Birds of a Feather Flock Together: Scaling RDMA RPCs with Flock

Sumit Kumar Monga, Sanidhya Kashyap, Changwoo Min

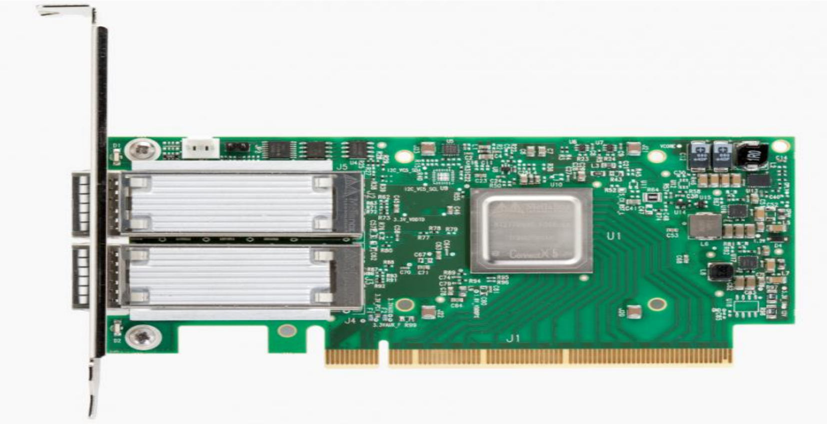
VIRGINIA TECH

EPFL

Datacenters adopting RDMA

To achieve good performance, datacenter applications require the network to deliver

- high throughput
- low latency



Within datacenters RDMA deployment ⬆

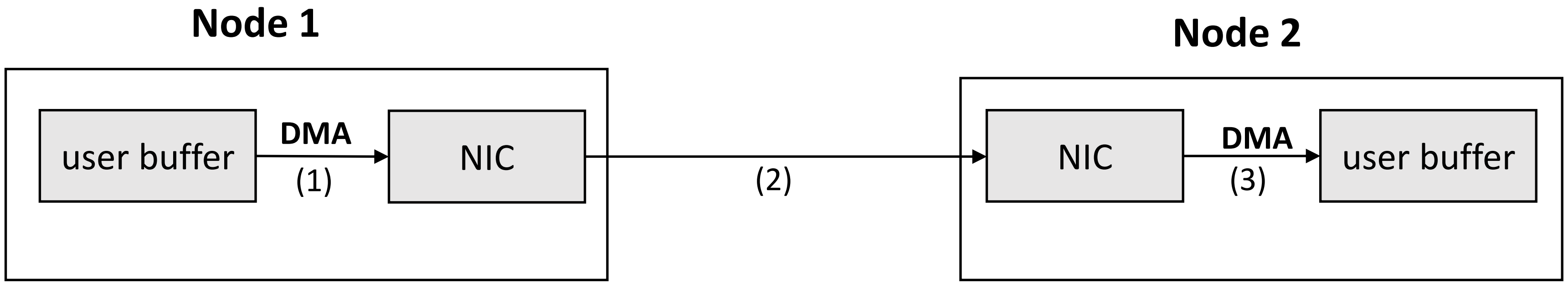
- ✓ high throughput and low latency
- ✓ drop in RDMA hardware prices

[1] https://www.datacenterknowledge.com/archives/2015/06/17/rdma-replaces-tcpip-in-linbits-data-replication-tool

[2] https://www.nextplatform.com/2018/03/27/in-modern-datacenters-the-latency-tail-wags-the-network-dog/

Remote direct memory access (RDMA)

- enables direct access to memory of a remote machine
- low latency (1 μs)
- kernel bypass + CPU bypass



RDMA background

Transport Types

- Reliable Connection (RC)
- Unreliable Connection (UC)
- Unreliable Datagram (UD)

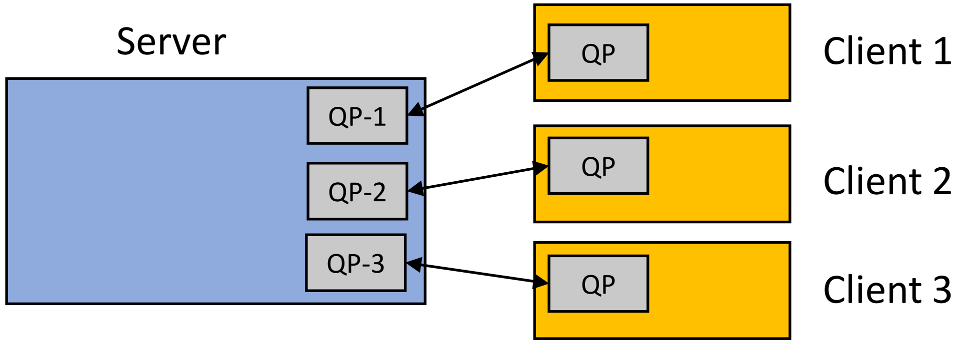
Queue Pair : hosts establish queue pairs (QP) to communicate with each other

RDMA background

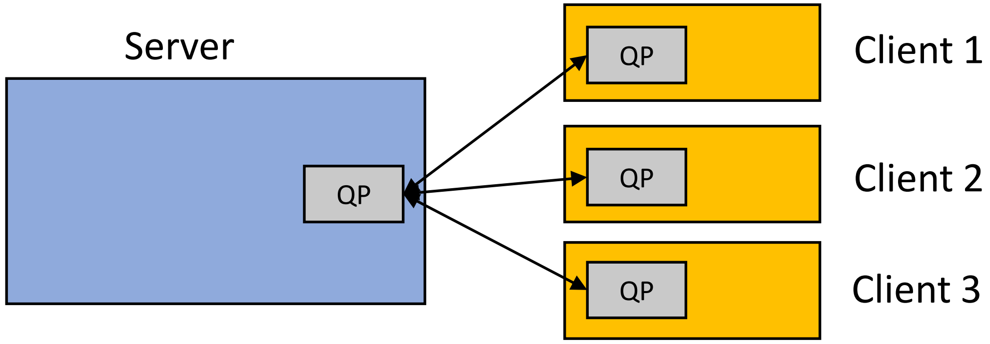
Transport Types

- Reliable Connection (RC)
- Unreliable Connection (UC)
- Unreliable Datagram (UD)

Queue Pair : hosts establish queue pairs (QP) to communicate with each other



Connected transport (RC, UC)



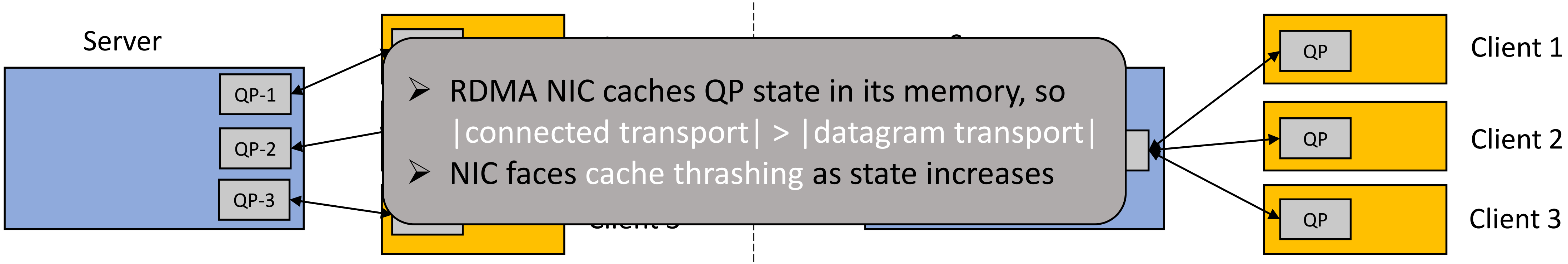
Datagram transport (UD)

RDMA background

Transport Types

- Reliable Connection (RC)
- Unreliable Connection (UC)
- Unreliable Datagram (UD)

Queue Pair : hosts establish queue pairs (QP) to communicate with each other



- RDMA NIC caches QP state in its memory, so |connected transport| > |datagram transport|
- NIC faces cache thrashing as state increases

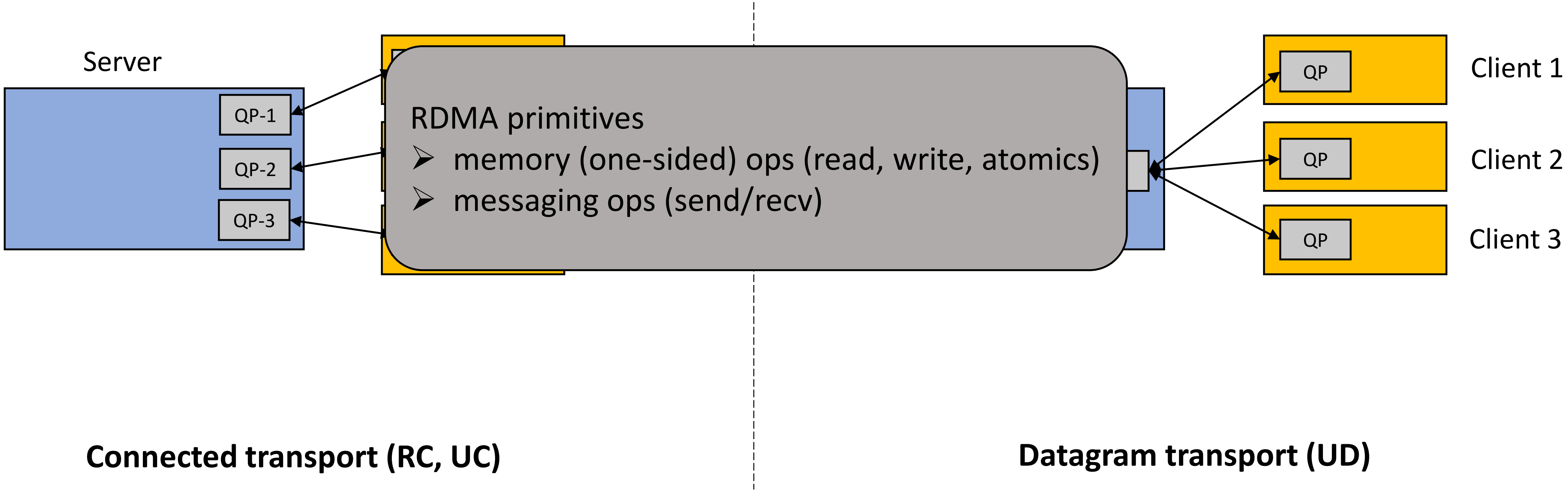
Connected transport (RC, UC) **Datagram transport (UD)**

RDMA background

Transport Types

- Reliable Connection (RC)
- Unreliable Connection (UC)
- Unreliable Datagram (UD)

Queue Pair : hosts establish queue pairs (QP) to communicate with each other



Connected transport (RC, UC)

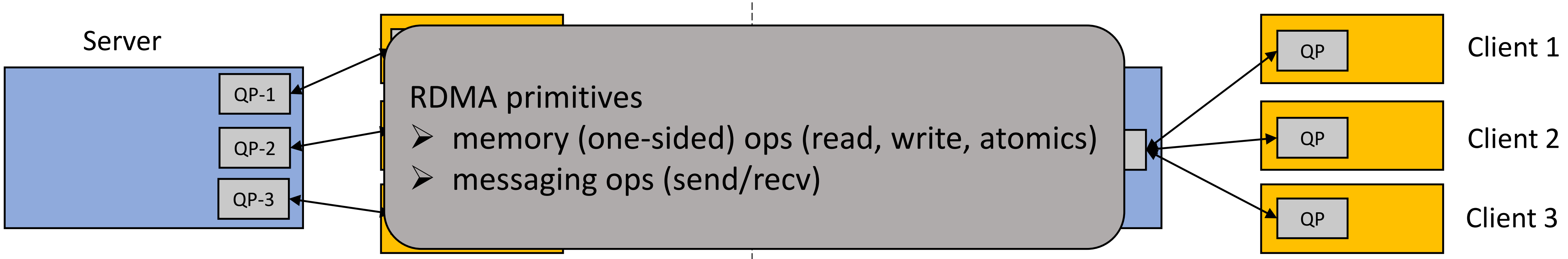
Datagram transport (UD)

RDMA background

Transport Types

- Reliable Connection (RC)
- Unreliable Connection (UC)
- Unreliable Datagram (UD)

Queue Pair : hosts establish queue pairs (QP) to communicate with each other



RC supports all RDMA primitives

UD supports only send/recv

Connected transport (RC, UC)

Datagram transport (UD)

Large cluster RDMA scalability challenges

- Non-scalable RC
  - + Provides one-sided ops
  - - Limited on-chip memory on the RDMA NIC
- Limited UD functionality:
  - + Enables one-to-many communication
  - - Lacks CPU-efficient one-sided ops

Which RDMA transport to use for scalable communication ?

