# Infrastructure for Linearizable RPCs in RAMCloud

Seo Jin Park

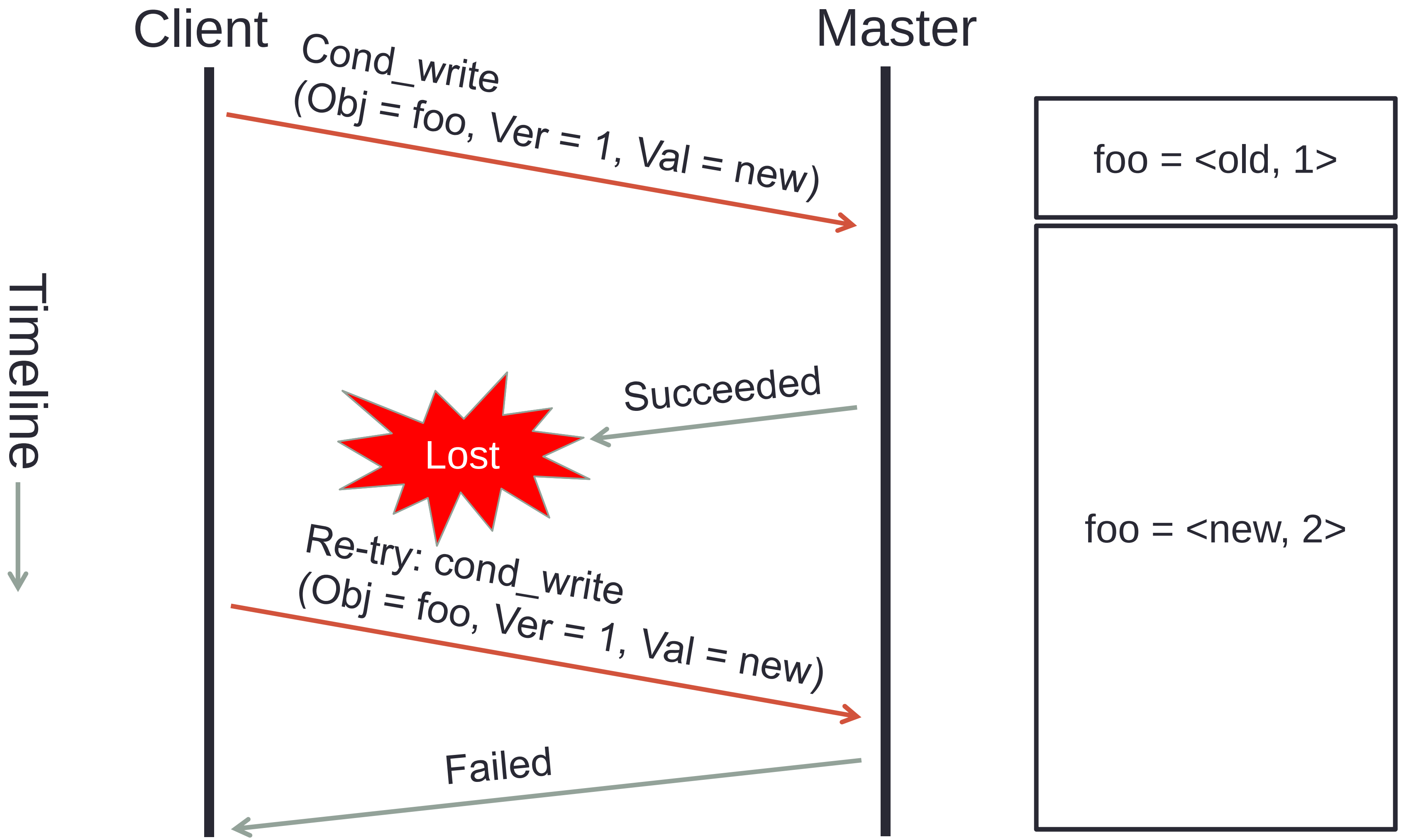
Stanford University

# What is Linearizability and Problem?

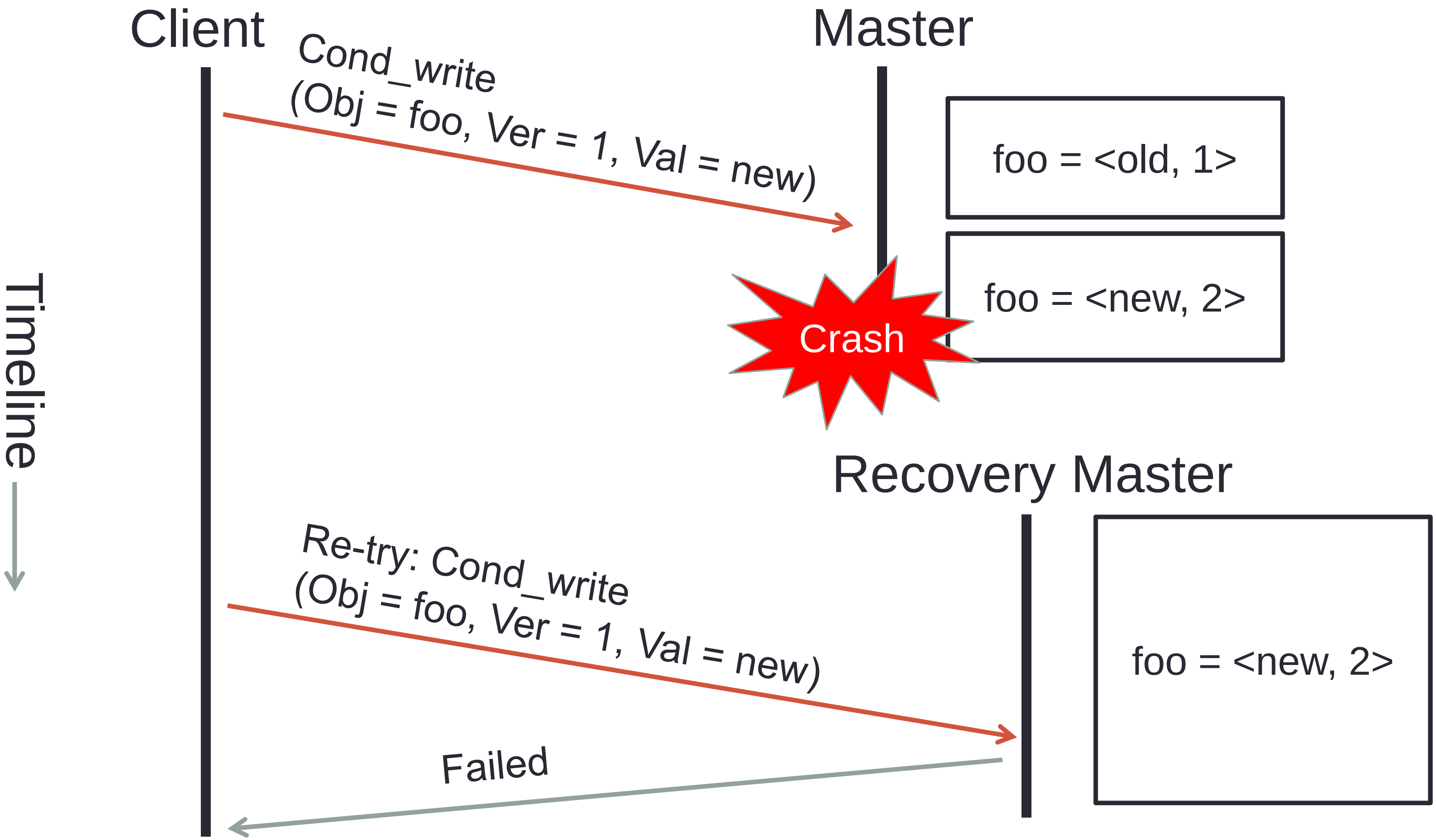
- In concurrent programming, an operation (or set of operations) is ***linearizable*** if it appears to the rest of the system to occur instantaneously.

