At-most-once Algorithm for Linearizable RPC in Distributed Systems

Seo Jin Park[1]

[1]Stanford University

***Abstract*—In a distributed system, like RAMCloud, supporting automatic retry of RPC, it is tricky to achieve linearizability correct along with durability and liveness. Automatic retry may cause duplicate executions when a server fails after durably recording the original execution and before sending its response. We can cope with the problem by implementing at-most-once mechanism for RPC system. But in a massively distributed system, the implementation of the algorithm is not simple; correct recovery of RPC information and garbage collection for the information in cluster level are pretty challenging. In this project, I am presenting solutions to the problems and implementation in a distributed storage system, called RAMCloud.**

I. Introduction

We say that a system supports a linearizable RPC if the system acts as though it executed the RPC instantaneously and exactly once at some point between the RPCs invocation and response [2]. Unfortunately, RPCs currently supported by RAMCloud are neither linearizable during normal operation nor in the face of certain expected events (e.g., master failure). For example, consider a write RPC invoked by a client on a master where the master receives the request, performs the write, but whose response is never received by the client. This may occur due to a number of acceptable events such as dropped network packets or failed masters. RAMCloud handles missing responses by retrying the RPC. Retrying RPCs with missing responses is appropriate for ensuring that the RPC is executed at-least-once since it is possible that the reason for the missing response is a dropped request that never made it to the master. However, if the master did in fact execute the request, any subsequently received request will amount to re-execution where the re-execution may even occur after a response has been received by the client (in the case a network delayed requests). In this way the fundamental RAMCloud RPC is not linearizable.

Non-linearizable RPCs endanger the correctness of the whole system. Particularly network-delayed request can be fatal. Assume a single client is setting up a new bank account for a customer; client write balance to be zero, followed by subsequent transactions. The original request for the write zero balance was delayed on network layer, and client retried the request and succeeded. The network-delayed request to write zero can reach master server later, resulting in deletion of all transactions so far. To avoid this problem, client should not retry initial write and any subsequent write should be conditional write although only a single client modified the balance synchronously and there are no concurrent accesses.

In another example, a different problem exists for conditional write; a client sends a conditional write RPC request, and the responsible master crashes in between processing the conditional write and replying with the result of the conditional write. Then the client will retry the conditional write since it thinks the original request was lost. A recovery master will have the updated value (since the original master had committed the result on the log) and will respond to the re-tried conditional write with a failure due to a version number mismatch. The correct behavior should be that the recovery master understands that the request is being retried and responds with the original result *success*.

Outside of RAMCloud, many systems use retry and at-least-once mechanism and handle their non-linearizable RPCs with external resolution mechanism [1]. Even in system without retry, a packet for an RPC may be duplicated and delayed in network layer. This is dangerous as well and any nave RPC systems can be vulnerable.

The non-linearizable nature of the system stems from our inability to distinguish been RPCs that have and have not been previously serviced. The solution, therefore, is to maintain enough information so that it is possible to distinguish between serviced and unserviced RPCs. Specifically, we resolve this problem by saving 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. We first introduce RAMCloud in section II to help understanding of our design decisions, followed by description of our design during normal operation (section III) and failure situation (section IV). To build a practical system, we discuss a mechanism to bound the amount of data that is accumulated and a mechanism for safely garbage-collecting accumulated data in cluster-level (section V). The performance analysis in section VI shows that adopting linearizable RPC costs negligible latency penalty and minimal DRAM space.

II. Background: Intro to RAMCloud

RAMCloud is a key-value storage system that provides low-latency access with durability. The system supports large-scale datasets, which may span thousands of servers. RAMCloud consists of multiple master servers (for holding data) and a single coordinator server (for sharding key hash and recovery coordination) as shown in figure 1. To achieve low-latency access, RAMCloud stores at least a copy of data in DRAM and provides durability of the data by keeping backup copies on secondary storage of other servers. The system utilizes log-structured storage mechanism both for DRAM and secondary storage, which provides high utilization and performance [5]. RAMCloud also provides high availability of data by recovering from crashes within 1-2 seconds [3].

