Replication

Contents

- Why and what is replication
- Replica Set Lifecycle
- Replica Set Roles and Configuration
- Elections
- Oplogs
- Local database
- Write concern

Questions

1. why replication?
2. when to use replication or not?

Why Replication Exists?

- redundancy/extra copies of data (DR)
- failover (HA)
- reduce latency
- maintenance, upgrades (SW or HW)
- offload “special” reads

Using Replication for Scaling

- 2 special situations:
  - atypical read, aggregation/analytical query
    - Messing up the performance on the secondary does not have the same impact with collection scans if all important reads go to the primary
  - reduce latency
    - Needed data is based on the geography.
      For example, users data in each location, publishing, …
    - Improve round trip latency of a request.

Database High Availability

- RDBMS: replication across datacenters
- MongoDB: oplog replication scheme
  - asynchronous replication – v1.2
  - failover – v1.6
  - automatic recovery – v1.6(?)

  => Changes the definition of durability

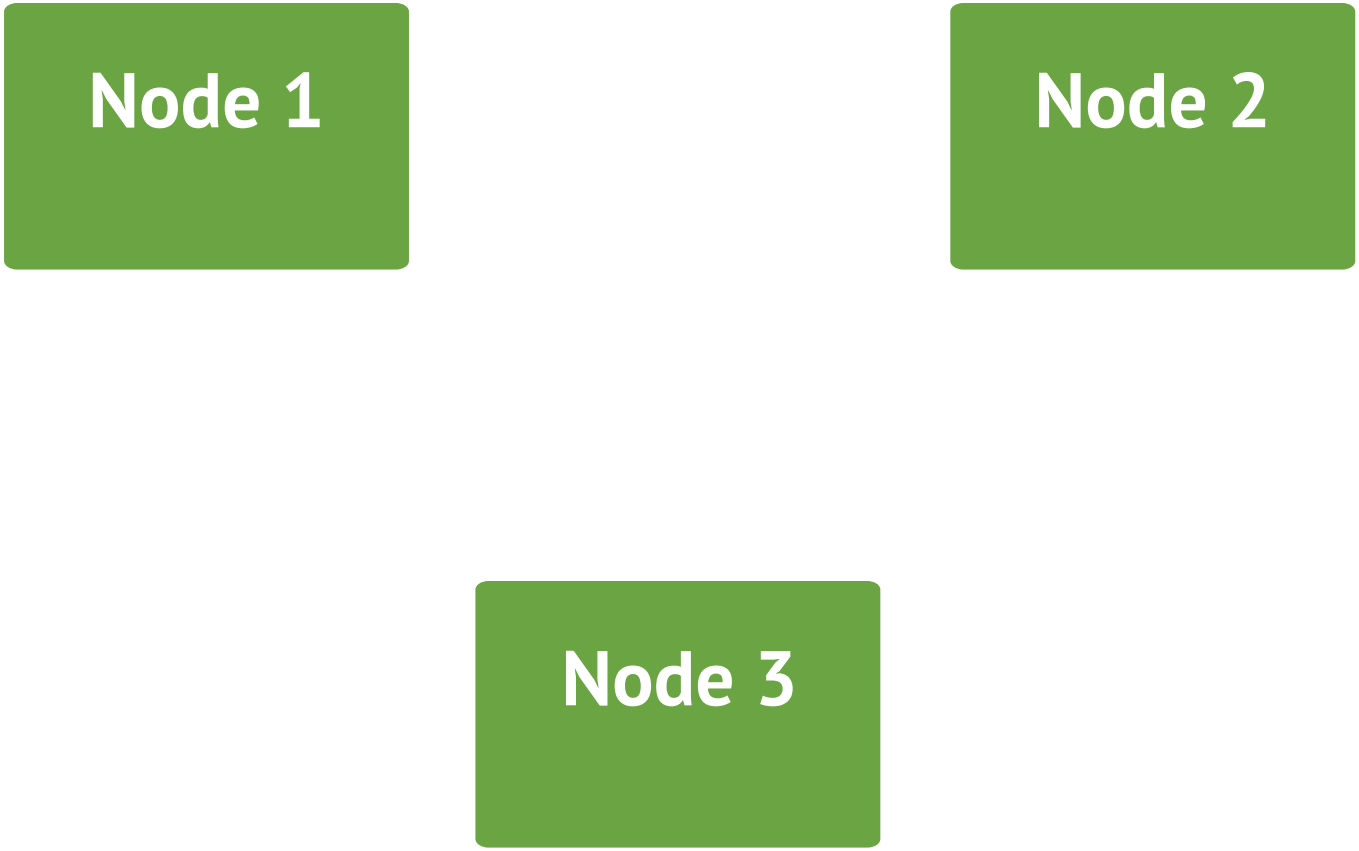
Mastership of data

- Single master
  - repeat writes on other nodes
  - disadvantages:
    - in case of failure, a new master must be elected
    - can't write when there are no primary
    - possibly long latency
- Multi masters
  - merge data or the last one wins or quorum
- Selective mastership
  - every piece of data has one "master" location

