguished *leader* replica ().
  2. Provide fault tolerance by applying operations on *follower* replicas.

State Machine Replication (SMR)

- Databases use an SMR algorithm to provide linearizability and fault tolerance.

  e.g. Paxos, Raft, etc...

- SMR algorithms:

  1. Order all operations by electing a distinguished *leader* replica ().
  2. Provide fault tolerance by applying operations on *follower* replicas.

State Machine Replication (SMR)

- Databases use an SMR algorithm to provide linearizability and fault tolerance.

  e.g. Paxos, Raft, etc...

- SMR algorithms:

  1. Order all operations by electing a distinguished *leader* replica ().
  2. Provide fault tolerance by applying operations on *follower* replicas.

State Machine Replication (SMR)

- Databases use an SMR algorithm to provide linearizability and fault tolerance.

  e.g. Paxos, Raft, etc...

- SMR algorithms:

  1. Order all operations by electing a distinguished *leader* replica ().
  2. Provide fault tolerance by applying operations on *follower* replicas.

State Machine Replication (SMR)

- Databases use an SMR algorithm to provide linearizability and fault tolerance.

  e.g. Paxos, Raft, etc...

- SMR algorithms:

  1. Order all operations by electing a distinguished *leader* replica ().
  2. Provide fault tolerance by applying operations on *follower* replicas.

State Machine Replication (SMR)

- Databases use an SMR algorithm to provide linearizability and fault tolerance.

  e.g. Paxos, Raft, etc...

- SMR algorithms:

  1. Order all operations by electing a distinguished *leader* replica ().
  2. Provide fault tolerance by applying operations on *follower* replicas.

State Machine Replication (SMR)

- Databases use an SMR algorithm to provide linearizability and fault tolerance.

  e.g. Paxos, Raft, etc...

- SMR algorithms:

  1. Order all operations by electing a distinguished *leader* replica ().
  2. Provide fault tolerance by applying operations on *follower* replicas.

State Machine Replication (SMR)

- Databases use an SMR algorithm to provide linearizability and fault tolerance.

  e.g. Paxos, Raft, etc...

- SMR algorithms:

  1. Order all operations by electing a distinguished *leader* replica ().
  2. Provide fault tolerance by applying operations on *follower* replicas.

State Machine Replication (SMR)

- Databases use an SMR algorithm to provide linearizability and fault tolerance.

  e.g. Paxos, Raft, etc...

- SMR algorithms:

  1. Order all operations by electing a distinguished *leader* replica ().
  2. Provide fault tolerance by applying operations on *follower* replicas.

Improving Read Performance

Improving Read Performance

- Many databases improve performance by **not replicating reads**.

Improving Read Performance

- Many databases improve performance by **not replicating reads**.
  - Possible because reads do not modify the database's state.

Improving Read Performance

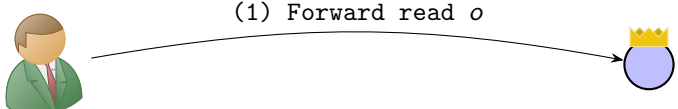
- Many databases improve performance by **not replicating reads**.
  - Possible because reads do not modify the database's state.
- Instead all reads are immediately applied on the leader's replica ().

Improving Read Performance

- Many databases improve performance by **not replicating reads**.
  - Possible because reads do not modify the database's state.
- Instead all reads are immediately applied on the leader's replica ().

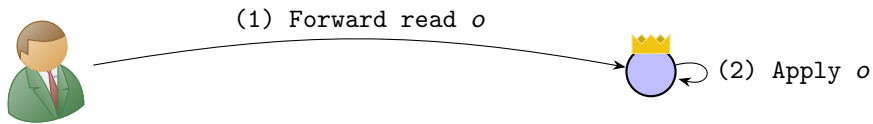
Improving Read Performance

- Many databases improve performance by **not replicating reads**.
  - Possible because reads do not modify the database's state.
- Instead all reads are immediately applied on the leader's replica ( ).



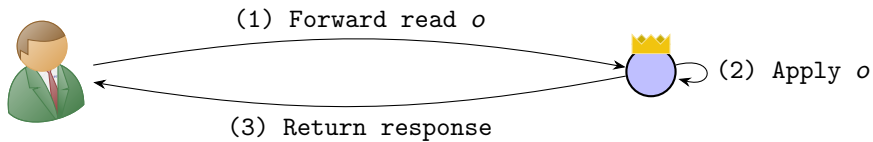
Improving Read Performance

- Many databases improve performance by **not replicating reads**.
  - Possible because reads do not modify the database's state.
- Instead all reads are immediately applied on the leader's replica ( ).



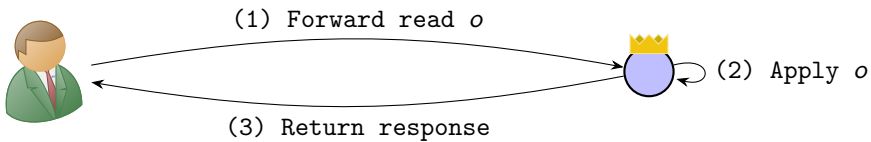
Improving Read Performance

- Many databases improve performance by **not replicating reads**.
  - Possible because reads do not modify the database's state.
- Instead all reads are immediately applied on the leader's replica ( ).



Improving Read Performance

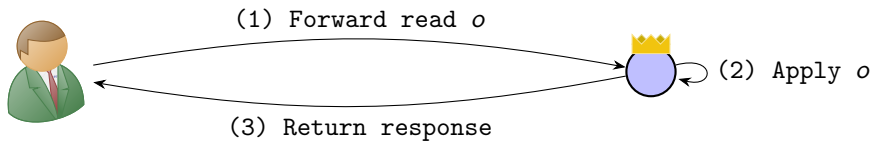
- Many databases improve performance by **not replicating reads**.
  - Possible because reads do not modify the database's state.
- Instead all reads are immediately applied on the leader's replica ().



✓ Faster than Paxos and Raft because it eliminates one round of communication.

Improving Read Performance

- Many databases improve performance by **not replicating reads**.
  - Possible because reads do not modify the database's state.
- Instead all reads are immediately applied on the leader's replica ( ).



✓ Faster than Paxos and Raft because it eliminates one round of communication.

✗ When the leader is in another datacenter read latency can be **100s of milliseconds!**

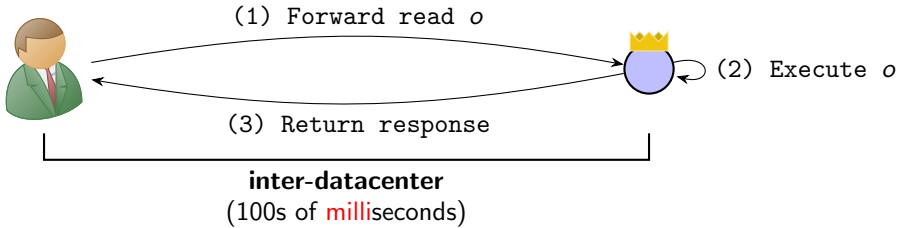
Local Read Algorithms

Local Read Algorithms

- Reduces read latency by enabling **all** replicas to apply reads locally.

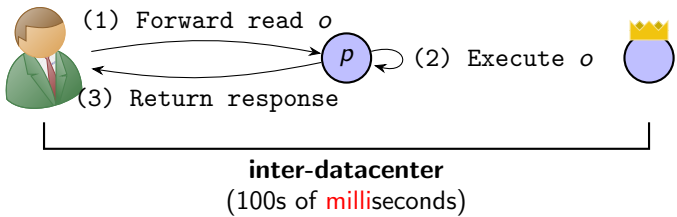
Local Read Algorithms

- Reduces read latency by enabling **all** replicas to apply reads locally.



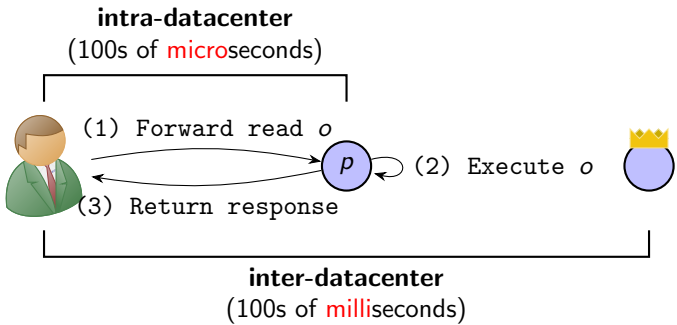
Local Read Algorithms

- Reduces read latency by enabling **all** replicas to apply reads locally.



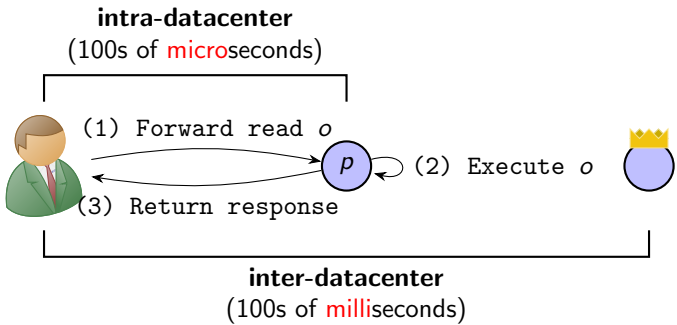
Local Read Algorithms

- Reduces read latency by enabling **all** replicas to apply reads locally.



Local Read Algorithms

- Reduces read latency by enabling **all** replicas to apply reads locally.