Large cluster RDMA scalability challenges

- Non-scalable RC
  - + Provides one-sided ops
  - - Limited on-chip memory on the RDMA NIC
- Limited UD functionality:
  - + Enables one-to-many communication
  - - Lacks CPU-efficient one-sided ops

Only RC	Only UD	Hybrid
FaRM [NSDI 14, SOSP 15] Storm [SYSTOR 19] ScaleRPC [EuroSys 19]	FaSST [OSDI 16] eRPC [NSDI 19]	HERD [SIGCOMM 14] (UC + UD) DrTM+H [OSDI 18] (RC + UD)

Scalability comparison of RC vs UD

Benchmark setup: 1 server with increasing clients

Each client issues

- a 16B one-sided read from server memory (RC)
- a 16B RPC with server (UD)

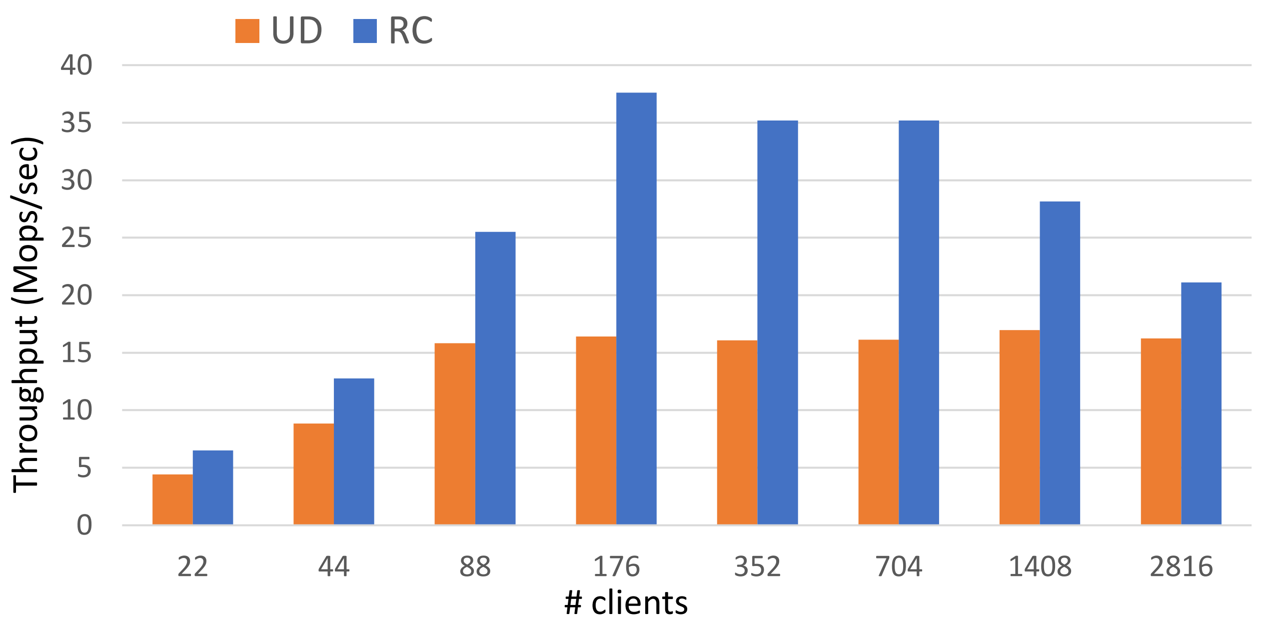
* each machine has a Mellanox ConnectX-5 100 Gbps NIC

Scalability comparison of RC vs UD

Benchmark setup: 1 server with increasing clients

Each client issues

- a 16B one-sided read from server memory (RC)
- a 16B RPC with server (UD)



- NIC's cache thrashing after 700 clients
- UD:
  - Lower throughput and saturates earlier than RC
  - Server CPU cycles are used

* each machine has a Mellanox ConnectX-5 100 Gbps NIC

Connection scalability with RDMA full flexibility?

Goals:

- Maintain **connection scalability**
- Expose all RDMA features, i.e., **versatility**
- Minimal **software-induced overheads**

Connection scalability with RDMA full flexibility?

Goals:

- Maintain **connection scalability**
- Expose all RDMA features, i.e., **versatility**
- Minimal **software-induced overheads**

FLOCK: An RDMA communication library

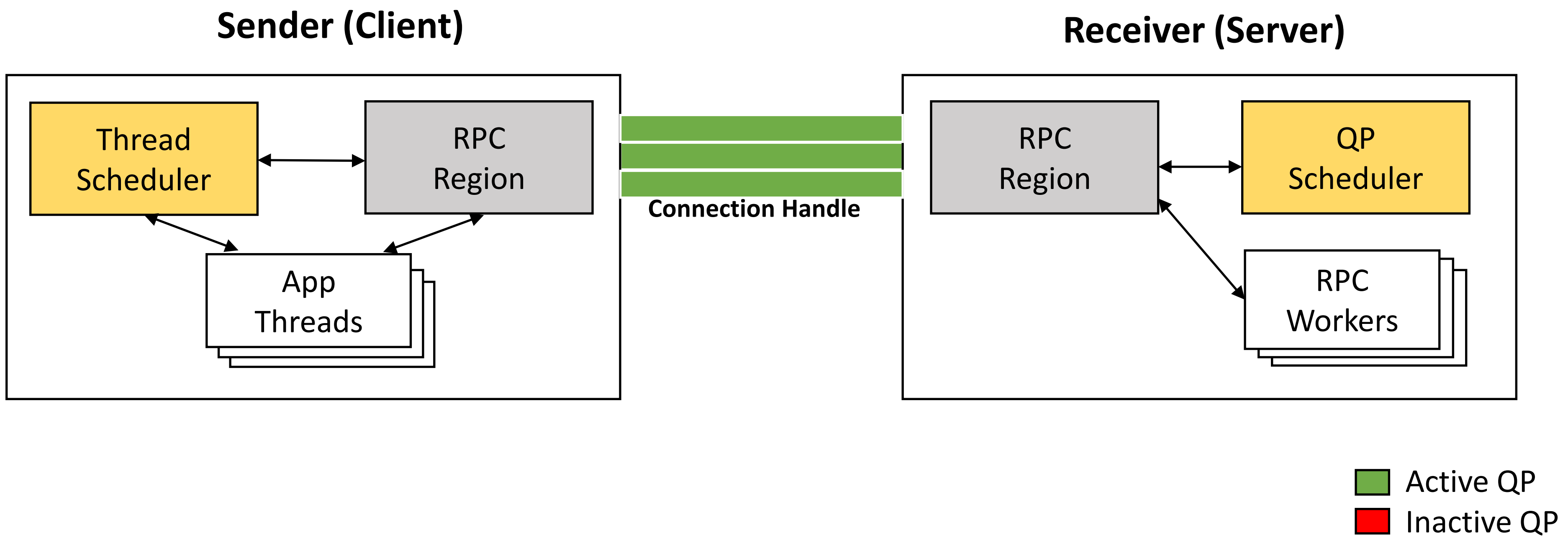
FLOCK

- Uses RC
  - + Exposes all RDMA capabilities
- Uses QP sharing among threads[1,2]
  - + Uses FLOCK synchronization for connection scalability
- Introduces **symbiotic send-recv scheduling**
  - + A cooperative scheduling policy between sender and receiver
  - + Enables efficient network resource allocation and utilization at the end-hosts

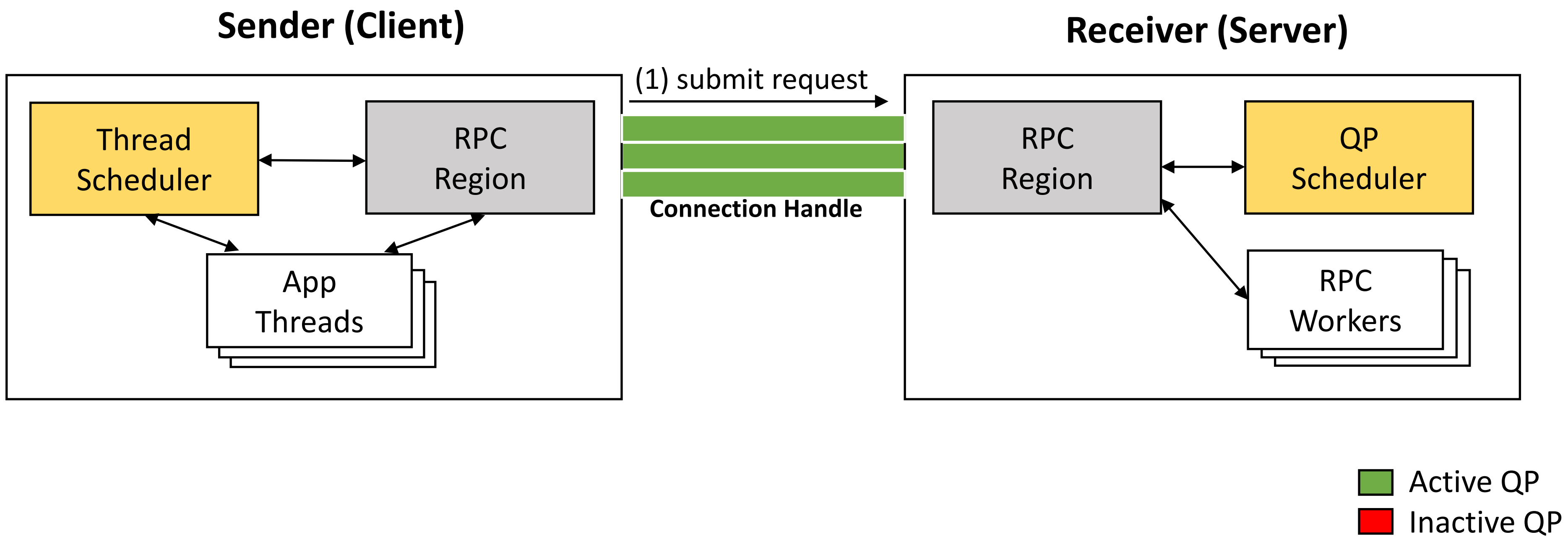
[1] FaRM : Fast remote memory, NSDI 2014

[2] No compromises : distributed transactions with consistency, availability, and performance, SOSP 2015