- A RPC in RAMCloud is not linearizable for re-executions in certain circumstances (eg. server crash) because the same RPC could be executed multiple times.

# Broken Conditional Write



# Broken Conditional Write 2

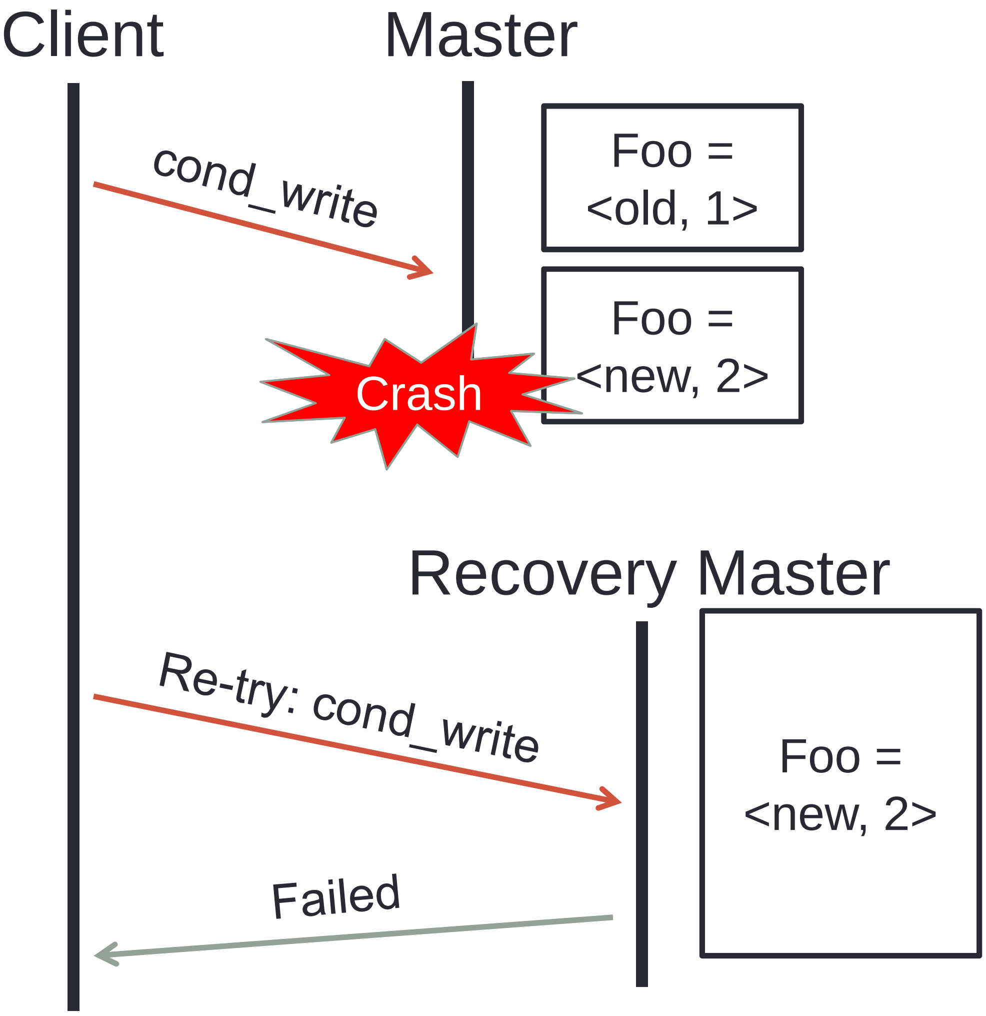


# Solution

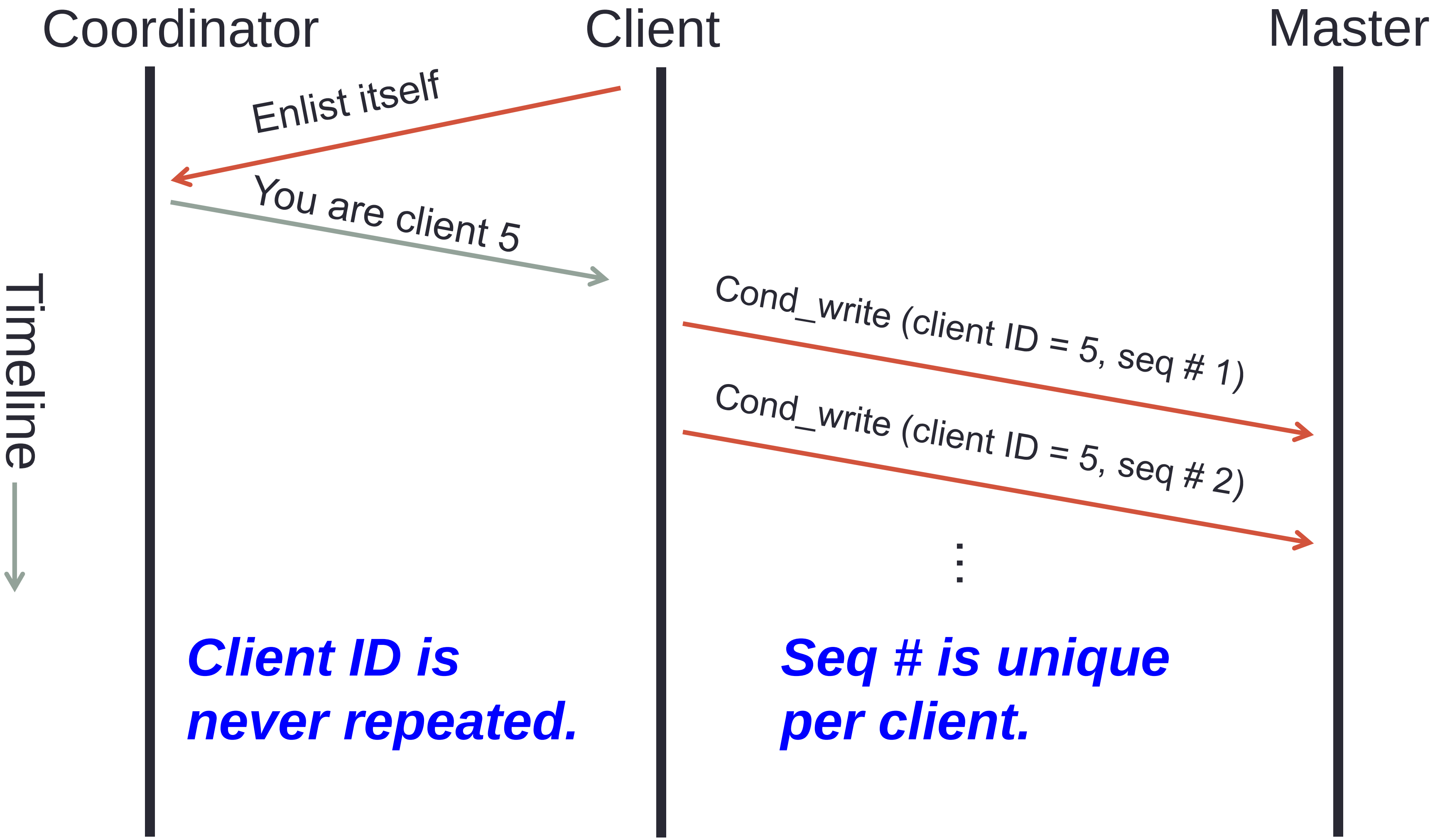
- *Save the results of RPCs on masters.*
- *If a master is asked to execute the same RPC again, it just returns the old result instead of re-executing.*