- In the best case read latency is reduced by **orders of magnitude!**

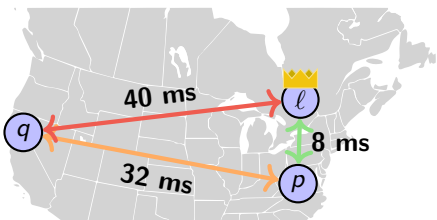
Local Read Algorithms in the Worst Case

Local Read Algorithms in the Worst Case

- Reads block while the database transitions to a newer version to guarantee: If someone reads version V of the database, no one will later read a version $< V$.

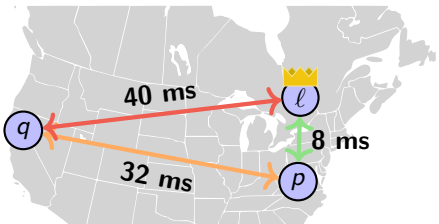
Local Read Algorithms in the Worst Case

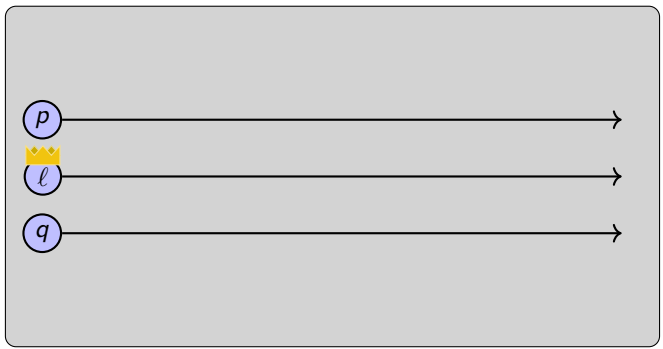
- Reads block while the database transitions to a newer version to guarantee: If someone reads version V of the database, no one will later read a version $< V$.



Local Read Algorithms in the Worst Case

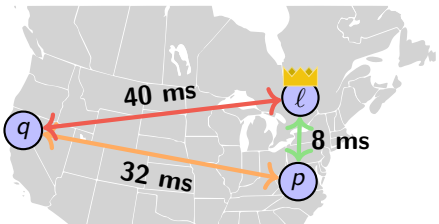
- Reads block while the database transitions to a newer version to guarantee: If someone reads version V of the database, no one will later read a version $< V$.

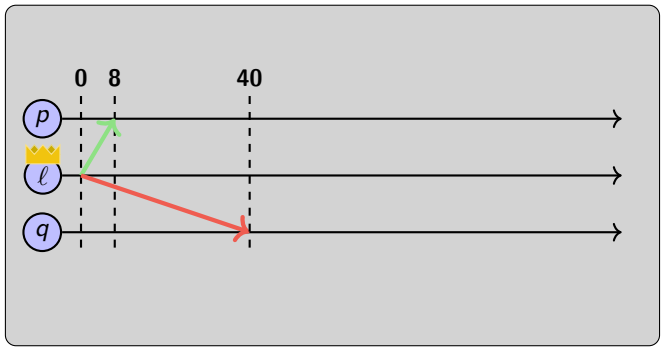




Local Read Algorithms in the Worst Case

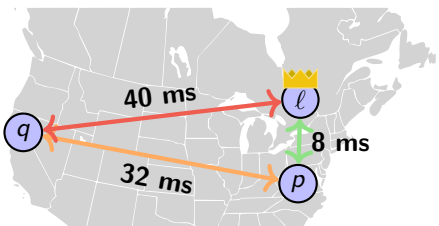
- Reads block while the database transitions to a newer version to guarantee: If someone reads version V of the database, no one will later read a version $< V$.

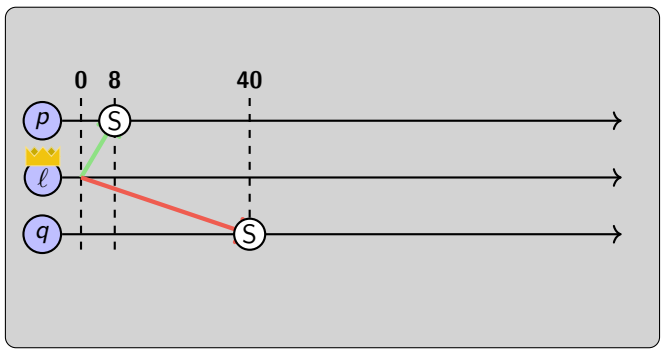




Local Read Algorithms in the Worst Case

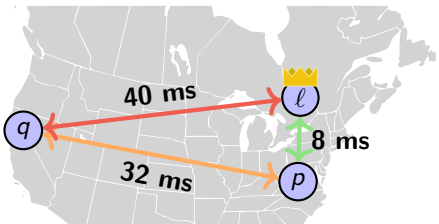
- Reads block while the database transitions to a newer version to guarantee: If someone reads version V of the database, no one will later read a version $< V$.

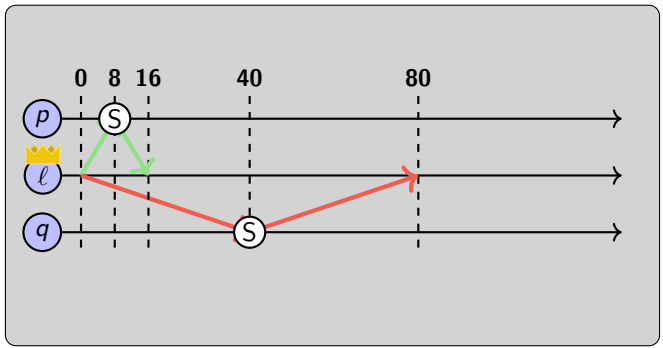




Local Read Algorithms in the Worst Case

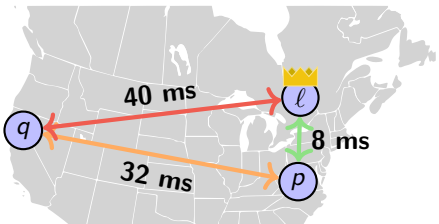
- Reads block while the database transitions to a newer version to guarantee: If someone reads version V of the database, no one will later read a version $< V$.

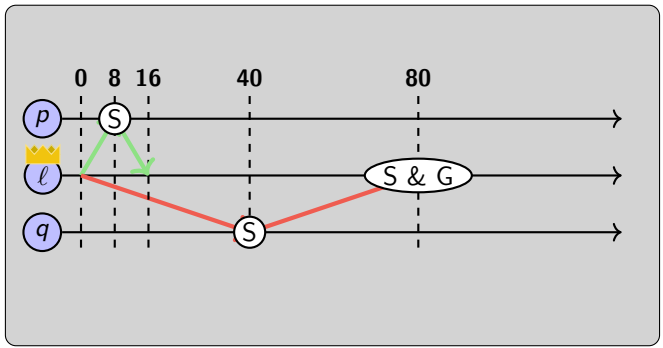




Local Read Algorithms in the Worst Case

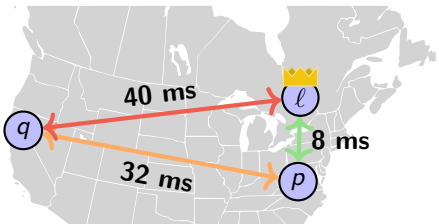
- Reads block while the database transitions to a newer version to guarantee: If someone reads version V of the database, no one will later read a version $< V$.

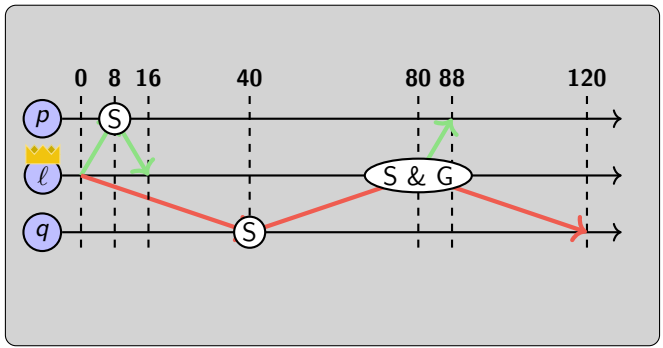




Local Read Algorithms in the Worst Case

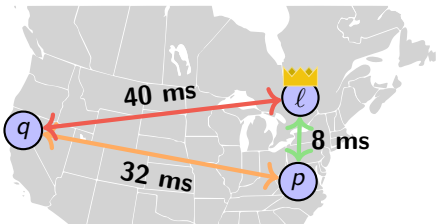
- Reads block while the database transitions to a newer version to guarantee: If someone reads version V of the database, no one will later read a version $< V$.

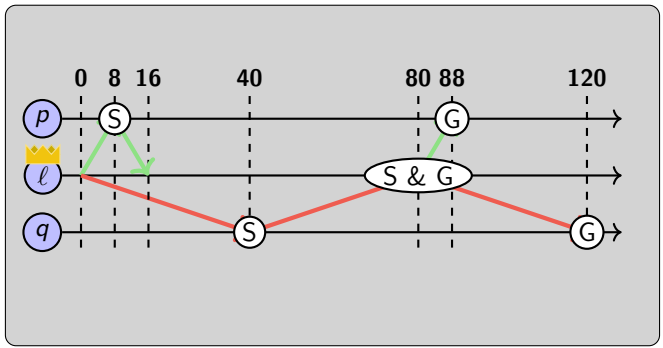




Local Read Algorithms in the Worst Case

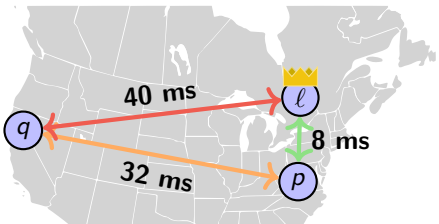
- Reads block while the database transitions to a newer version to guarantee: If someone reads version V of the database, no one will later read a version $< V$.