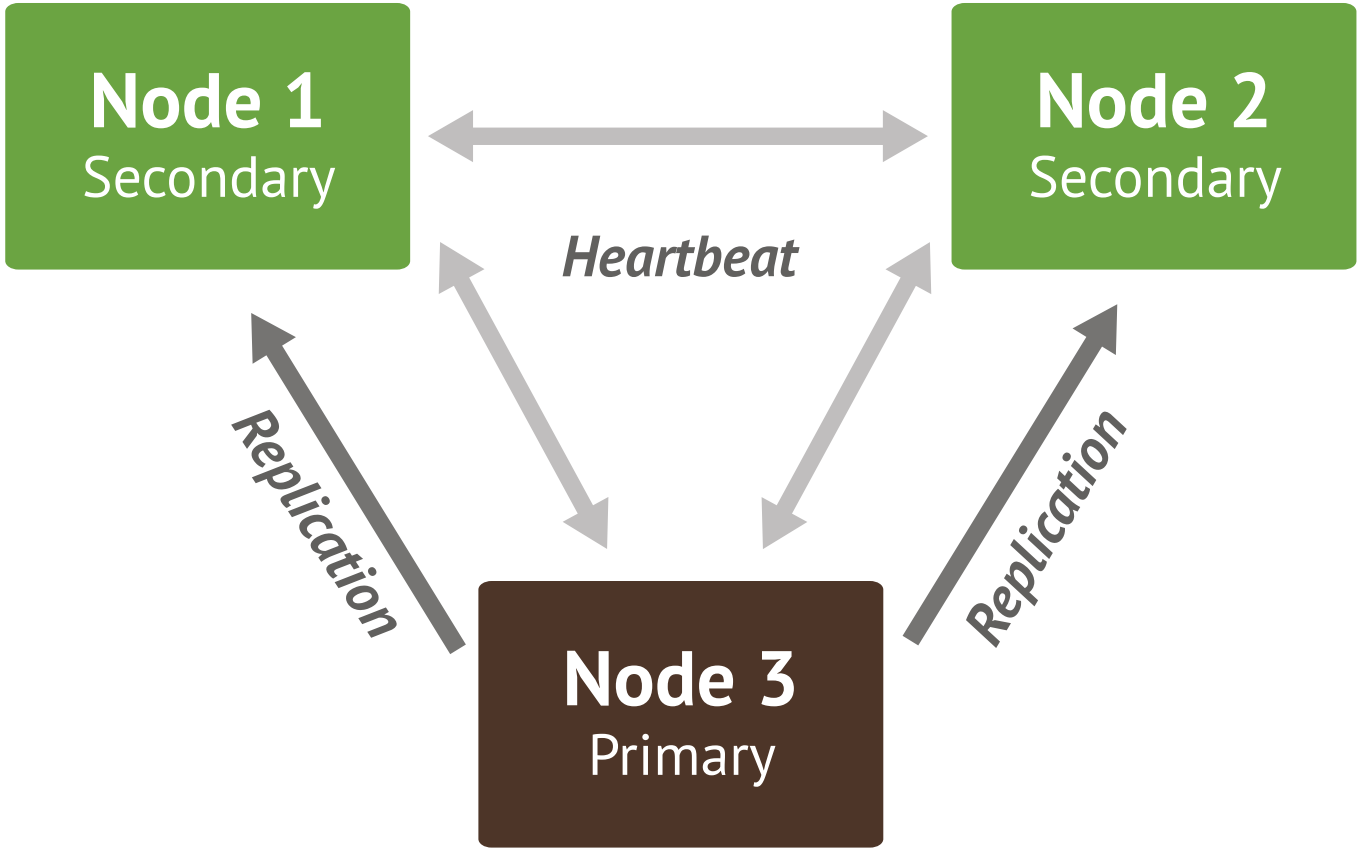
Replication protocol

- Version 0
  - Up to 3.2
- Version 1
  - Starting in 3.2
  - Based on RAFT
    - Differences: arbiters, priorities, operations pulled by secondaries (needed for chained secondaries)
    - Faster election, may not let the new Primary catch up

