Synchronous Replication

Meta

- RFC Name: Synchronous Replication
- RFC ID: 0046
- Start date: 2020-12-03
- Owner: Sergey Avseyev
- Current Status: DRAFT

Summary

This RFC describes new interfaces to support server feature known as "Synchronous Replication".
The major improvement over previous implementation is that mutations are not visible to any other clients until they have met their durability requirements.

Motivation

Today Couchbase Server provides durability requirements built in the SDKs on top of the Observe operation for clients to check if a mutation has been replicated and/or persisted to a specified number of cluster nodes. This allows applications to determine when a write is durable (will still be present after some amount of failure).

However, the current OBSERVE operation-based API has a number of limitations which make it difficult / complex for users to write correct, safe applications using it:

- Mutations are visible to clients before they are durable. As mutations are visible on the active node as soon as the mutation has been processed locally (without waiting for the replica(s) to receive them), clients can read the "new" value before it has been replicated / persisted to disk. This can cause problems in failure scenarios, for example if an active node is failed over before a mutation was replicated to the newly promoted active; then the document is inconsistent - an earlier read (pre-failover, to the old active node) saw the new value, but a later read (post-failover, to the new active node) would see the old one.
- No Server-side concept of durability failing. Observe() is essentially just a "post-write" operation - the Set has already logically "happened" on the master node, Observe is just waiting for the replica to receive the mutation. As such, there's no server-side concept of durability failing (e.g. due to timeout, unavailability of replica, etc). This means that even if a client considered a Durable Write to have failed, the active node in the cluster may consider the write to have happened, and the client is responsible for handling this - perhaps undoing the operation.

The current observe semantics makes it difficult for applications which require durability to be written correctly - they must either avoid reading a document until it is durable (challenging if the application consists of multiple threads / actors); or they must implement application-level logic to deal with observed writes which could later be "undone".
To address these problems, we propose to introduce a new variant of mutation - a Synchronously Replicated Mutation - or Sync Write. Such a write is not made visible to any observers until it has met its durability requirements.

Terminology

Synchronous Mutation - A mutation to a vBucket which is only made visible to clients once it has been successfully replicated/persisted to an operation-controlled number of nodes. A.k.a. Sync Write

Asynchronous Mutation - A mutation to a vBucket which is made visible to clients as soon as it has been processed by the active vBucket. All KV-Engine mutations currently (as of Vulcan) are asynchronous.

Prepare - The first phase of a Synchronous Mutation; where the "future" value of the Key is calculated and distributed to replica nodes. See also: Commit, Abort.

Commit - The second phase of a Synchronous Mutation when the mutation was successful - the value previously prepared is committed such that it is visible to clients. Once commit is complete the Sync Write has finished.

Abort - The second phase of a Synchronous Mutation when the mutation was unsuccessful - the value previously prepared is logically discarded; and on-disk state is updated as necessary to undo any state changes by the Sync Write.

Rollback - The act of discarding the last N modifications to a vBucket to essentially rewind time to some previous point. Currently used by KV-Engine in the event of a failover when nReplicas > 1, or in delta-node recovery; in the situation where a replica node ends up "ahead" of the current active and hence must discard it's alternative "future" timeline.

Protocol Changes

New HELLO flags

- **ALT_REQUEST_SUPPORT (0x10)** - Must be sent to allow the server to accept requests with flexible frame extras, The request magic for new requests is 0x08.
- **SYNC_REPLICATION (0x11)** - Must be sent to be able to specify durability requirements for mutations.

New Status Codes

- **DURABILITY_INVALID_LEVEL (0xa0)** - Invalid request. Returned if an invalid durability level is specified
- **DURABILITY_IMPOSSIBLE (0xa1)** - Valid request, but given durability requirements are impossible to achieve - because insufficient configured replicas are connected. Assuming level=majority and C=number of configured nodes, durability becomes impossible if floor((C + 1) / 2) nodes or greater are offline.
- **SYNC_WRITE_IN_PROGRESS (0xa2)** - Returned if an attempt is made to mutate a key which already has a SyncWrite pending. Transient, the client would typically retry (possibly with backoff). Similar to ELOCKED.
- **SYNC_WRITE_AMBIGUOUS (0xa3)** - The SyncWrite request has not completed in the specified time and has ambiguous result - it may Succeed or Fail; but the final value is not yet known.

Packet Changes

Durability requirements might be optionally specified with the following operations:

- Set (0x1)
- Add (0x2)
- Replace (0x3)
- Delete (0x4)
- Increment (0x5)
- Decrement (0x6)
- Append (0xe)
- Prepend (0xf)
- SubdocDictAdd (0xc7)
- SubdocDictUpsert (0xc8)
- SubdocDelete (0xc9)
- SubdocReplace (0xca)
- SubdocArrayPushLast (0xcb)
- SubdocArrayPushFirst (0xcc)
- SubdocArrayInsert (0xcd)
- SubdocArrayAddUnique (0xce)
- SubdocCounter (0xcf)
- SubdocMultiMutation (0xd1)

In order to pass durability requirements, they have to be encoded as flexible framing extras, and magic of the packet set to 0x08.

The ID of the flexible frame extra is 0x01. The length might be 1 or 3.

