RTable: Synchronous Azure Table replication for disaster tolerance

Parveen Patel, Ashwin Murthy, Jim Johnson, Tony Chan, Meetu Gupta

Microsoft

# **Summary**

Azure table storage provides a highly scalable NoSQL storage to store data as rows in a table. It allows queries in a table based on partition and primary keys. It is a highly popular service among Azure customers. As of Jan 2015, it stores 40 trillion objects with 4.5 million transactions per second. The service high durability by replicating data within and across regions. To maintain high throughput and low latency, data is replicated synchronously within a region, and asynchronously across regions. Azure storage provides 99.9% uptime SLA.

Due to asynchronous cross-region replication in Azure storage, services that need strong consistency of their data are not able to tolerate regional outages of storage. In other words, if storage in a region is unavailable (due to network, storage or other issues) these services are unable to read or write data.

To address this problem, we provide a design of RTable that synchronously replicates Azure Table across regions to provide the following guarantees despite regional outages: (1) high data durability by preventing data loss as long as at least one region is available, (2) high availability by providing consistent and quick recovery from failures without giving up on durability guarantee. These guarantees enable immediate failover to a secondary storage region while ensuring zero data loss and strong consistency.

# **Design constraints and assumptions**

We would like to meet the following design constraints while providing the above durability and availability guarantees:

1. **Replication cost:** We would like to keep the replication cost (storage and networking) low. Specifically, we would like to use protocols that tolerates *t* data center failures with as few replicas as possible. To meet the durability guarantees, we know that we cannot use fewer than *t+1* replicas. We treat replication cost (not to exceed *t+1 replicas*) and durability guarantee (no data loss despite *t* failures) as primary constraints in designing our protocol.
2. **Compatibility:** Many existing tools (e.g., analytical and monitoring) use Azure Table interfaces to read data from a single replica. Most such services can run unmodified and uninterrupted even after we replicate data. RTable protocol will ensure that such queries are reading consistent data even under network partitions.
3. **Client library:** We would like to build replication on top of unmodified Azure Table storage. Specifically, the replication protocol should be run by a stateless client library assuming only a passive Azure Table storage system. Hence, our system should be robust to client failures.
4. **Monitoring and configuration service:** Any replication protocol requires a service where the information on current view of the system (set of replicas) is maintained. For our very first code iteration, this can be done by human operators, who change the replica membership on failures. We intend to build this service into the client library (using a fault-tolerant Azure Storage-based leader election protocol, which can tolerate a single Azure Storage stamp failure) that will automate the monitoring and configuration process.

## **RTable API**

Our goal is to support all Azure Table calls in RTable API so that client applications have to make minimal changes. In this initial draft, we just describe how RTable implements *Retrieve* and *InsertOrReplace* calls to read and update rows in replicated table.

## **RTable uses chain replication protocol**

Broadly speaking, chain replication (sequential protocols) and quorum-based replication (parallel protocols) are the two main techniques for synchronously replicating data. The main reason for choosing chain replication is its low replication cost (t+1 replicas can tolerate t failures), low read overheads (read from a single replica), better load balancing of reads (read from any replica), and its simplicity (simpler recovery mechanisms on failures). We discuss the tradeoffs and the design rationale for choosing chain replication in RTable in detail later in this document.

Client applications read and write to RTable rows using the RTable *Retrieve* and *InsertOrReplace* API calls. Internally, RTable library synchronously replicates data across multiple data centers to tolerate individual Azure Table failures.

### RTable library.

The replication protocol is completely implemented inside the client library on top of Azure Table API. When a client application calls RTable library to read/write, the RTable library interacts with individual Azure Table replicas underneath using existing Azure Table APIs. RTable uses chain replication where replicas are arranged in an ordered chain so that locks are acquired and released in order along with writing and committing data. The first replica is called the *head* replica, the last replica is called the *tail* replica and others are called *middle* replicas.