FLOCK Architecture

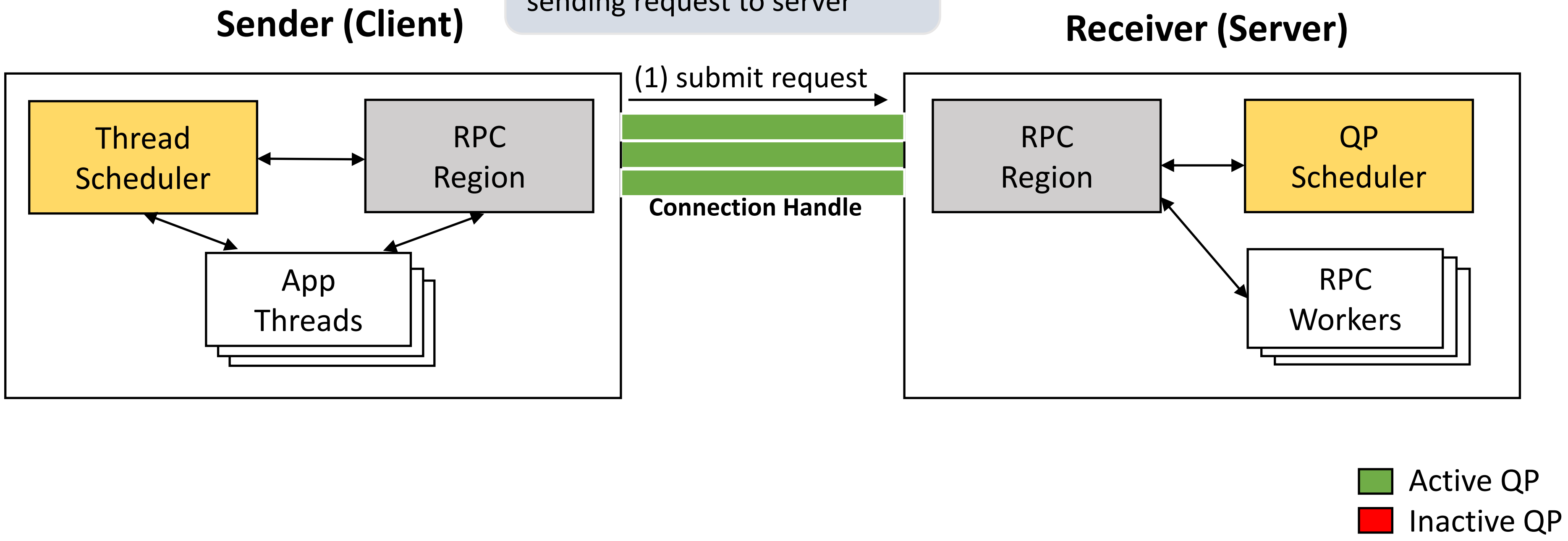


FLOCK Architecture

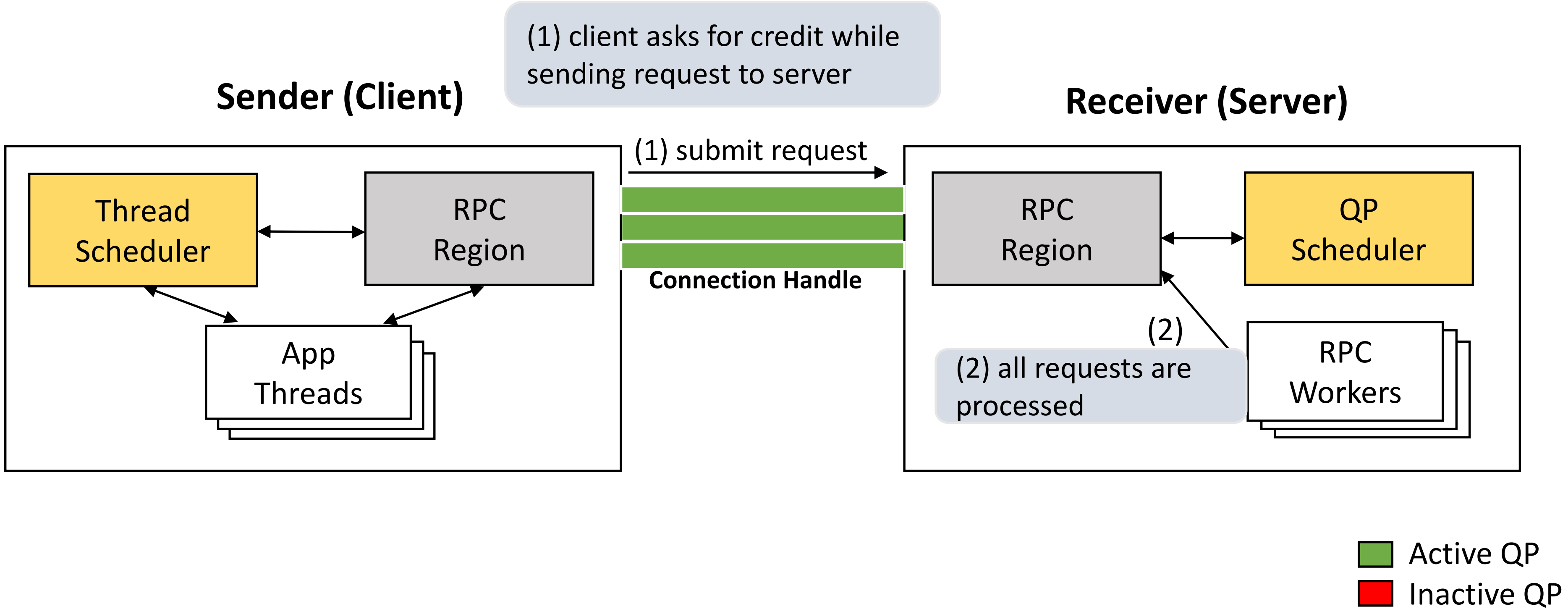


FLOCK Architecture

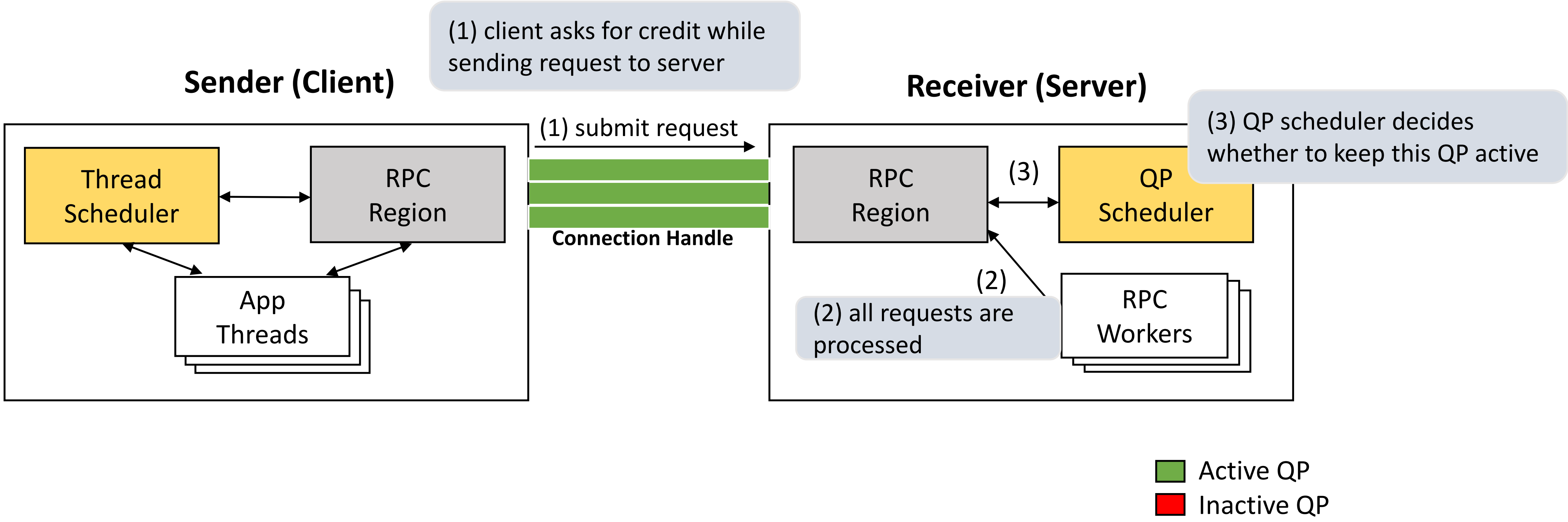
(1) client asks for credit while sending request to server