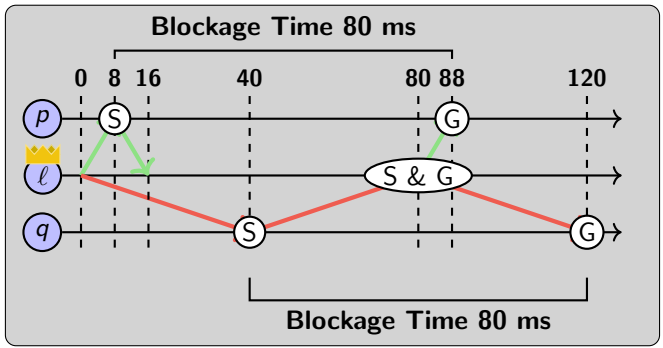




Local Read Algorithms in the Worst Case

- Reads block while the database transitions to a newer version to guarantee: If someone reads version V of the database, no one will later read a version $< V$.

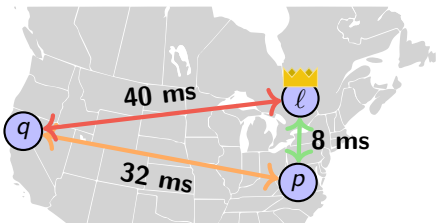




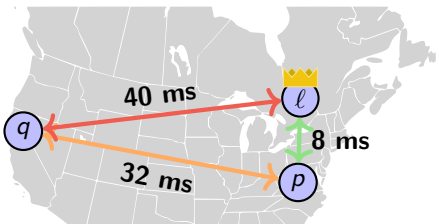
- During fault-free periods with fixed message delays, reads can block for **80 ms**.

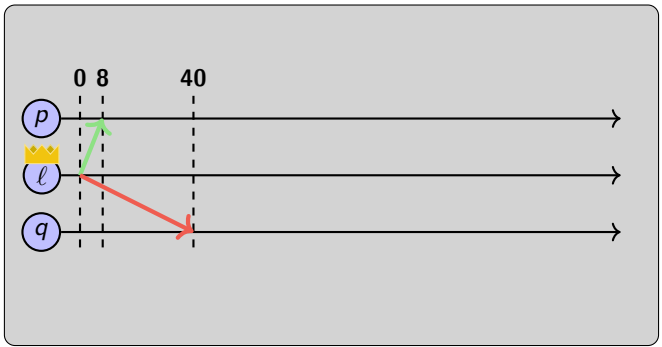
The State-Of-The-Art ("Delayed Stamping")

The State-Of-The-Art ("Delayed Stamping")

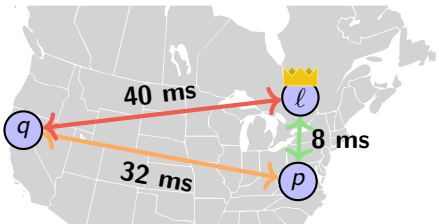


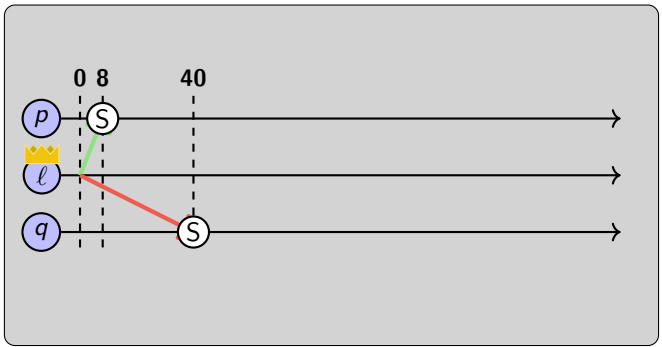
The State-Of-The-Art ("Delayed Stamping")



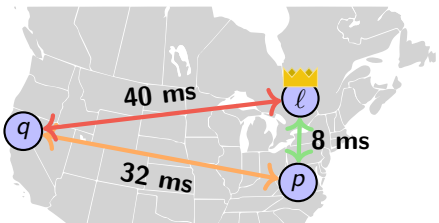


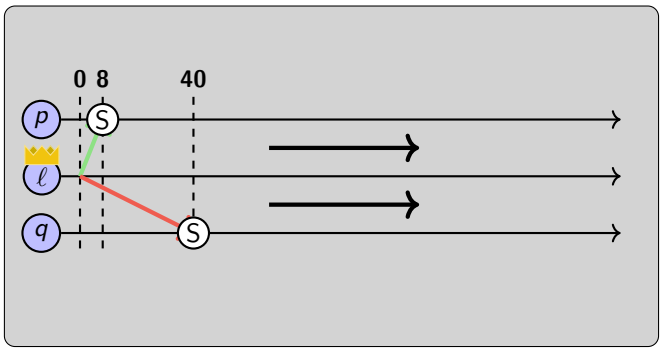
The State-Of-The-Art ("Delayed Stamping")



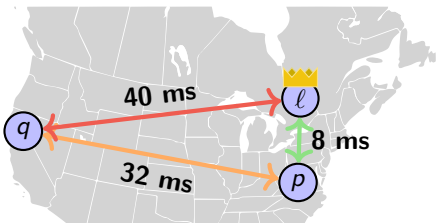


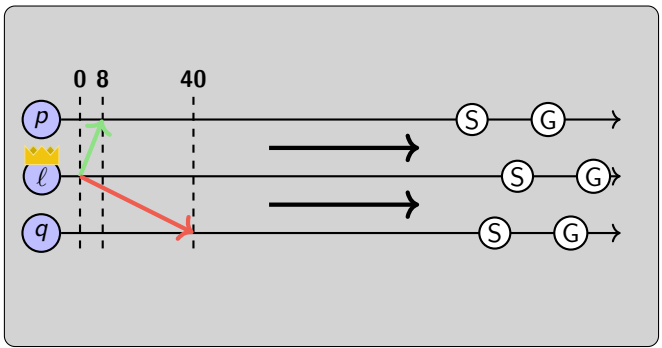
The State-Of-The-Art ("Delayed Stamping")





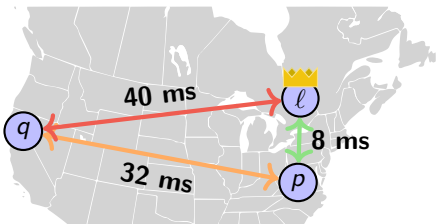
The State-Of-The-Art ("Delayed Stamping")

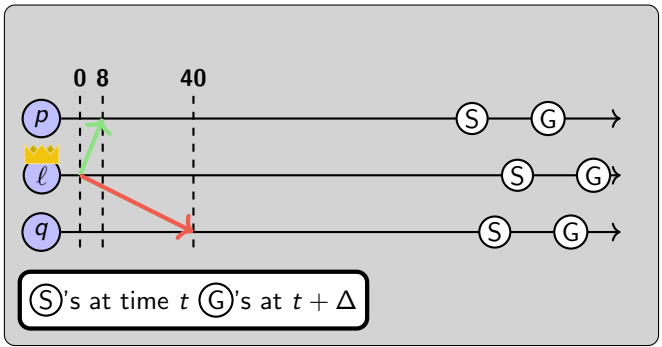




The State-Of-The-Art ("Delayed Stamping")

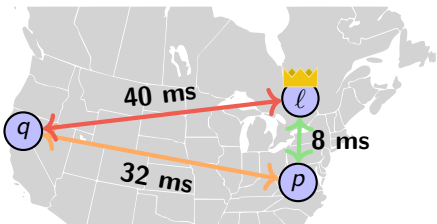
- **Assumes:** globally synchronized clocks with skew that is always bounded by Δ.

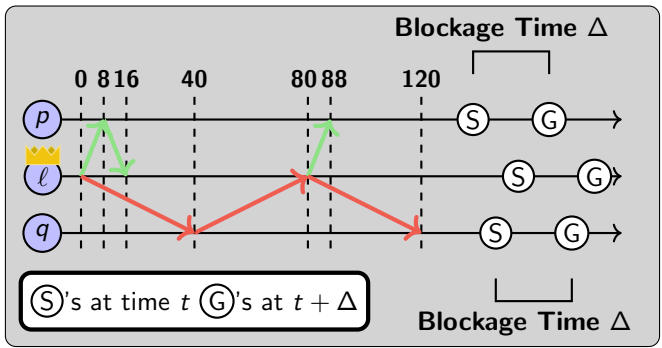




The State-Of-The-Art ("Delayed Stamping")

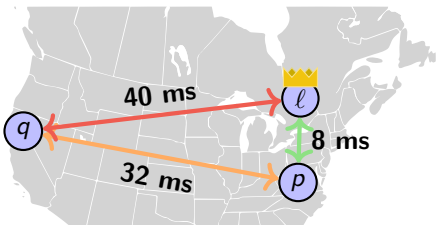
- **Assumes:** globally synchronized clocks with skew that is always bounded by Δ.