RTable library gets a current set of replicas from a configuration service along with a lease time during which the set of replicas cannot change. RTable updates the replica set (or *view*) periodically, after the expiry of lease interval. The configuration service is used to update the *view* (set of replicas) at the end of current lease interval if individual Azure Table replicas are not reachable and are deemed as failed replicas, and start providing the new *view* in the next lease period. When there are Azure Table failures, clients observe timeouts and failures for writes until they get a new *view* once the lease expires. For reads, client can fetch data from any replica as explained below if there are no conflicting writes in progress or fetch from the *tail* when there are conflicting writes in progress. Reads are only blocked if there is a *tail* node failure.

### Protocol state

RTable maintains per-row protocol meta-data to recover rows correctly from client and data center failures. RTable adds three properties (columns) to entities (rows) to maintain its per-row replication state: Version (V), lock bit(L), lock acquisition time (Ltime), view id (Vid) as shown in Figure 1.

Figure 1: RTable replicas with protocol state



*LTime* is wall clock time set when a client acquires the lock. RTable uses LTime to detect incomplete operations (due to client failures) so that other clients (or a periodic cleanup process) can complete them and unlock this row. RTable uses the *Version* number to keep track of the latest version of a row. RTable also uses version number as a virtual *etag* for the replicated row when it performs read-modify write operations on rows (such as Replace) given that underlying replicas may have different physical etags for a row. *View id* is an index supplied by the configuration service, which internally maps the index to a set of replicas that were active when a row was modified. RTable uses this information to keep track of the chain membership, reconcile the replicated state of surviving replicas on replica failures, and bring a reinstated replica back to a consistent and up-to-date state when it is added back to the chain.

### Write protocol

When a client updates or inserts a row, RTable uses chain replication protocol to synchronously replicate the new data across all replicas before sending the response back to the client. RTable interacts with replicas sequentially in an ordered chain from the *head* replica to the *tail* replica during the execution of two phase protocol as explained below.

**Assumptions:**

1. View and version numbers are monotonically increasing and don’t wrap around.
2. Any new replica is always introduced at the head with a view change.

***1st Phase (Prepare and lock phase)****.* When an application modifies a row in RTable, the client library first reads that row at the *head*. If the view number it reads at the head is same as its current view, it attempts to atomically set the lock bit at the head first. This can be done using Azure Table Replace() call that provides read-modify-write semantics using etags.

If the view at the head replica is higher than the client’s view, it refreshes its view and reattempts to acquire the lock at the head in the new view.

If a client cannot acquire the lock – either because the lock bit is already set or setting the lock bit fails with a conflict exception – it backs off and waits until the current lock owner releases the lock or LTime expires for the current lock, at which point the client finishes any unfinished update left over by a faulty client before acquiring the lock.

The *head* replica acts as a synchronization point so that only one client can proceed with a write when there are concurrent updates to the same row from multiple clients. However, updates to different rows can happen in parallel as there is a separate lock bit per row.

If and when a client acquires the lock at the *head*, RTable also overwrites data (and its version) in place along with the lock bit atomically as it is part of the same entity. RTable then acquires locks and updates data at other Azure Table replicas sequentially in the chain until it reaches the *tail*. The locking phase ends when locks are acquired at all replicas. Figure 2 shows the state of replicas at the end of this phase.

If a client fails to acquire a lock at a replica other than the *head* due to a conflict exception, this must be due to a view change and the current replica has become the head in the new view. The client refreshes its view and follows the protocol described above for writing to the new head.



Figure RTable replicas after 1st phase (prepare and lock)

***2nd Phase (Commit and Unlock).*** RTable starts the second phase by unlocking and committing data at the *tail* replica first. It then continues by unlocking and committing at every replica sequentially in the reverse path of the chain, starting at the *tail* towards the *head*. It then returns success to the client application for the write operation. Figure 3 shows the state of table replicas after the commit phase.

Figure 3: RTable replicas after second phase (commit and unlock)



**Optimization**: Given that the last step of the 1st phase and the first step of the 2nd phase happen at the tail node consecutively, we can skip the prepare and lock step at the tail node by directly committing the data at the tail node in a single step, and starting the 2nd phase from the predecessor of the tail node.