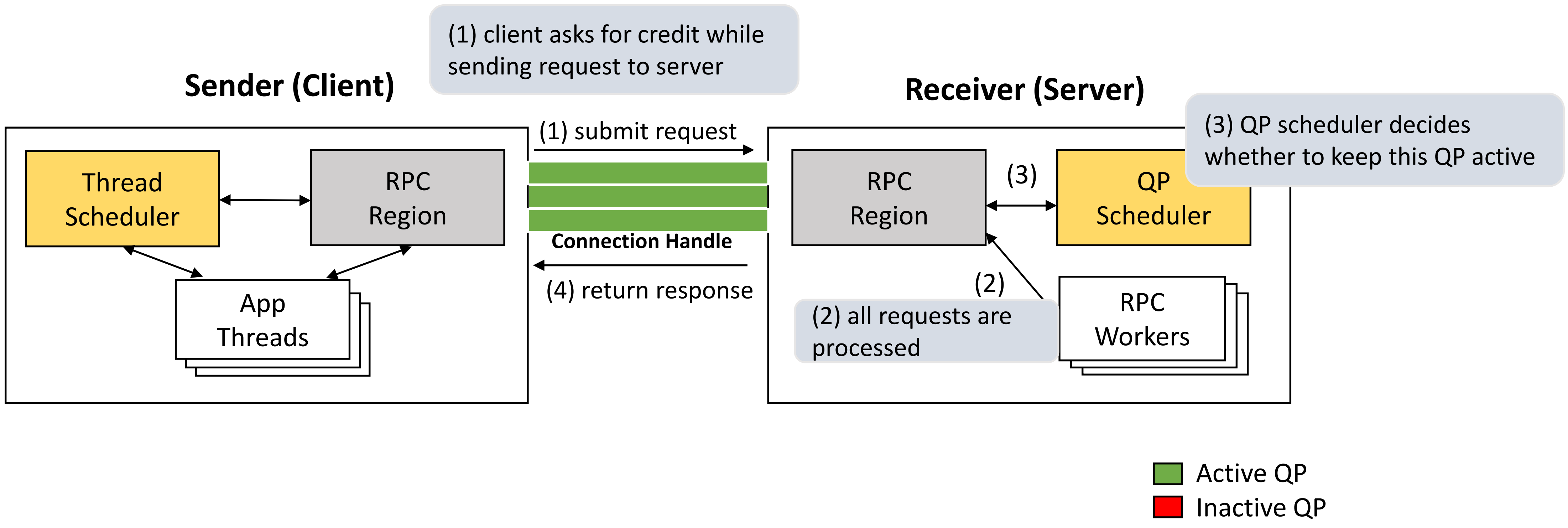
FLOCK Architecture



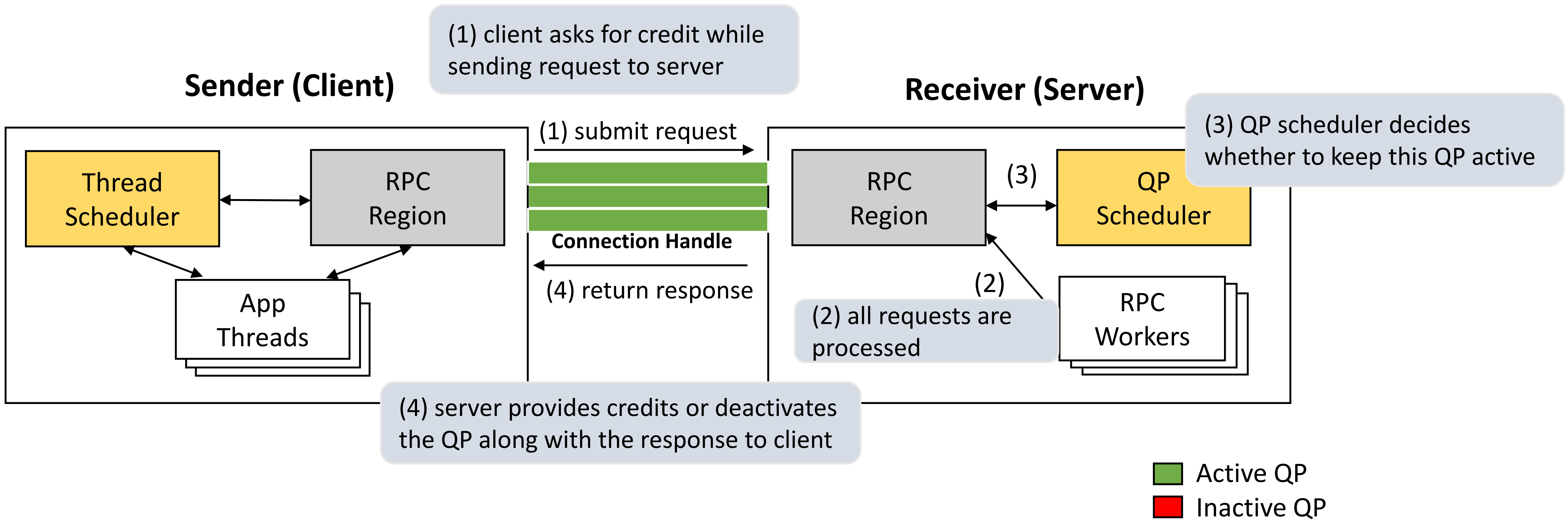
FLOCK Architecture



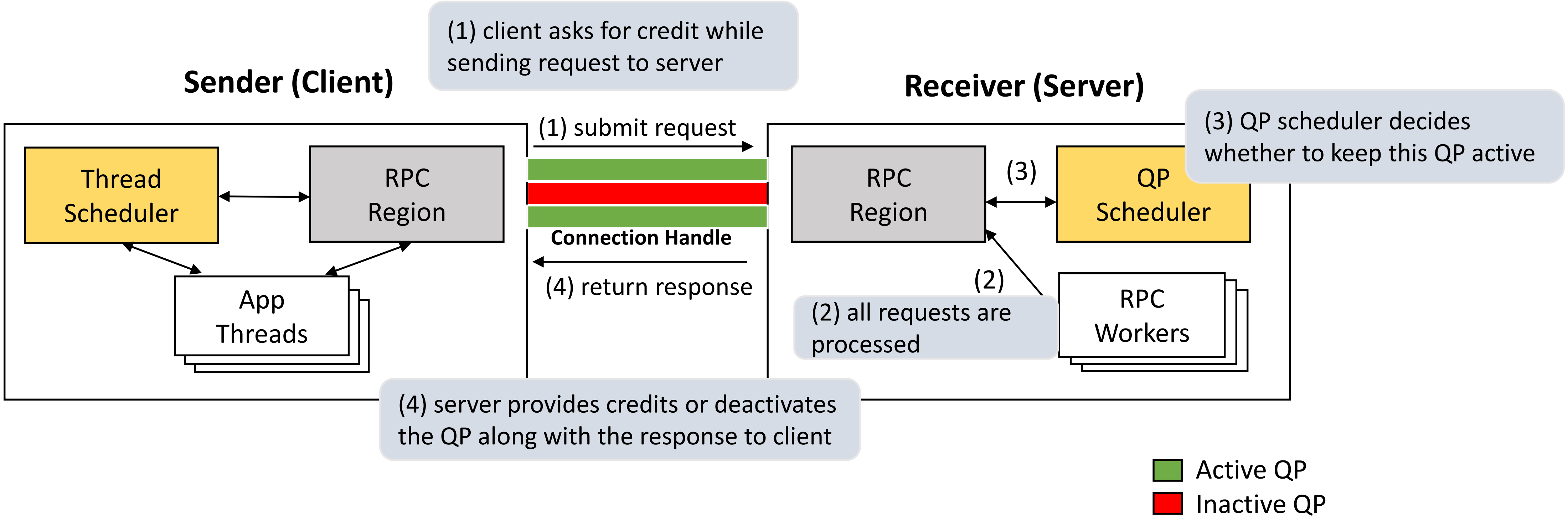
FLOCK Architecture



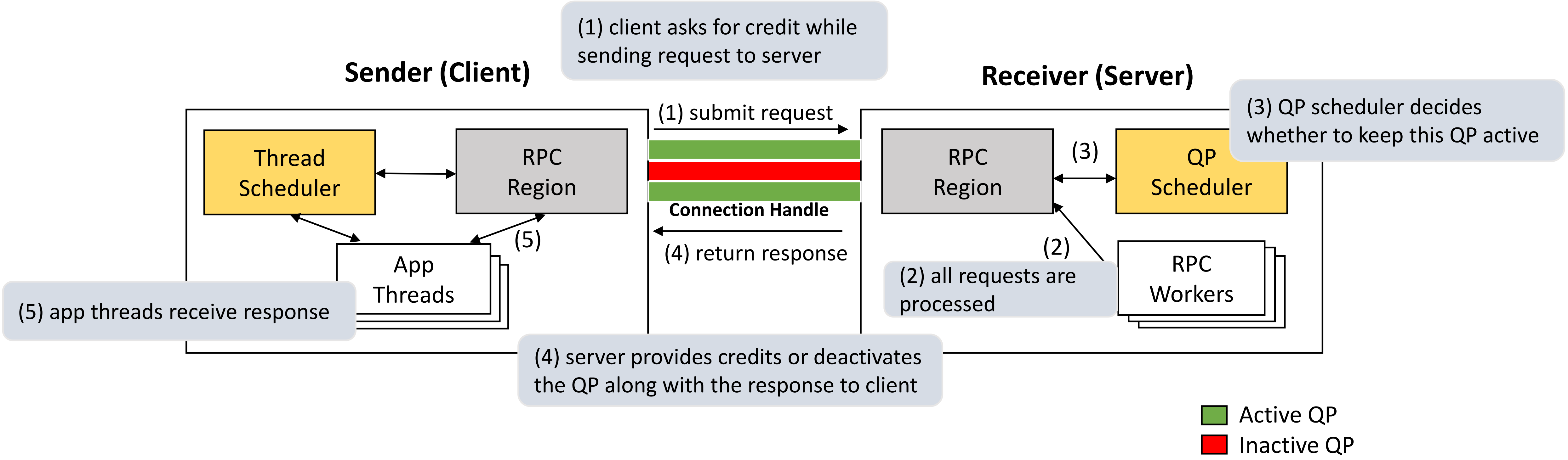
FLOCK Architecture



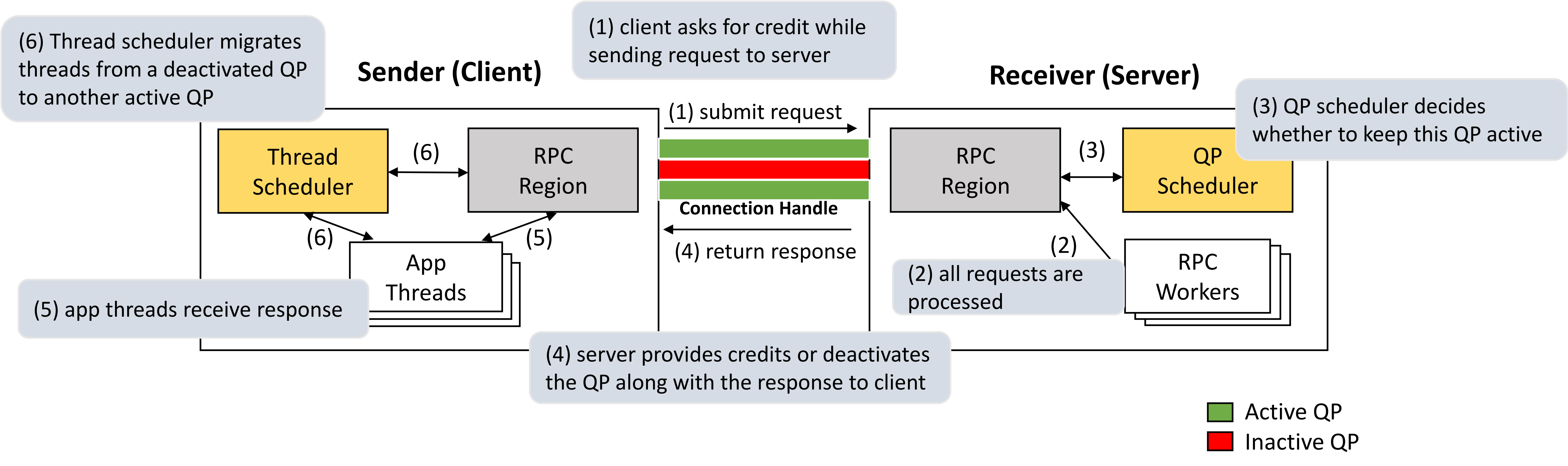
FLOCK Architecture



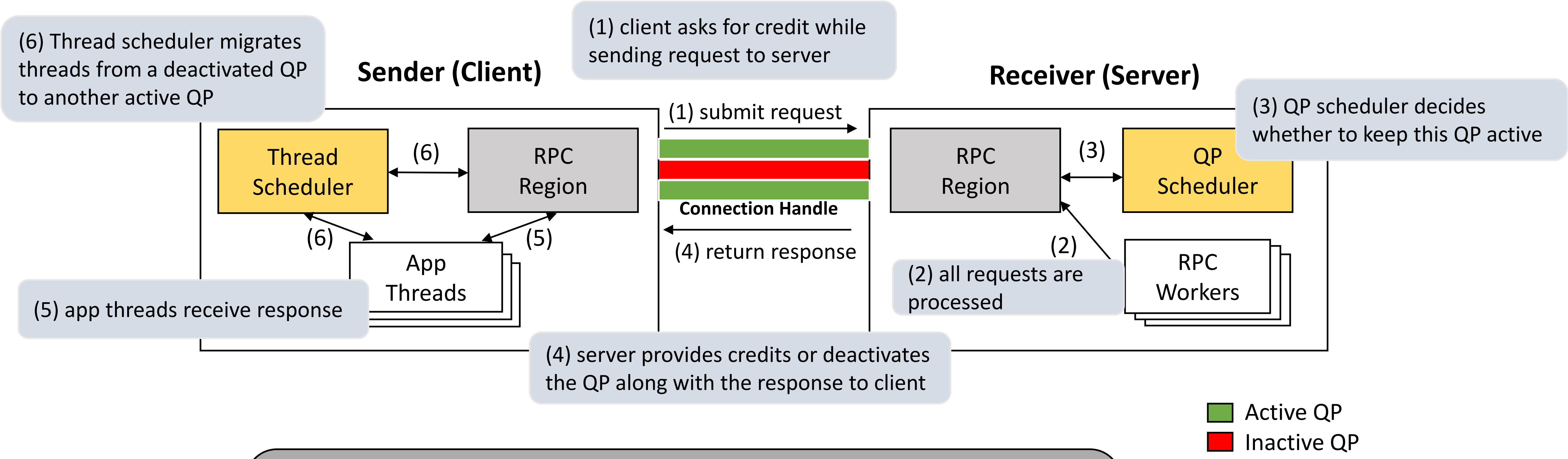
FLOCK Architecture



FLOCK Architecture



FLOCK Architecture



- Receiver-side QP Scheduler dynamically activates/deactivates QPs on a per-sender basis to avoid NIC cache pressure
- Sender-side Thread scheduler multiplexes active QPs among application threads

But isn’t QP sharing bad for performance

QP sharing among threads is detrimental to performance[1,2]