# Required Features

- Identify the same re-tried RPCs.
- Save the results of RPCs on masters log.
- Fast lookup for the saved results.
- After crash, distribute result log entry to correct recovery master.
- On recovery master, reconstruct lookup table from log.
- Garbage collection for lookup table, client state, and log entries.



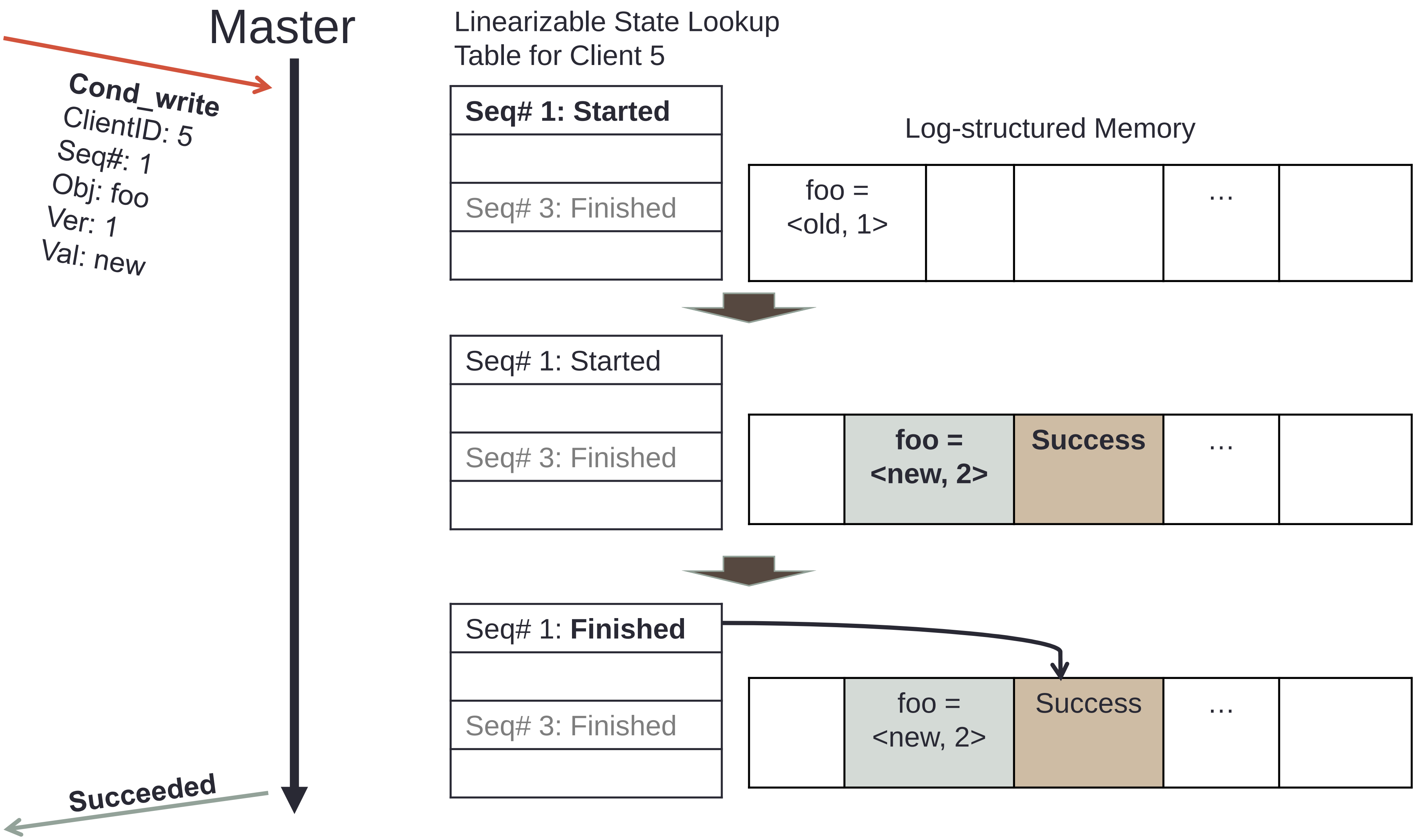
# Client: provides a unique RPC ID


Coordinator
Client
Master
Enlist itself
You are client 5
Cond_write (client ID = 5, seq # 1)
Cond_write (client ID = 5, seq # 2)
⋮
Timeline