A. Data Model

RAMCloud is basically a key-value store, with a few extensions. Data in RAMCloud is organized by *tables*, each of which is identified by a unique 64-bit identifier. A table contains any number of *objects*, each of which is composed of a variable-length unique *key*, a variable-length *value*, and a monotonically increasing 64-bit *version number*.

A client of RAMCloud may use read, write, and delete operations on individual objects, and RAMCloud also provides two atomic operations, **conditionalWrite** and **increment**, which can be used to synchronize concurrent accesses. The **conditionalWrite** overwrites an object only if the condition specifying version number of object is met.

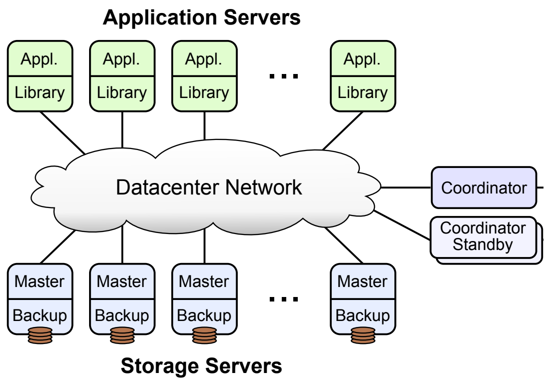

Fig. 1: Overal architecture of RAMCloud

B. Log-structured Storage [5]

RAMCloud uses log-structured mechanism for both DRAM and secondary storage to implement the key-value store, which is similar to a log-structured file system [4]. Each master maintains an append-only log, which is the only storage for data of all objects in its assigned key-hash range. RAMCloud keeps the same log structure both for primary copies in memory and backup copies on secondary storage.

Log-structured storage provides several benefits: high throughput, crash recovery, efficient memory utilization, and consistency. Especially, the log simplifies crash recovery problem; in case of masters crash, its log can be replayed to reconstruct the information that was in memory.

C. Master Crash Recovery [3]

RAMCloud divides the recovery work across many nodes for fast recovery. During normal operation, each master scatters its log segment replicas across all masters since the replicas can be read concurrently during crash recovery. If a master crashes, the coordinator selects a set of existing servers, called *recovery masters*, to take over the crashed master's data. Each of the recovery masters gets assigned a subset of the crashed master's tablets (each tablet is defined by key-hash ranges).

At this point a massive data shuffle takes place: each backup reads segment replicas, divides their log entries into buckets for each recovery master, and transmits the buckets to the corresponding recovery masters. Each recovery master appends the incoming log entries to its log and creates a hash table entry for the current version of each live object. Once this process completes, all requests on the crashed servers tablets are directed to the recovery masters.

III.Normal operation

As mentioned in section I, a RPC in RAMCloud is not linearizable with respect to re-execution in certain circumstances (eg. server crash or network delay) because the same RPC could be executed multiple times.

We resolve this problem by saving the results of RPCs on master servers. If a master is asked to execute the same RPC again, it just returns the old result instead of re-executing. To implement this solution, modifications on client, master, and logging system were required.

A. Client provides a unique ID for a RPC

Each client provides a unique ID for each RPC, so that masters can detect duplicate RPCs. Also, a client has to tell masters about results it has received, so that master can discard saved results that are no longer needed (Detail of garbage collection is in V-A).

Client ID assignments The identification of an RPC has two components: an rpc sequence number (shortened to *RPC ID*) and a *client ID*. Since the rpc ID is only unique within a client, a master needs an ID for the client to distinguish duplicate RPCs. When a client starts, it enlists itself with the Coordinator and obtains a unique client ID, which is attached to every linearizable RPC later.

Rpc ID & ack ID assignment A client-side data structure, called *RpcTracker*, keeps the last assigned rpc ID and the status of all outstanding RPCs whose results have not yet been received. We use RpcTracker to assign new rpc IDs and calculate *ack ID*s (which indicate that responds have been received for all RPCs with rpc ID below the number).