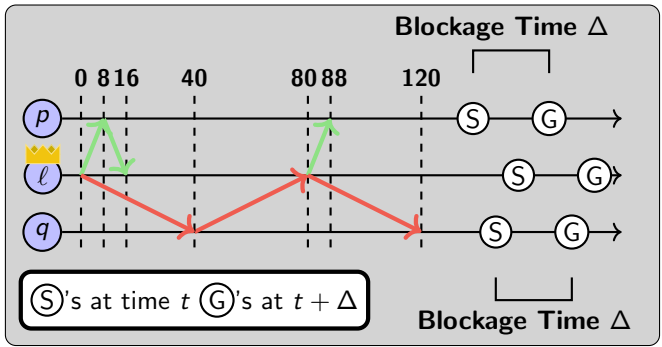




The State-Of-The-Art ("Delayed Stamping")

- **Assumes:** globally synchronized clocks with skew that is always bounded by Δ.

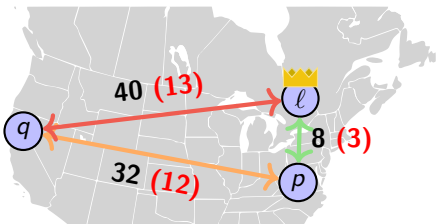


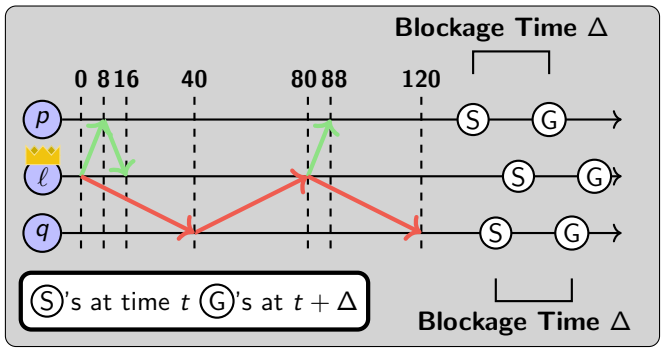


- Without specialized clock synchronization hardware (e.g. Truetime), $\Delta \gtrapprox$ **40 ms**.

The State-Of-The-Art ("Delayed Stamping")

- **Assumes:** globally synchronized clocks with skew that is always bounded by Δ.

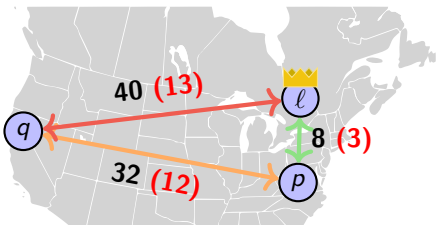


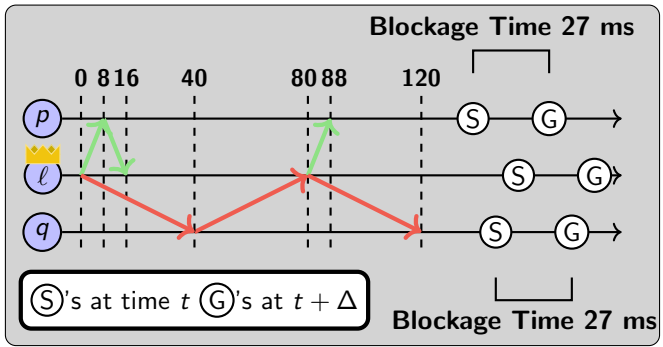


- Without specialized clock synchronization hardware (e.g. Truetime), $\Delta \gtrapprox$ **40 ms**.

The State-Of-The-Art ("Delayed Stamping")

- **Assumes:** globally synchronized clocks with skew that is always bounded by Δ.





- Without specialized clock synchronization hardware (e.g. Truetime), $\Delta \geq$ **27 ms**.

Can lower blockage times be achieved?

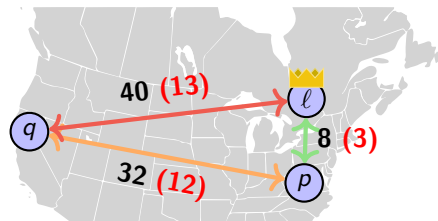
This Work: Lower Blockage Times by Leveraging Network Asymmetry

This Work: Lower Blockage Times by Leveraging Network Asymmetry

- Two new local reads algorithms that reduce worst-case blockage times at followers:
  1. close to the leader, or
  2. close to the network's center.

This Work: Lower Blockage Times by Leveraging Network Asymmetry

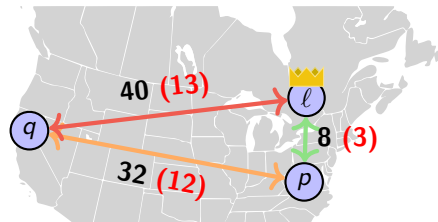
- Two new local reads algorithms that reduce worst-case blockage times at followers:
  1. close to the leader, or
  2. close to the network's center.



	Blockage Time @ p	Blockage Time @ q
Eager Stamping	80 ms	80 ms
Delayed Stamping	27 ms	27 ms

This Work: Lower Blockage Times by Leveraging Network Asymmetry

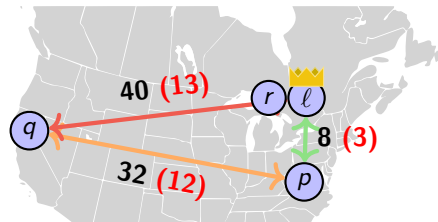
- Two new local reads algorithms that reduce worst-case blockage times at followers:
  1. close to the leader, or
  2. close to the network's center.



	Blockage Time @ p	Blockage Time @ q
Eager Stamping	80 ms	80 ms
Delayed Stamping	27 ms	27 ms
Pairwise-Leader (ours)	10 ms	54 ms

This Work: Lower Blockage Times by Leveraging Network Asymmetry

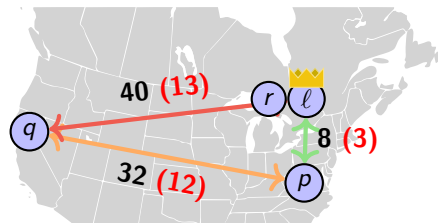
- Two new local reads algorithms that reduce worst-case blockage times at followers:
  1. close to the leader, or
  2. close to the network's center.



	Blockage Time @ p	Blockage Time @ q
Eager Stamping	80 ms	80 ms
Delayed Stamping	27 ms	27 ms
Pairwise-Leader (ours)	10 ms	54 ms

This Work: Lower Blockage Times by Leveraging Network Asymmetry

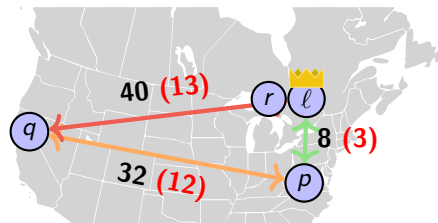
- Two new local reads algorithms that reduce worst-case blockage times at followers:
  1. close to the leader, or
  2. close to the network's center.



	Blockage Time @ p	Blockage Time @ q	Blockage Time @ r
Eager Stamping	80 ms	80 ms	80 ms
Delayed Stamping	27 ms	27 ms	27 ms
Pairwise-Leader (ours)	10 ms	54 ms	≈ 0 ms

This Work: Lower Blockage Times by Leveraging Network Asymmetry

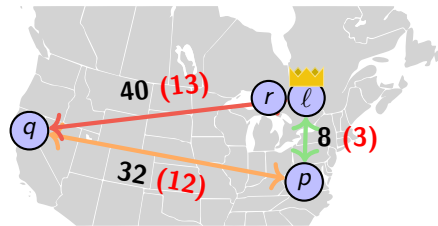
- Two new local reads algorithms that reduce worst-case blockage times at followers:
  1. close to the leader, or
  2. close to the network's center.



	Blockage Time @ p	Blockage Time @ q	Blockage Time @ r
Eager Stamping	80 ms	80 ms	80 ms
Delayed Stamping	27 ms	27 ms	27 ms
Pairwise-Leader (ours)	10 ms	54 ms	10s-100s of μs

This Work: Lower Blockage Times by Leveraging Network Asymmetry

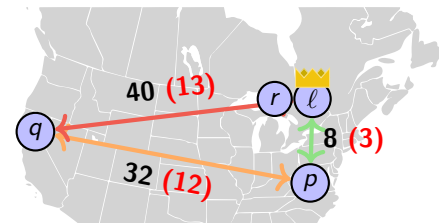
- Two new local reads algorithms that reduce worst-case blockage times at followers:
  1. close to the leader, or
  2. close to the network's center.



	Blockage Time @ p	Blockage Time @ q	Blockage Time @ r
Eager Stamping	80 ms	80 ms	80 ms
Delayed Stamping	27 ms	27 ms	27 ms
Pairwise-Leader (ours)	10 ms	54 ms	10s-100s of μs
Pairwise-All (ours)	20 ms	27 ms	27 ms

This Work: Lower Blockage Times by Leveraging Network Asymmetry

- Two new local reads algorithms that reduce worst-case blockage times at followers:
  1. close to the leader, or
  2. close to the network's center.