***Client ID is never repeated.***

***Seq # is unique per client.***

# How does Master Save Results of RPCs



Master

**Cond_write**
ClientID: 5
Seq#: 1
Obj: foo
Ver: 1
Val: new

**Succeeded**

Linearizable State Lookup Table for Client 5

**Seq# 1: Started**

Seq# 3: Finished

Log-structured Memory

foo = <old, 1>

…

Seq# 1: Started

Seq# 3: Finished

**foo = <new, 2>**

**Success**

…

Seq# 1: **Finished**

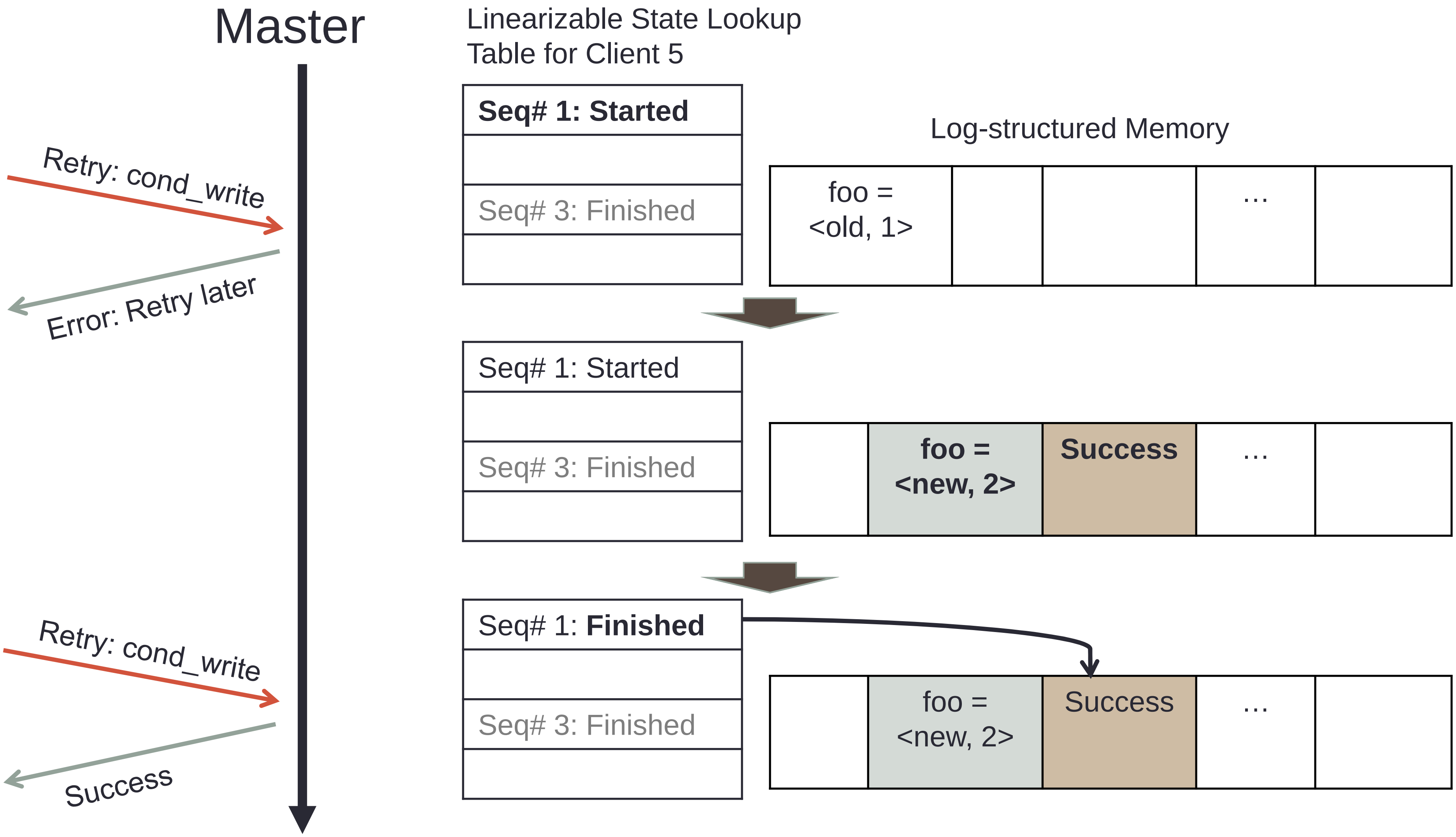
Seq# 3: Finished

foo = <new, 2>

Success

…

# What happens for re-tries RPCs?



Master

Retry: cond_write

Error: Retry later

Retry: cond_write

Success

Linearizable State Lookup Table for Client 5

**Seq# 1: Started**

Seq# 3: Finished

Log-structured Memory

foo = <old, 1>

…

Seq# 1: Started

Seq# 3: Finished

**foo = <new, 2>**

**Success**

…

Seq# 1: **Finished**

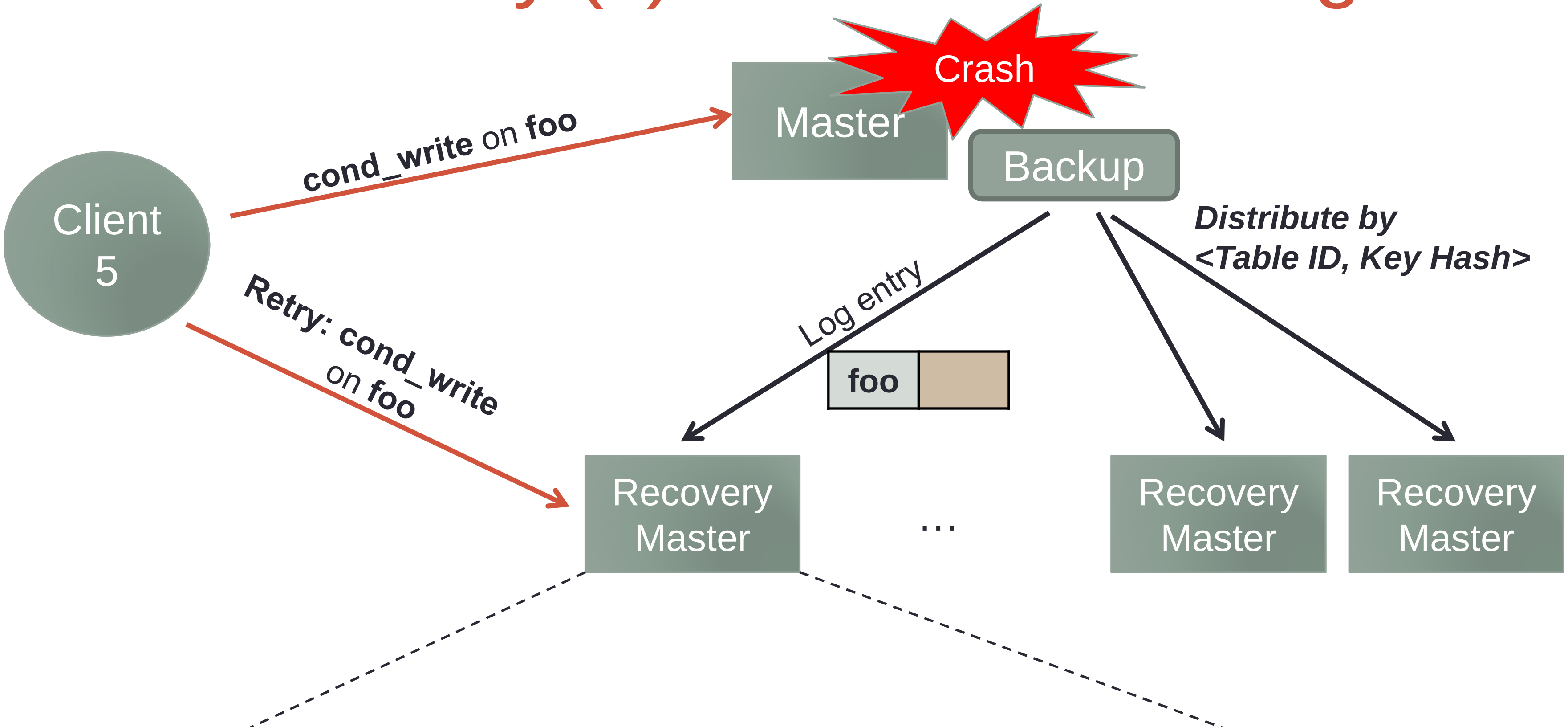
Seq# 3: Finished

foo = <new, 2>

Success

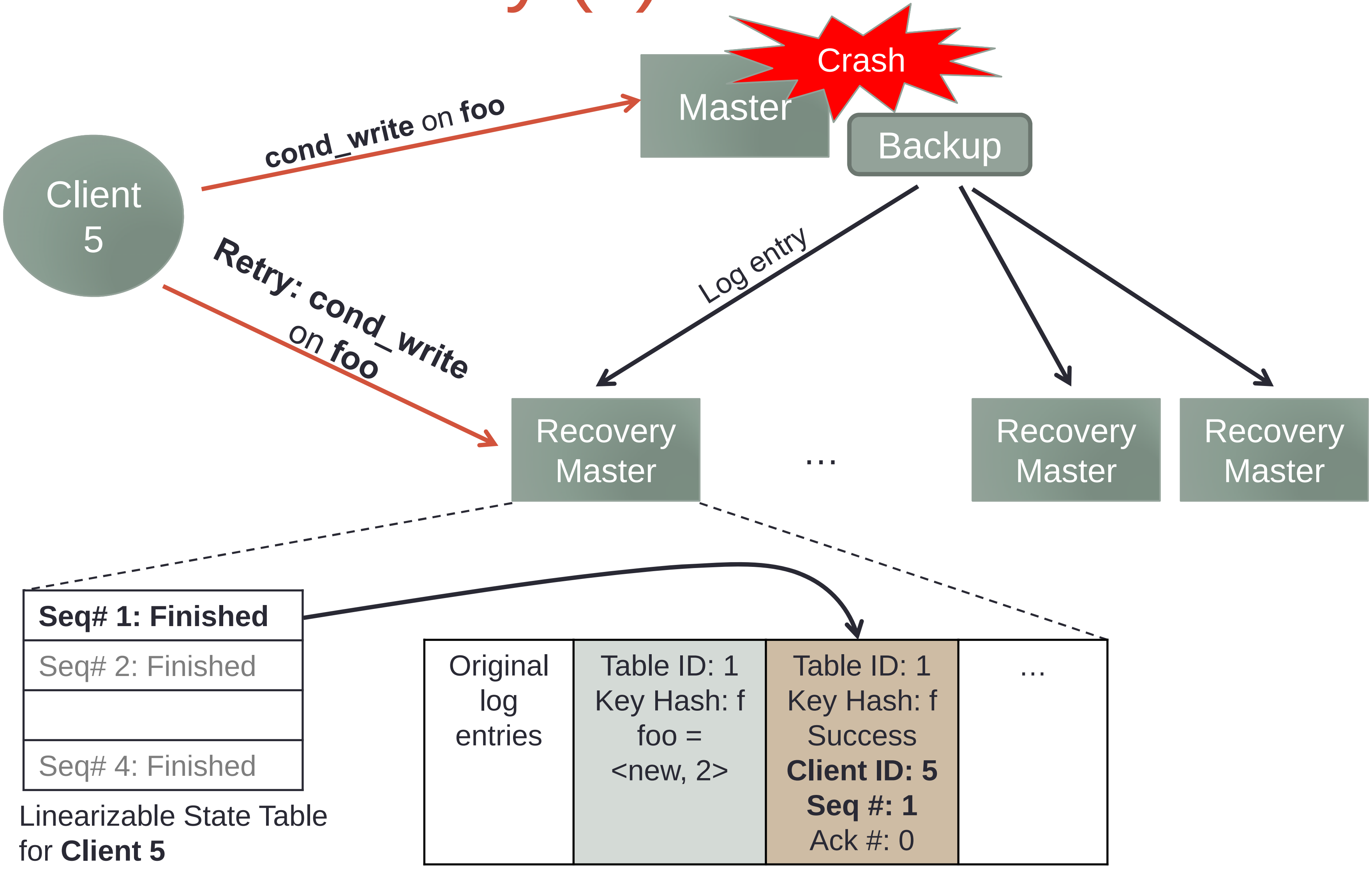
…

# Crash Recovery (1): Distribution of Log



| | | | | |