**Tail replica has the authoritative copy.** Note that the tail replica atomically commits the data first before any other replica. Hence, it acts as an authoritative copy from which clients can read the latest committed data and provide strong consistency (linearizability for single row reads/writes).

### Read protocol

When a client reads data from RTable, RTable can always fetch data from the tail node. Since all writes are committed on the tail first, it is guaranteed to have the latest committed data.

**Optimization**: RTable uses the lock bit to implement a read optimization where clients can read from any replica instead of just the tail replica. If the lock bit is not set, RTable library returns the data. If the lock bit is set, it discards the result and fetches the data from the tail replica.

Note that our protocol is very similar to CRAQ [1], a variation of the original chain replication protocol [2]. However, our protocol runs on the client side and takes care of client failures whereas CRAQ protocol is implemented on the server side. These features make it unique and more generally-applicable compared to the existing chain replication protocols.

## **Handling** **client failures.**

RTable recovers from client failures using the protocol meta-data that we store in a row. We do so by running a recovery operation completely on the client side without requiring any support from external services.

1. **Read operations.** Client failures in the middle of a read operation does not require any clean up as reads do not change any state in the system. Clients retry on failed reads and they are guaranteed to get consistent data when they succeed.

Also, note that reads are not blocked by incomplete write operations due to faulty clients because the tail node collates the phase 1 and phase 2 operations into a single, atomic Azure Table write operation. That is, RTable does not set the lock bit at the tail node. A client can always read from the tail node independent of the status of other concurrent write operations.

1. **Write operations.** Clients can fail in the middle of a write operation and leave the system in an inconsistent state across replicas. However, our recovery process ensures correctness by allowing other clients to finish incomplete write operations left over by faulty clients. Here are the various failure scenarios and how we handle them correctly.

If a client fails before it acquires the lock on head, we do not have to do anything as the operation has not started from replicas’ perspective.

If a client fails after acquiring the lock but before the success is returned to the application. There are three cases. First, it can fail before the end of first phase where it acquires a lock on the head node but a lock is not acquired on other replicas. (Note that a lock bit is never set on the tail replica.) In such a case, other clients cannot write to the row until Ltime (lock expiration time) for the row expires. Until then writes to the row fail. After the expiration of Ltime, other clients clean up the row on behalf of the faulty client by completing the incomplete operation. They proceed by starting the write operation from where the faulty client left off, acquire locks in the first phase, commit and release the locks in the second phase. As locks and data are updated atomically at the head replica, the cleanup process has all the data required to complete the operation by reading the head replica. After the cleanup is done, clients can continue to write to the same row. Second, a client can fail in the second phase before the head node unlocks and commits the data (atomic operation). In this case, the data was at least committed at the tail node, which may have been read by some other client(s). We have to make sure that this data is committed at other replicas eventually to ensure consistency.

Given that the head node has not released the lock, no other client writes to the row until a client runs the cleanup process to release the locks first. This means that eventually a client that is trying to acquire the lock, completes the operation by running the second phase and committing the operation before releasing the lock at the head replica. Third, a client can fail after the end of second phase but before the client receives the acknowledgement. In this case, the operation has been completed at all replicas and there is nothing to be done. Other clients are not blocked because the lock on head replica is released.

Therefore, in all cases, we recover from client failures correctly by always completing incomplete write operations, eventually, by other correct clients so that a client never sees data that is revoked later. Once an update reaches the head, it is never lost due to client failures.

## **Handling** **data center outages**

When a data center is not reachable due to network partitions or geographical disasters, RTable ensures consistent recovery. We do so by changing the *view* in the configuration service, which removes the faulty replica (data center) from the set of replicas. Clients read the current view from the configuration service, cache it locally, and update their local copy periodically when the view lease time expires.

**Read availability.** Read operations are not affected by non-tail replica failures as tail replica can continue to service read requests despite failures. However, when a tail replica fails, other replicas can still provide data if the lock bit is not set for the row being accessed because other replicas can respond after they commit and unlock their local bits. However, if the lock bit is set at every available replica then read operations to a row fails until the view is changed and the row write is completed using the recover protocol mentioned above.