	Blockage Time @ p	Blockage Time @ q	Blockage Time @ r
Eager Stamping	80 ms	80 ms	80 ms
Delayed Stamping	27 ms	27 ms	27 ms
Pairwise-Leader (ours)	10 ms	54 ms	10s-100s of μs
Pairwise-All (ours)	20 ms	27 ms	27 ms

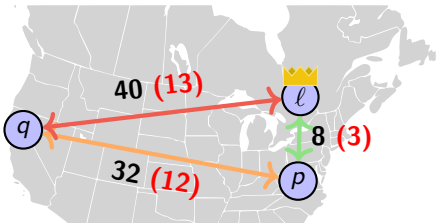
Pairwise-Leader

Pairwise-Leader

- **Core idea:** schedule Ⓢ and Ⓖ using a pairwise event scheduling primitive instead of globally synchronized clocks (like Delayed Stamping algorithms do).

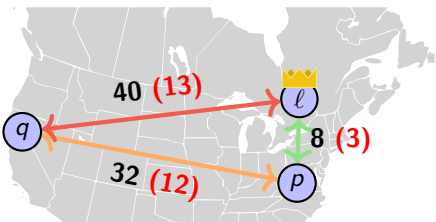
Pairwise-Leader

- **Core idea:** schedule Ⓢ and Ⓖ using a pairwise event scheduling primitive instead of globally synchronized clocks (like Delayed Stamping algorithms do).



Pairwise-Leader

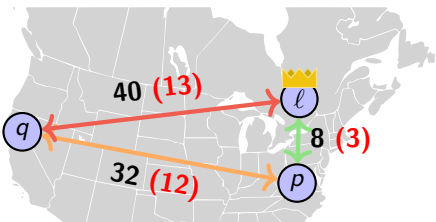
- **Core idea:** schedule Ⓢ and Ⓖ using a pairwise event scheduling primitive instead of globally synchronized clocks (like Delayed Stamping algorithms do).



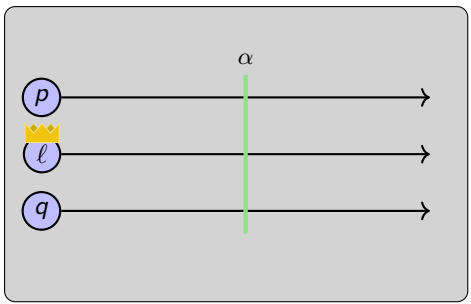
Event Structure

Pairwise-Leader

- **Core idea:** schedule Ⓢ and Ⓖ using a pairwise event scheduling primitive instead of globally synchronized clocks (like Delayed Stamping algorithms do).

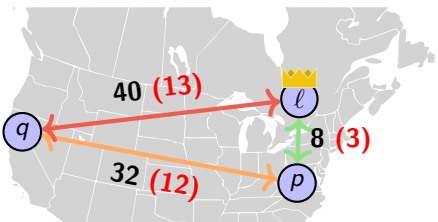


Event Structure

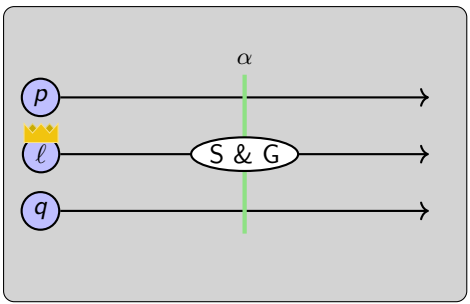


Pairwise-Leader

- **Core idea:** schedule Ⓢ and Ⓖ using a pairwise event scheduling primitive instead of globally synchronized clocks (like Delayed Stamping algorithms do).

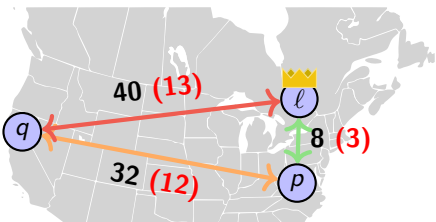


Event Structure

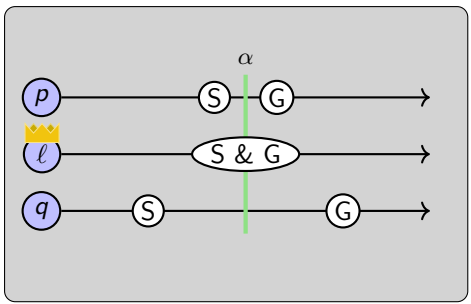


Pairwise-Leader

- **Core idea:** schedule Ⓢ and Ⓖ using a pairwise event scheduling primitive instead of globally synchronized clocks (like Delayed Stamping algorithms do).

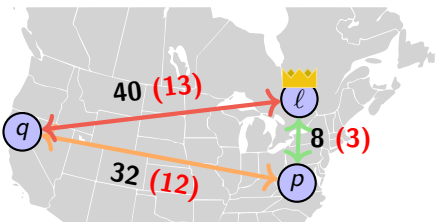


Event Structure

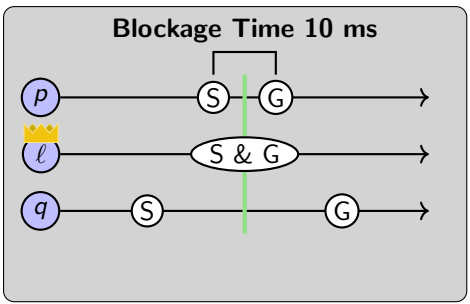


Pairwise-Leader

- **Core idea:** schedule Ⓢ and Ⓖ using a pairwise event scheduling primitive instead of globally synchronized clocks (like Delayed Stamping algorithms do).

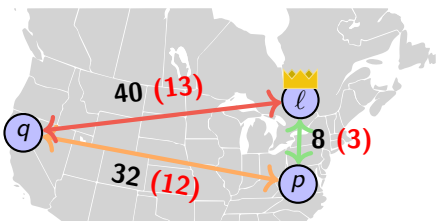


Event Structure

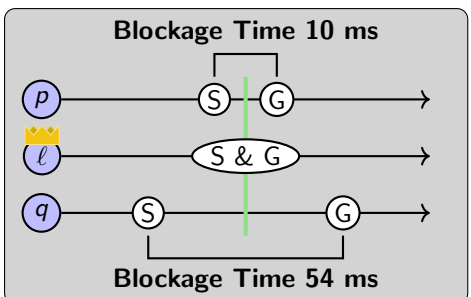


Pairwise-Leader

- **Core idea:** schedule Ⓢ and Ⓖ using a pairwise event scheduling primitive instead of globally synchronized clocks (like Delayed Stamping algorithms do).

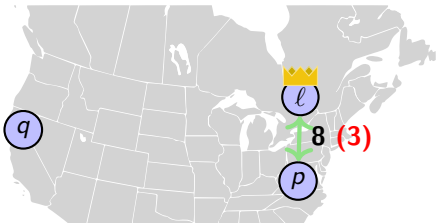


Event Structure



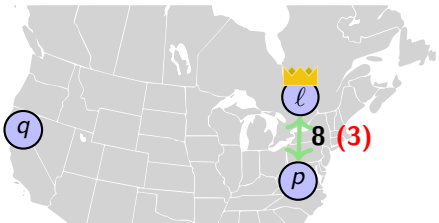
Pairwise-Leader

- **Core idea:** schedule Ⓢ and Ⓖ using a pairwise event scheduling primitive instead of globally synchronized clocks (like Delayed Stamping algorithms do).



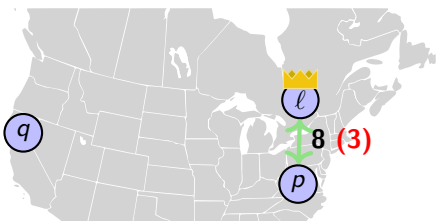
Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.



Pairwise-Leader

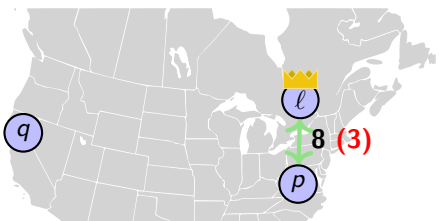
- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.



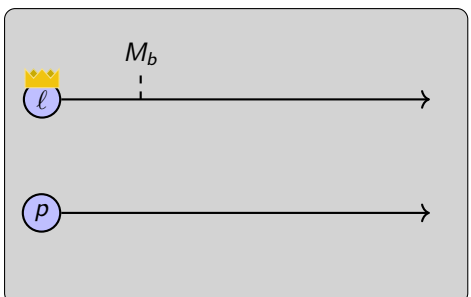
Marker establishment

Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.

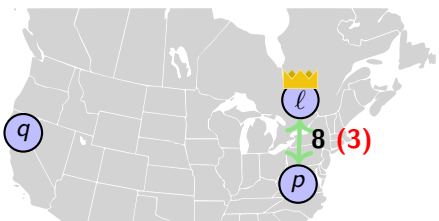


Marker establishment

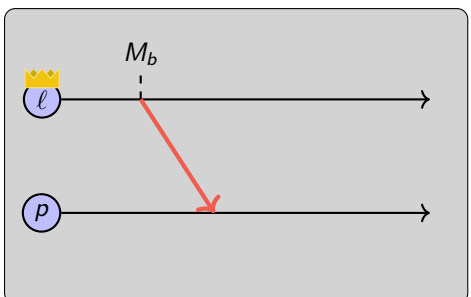


Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.

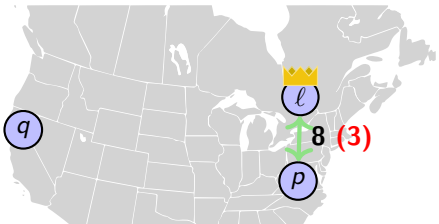


Marker establishment

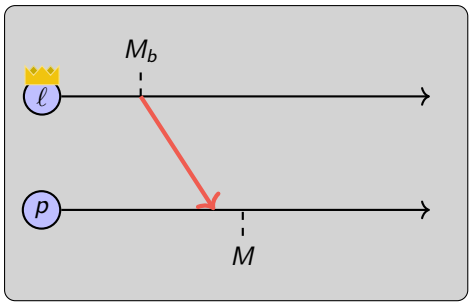


Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.