|---|---|---|---|---|
| Original log entries | **Table ID: 1<br>Key Hash: f<br>foo = <new, 2>** | **Table ID: 1<br>Key Hash: f<br>Success** | … | |

# Crash Recovery (2): Reconstruction



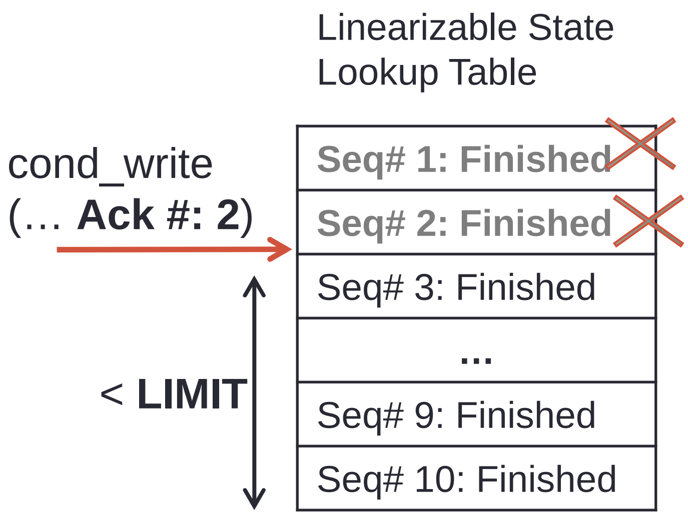
| |
|---|
| **Seq# 1: Finished** |
| Seq# 2: Finished |
| |
| Seq# 4: Finished |

Linearizable State Table
for **Client 5**

| | | | |
|---|---|---|---|
| Original log entries | Table ID: 1<br>Key Hash: f<br>foo =<br><new, 2> | Table ID: 1<br>Key Hash: f<br>Success<br>**Client ID: 5**<br>**Seq #: 1**<br>Ack #: 0 | … |

# Garbage Collection 1: Linearizable States

- Client attaches **Ack #** to every linearizable RPC. (Acknowledging the receipt of all results for **Seq #** <= **Ack #**)
- Master can clean up all records up to highest **Ack #** seen.
- Client limits the number of outstanding RPCs by keeping (**Seq #** – **Ack #**) < **LIMIT**, so that a master only needs O(LIMIT) space per client.



Linearizable State Lookup Table

cond_write (… **Ack #: 2**)