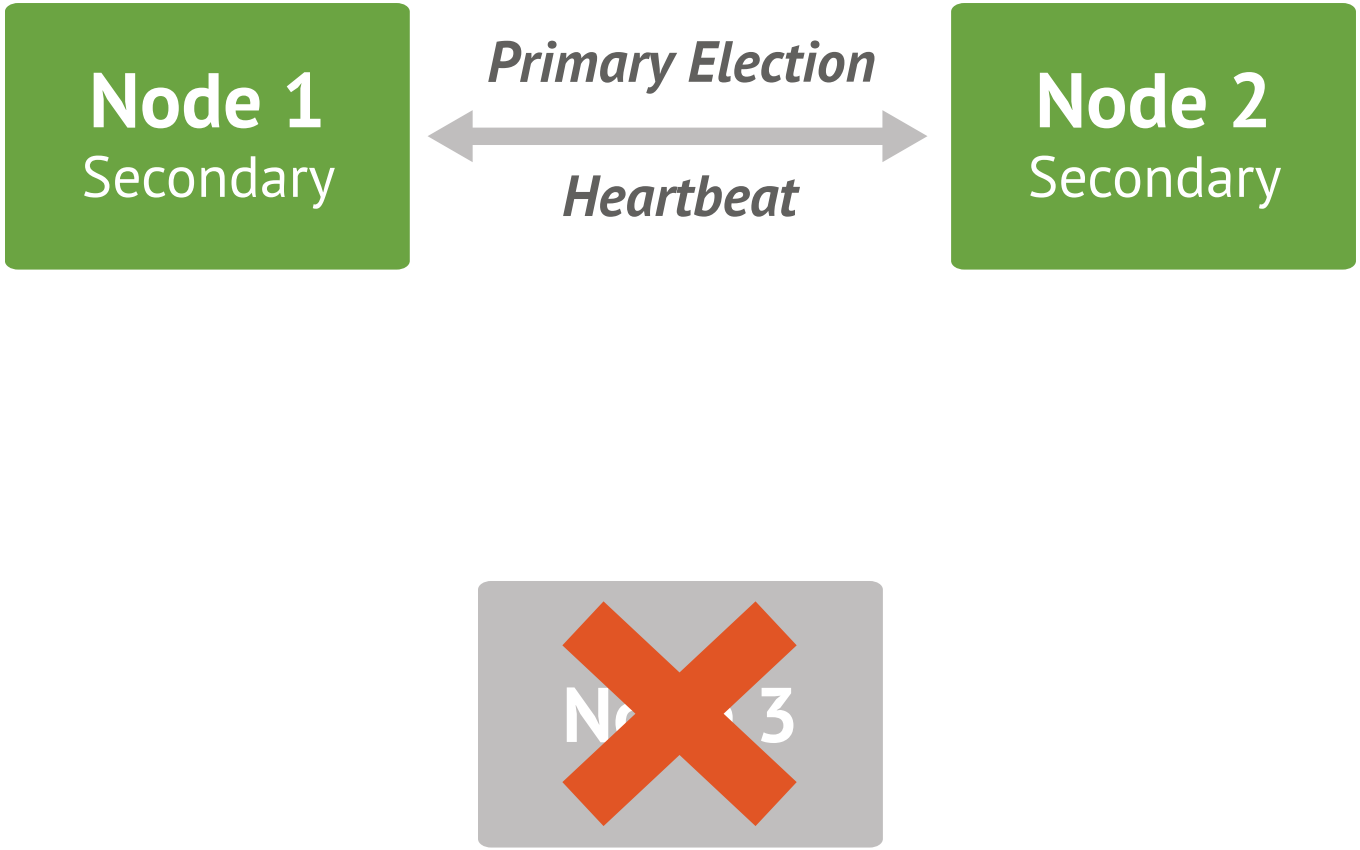
Replica Set Lifecycle - Creation



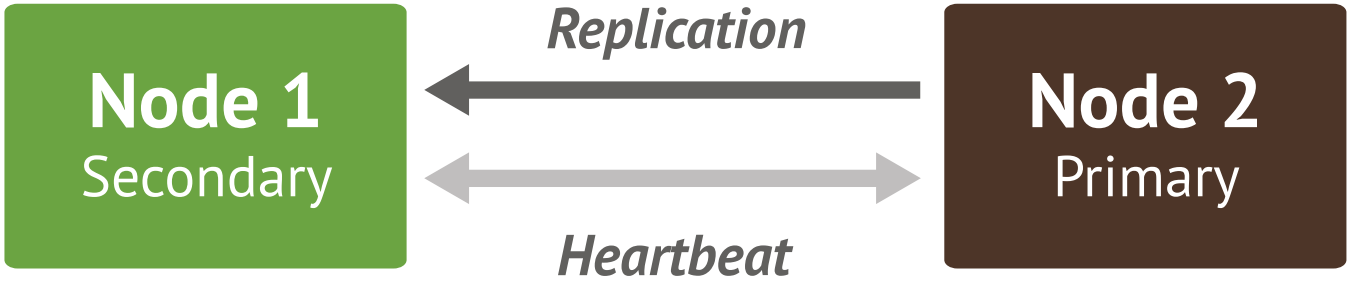
Replica Set Lifecycle - Initialize



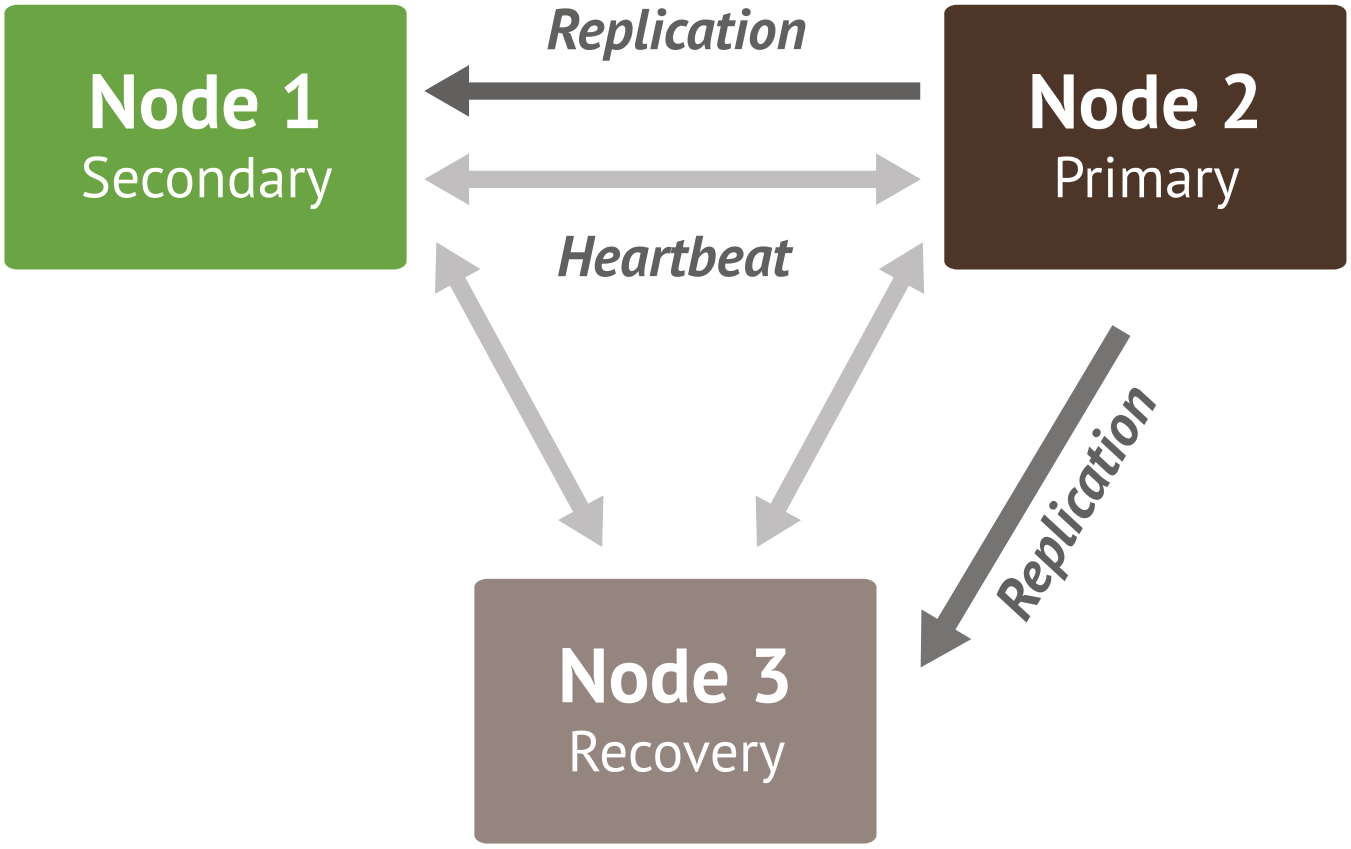
Replica Set Lifecycle - Failure



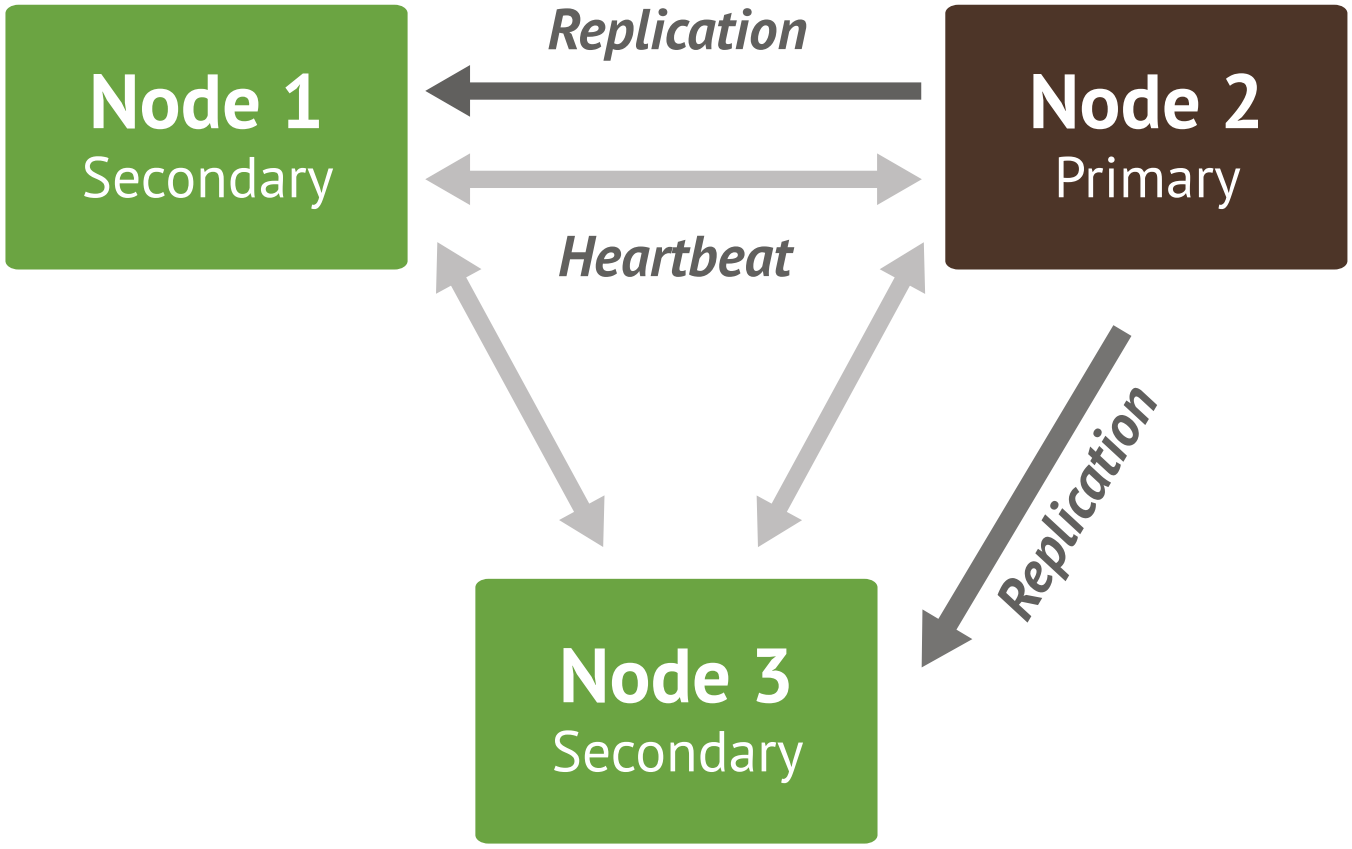
Replica Set Lifecycle - Failover



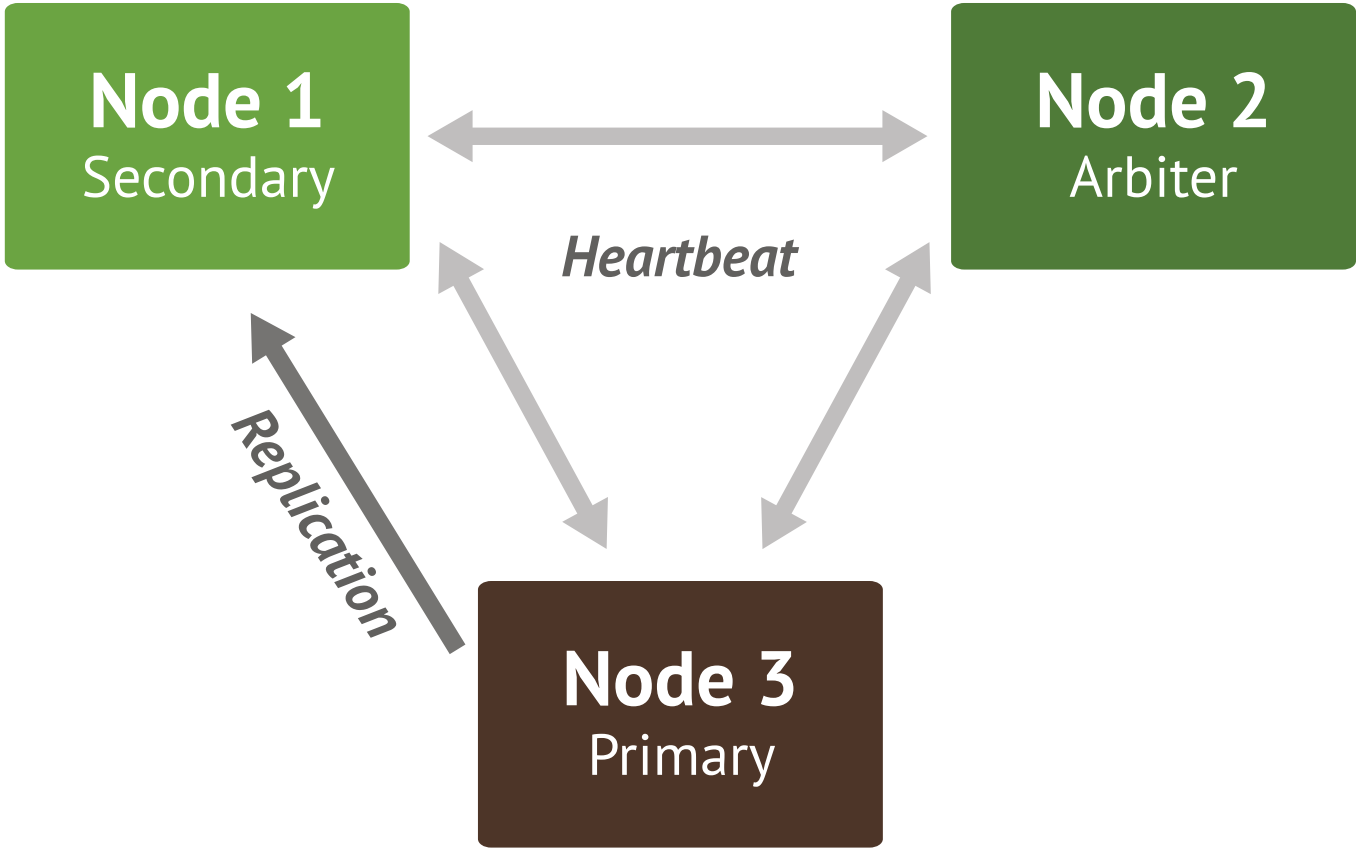
Replica Set Lifecycle - Recovery



Replica Set Lifecycle - Recovered



Replica Set Roles



Replica Set Configuration Options

```
> conf = {
    _id : "mySet",
    members : [
        {_id : 0, host : "A", priority : 3},
        {_id : 1, host : "B", priority : 2},
        {_id : 2, host : "C"},
        {_id : 3, host : "D", hidden : true},
        {_id : 4, host : "E", hidden : true, slaveDelay : 3600}
       ]
}

> rs.initiate(conf)
```