Level - the first and mandatory byte of the frame. It might take one of the following values:

- **MAJORITY (0x01)** - Mutation must be replicated to (i.e. held in memory of that node) a majority ((configured_nodes / 2) + 1) of the configured nodes of the bucket.
- **MAJORITY_AND_PERSIST_TO_ACTIVE (0x02)** - As MAJORITY, but additionally persisted to active node.
- **PERSIST_TO_MAJORITY (0x03)** - Mutation must be persisted to (i.e. written and fsync'd to disk) a majority of the configured nodes of the bucket.

Any of other value of level is invalid and results in the request failing with DURABILITY_INVALID_LEVEL(0xa0). For level=MAJORITY, if zero replicas are configured then the request will fail with DURABILITY_IMPOSSIBLE(0xa1) error. Configured nodes is defined as the number of replicas + 1 (for the active node) at the point the SyncWrite was processed by the active node.

Timeout - the second and third optional bytes represent 16-bit unsigned integer, which contains time in milliseconds the durability must be met within, otherwise the mutation will be aborted. If not specified, then a server-defined default is used.

Early-scheduled mutations behaviour

The SDK must use bucket capability to determine if Synchronous Replication supported by server. If it is not available API operations, which explicitly use Durability Level, have to fail with error.

Ambiguous Response

There will always be situations where the client cannot know if the request succeeded or failed, and hence the result is ambiguous. The simplest example of this is when the SyncWrite was successful on the server-side and it sent a SUCCESS response to client, but the client disconnects just before receiving it. Without additional communication to the server, the client cannot determine if the request succeeded or failed - and hence sees a "result is ambiguous" (SYNC_WRITE_AMBIGUOUS) response from the SDK.

If the client receives this response and the SyncWrite is idempotent, it can simply retry the request. If the SyncWrite is not idempotent, the application needs to determine what is the least-worst option - applying the SyncWrite twice (and hence retry it), or applying the SyncWrite never (don’t retry). The client could also read the current status of the document, if that allows them to figure of if their SyncWrite happened - however there could well have been subsequent writes since.

API Details

Synchronous Replication Timeout Adaptive Algorithm

Take time value from SDK timeout specified by the app developer, make deadline 90% of SDK timeout or 1500ms whichever is greater. If SDK operation timeout (configured by application, and specified with mutation API) is lower than 1500ms, coerce it to 1500ms and log a warning. The only variable in the API is timeout from which a deadline is derived in the network request. Environment: **persistence_timeout_floor** defaults to 1500ms, attempts to go below this will fast fail with an error.

According to server team, they don't have any plans right now to have ceiling for durability timeout which would be less than uint16_t maximum value.

Language Specifics

C

```
/* TBD */
```

Go

```
// TBD
```

Java

```
/* TBD */
```

.NET

```
public enum DurabilityLevel
{
        None,
        Majority = 0x01,
        MajorityAndPersistActive = 0x02,
        PersistToMajority = 0x03

}

var result = collection.Replace(
    “user:4123”,
    new {age = 45, arms = 1},
    options => options.DurabilityLevel = DurabilityLevel.Majority
```

```
);
```

NodeJS

```
// TBD
```

PHP

```
class DurabilityLevel
{
    const MAJORITY = 0x01,
    const MAJORITY_AND_PERSIST_ON_MASTER = 0x02,
    const PERSIST_TO_MAJORITY = 0x03
}

$mutateResult = $collection->replace(
    'user:4123',
    ['age' => 45, 'arms' => 1],
    Options::replaceOptions()->withDurabilityLevel(DurabilityLevel::MAJORITY)
);
// The old style of durability
Options::replaceOptions()
        ->withDurability(PersistTo::TWO, Replicate::ONE)
```

Python

```
# TBD
```

Scala

```
sealed trait Durability
       case object Disabled extends Durability
       case class ClientVerified(replicateTo: ReplicateTo.Value, persistTo:
       PersistTo.Value) extends Durability
       case object ReplicateToMajority extends Durability
       case object PersistToMajority extends Durability

collection.replace(“id”, SomeData, durability = ReplicateToMajority);
collection.replace(“id”, SomeData, durability = ClientVerified(ReplicateTo.Two,
PersistTo.None));

// Error handling for idempotent op
def retryIdempotentOp(op: => Try[MutateResult, guard: Int = 3): Try[MutateResult] = {
   val result = op()
   result match {
      case Success(_) =>
      case Failure(err: RetryableOperationException) =>
        // All ops that an app can retry immediately inherit RetryableOperationException
```

```
        if (guard > 0) retryOp(op, guard - 1) else Failure(err)
      case Failure(err) => Failure(err)
   }
}

retryIdempotentOp(collection.replace("id", SomeData, durability = ReplicateToMajority))

// Error handling for non-idempotent op: very situation/app specific.  Suggestions welcome!

// Upgrade scenario, writing app for both server <6.0 and >=6.5
def durabilityForServer() = {
   if (serverVersion >= 6.5) ReplicateToMajority else ClientVerified(ReplicateTo.Two)
}
collection.replace("id", SomeData, durability = durabilityForServer())
```

Questions

- Should we be able to retrieve the server default timeout?
-

Open Items

- This RFC does not describe work of Synchronous Replicate when the SDK supports bidirectional communication (HELLO flag DUPLEX(0x0c)). Without this feature enabled, the SDK will wait until the server will meet durability requirements, timeout or return unambiguous code.
- This RFC does not cover any changes related to synchronous replication, which might affect other services inside SDKs.

Signoff

Language	Representative	Date
C		
Go		
Java		
.NET		

NodeJS		
PHP		
Python		
Scala		