< **LIMIT**

~~**Seq# 1: Finished**~~

~~**Seq# 2: Finished**~~

Seq# 3: Finished

**…**

Seq# 9: Finished

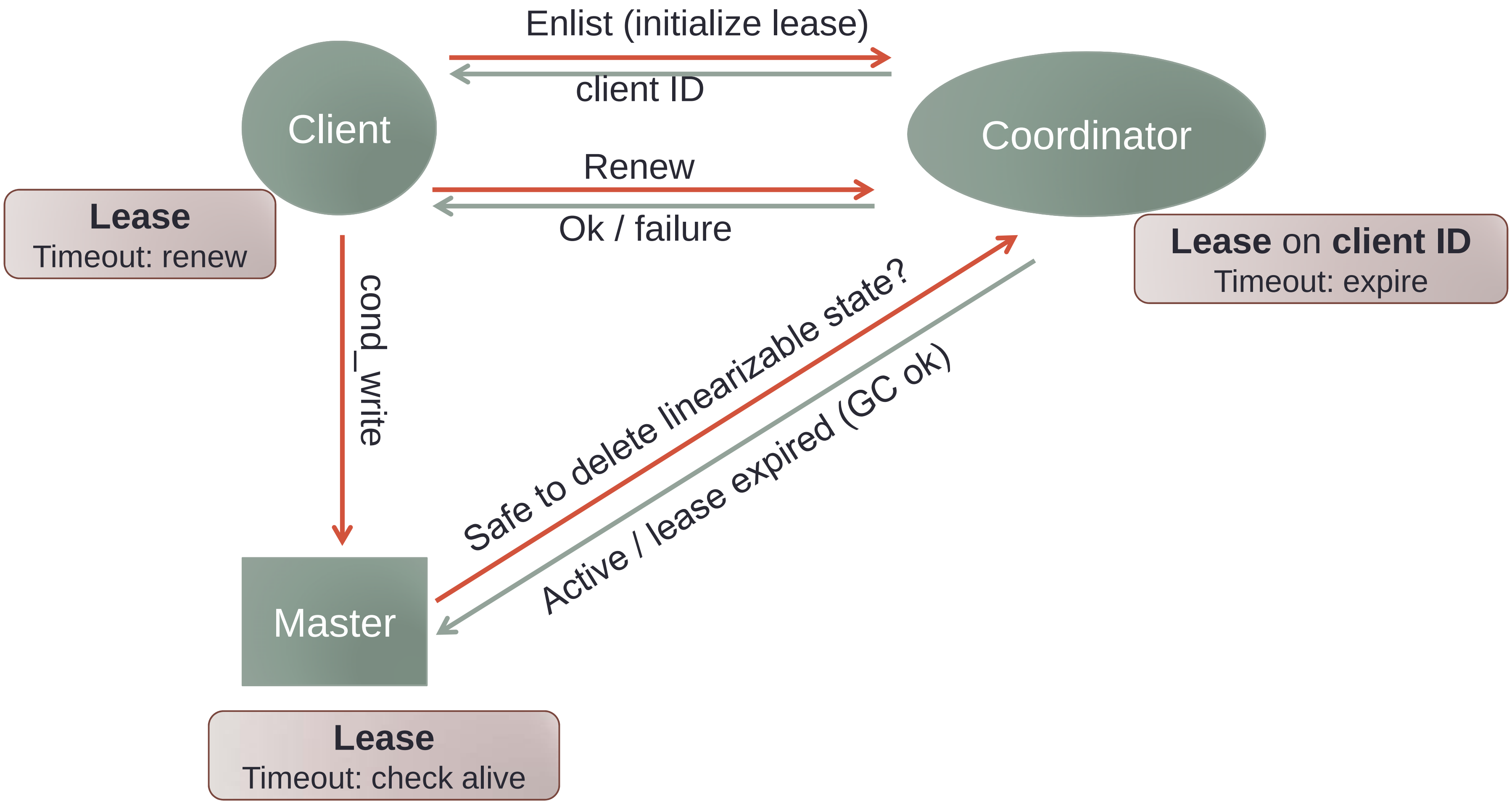
Seq# 10: Finished

# Garbage Collection 2: Client State

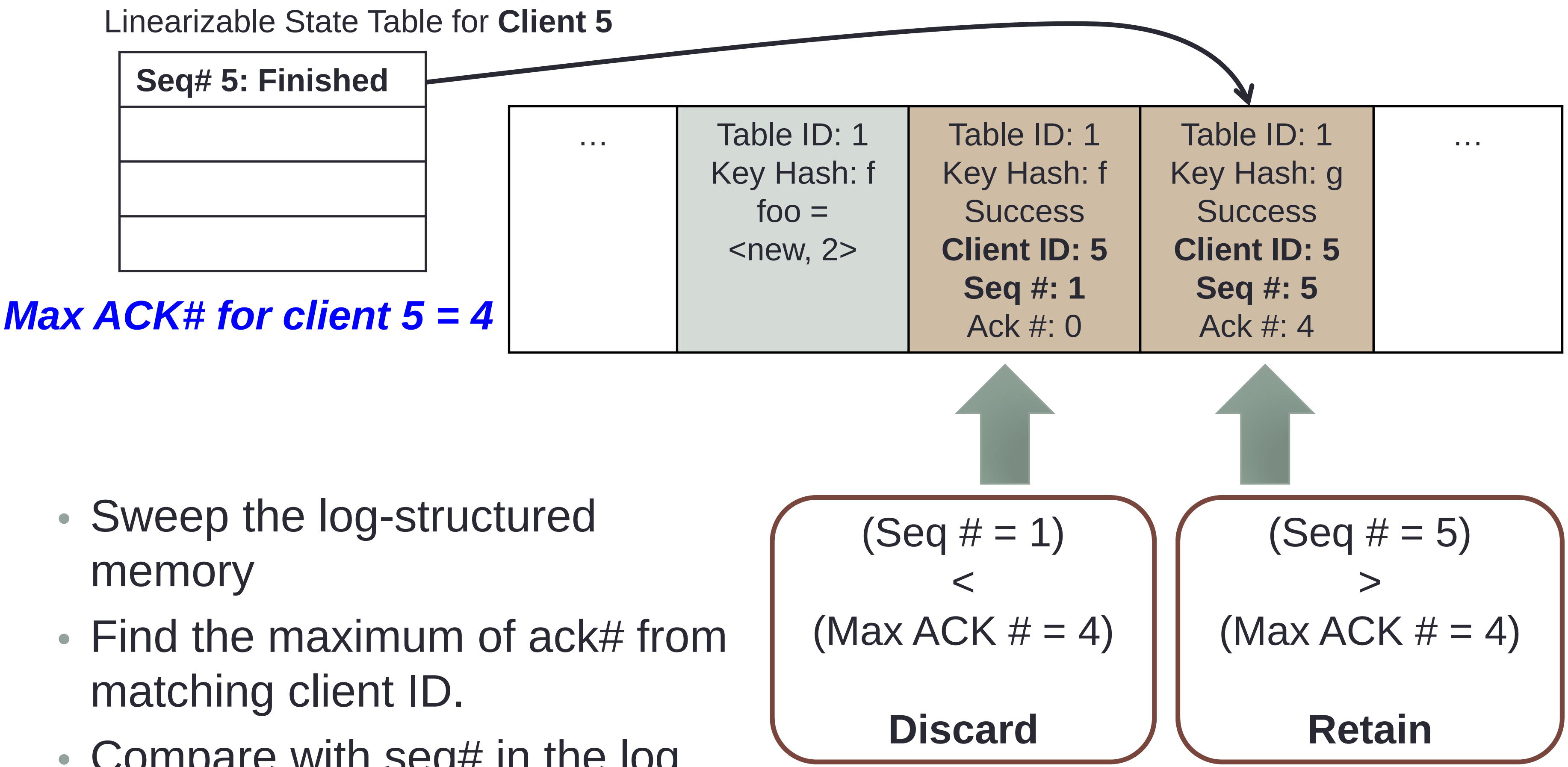
**Problem**: when is safe to clean up client state on masters? (If the client is alive after clean up, master may re-execute RPCs.)

- Client maintains lease for its client id and renews it as long as it want to keep its linearizable states on masters.
- Coordinator keeps the main lease.
- Master keeps a local lease. On timeout, master asks coordinator whether lease is alive.

# Garbage Collection 2: Client State



# Garbage Collection 3: Log cleaner



Linearizable State Table for **Client 5**

| **Seq# 5: Finished** |
|---|
| |
| |
| |

***Max ACK# for client 5 = 4***

| … | Table ID: 1<br>Key Hash: f<br>foo =<br><new, 2> | Table ID: 1<br>Key Hash: f<br>Success<br>**Client ID: 5**<br>**Seq #: 1**<br>Ack #: 0 | Table ID: 1<br>Key Hash: g<br>Success<br>**Client ID: 5**<br>**Seq #: 5**<br>Ack #: 4 | … |
|---|---|---|---|---|

(Seq # = 1)
<
(Max ACK # = 4)

**Discard**

(Seq # = 5)
>
(Max ACK # = 4)

**Retain**

- Sweep the log-structured memory
- Find the maximum of ack# from matching client ID.
- Compare with seq# in the log.