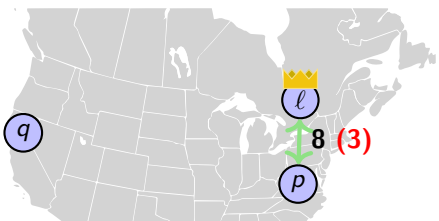


Marker establishment

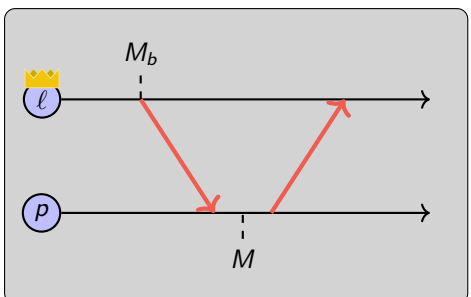


Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.

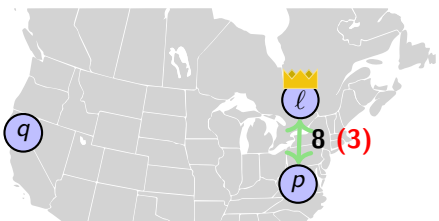


Marker establishment

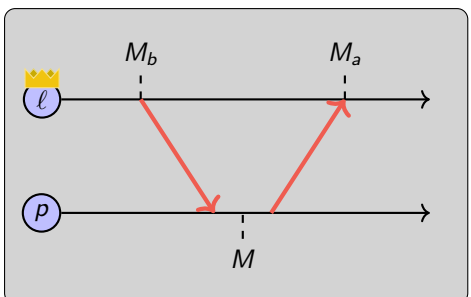


Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.

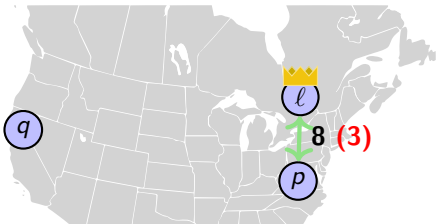


Marker establishment

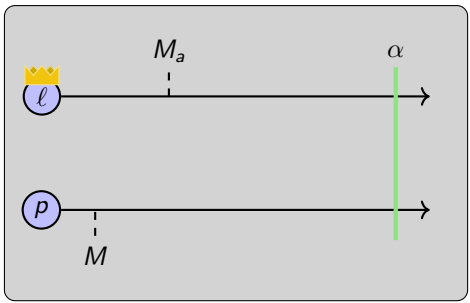


Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.

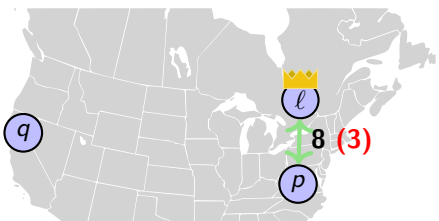


Scheduling Ⓢ on p

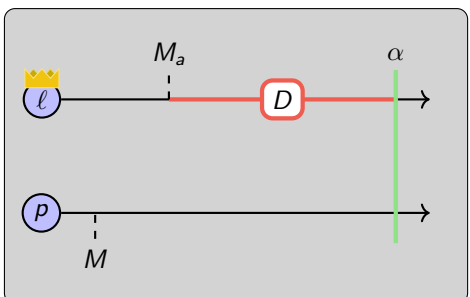


Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.

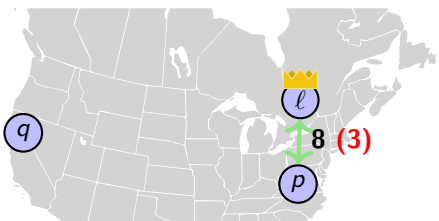


Scheduling Ⓢ on p

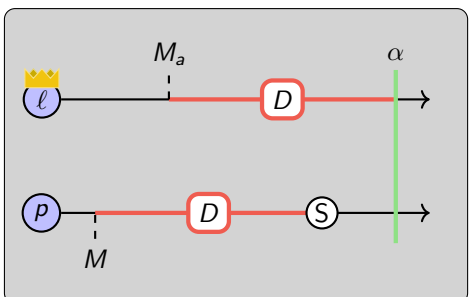


Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.

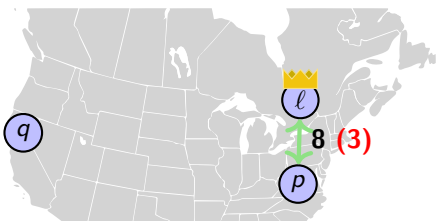


Scheduling Ⓢ on p

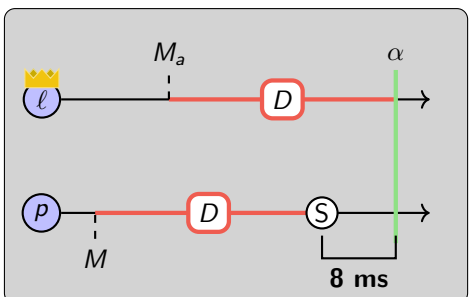


Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.

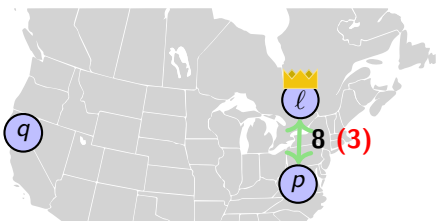


Scheduling Ⓢ on p

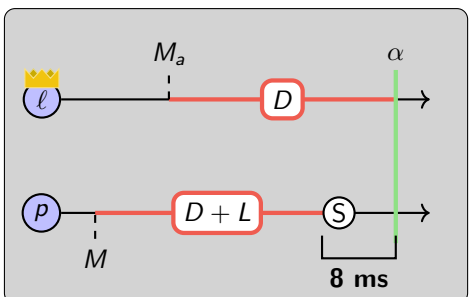


Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.

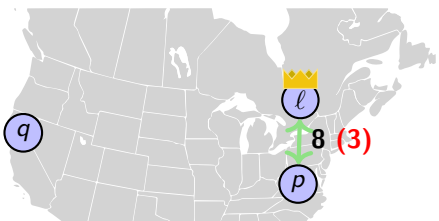


Scheduling Ⓢ on p

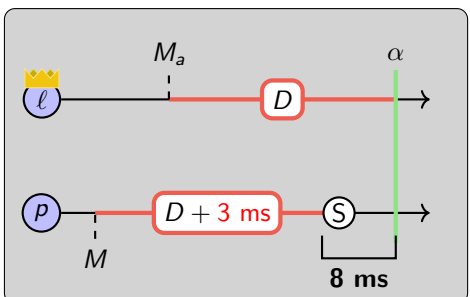


Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.

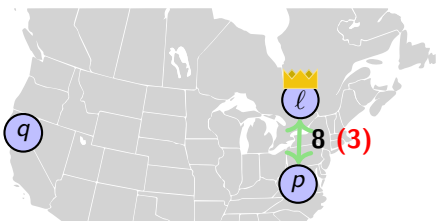


Scheduling Ⓢ on p

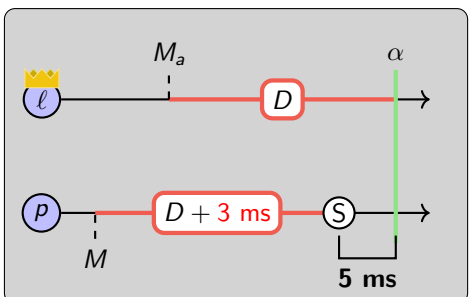


Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.

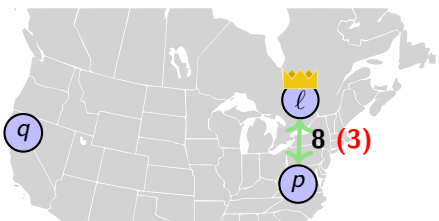


Scheduling Ⓢ on p

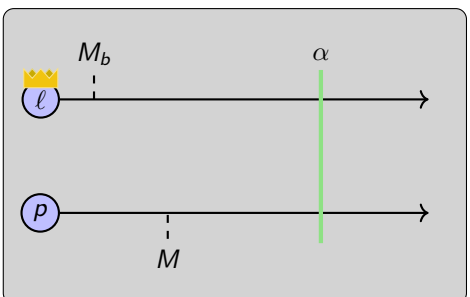


Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.

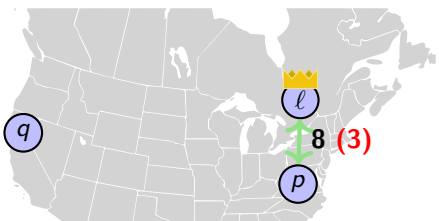


Scheduling Ⓖ on p

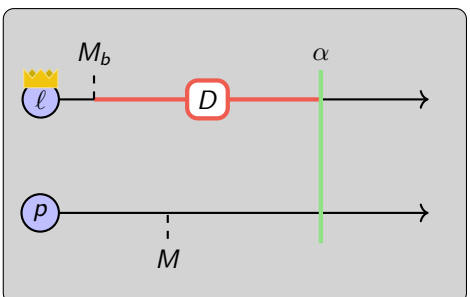


Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.