Creating a replica set

Elections

- What triggers an election?
  - there is no primary
  - nobody else has called for an election
- What conditions to be elected?
  - must be up to date to win to avoid a rollback
  - priority different than 0
  - if you have a higher priority, you have 10 seconds to catch up
- Each member has one vote or none (since 2.6)

Number of nodes in a replica set

- Odd number
- Beware of more than one arbiter

Election examples

- 3 data nodes: 1 down, 1 with more data
- Same, the one with more data has priority=0
- 5 data nodes: 2 with vote=0
- 2 data nodes + arb

=> 3 data nodes is the optimum configuration for cost and HA for most cases.

Replica States

Number	Name	State Description
0	`STARTUP`	Not yet an active member of any set. All members start up in this state. The `mongod` parses the replica set configuration document while in `STARTUP`.
1	`PRIMARY`	The member in state primary is the only member that can accept write operations.
2	`SECONDARY`	A member in state secondary is replicating the data store. Data is available for reads, although they may be stale.
3	`RECOVERING`	Can vote. Members either perform startup self-checks, or transition from completing a rollback or resync.
5	`STARTUP2`	The member has joined the set and is running an initial sync.
6	`UNKNOWN`	The member's state, as seen from another member of the set, is not yet known.
7	`ARBITER`	Arbiters do not replicate data and exist solely to participate in elections.
8	`DOWN`	The member, as seen from another member of the set, is unreachable.
9	`ROLLBACK`	This member is actively performing a rollback. Data is not available for reads.