Maintaining lease on coordinator Before sending a linearizable RPC to a master, a client must make sure that the lease for its client ID is active. Enlisting with coordinator (for obtaining client ID) initiates the lease on coordinator side, and the client should maintain the lease by sending renew requests to coordinator before expiration as long as it want masters to keep its linearizable states. This lease-like mechanism is used to ensure safe garbage collection for client states (See section V-B for how masters use this lease.)

B. Master keeps records of RPCs

With the client ID, rpc ID, and ack ID from a client, a master avoids re-execution of duplicate RPC by saving the results of RPCs in its internal memory (for fast lookup) and in its log (for durability). I will talk about the details of logging in section IV, and now focus on the internal memory data structure.

Memory data structure: UnackedRpcResults After a master completes an RPC, it records its result in the log. To locate the result in the log efficiently, each master keeps a data structure called UnackedRpcResults. A master records the start and the completion of an RPC execution in UnackedRpcResults, so that it can check whether an incoming RPC is a duplicate later.

The scenario of executions is shown in figure 2. When a linearizable RPC arrives at a master, the master checks whether a duplicate RPC is in progress or completed. If not, it marks the start of processing in UnackedRpcResults. When the master durably writes new value for object on log, it also appends information of the RPC together atomically. After logging the RPC record, it updates the status of the RPC in UnackedRpcResults to *completed* with the pointer to the RPC record in log. Finally, the master responds back to client.

RAMCloud master can service multiple RPCs concurrently. If a duplicate RPC comes in while processing the original RPC (after marking its start in UnackedRpcResults), the master respond the duplicate request with **Retry Later**, which triggers the client to retry the same RPC later. If a duplicate

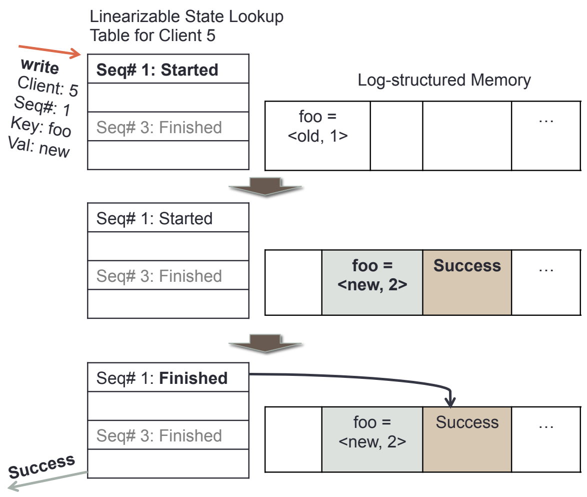

Fig. 2: Normal operation

RPC arrives after the original RPC is finished (after marking its completion in UnackedRpcResults), the master retrieves the result of the original RPC from log and respond with the saved information.

IV. Crash Recovery

For durability, we store the result of each linearizable RPC in log (both in a master's log-structured memory and backups). For that, we defined a new type of log entry and we can recover all necessary information from the log entries after a crash.

A. Structure of the new type of log entry: rpc log entry

For each completed linearizable RPC, the new type of log entry is written. The log entry has **Result** (for replying to duplicate rpcs in future), ⟨**TableId, KeyHash**⟩ (for finding the correct recovery master during recovery, more details in section IV-B), and ⟨**clientId, rpcId, ackId**⟩ (for re-constructing UnackedRpcResults during recovery). I will call this new type of log entry an rpc log entry. On each update of an object with linearizable RPC, we should write both the new object log entry and the new rpc log entry atomically. As we write them atomically, we can guarantee the consistent behavior on crash: either an RPC is never executed, or it is executed and duplicate RPCs will be detected.

B. Distribution of rpc log entries after crash

During crash recovery, log entries get split to many recovery masters. After crash recovery, a re-tried linearizable RPC will be directed to the new recovery master (which received the corresponding log entries), so relevant rpc log entries should also migrate to the master. We associate an object with each linearizable RPC and refer to the ⟨TableId, KeyHash⟩ value in a log entry to decide which recovery master should receive the rpc log entry.

C. Reconstruction of UnackedRpcResults table during crash recovery

On crash recovery, we can reconstruct UnackedRpcResults with the following steps. After a recovery master gathers relevant portion of rpc log entries, it just replays the rpc log entry by using ⟨clientId, rpcId, ackId⟩ to re-construct (or add new info to) UnackedRpcResults on the recovery master.