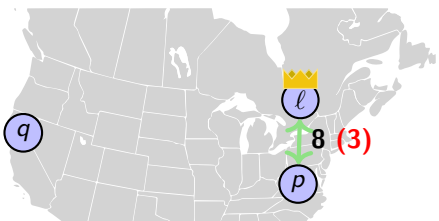


Scheduling Ⓖ on p

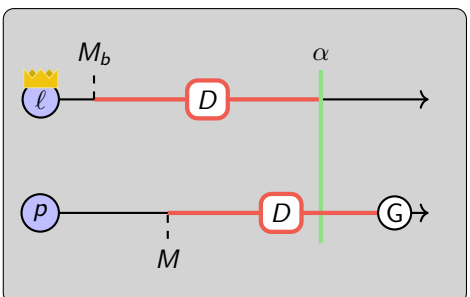


Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.

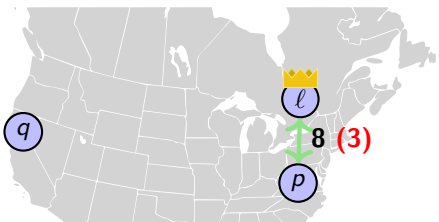


Scheduling Ⓖ on p

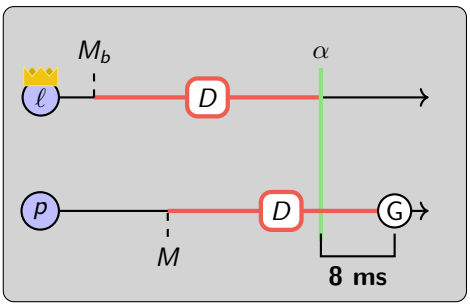


Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.

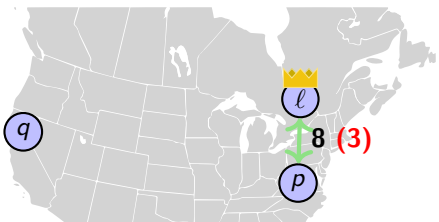


Scheduling (G) on p

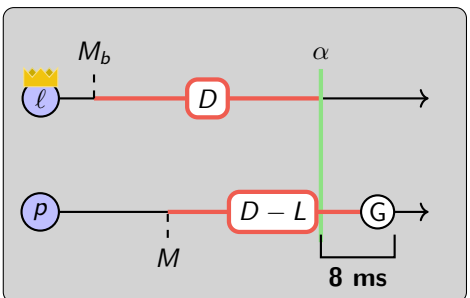


Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.

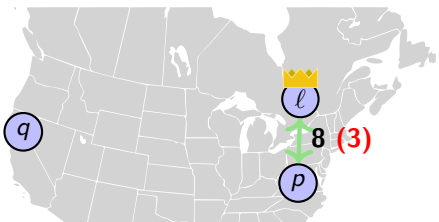


Scheduling (G) on p

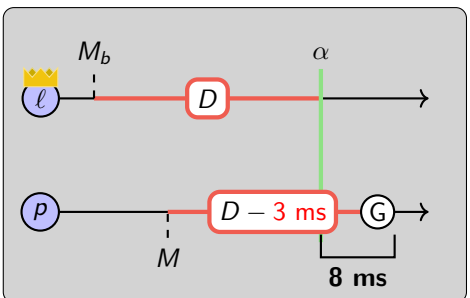


Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.

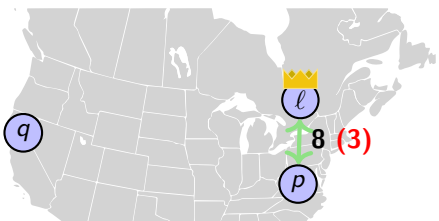


Scheduling Ⓖ on p

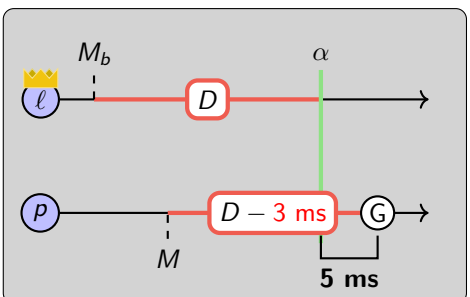


Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.

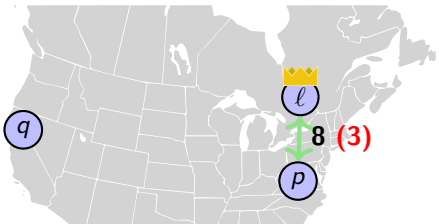


Scheduling Ⓖ on p

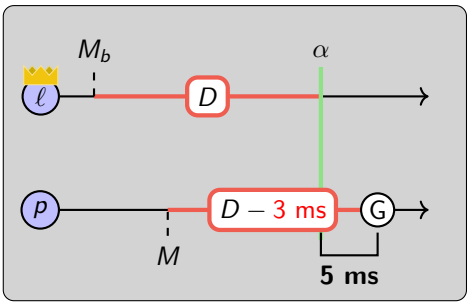


Pairwise-Leader

- **Assume:** for simplicity that all replica's clocks tick at the same rate, i.e., if 10 minutes pass in real-time, then every replica's clock advances by 10 minutes.



Scheduling Ⓖ on p



- Can tolerate at most ϵ drift by scaling $D + L$ and $D - L$ proportionally to ϵ.

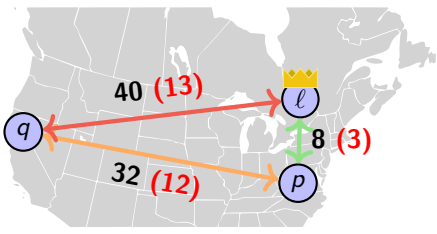
Pairwise-Leader Summary

Pairwise-Leader Summary

- **Core idea:** schedule Ⓢ and Ⓖ using a pairwise event scheduling primitive instead of globally synchronized clocks (like Delayed Stamping algorithms do).

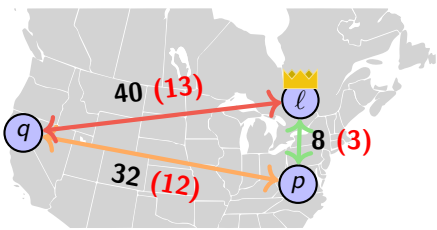
Pairwise-Leader Summary

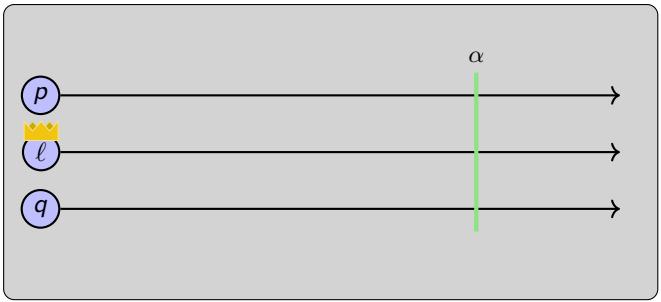
- **Core idea:** schedule Ⓢ and Ⓖ using a pairwise event scheduling primitive instead of globally synchronized clocks (like Delayed Stamping algorithms do).



Pairwise-Leader Summary

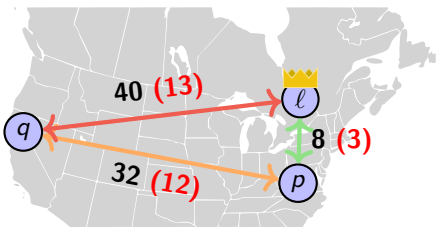
- **Core idea:** schedule Ⓢ and Ⓖ using a pairwise event scheduling primitive instead of globally synchronized clocks (like Delayed Stamping algorithms do).

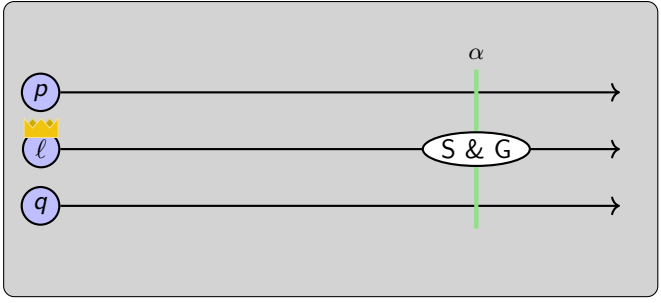




Pairwise-Leader Summary

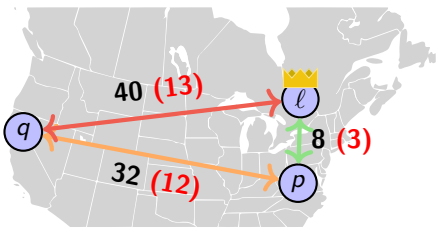
- **Core idea:** schedule Ⓢ and Ⓖ using a pairwise event scheduling primitive instead of globally synchronized clocks (like Delayed Stamping algorithms do).