**Write availability.** When a data center fails, writes fail until the view is changed and the faulty replica is removed from the chain. Hence, write availability depends on how quickly the monitoring and reconfiguring service detects and repairs the chain by replacing the faulty replica (to run with the same durability guarantees) or by removing the faulty replica (to run at lower durability with fewer replicas until the faulty replica is replaced). Unfortunately, it is impossible to get around this constraint while meeting the durability guarantee of tolerating *t* data center failures using only *t+1* replicas. In a later section, we discuss future extensions to provide better write availability by paying additional replication cost.

## **Introducing new replicas or reintroducing partitioned replicas**

RTable allows introduction of new replicas or reintroduction of old partitioned replicas into the system. However, both new and old replicas do not have data that is consistent with existing replicas in the chain. It is important that these replicas are (re)introduced carefully without affecting consistency.

The configuration service uses an active recovery mechanism to introduce new replicas or reintroduce existing replicas into the chain without stalling reads or writes. It uses the notions of *read-view* and *write-view* to achieve this. A *read-view* has only the existing replicas of chain and provide consistent view for reads. *Write-view* has both existing replicas and the new replicas prefixed to the *read-view* chain so that new writes go to both set of replicas. Client applications use *read-view* to read data and *write-view* to write data. During a write (during lock or commit phase; and before returning write success to a client), if a client detects that RTable read-view and write-view are different, it will first invoke the recovery protocol described in Figure 4 before continuing with its write. The client only needs to invoke the recovery protocol for a row once, so it can continue writing to the same row without invoking recovery again.

The configuration service fires a recovery agent to asynchronously copy data from the read-view to the new replicas by iterating over rows in the head of the read-view and following the protocol described below.



Figure : protocol to recover a row

After the recovery client is done iterating over the rows, the new replicas are up to date. Any writes that happened while the recovery was in progress, have already been written to the new replicas. Tail replica continues to hold the authoritative copy.

The recovery client returns success to the configuration service, which in turn changes the read-view and write-view to include all replicas. Clients get the new read-view and write-view with all replicas included in them. This change to the view does not need to happen atomically and is propagate to all clients eventually. The protocol still ensures correctness if some clients are operating in the old view while the others are operating in the new view since they all start writes from the write-view head and read from the same tail.

Multiple view changes can take place before a previous view becomes stable, i.e., the read-view and the write-view have the same replicas. The protocol handles this by following the above protocol for the chain that is prefixed to the read-view.

**Note on Insert operation**: For *insert* operation, the client first creates a tombstone entry (without any data) for the row with the lock bit set on the head and then subsequent replicas. It then inserts the row at the head first and then at the other replicas along the chain. This prevents a race condition between the client and the recovery agent due to lack of timeout (sunset) on Azure tables. The following scenario could occur:

1. At time T1, there is only replica R1 in view v1. R1 is both the head and the tail.
2. A client C1 issues an insert for row K1 on R1.
3. View changes to v2 and replica R2 is introduced.
4. The recovery agent finds that K1 does not exist on R1 and declares that R2 is in sync with R1.
5. View changes to v3.
6. Insert on R1 from step #2 above completes on R1.

If the tombstone entry is not inserted first then the recovery agent might conclude that the row does not exist while the insert might complete at a later time on the tail. This will leave the tail ahead of the other replicas, which violates our chain protocol. This is fixed by forcing client C1 to insert a tombstone entry for K1 first before the real row is written. The tombstone will avoid the race condition between the original client and the recovery agent.

**Optimization**: if the replica being introduced has updates up to a certain view id (M), then the recovery client can do incremental updates by skipping any rows that were updated in views < M-1. This does not handle deletes as those entries will not be present in the read-view. In order to ensure faster recovery, it may be worthwhile to create a new table to keep track of the all the deleted entries while RTable is running with low replica count. This will ensure that any deletes that happened while the replica being re-introduced had an outage can be quickly deleted on the replica before stabilizing the view.