# What’s done so far?

## Features implemented

- Identify the same re-tried RPCs.
- Fast lookup for the saved results.
  - 70 nanoseconds overhead for turning on linearizability
- Garbage collection for lookup table.

## Future work

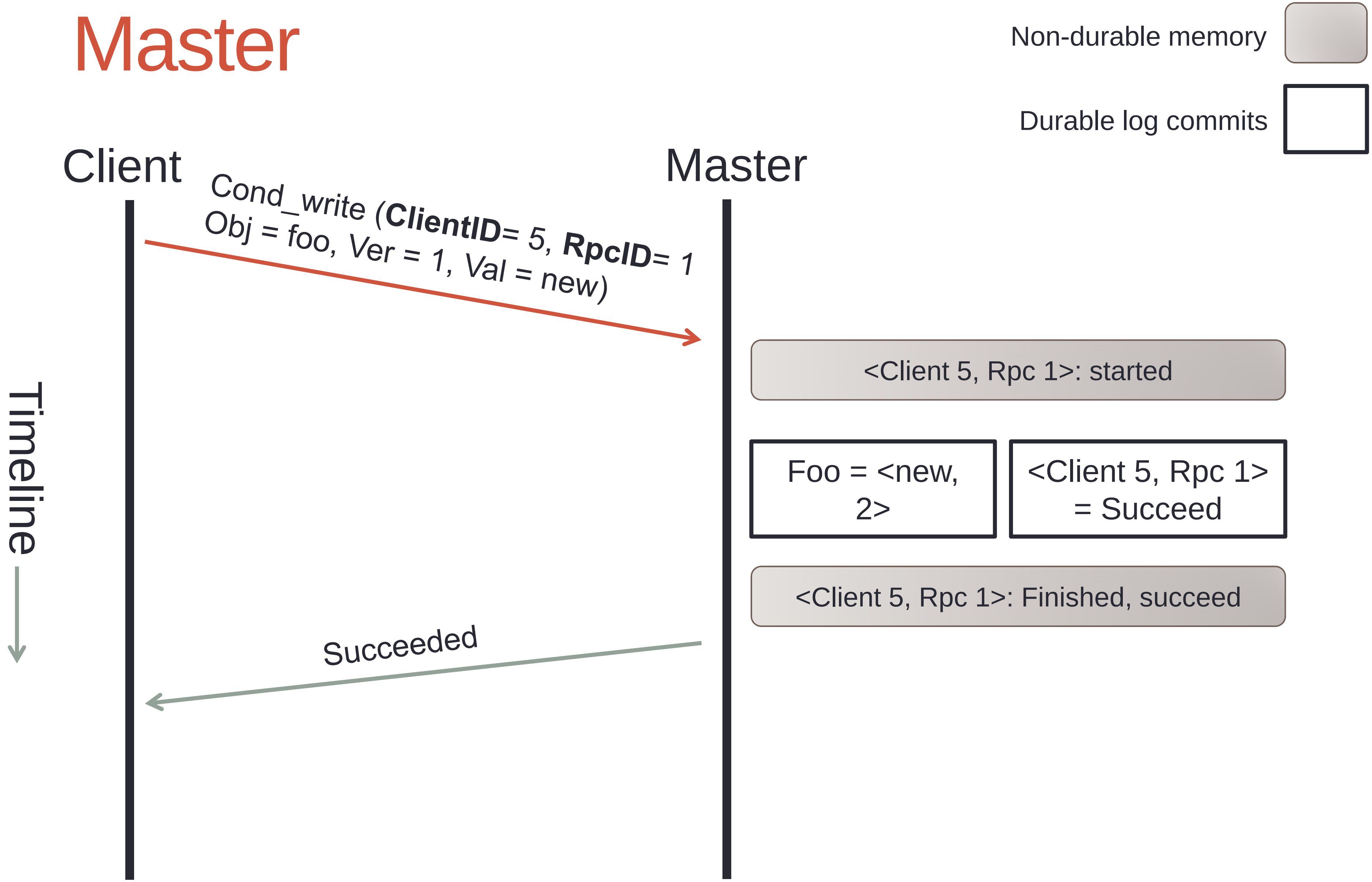
- Save the results of RPCs on masters log.
- After crash, distribute result log entry to correct recovery master.
- On recovery master, reconstruct lookup table from log.
- Garbage collection for client state and log entries.

# Conclusion

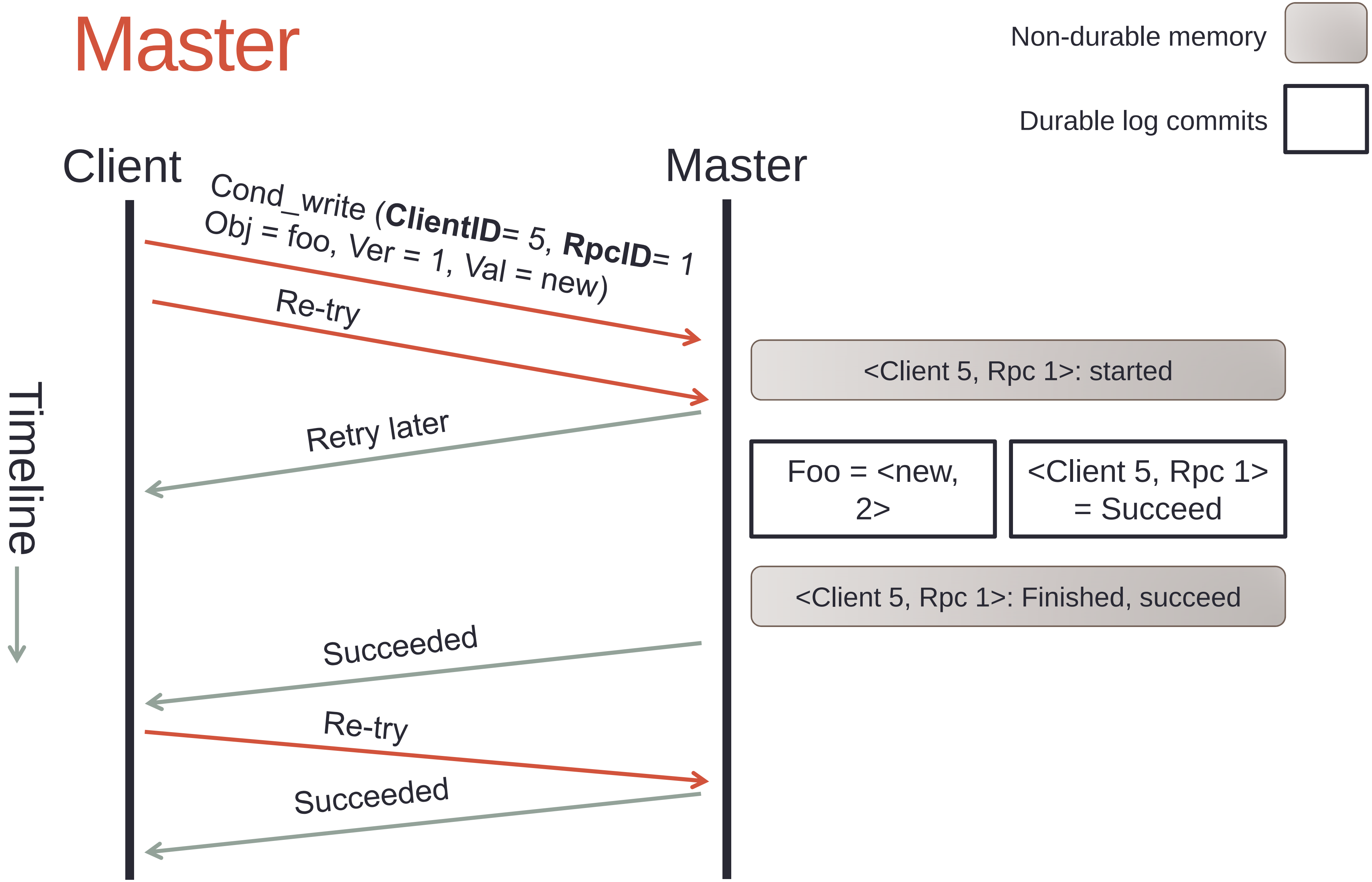
- We build high performance distributed system without compromising consistency.
- Durable logging system was key component and made design simple.
- The most trickiest part to design correctly was garbage collection. (~40% of time)