10	`REMOVED`	This member was once in a replica set but was subsequently removed.

The “local” database

```
training:PRIMARY>
db.version()
3.0.7

training:PRIMARY> show
collections

me
oplog.rs
startup_log
system.indexes(MMAP only)
system.replset
```

```
training:PRIMARY>
db.version()
3.2.0-rc6

training:PRIMARY> show
collections

me
oplog.rs
replset.election
startup_log
system.replset
```

Oplog

- Logical replication (or statement replication)
- Size (default to 5% of available disk space)
  - With min and max
  - Should be set in relationship with write rates
- Optime (timestamp + counter)
- Capped collection
- Idempotency
  - As long as you replay in order, even overlapping, you are guaranteed to have the right results

Oplog example

```
> db.oplog.rs.find()
{
    "ts" : { "t" : 1286821527000, "i" : 1 },
    "h" : NumberLong(0),
    "op" : "n",
    "ns" : "",
    "o" : { "msg" : "initiating set" }
}
{
    "ts" : Timestamp(1424838907, 1),
    "h" : NumberLong("-6646643812489756655"),
    "v" : 2,
    "op" : "i",
    "ns" : "test.test",
    "o" : { "_id" : ObjectId("54ed50fbd0ace4fb57c0dfd8") }
}
```

ts: the time this operation occurred.
h: a unique ID for this operation. Each operation will have a different value in this field.
op: the write operation that should be applied to the slave. n indicates a no-op, this is just an informational message.
ns: the database and collection affected by this operation. Since this is a no-op, this field is left blank.
o: the actual document representing the op. Since this is a no-op, this field is pretty useless.