# **RTable: design tradeoffs, rationale, optimizations, and future work**

Broadly, there are two approaches to replicate data synchronously with different tradeoffs: chain replication (sequential) and quorum-based replication (parallel) protocols. Here we discuss the tradeoffs, explain our initial assumptions and the metrics that we are optimizing for, to justify our choice of using chain replication protocol in the initial RTable design.

1. **Simplicity**: RTable’s chain replication protocol has a rather simple recovery protocol. To recover a row, the recovery process picks up a locked row from the head and proceeds through the chain using the two phase chain replication protocol. There is never a need to rollback an incomplete write. This makes it possible to implement the recovery protocol entirely in client library as part of regular read and write operations. The same protocol is used to bring new replicas into rotation without blocking reads.

*Why not parallel protocols?* Quorum-based protocols, including Paxos, are harder to implement entirely in the client. Furthermore, they require a different read protocol, which means existing clients cannot work as is. Paxos has an additional disadvantage of serializing all requests and generating a global order whereas RTable protocol executes independent requests in parallel by maintain fine-grained locks.

1. **Minimal replication cost:** Chain replication keeps the replication cost low and meets the lower bound of *t+1* replicas. Note that to tolerate t simultaneous data center failures we need to replicate data at least to *t+1* data centers. In addition, RTable does not require any write-ahead log or maintaining multiple versions of data.

*Why not parallel protocols?* Majority-based quorum systems require 2t+1 replicas to tolerate t failures.  Our chain replication protocol can be seen as a specific type of ROWA (Read-one-write-all) quorum system. In the future, we will explore other protocols by relaxing the cost constraints if applications are fine with higher replication cost.

1. **Read availability and latency:**  Chaining provides significant advantages to read-heavy workloads compared to quorum-based systems.
   1. Low overhead reads: In RTable, clients usually reads from a single replica (or at most two) with fewer bytes transferred in the network. Majority-based quorum requests have to read from t+1 replicas requiring more network bandwidth.
   2. Load balancing: RTable provides better load balancing for reads than majority quorums as clients can read from any replica when there are no concurrent updates (details below).
   3. Compatibility: Existing tools that read from un-replicated Azure Tables continue to work unmodified; they can just read from the tail replica, which always has the latest data.
   4. Read availability: In RTable, reads are non-blocking when any or all of the *t* non-tail nodes fail. Reads, however, block when there is a tail node failure until the fault is detected, the faulty tail node is ejected from the chain (view change) and another live replica (predecessor) is chosen a new tail replica. On the contrary, majority quorum systems provide better read availability as they do not block on any node failure up to *t* failures out of *2t+1* replicas.
2. **Writes:**  Chain replication trades the above advantages with the following compromises for write availability and latency.
   1. Write latency: Writes to replicas proceed sequentially in a chain leading to higher latency (end-to-end latency = f(sum of latencies to replicas)) compared to quorum-based systems which exchange messages in parallel (end-to-end latency = f(max latency of all replicas)).

Latency in RTable can be reduced by writing to non-head and non-tail nodes concurrently, but it slightly complicates the recovery mechanism. We will explore this option in future.

* 1. Availability: Writes are blocked when a replica fails until the chain is reconfigured by removing it. Note that it is impossible to get around this drawback using *t+1* replicas as writes cannot return success to the application without writing to all of them to provide durability despite *t* failures.

We will explore extending our protocol to use quorum based approaches for writing to the middle nodes and provide better availability by paying additional replication cost.

# Configuration service

The configuration service is responsible for storing and updating the current view (chain) of the system while ensuring safety. Configuration service consists of (a) a highly-available configuration store (for example, using replicated blob store) to store the current configuration and (b) a configuration agent that is responsible for failure detection and reconfiguration of the system to ensure progress. RTable clients learn the current configuration by reading the configuration store, which is initialized and updated by the configuration agent(s). Figure 4 shows the architecture of the configuration service.