- Low parallelism
- High synchronization overheads

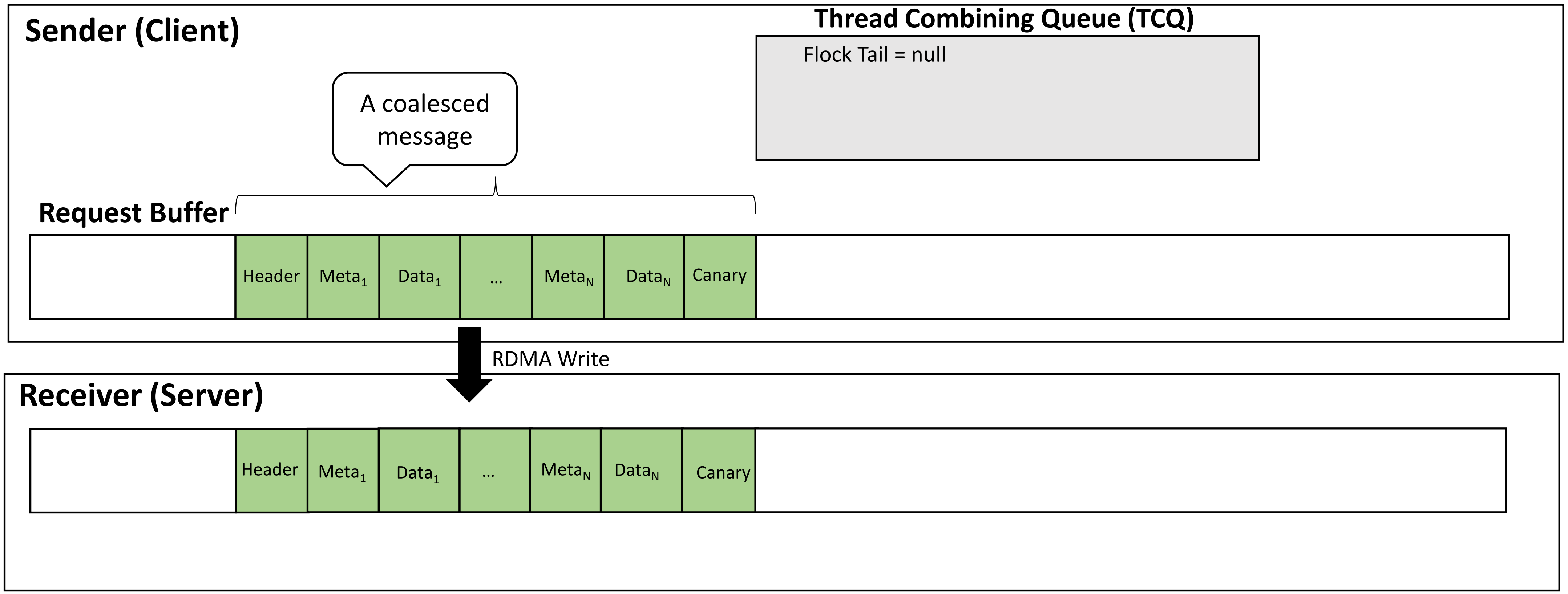
FLOCK synchronization overcomes these challenges

- Threads sharing a QP progress concurrently with minimal synchronization
- Coalesces smaller messages utilizing network bandwidth + CPU efficiently

[1] FaRM : Fast remote memory, NSDI 2014

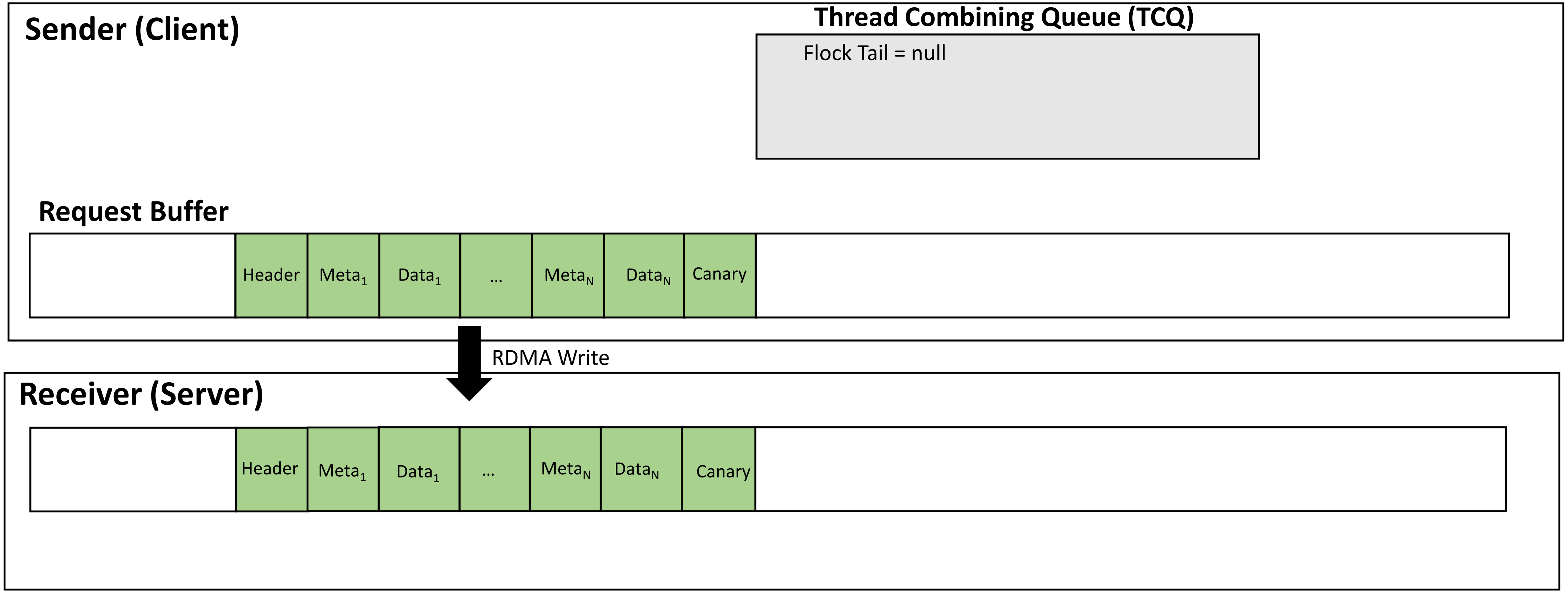
[2] FaSST : Fast, Scalable and Simple Distributed Transactions with Two-Sided RDMA Datagram RPCs, OSDI 2016

FLOCK Synchronization



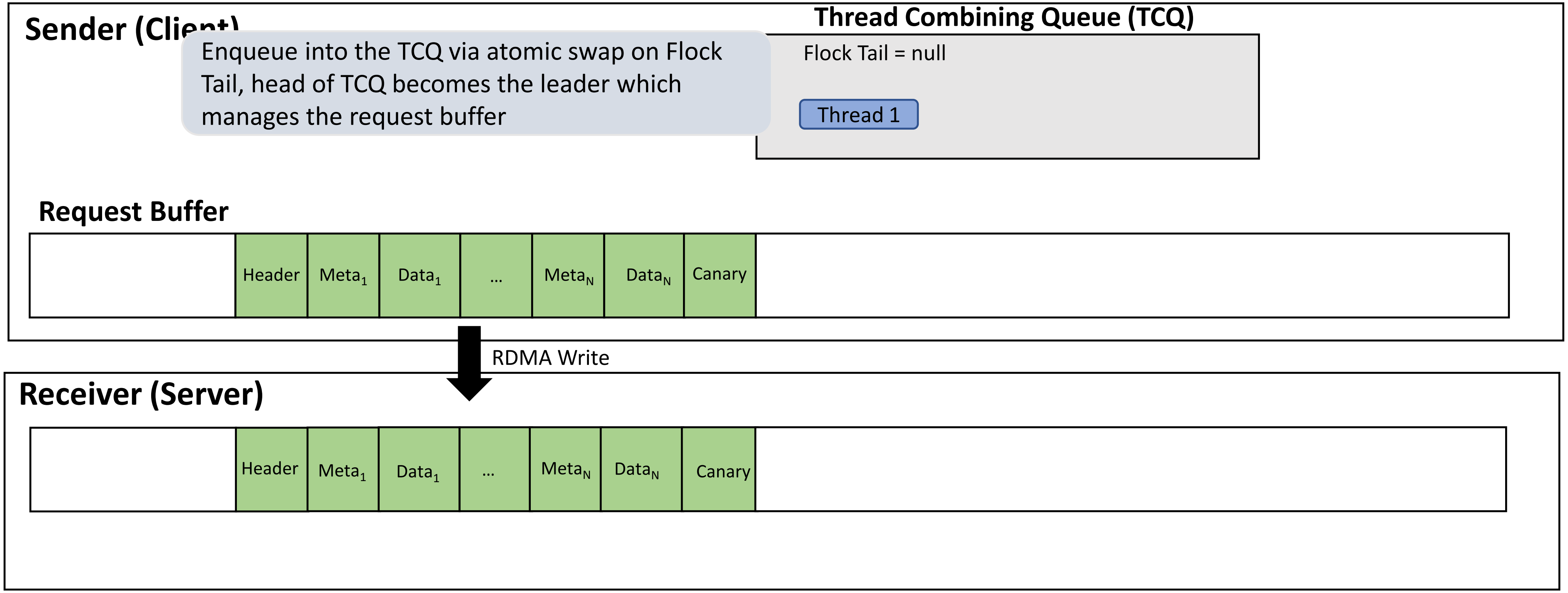
FLOCK Synchronization

- QP sharing using leader-follower coordination
- leader coalesces requests from followers



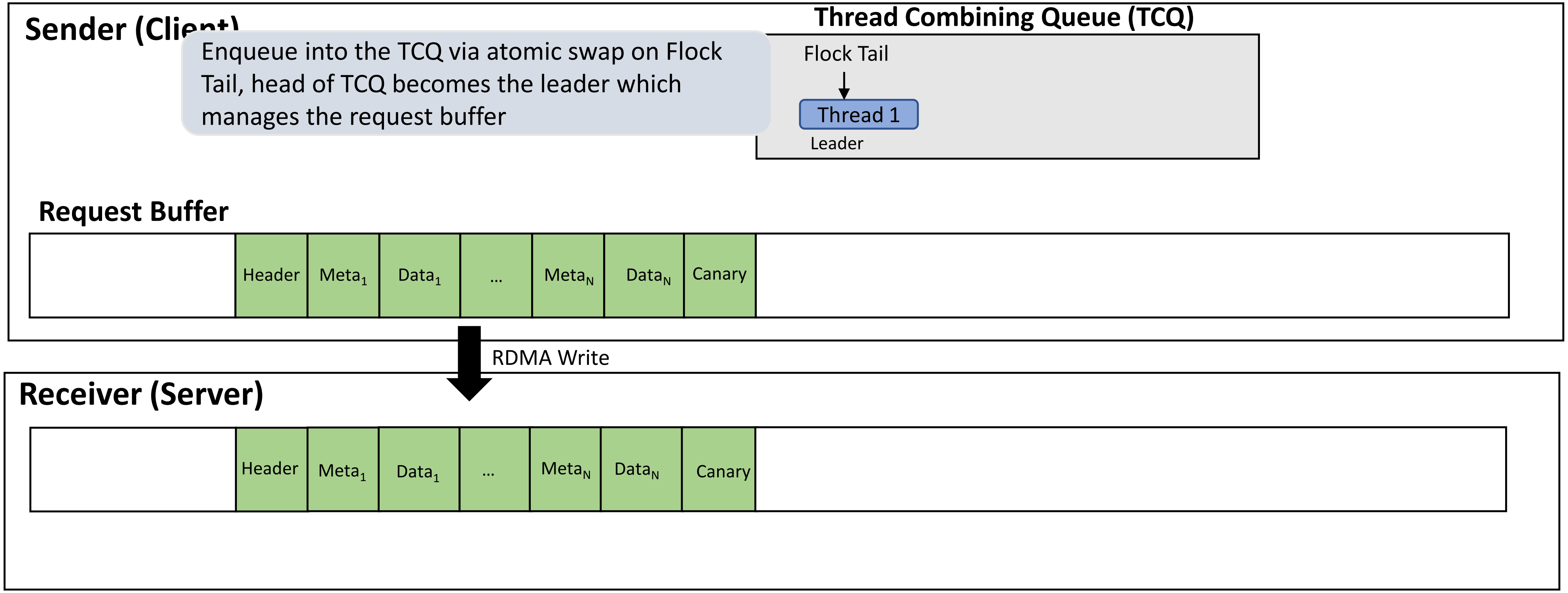
FLOCK Synchronization

- QP sharing using leader-follower coordination
- leader coalesces requests from followers