# Q & A

# Master


Non-durable memory
Durable log commits
Client
Master
Cond_write (**ClientID**= 5, **RpcID**= 1
Obj = foo, Ver = 1, Val = new)
<Client 5, Rpc 1>: started
Foo = <new, 2>
<Client 5, Rpc 1> = Succeed
<Client 5, Rpc 1>: Finished, succeed
Succeeded
Timeline

# Master



Non-durable memory

Durable log commits

Client

Master

Cond_write (**ClientID**= 5, **RpcID**= 1
Obj = foo, Ver = 1, Val = new)

Re-try

<Client 5, Rpc 1>: started

Retry later

Foo = <new, 2>

<Client 5, Rpc 1> = Succeed

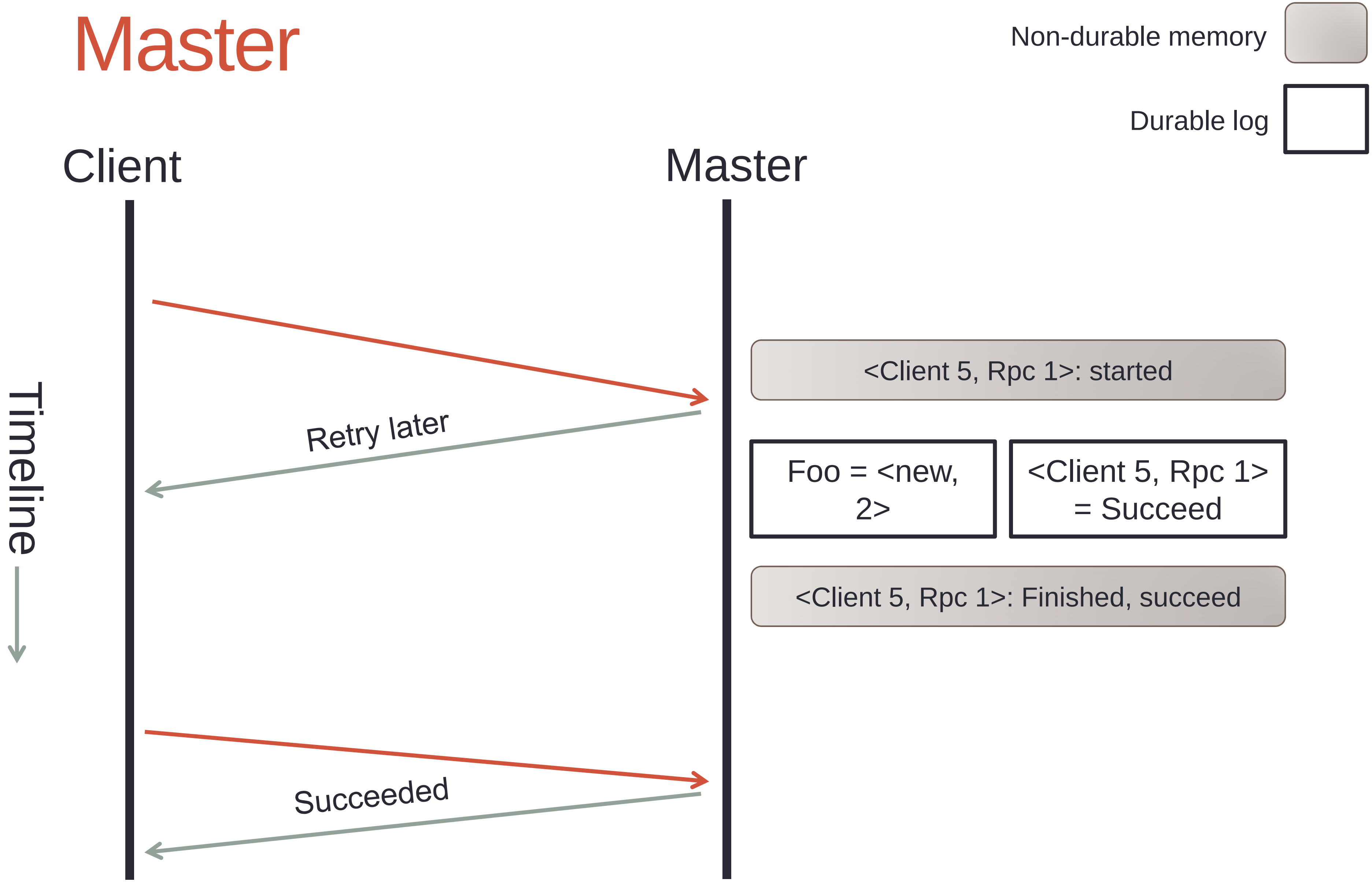
<Client 5, Rpc 1>: Finished, succeed

Succeeded

Re-try

Succeeded

# Master



Non-durable memory
Durable log
Client
Master
Timeline
Retry later
Succeeded
<Client 5, Rpc 1>: started
Foo = <new, 2>
<Client 5, Rpc 1> = Succeed
<Client 5, Rpc 1>: Finished, succeed

# Structure of Rpc Log Entry

| Field | Purpose |
|---|---|
| **Result** | Replying duplicate rpcs in future |
| **<Table ID, Key Hash>** | Distributing log entries to correct recovery masters during recovery |
| **<Client ID, Rpc ID, Ack ID>** | Reconstructing master's linearizable state during recovery |

***A Master atomically writes this log entry and new object on log.***

# Distribution of log entry

- During crash recovery, log entries get split to many recovery masters.
- After recovery, re-tried RPCs will be directed to new recovery masters.
- Every linearizable RPC is tied to an object.
- Linearizable RPC is routed to a master by <Table ID, KeyHash>
- By referring <Table ID, KeyHash> value in a log entry, we can decide which recovery master is in charge.

# Reconstruction of linearizable state

- On crash recovery, a recovery master should incorporate old master's linearizable state, so that it can still avoid re-execution of linearizable RPCs executed in old master.
- As recovery master receives rpc log entries, it adds new entries to its linearizable state by referring <Client ID, Rpc ID, Ack ID> and Result.