Configuration Agent

RTable client

RTable client

Configuration Store (replicated)

Figure : Configuration service

The configuration service uses a leasing mechanism to allow RTable clients to read the configuration state and cache it for the lease duration without compromising safety under the assumptions stated below. Thus, we improve (a) the latency of RTable read or update operations and (b) provide good scalability with the number of clients by avoiding reading the configuration store on every read or update operation.

## Assumptions

The configuration service design makes the following assumptions:

1. Configuration agent’s clock does not advance faster than RTable clients’ clock more than a known constant bound, called clock factor (CF). This is similar to the Chubby system [3].
2. No new operation is issued to Azure table after the lease time has expired. Note that the operation may finish after the lease has expired since we do not control the server side.

## Data stored in the configuration store

The configuration store has the chain replica set, an associated lease duration and a version number. Table 1 shows the logical data structure stored in the configuration store.

Table : Data stored in configuration store

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Service Name** | **Replica chain** | **Read head index** | **ViewId** | **Lease duration** | **Timestamp** |
| Svc1 | Svc1RTable-useast, 5; Svc1RTable-uswest, 1 | 0 | 6 | 60 | 00:00:00::01012014 (UTC) |
| Svc2 | Svc2RTable-asiawest, 7; Svc2RTable-asiaeast, 2 | 0 | 8 | 60 | 00:00:00::01012016 (UTC) |

*Replica chain* stores the ordered list of replicas. Write view always starts at index 0. Read view starts at replica index specified in *Read head index*. Each replica also has an associated view number during which that replica was introduced. This version number is used during repair as described in **Introducing new replicas or reintroducing partitioned replicas**. *ViewId* is incremented anytime there is a change to the row. For example, when introducing a replica back into the system, the ViewId is incremented and the read view and write view are different. When the write view has caught up to the read view, the ViewId is incremented again and the read-head-index is set to zero.

*Lease duration* is the time duration (in seconds) for which the chain configuration is valid from the time it is read. Every time a client reads the configuration, its lease is renewed for *Lease duration*. The client must not use a configuration for which the lease has expired. The client should account for transmission delays in the network as well.

## Client operation

Client will read the configuration and assume that the lease is valid for lease duration. The client will renew the lease, e.g., read the configuration from configuration store, at the following events:

1. The client will periodically renew the lease, ideally before it has expired. In order to ensure that the client makes at least two attempts at acquiring the lease before it expires, it is recommended to try the renewal every ((lease duration / 2) – 1) seconds.
2. Before starting a new read or write transaction, the client should check if the lease will remain valid when the transaction finishes (based on maximum transaction time). If the lease might expire before the transaction finishes, the client should start an asynchronous renewal request in parallel to the transaction. If the lease has expired when the transaction finishes, the client should discard the result and wait for lease renewal before retrying the transaction.

## Configuration agent operation

### Configuration agent selection

Any RTable or external client can serve as a configuration agent. At any given time only one configuration agent can update configuration. This is ensured via reliable leader election. Reliable leader election can be done using Azure blob leases, as described in [this](http://sharepoint/sites/CIS/Networking/Engineering/Hybrid/Brooklyn/Reliable%20gateway%20tenant%20leader%20election%20in%20presence%20of%20storage%20failures.docx) document.

### Failure detection

For now, we assume that failure of an RTable replica is detected manually. This can be easily automated once we build the configuration agent into RTable client. The configuration agent can actively and passively monitor all the storage accounts in a given chain and determine whether they are up or down. Further, each rtable client can send health reports to the configuration agent. This allows us to handle cases where the configuration agent finds the replicas healthy but some of the RTable clients may not.

Once a failure is detected, the configuration agent is used to reconfigure the chain appropriately. Similarly, when a failed replica is back online or a new replica is to be brought online, the configuration agent is used to add the new replica back to the pool. The configuration agent is also responsible for bringing the new replica up to speed following the protocol described earlier.