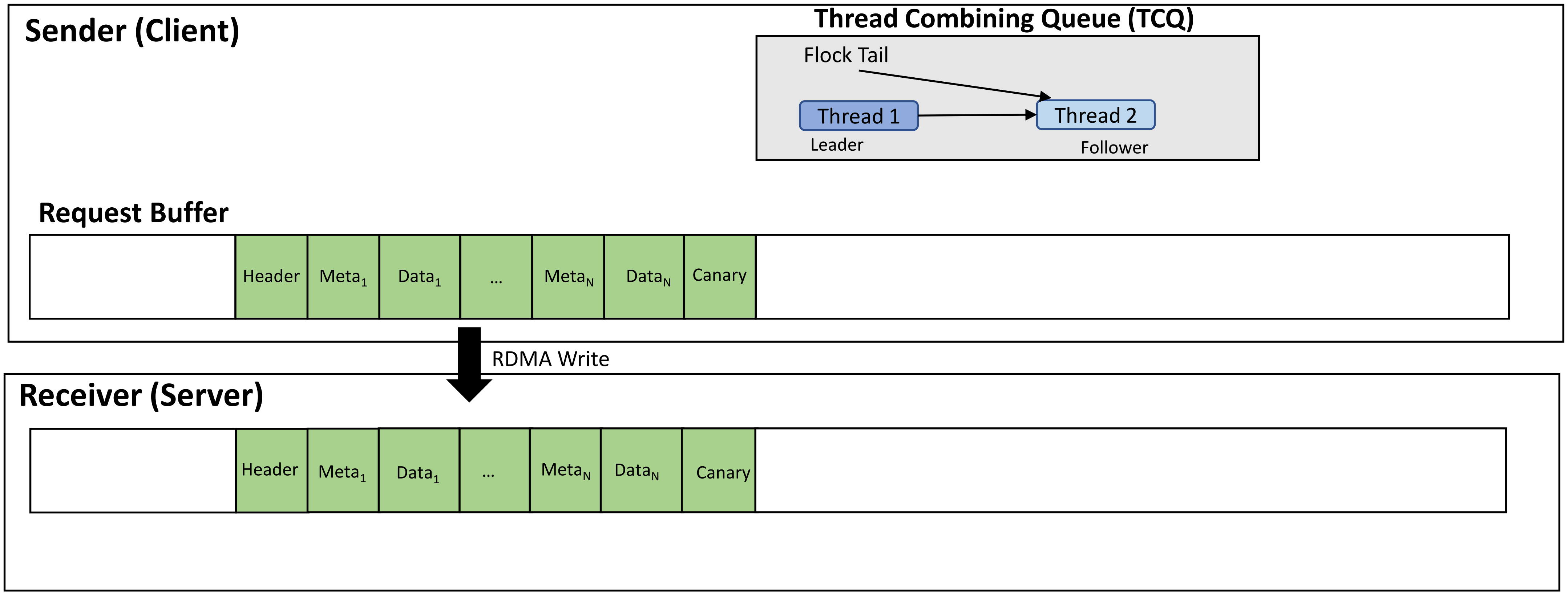
FLOCK Synchronization

- QP sharing using leader-follower coordination
- leader coalesces requests from followers



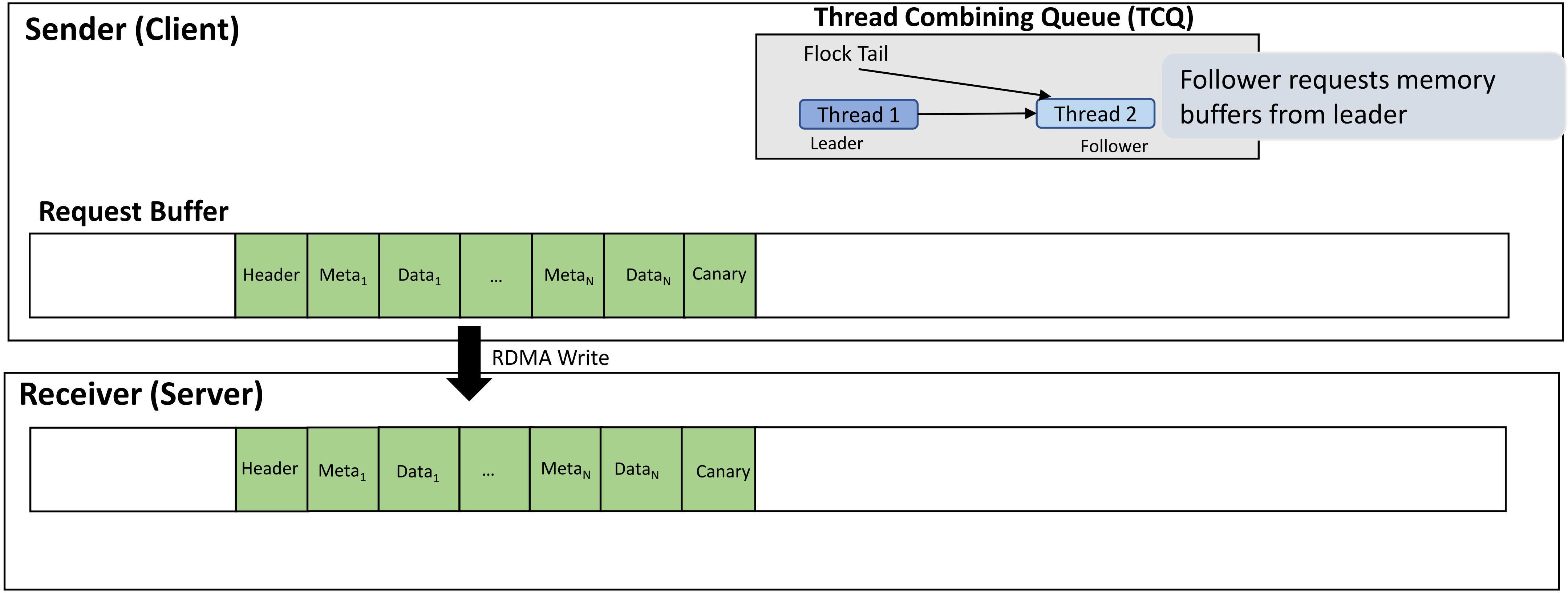
FLOCK Synchronization

- QP sharing using leader-follower coordination
- leader coalesces requests from followers



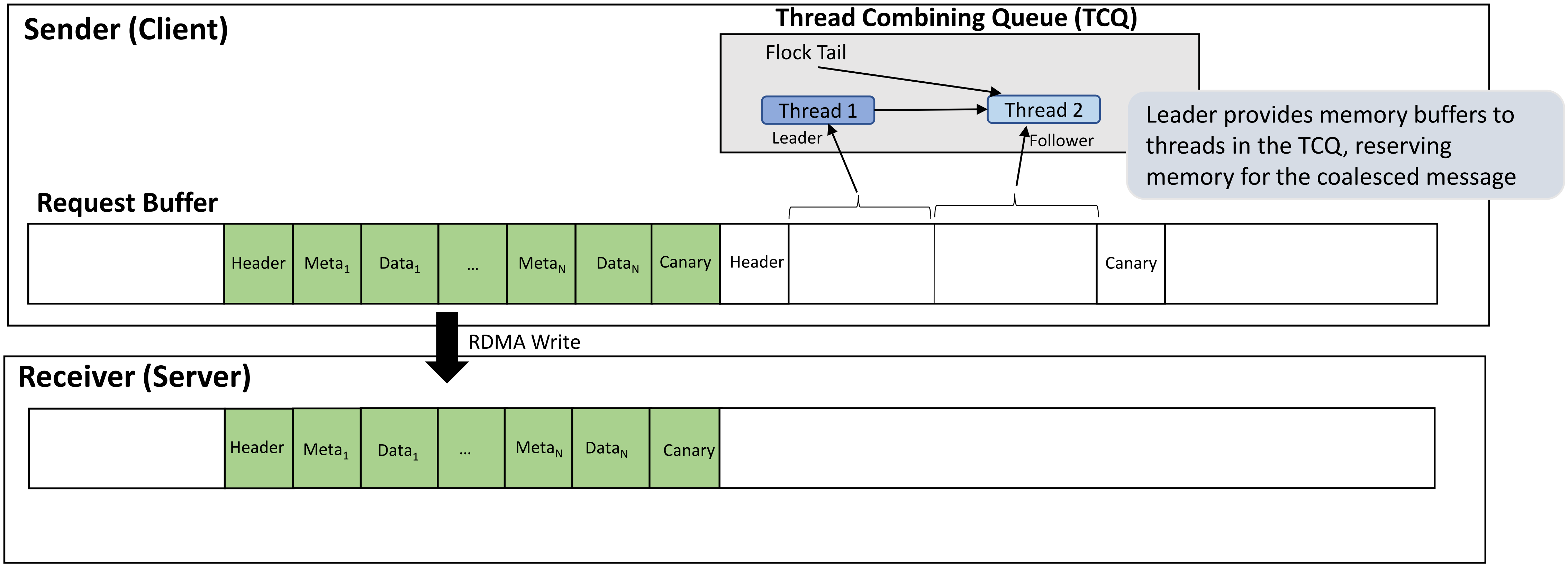
FLOCK Synchronization

- QP sharing using leader-follower coordination
- leader coalesces requests from followers



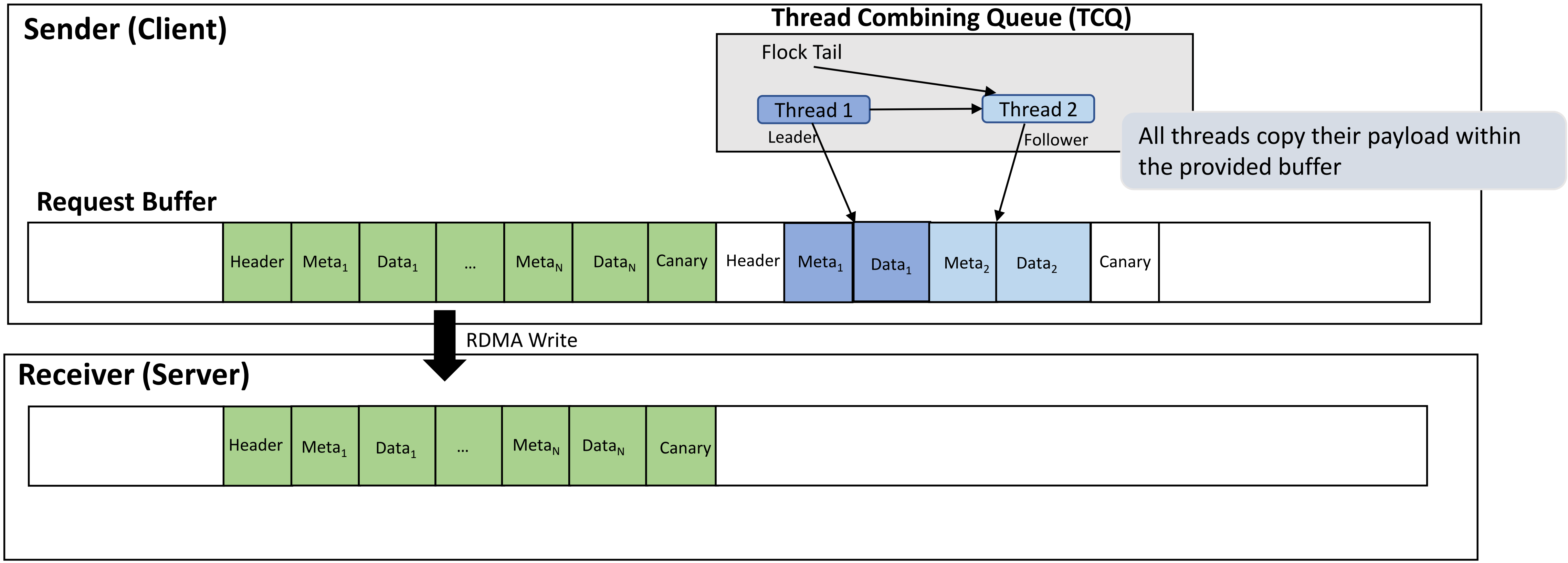
FLOCK Synchronization

- QP sharing using leader-follower coordination
- leader coalesces requests from followers