D. Log cleaning for new log entry type

The existing log cleaner should be modified to handle cleaning of rpc log entries. When it finds an rpc log entry with ⟨clientId, rpcId, ackId⟩, the log cleaner checks the UnackedRpcResults to figure out whether the rpc is already acknowledged. If it is, we can clean it up. If not, the log entry is still valid and needs to be copied to a new location in the log. After relocating the log entry, we should update UnackedRpcResults to have correct pointer to the log entry.

V. Garbage Collection

As a practical system, our design bounds the amount of data that is accumulated in two ways: 1) limiting data per client 2) garbage collecting expired client state.

A. Garbage collection by ack ID

The memory data structure, UnackedRpcResults, may grow very quickly as we need a separate entry for every linearizable RPCs. We need a way to clean up information that is no longer needed. We use the ack IDs attached to linearizable RPCs to clean up the stale records safely. The difference of rpcId - ackId is bounded by a fixed constant,

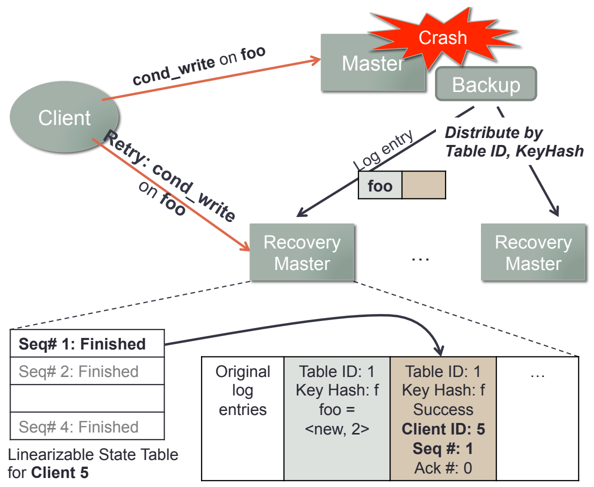

Fig. 3: While recovery of master, rpc log entry is distributed by tableID and keyHash, so that it always comes with object log. Recovery master replays the incoming rpc log entries and reconstructs UnackedRpcResults data.

and a client waits for ackId to increase before sending more linearizable RPCs, guaranteeing bounded memory use for the client on a master.

B. Cleanup for disconnected client

By our scheme of garbage collection by ackId, disconnected (or dead) clients always leave the last entry unacknowledged. We garbage collect this by implementing a lease-like mechanism. As in section III-A, a client maintains the lease with its client ID on coordinator for its lifetime. Our lease is defined under no clock synchronization assumption. A single cluster-level time is used for deciding expiration of lease. The coordinator declares 64-bit integer **ClusterTime**, which advances monotonically by coordinators clock. The coordinator broadcasts the ClusterTime to every master and client. (In practice, it is piggybacked on other messages.)

When a client renews its lease from the coordinator, it obtains a new lease expiration time and current ClusterTime. The client attaches its best knowledge of the current ClusterTime and its lease expiration time to every linearizable RPC. As a master handles the linearizable RPC, it updates its cache of ClusterTime, and check whether provided lease expiration time is larger than current ClusterTime. If not, master checks with coordinator for lease validity before processing the request. If coordinator tells the lease is already expired, master rejects the request and deletes the state data of the client.

In addition to the garbage collection driven by stale RPC, master periodically check the validity of lease and garbage collect them.

Thanks to the monotonicity of ClusterTime, our lease mechanism doesn't depend on clock synchronization but only on bounded clock drift for liveness. (Correctness doesn't depend on any assumption on clock.)

VI. Performance Analysis

Implementing linearizability required additional operations such as durable log append and hash table lookup. We implemented the described algorithm on RAMCloud and measured performance impact of the additional operations. The latency of write, conditionalWrite and increment operations increased slightly, enough to be considered as no penalty. We also calculated storage burden caused by client state table (called UnackedRpcResult) and in-memory log entries. With the assumption of ten million concurrent clients, our scheme only required one percent of total memory space.