### Updating the replica set

Let’s assume that the maximum time clock factor time for lease duration L is CF. In the general case, the configuration is updated from version v1 to v2 as follows:

1. The configuration agent deletes the current configuration version v1 from the configuration store.
2. The configuration agent waits a constant time L + CF. This will ensure that no new transactions start in the old view. A transaction started in v1 may still complete in v2 but that is handled by the repair protocol.
3. The configuration agent writes the new configuration v2 to the configuration store.

Figure 5 depicts a flow chart of this algorithm.

Figure : Updating chain configuration

## Configuration store

Configuration store can be implemented using replicated blobs. We use majority quorum (2t+1 replicated blobs to tolerate t failures) to store configuration state with high availability. RTable clients do a quorum read on the configuration store to determine the current configuration. On a quorum read to the replicas, a client accepts the configuration state if it receives a majority of blobs with the same version number and the configuration state.

Writes to the configuration store are not complete until the state is written to a majority of the replicas. The configuration agent is responsible for ensuring that all replicas of the configuration store are in sync eventually. To deal with temporary unavailability of replicas, the agent will periodically read the configuration from all the replicas and update any replicas that have fallen behind. If it fails to update any replica for an extended period of time, e.g., 48 hours, it will raise a critical alert that needs to be attended to by an administrator.

## Rationale for separating the control path from the data path

We use RTable to replicate the application data and a different configuration service to replicate the configuration state. We separate the control path (using the configuration service) from the data path (using RTable) as they involve different cost-reliability-performance tradeoffs. Specifically, RTable uses t+1 replicas to store the application state while using the configuration service with 2t+1 replicas to store the control data (configuration state).

Note that configuration store can also be implemented using RSL like systems such as Zookeeper or WinFab or other Paxos based libraries.

Given that failures are uncommon, such a separation gives cost and performance benefits. RTable uses fewer replicas (t+1 as opposed to 2t+1 for any quorum based protocols including paxos or zookeeper) with improved read (just needs to read from a single replica) and recovery (any sub-chain has consistent state on failures) latencies; write latencies of RTable is comparable to any parallel quorum-based protocols for practical values of t (<=2) in most cases.

## Handling reconfigurations during in flight replication operations

Under certain failure scenarios, it is possible that a chain reconfiguration happens during an inflight write transaction. Further, since Azure Storage operations do not have a sunset time on them, a write request could be issued in one view but complete in a different view. The RTable write protocol deals with this scenario by refreshing the view at the end of any write. If a new replica has been added to the chain, it invokes the repair protocol. It then continues its write in the new chain.

The following invariants are claimed as part of the design of the write protocol:

1. Both version and view are monotonically increasing values. They don’t wrap around.
2. No committed data will be overwritten by concurrent Write (Serializability). This is guaranteed by the lock at the head.
3. Version at a replica >= version at replicas later in the chain.
4. Writes that reach the tail are always committed even in the presence of view changes.
5. If during the prepare phase, a client has acquired the lock on the head in the current view, then that write will always be eventually committed at the tail under the condition that at most one of the two – the head replica and the client doing the write fail simultaneously. This is possible because any client can flush incomplete writes.

The following invariants are claimed as part of the design of the read protocol:

1. Only committed data is returned from a read.

# Acknowledgements

This work benefited greatly from collaboration with Microsoft Research Silicon Valley. In particular, Rama Kotla, Doug Terry and Mahesh Balakrishnan made huge contributions to the initial design and implementation. Further, this work has improved due to discussions with various people and teams in Azure, including Mark Russinovich, Jeff Wilcox, Azure storage, Skype and AAD.

# References

|  |  |
| --- | --- |
| [1] | J. Terrace and M. J. Freedman, "Object storage on CRAQ: high-throughput chain replication for read-mostly workloads," in *USENIX Annual Technical Conference*, Berkeley, CA, 2009. |
| [2] | R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *OSDI'04 Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation* , 2004. |

[3] Mike Burrows, "The Chubby lock service for loosely coupled distributed systems", in *OSDI'06 Proceedings of the 7th conference on Symposium on Opearting Systems Design & Implementation* , 2006.