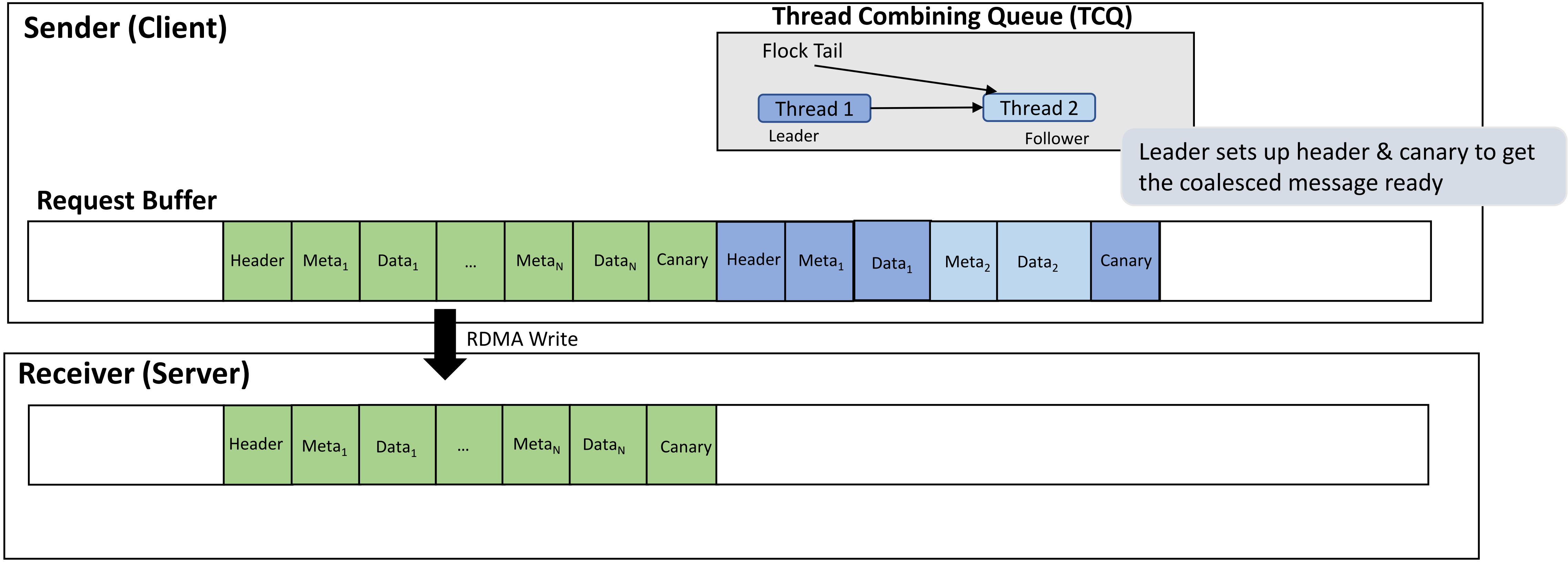
FLOCK Synchronization

- QP sharing using leader-follower coordination
- leader coalesces requests from followers



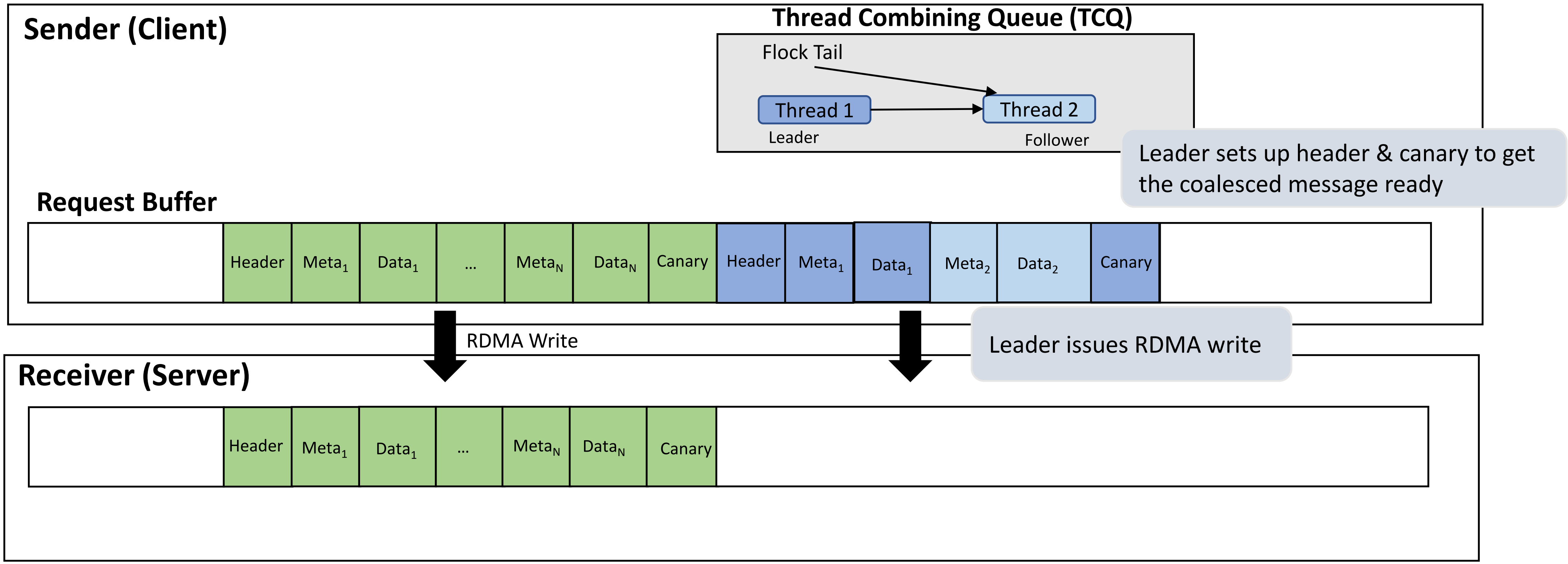
FLOCK Synchronization

- QP sharing using leader-follower coordination
- leader coalesces requests from followers



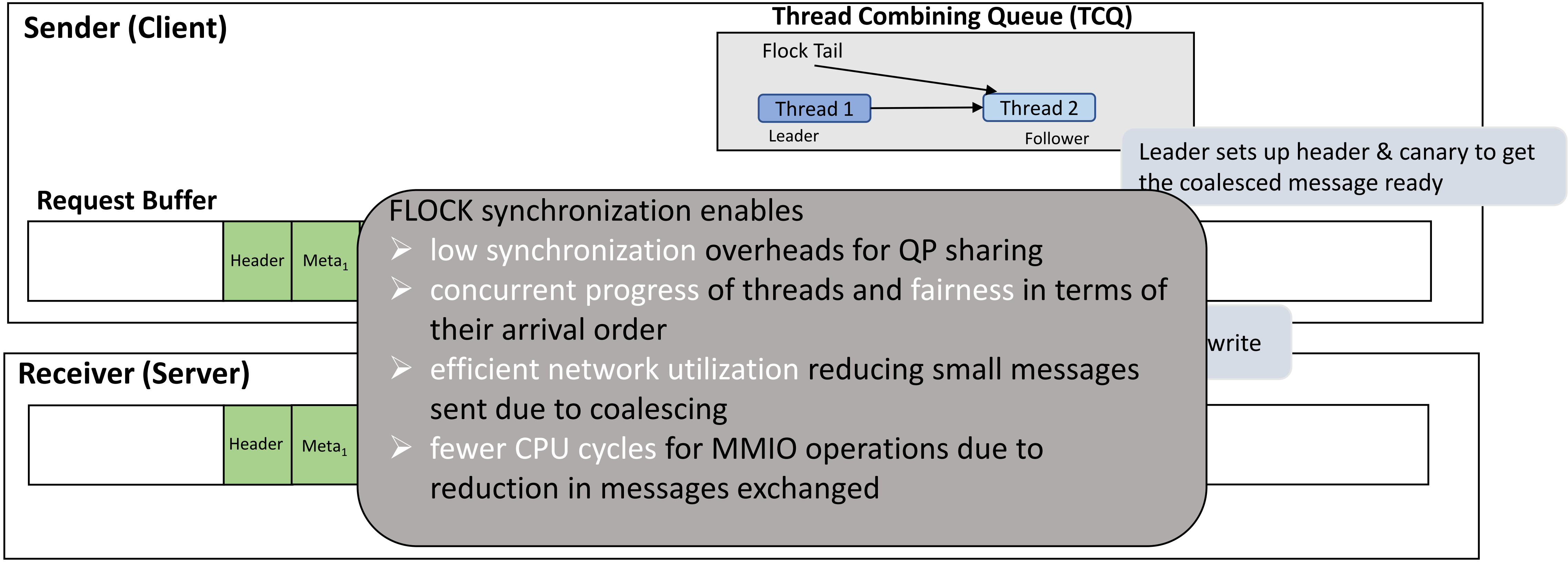
FLOCK Synchronization

- QP sharing using leader-follower coordination
- leader coalesces requests from followers



FLOCK Synchronization

- QP sharing using leader-follower coordination
- leader coalesces requests from followers



FLOCK synchronization enables

- low synchronization overheads for QP sharing
- concurrent progress of threads and fairness in terms of their arrival order
- efficient network utilization reducing small messages sent due to coalescing
- fewer CPU cycles for MMIO operations due to reduction in messages exchanged

Performance-Scalability Tradeoff

RDMA networks face a tradeoff between performance and scalability

Configuration	Performance	Scalability
Threads using dedicated QPs	✓ More parallelism within RDMA NIC	✘ Limited NIC cache
Threads sharing QP	✘ Hampers performance due to synchronization overheads	✓ Fewer NIC cache misses

FLOCK aims to resolve this tradeoff using **symbiotic send-recv scheduling**

Receiver-side QP Scheduling

- Limit active QP count to bound NIC state and prevent CPU overload
- Allocate fewer QPs to dormant clients and more to active clients

Clients categorized as active or dormant based on their utilization metrics

- **Credit renewal**: credits for future requests
- **Coalescing degree**: indicates the number of requests coalesced within a message. Higher values imply *QP contention*

Receiver-side QP Scheduling

- Limit active QP count to bound NIC state and prevent CPU overload
- Allocate fewer QPs to dormant clients and more to active clients

Clients categorized as active or dormant based on their utilization metrics

- **Credit renewal**: credits for future requests
- **Coalescing degree**: indicates the number of requests coalesced within a message. Higher values imply *QP contention*

Clients receive active QPs in proportion to their utilization

Evaluation Questions

- FLOCK vs state-of-the-art RDMA RPC systems
- Scalability with symbiotic scheduling
- Impact on a real-world application

Evaluation Environment

- 24 machines from CloudLab d6515 cluster
  - 32-core AMD 7452 2.5 GHz CPU
  - Mellanox ConnectX-5 100 Gbps NIC
- 100 Gbps switch connecting the machines
- Maximum active QP count at the server is 256