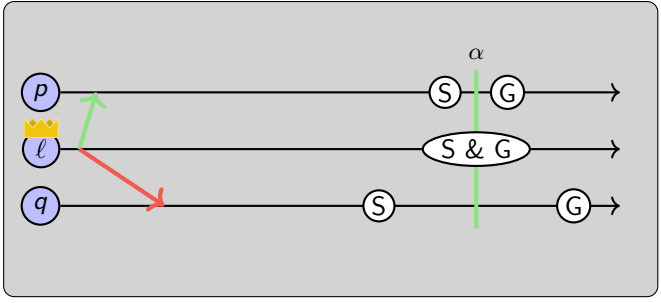




Pairwise-Leader Summary

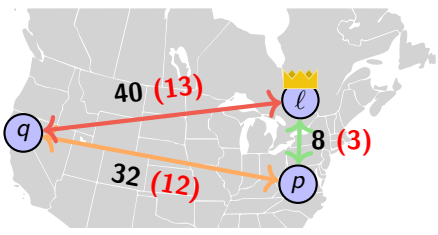
- **Core idea:** schedule Ⓢ and Ⓖ using a pairwise event scheduling primitive instead of globally synchronized clocks (like Delayed Stamping algorithms do).

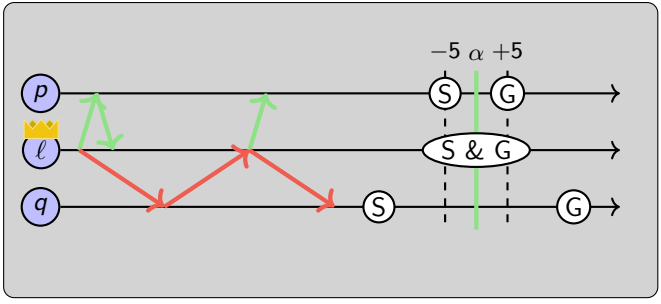




Pairwise-Leader Summary

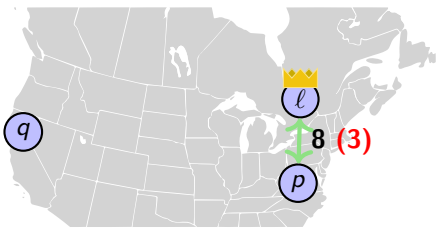
- **Core idea:** schedule Ⓢ and Ⓖ using a pairwise event scheduling primitive instead of globally synchronized clocks (like Delayed Stamping algorithms do).

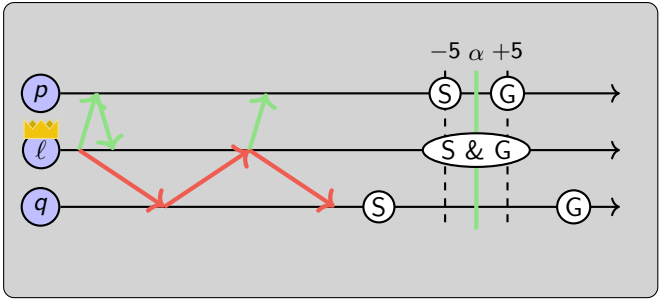




Pairwise-Leader Summary

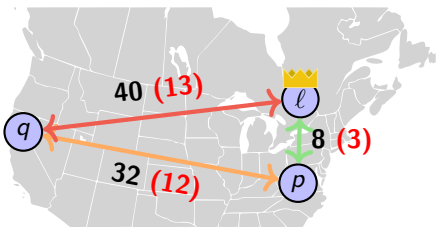
- **Core idea:** schedule Ⓢ and Ⓖ using a pairwise event scheduling primitive instead of globally synchronized clocks (like Delayed Stamping algorithms do).

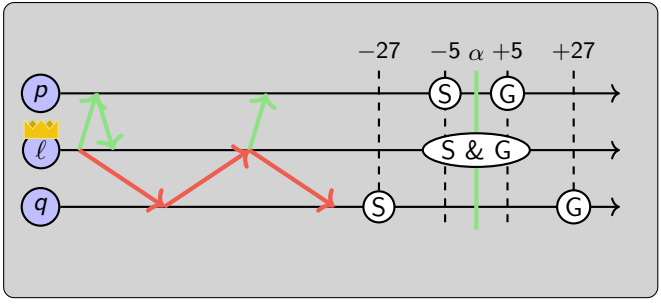




Pairwise-Leader Summary

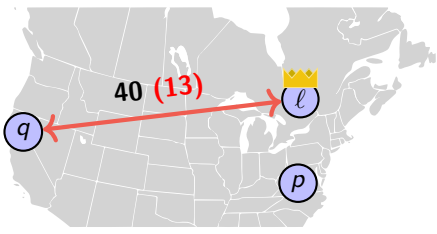
- **Core idea:** schedule Ⓢ and Ⓖ using a pairwise event scheduling primitive instead of globally synchronized clocks (like Delayed Stamping algorithms do).

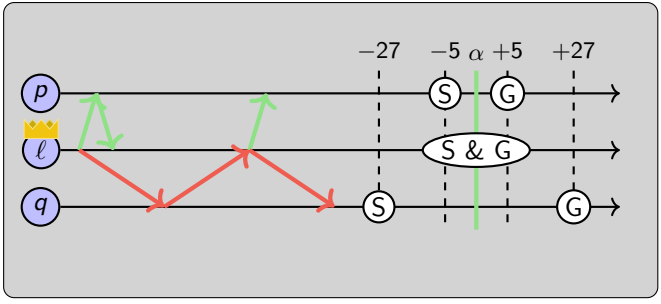




Pairwise-Leader Summary

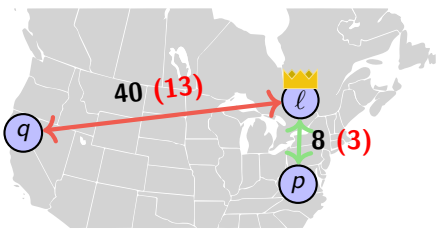
- **Core idea:** schedule Ⓢ and Ⓖ using a pairwise event scheduling primitive instead of globally synchronized clocks (like Delayed Stamping algorithms do).

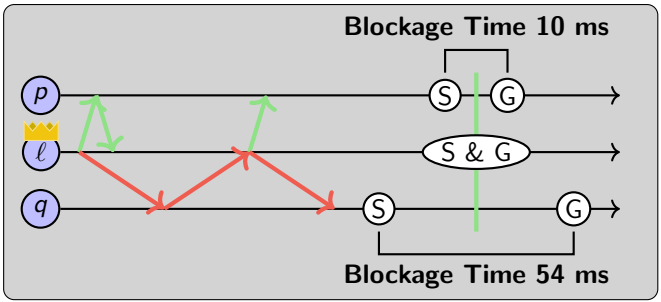




Pairwise-Leader Summary

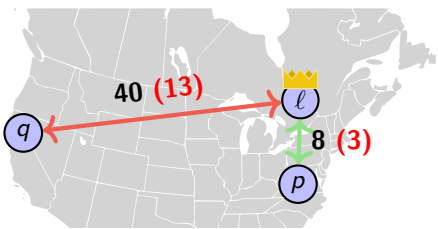
- **Core idea:** schedule Ⓢ and Ⓖ using a pairwise event scheduling primitive instead of globally synchronized clocks (like Delayed Stamping algorithms do).

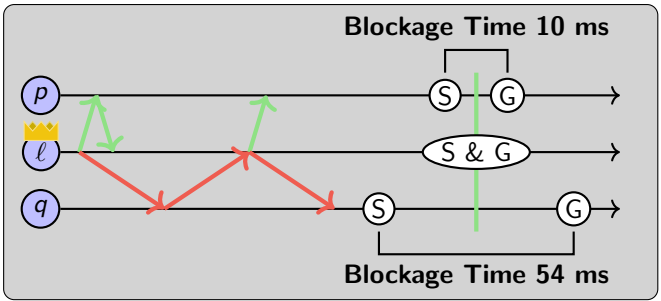




Pairwise-Leader Summary

- **Core idea:** schedule Ⓢ and Ⓖ using a pairwise event scheduling primitive instead of globally synchronized clocks (like Delayed Stamping algorithms do).

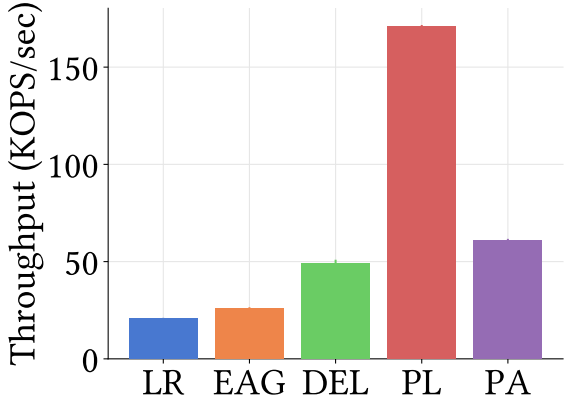




- Blockage time is the RTT minus 2 × LB between a follower and the leader (👑).

Benefits in RocksDB

- Geo-distributed RocksDB deployment spanning 5 AWS regions spread across North America and Europe.
- 5 replicas per region each running a YCSB workload.



Takeaways

Takeaways

1. Local read algorithms provide near zero read latency at all replicas in the best-case, but reads need to block for some time in the worst-case.

Takeaways

1. Local read algorithms provide near zero read latency at all replicas in the best-case, but reads need to block for some time in the worst-case.

2. Existing local read algorithms suffer from **large symmetric** blockage times.

Takeaways

1. Local read algorithms provide near zero read latency at all replicas in the best-case, but reads need to block for some time in the worst-case.

2. Existing local read algorithms suffer from **large symmetric** blockage times.

3. Our work overcomes this by presenting two new local read algorithms that achieve **asymmetric** blockage times through *pairwise event scheduling*.

Thank you!

Questions

mthiessen@cs.toronto.edu

Paper