A. Latency overhead

The experiment is performed on RAMCloud cluster, whose configuration is in table I. Latency comparison between normal operation and linearizable operation is shown in table II. There are

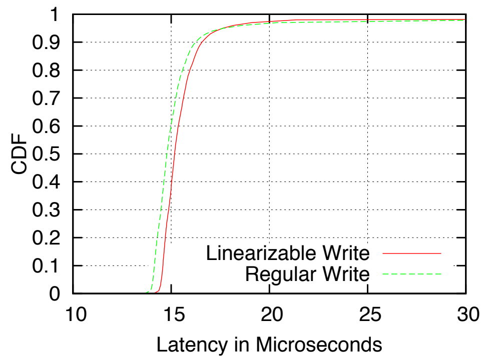

Fig. 4: CDF of latency of write operation

CPU	Xeon X3470 (4x2.93GHz cores, 3.6GHz Turbo)
RAM	24 GB DDR3 at 800 MHz
Disks	2 Crucial M4 SSDs CT128M4SSD2 (128GB)
NIC	Mellanox ConnectX-2 Infiniband HCA
Switches	Mellanox MSX6036 and Infiniscale IV

TABLE I: The hardware configuration of the 80-node cluster used for benchmarking. All nodes ran Linux 2.6.32 and were connected to a two-level Infiniband fabric with full bisection bandwidth.

	regular		**linearizable**	
Object Size	**Median**	**99%**	**Median**	**99%**
100 B	15.2	77.0	15.4	89.7
1000 B	19.2	75.4	19.3	87.4
10000 B	35.1	120.1	35.1	118.2
100000 B	225.5	405.7	224.6	481.6
1000000 B	2200	2500	2200	2500

TABLE II: Latency (in microseconds) of write operation with regular RPC and linearizable RPC: there are almost no latency penalty.

almost no penalty for using linearizable RPC instead of using regular RPC. The latencies of write, conditionalWrite, and increment are almost same, and only result of write is shown in this paper. The CDF of latency distribution is in figure 4.

B. Storage overhead

We also calculated storage burden caused by client state table (called UnackedRpcResult) and in-memory log entries. Thanks to the garbage collection by by ackID as described in section V-A, a master server keeps only one rpc record per client on average. Each entry in in-memory table, UnackedRpcResults, costs 100B, and a rpc log entry occupies 70B. To support 10M clients, each master server needs 1.7GB of memory. Considering the target host machine of RAMCloud has 256GB of DRAM, it is only 0.7 percent of total DRAM. With the assumption of commodity machine with 16GB of DRAM, we can still support 1 million concurrent clients with 1 percent of space overhead.

A client may reconnect after disconnection, which invokes new client ID assignment. To mitigate the gabage from disconnected client, we should have short lease for client ID as in section V-B. Current coordinator machine can accomodate 1M request per second. With the assumption of 1 percent of the coordinator's capacity is for lease renewal, 10M clients allow 1000 second lease expiration period.

VII.Conclusion

Non-linearizable RPCs may endanger the correctness of the whole system. The non-linearizable nature of the system stems from our inability to distinguish between serviced and not-serviced RPCs. The solution, therefore, is to maintain enough information so that it is possible to distinguish between serviced and unserviced RPCs. Implementing linearizable RPCs required additional operations such as durable log append and hash table lookup. Our implementation on top of RAMCloud showed that adoption of linearizable RPC brings almost no penalty on performance in terms of RPC latency and DRAM space consumption.

Having a linearizable RPC system hides many failure modes and simplifies distributed programing. Its implementation first seemed straight forward, but building a practical system was difficult because of crash recovery and cluster-level garbage collection. The garbage collection was very difficult to get correct while limiting performance impact, and we ended up with a cluster-level timeout mechanism.

VIII. Acknowledgments

Collin Lee helped for design and implementation, especially for cluster-level lease mechanism.

References

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[2] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[3] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[4] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992.

[5] S. M. Rumble, A. Kejriwal, and J. K. Ousterhout. Log-structured memory for dram-based storage.