FLOCK vs eRPC

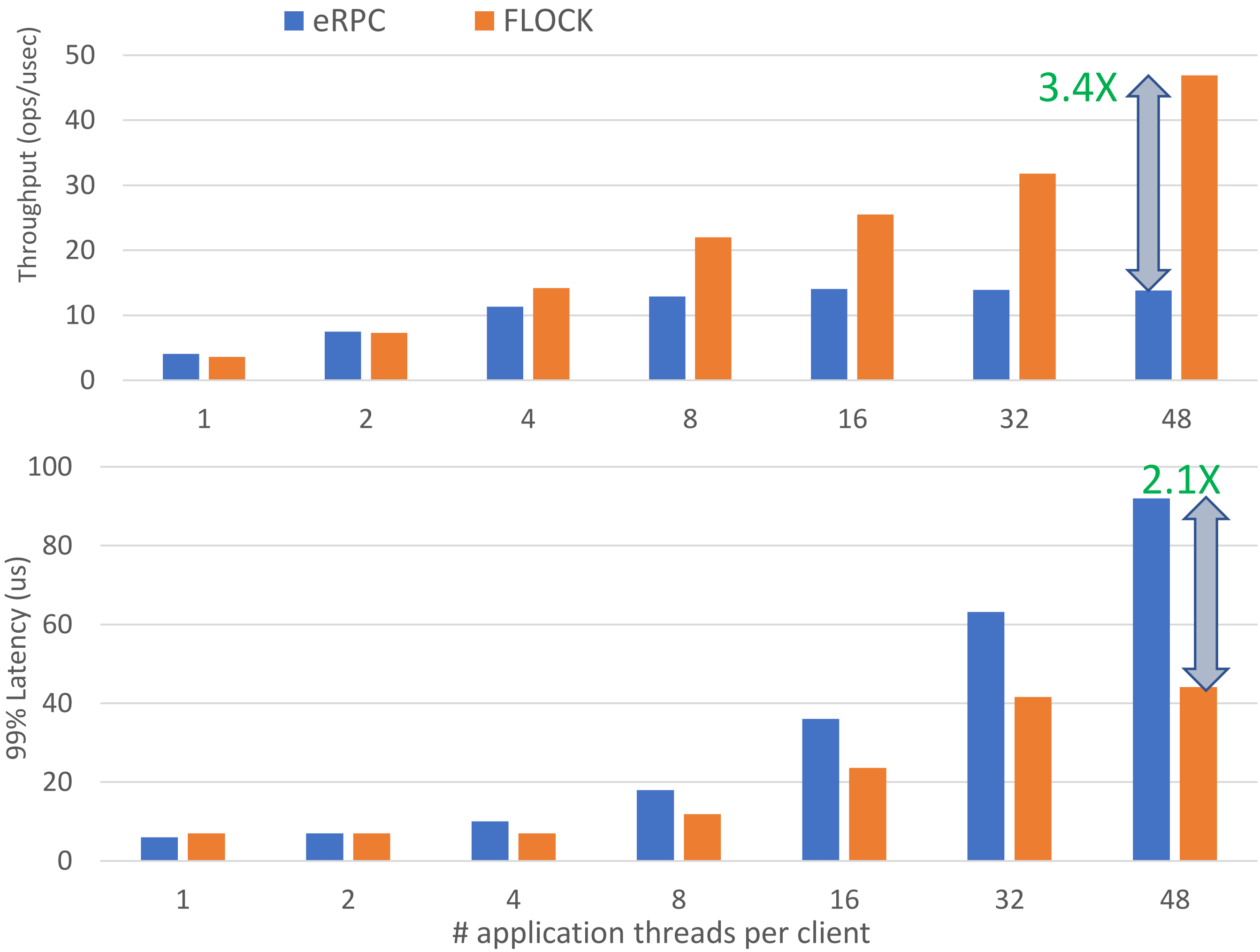
Configuration : 1 server, 23 clients

Workload : 64B request and 64B response

FLOCK vs eRPC

Configuration : 1 server, 23 clients

Workload : 64B request and 64B response

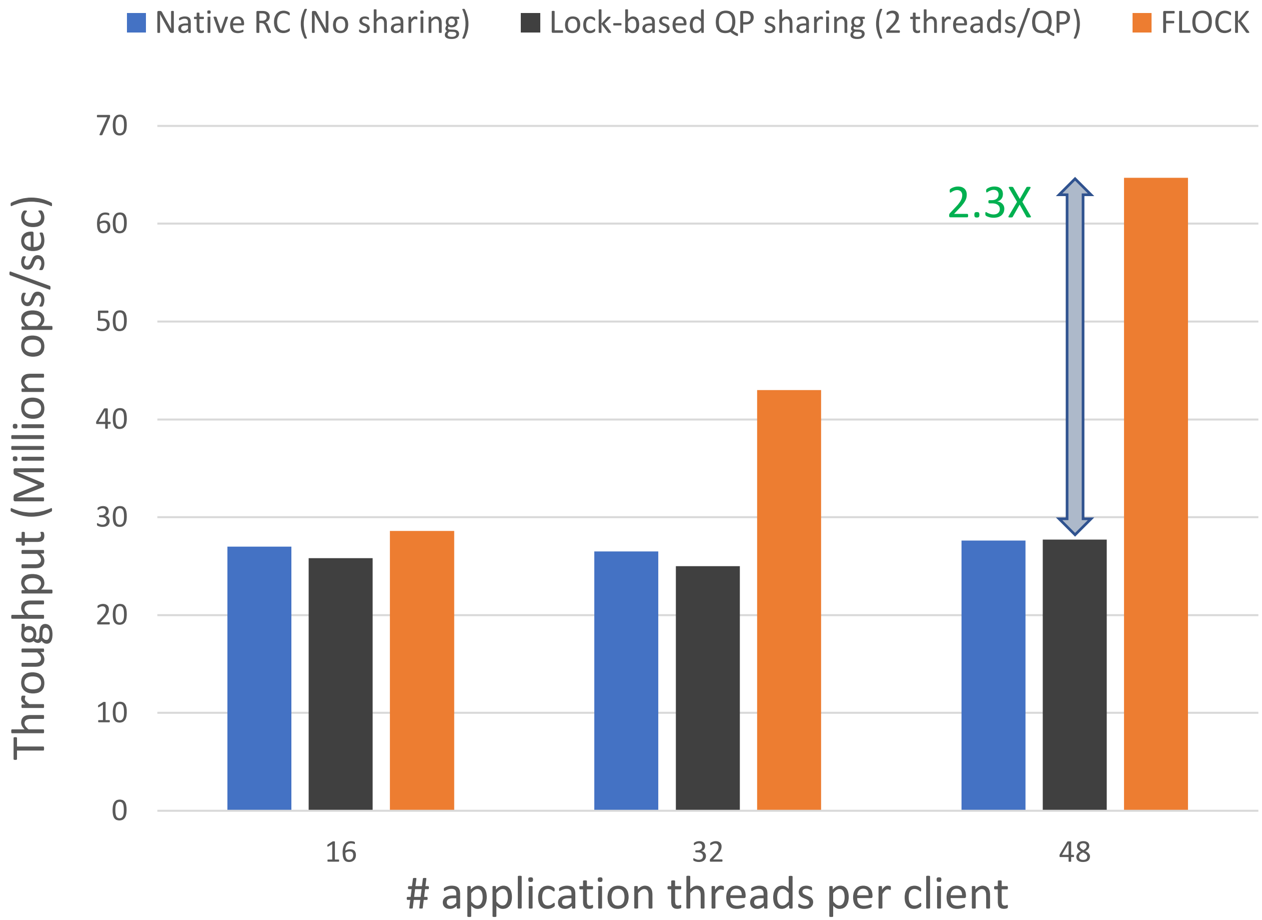


- ❑ FLOCK throughput up to 3.4X against eRPC
- ❑ Tail latency lower by up to a factor of 2

+ Coalescing enables more concurrency at the clients & scheduling limits active QP count

- UD-based RPCs have higher CPU overheads

Scalability with Symbiotic Scheduling



- ❑ Similar performance up to 16 threads
- ❑ FLOCK outperforms others by up to 133% at higher thread counts
- ❑ Sharing using spinlock serializes threads working on the same QP
- ❑ Coalescing in FLOCK enables concurrent request submission by threads sharing a QP, reducing messages transferred by client as well as server

Distributed Transaction Processing

Configuration

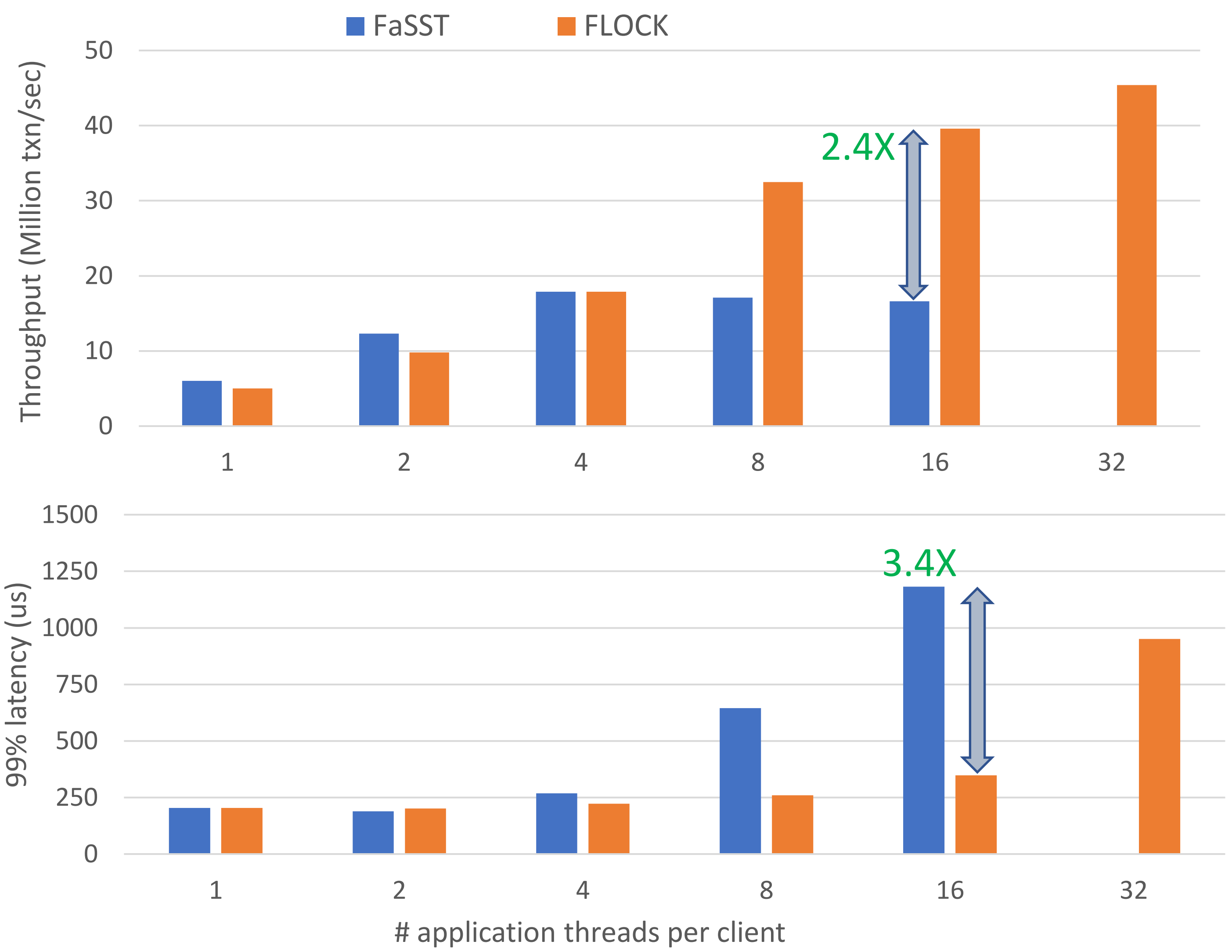
- comparison against FaSST, an RDMA-based transaction processing system
- Transaction protocol like FaSST : OCC[1] and 2-phase commit to provide serializable transactions
- 3 servers and 20 clients

Workloads

- TATP (read-intensive)
- Smallbank (write-intensive)

[1] Optimistic Concurrency Control

FLOCK vs FaSST for TATP



- ❑ FaSST performs better up to 2 threads, but its performance saturates at 4 threads
- ❑ Throughput in FLOCK up to 2.4X FaSST with lower median and tail latency
- ❑ FLOCK's performance improves with higher thread counts due to better coalescing and efficient network utilization

* FaSST suffers packet loss at 32 threads

Other evaluations

- Performance under increasing node counts
- Impact of coalescing on network and CPU utilization
- Head-of-line blocking mitigation using symbiotic scheduling
- Comparison with eRPC using in-memory index structure (HydraList)

Conclusion

FLOCK

- targets balancing the performance-scalability tradeoff in vanilla RDMA hardware
- offers low overhead QP sharing using leader-follower synchronization
- a cooperative scheduling mechanism between client and server to limit the maximum load at the server
- superior performance with efficient network utilization and reduced CPU usage

Thank you!