OpLog Idempotency

```
> db.replsettest.insert({_id:1,value:1})
{ "ts" : Timestamp(1350539727000, 1),
  "h" : NumberLong("6375186941486301201"),
  "op" : "i”,
  "ns" : "test.replsettest",
  "o" : { "_id" : 1, "value" : 1 } }

> db.replsettest.update({_id:1},{$inc:{value:10}})
{ "ts" : Timestamp(1350539786000, 1),
  "h" : NumberLong("5484673652472424968"),
  "op" : "u",
  "ns" : "test.replsettest",
  "o2" : { "_id" : 1 },
  "o" : { "$set" : { "value" : 11 } } }
```

OpLog Idempotency – cont…

- Single operations can have many entries

  > db.replsettest.update({},{$set:{name : ”foo”}, false, true})

  { "ts" : Timestamp(1350540395000, 1), "h" : NumberLong("-4727576249368135876"), "op" : "u", "ns" : "test.replsettest", "o2" : { "_id" : 2 }, "o" : { "$set" : { "name" : "foo" } } }
  { "ts" : Timestamp(1350540395000, 2), "h" : NumberLong("-7292949613259260138"), "op" : "u", "ns" : "test.replsettest", "o2" : { "_id" : 3 }, "o" : { "$set" : { "name" : "foo" } } }
  { "ts" : Timestamp(1350540395000, 3), "h" : NumberLong("-1888768148831990635"), "op" : "u", "ns" : "test.replsettest", "o2" : { "_id" : 1 }, "o" : { "$set" : { "name" : "foo" } } }

{multi:true} operation

```
training:PRIMARY> db.col1.update({},{$inc:{c2:1}},{multi:true})
WriteResult({ "nMatched" : 2, "nUpserted" : 0, "nModified" : 2 })
{
        "ts" : Timestamp(1424845573, 1),
        "h" : NumberLong("7952770730623398808"),
        "v" : 2,
        "op" : "u",
        "ns" : "test.col1",
        "o2" : {
                "_id" : ObjectId("54ed651f04461c9574eee5c4")
        },
        "o" : {
                "$set" : {
                        "c2" : 1
                }
        }
}
{
        "ts" : Timestamp(1424845573, 2),
        "h" : NumberLong("-4700050761066245270"),
        "v" : 2,
        "op" : "u",
        "ns" : "test.col1",
        "o2" : {
                "_id" : ObjectId("54ed65de2e279e3463caab90")
        },
        "o" : {
                "$set" : {
                        "c2" : 1
                }
        }
}
```

$push

```
training:PRIMARY> db.col1.update({_id:1},{$push:{array:"a"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

training:PRIMARY> db.col1.update({_id:1},{$push:{array:"b"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

training:PRIMARY> db.col1.update({_id:1},{$push:{array:"c"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

training:PRIMARY> db.getSiblingDB("local").oplog.rs.find().sort({$natural:-1}).limit(1).pretty()
{
      "ts" : Timestamp(1424845821, 1),
      "h" : NumberLong("8845512883118082322"),
      "v" : 2,
      "op" : "u",
      "ns" : "test.col1",
      "o2" : {
            "_id" : 1
      },
      "o" : {
            "$set" : {
                  "array.2" : "c"
            }
      }
}
```

$addToSet

```
training:PRIMARY> db.col1.update({_id:1},{$addToSet:{array:"e"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

training:PRIMARY> db.getSiblingDB("local").oplog.rs.find().sort({$natural:-1}).limit(1).pretty()
{
	"ts" : Timestamp(1424846204, 1),
	"h" : NumberLong("-8811591043923449945"),
	"v" : 2,
	"op" : "u",
	"ns" : "test.col1",
	"o2" : {
		"_id" : 1
	},
	"o" : {
		"$set" : {
			"array" : [
				"a",
				"b",
				"c",
				"e"
			]
		}
	}
}
```

Other commands and operators

- findAndModify()
  - Single oplog update (“u”), if found
- remove()
  - X entries in the oplog, one per removed document
- drop()
  - Command to drop the collection
- createIndex()
  - Inserts into the index collection

Initial Sync

- dropAllDatabasesExceptLocal
- Repeat until done
  - sync/cloning admin, test, datadbs
  - apply some oplogs
  - sync/cloning indexes

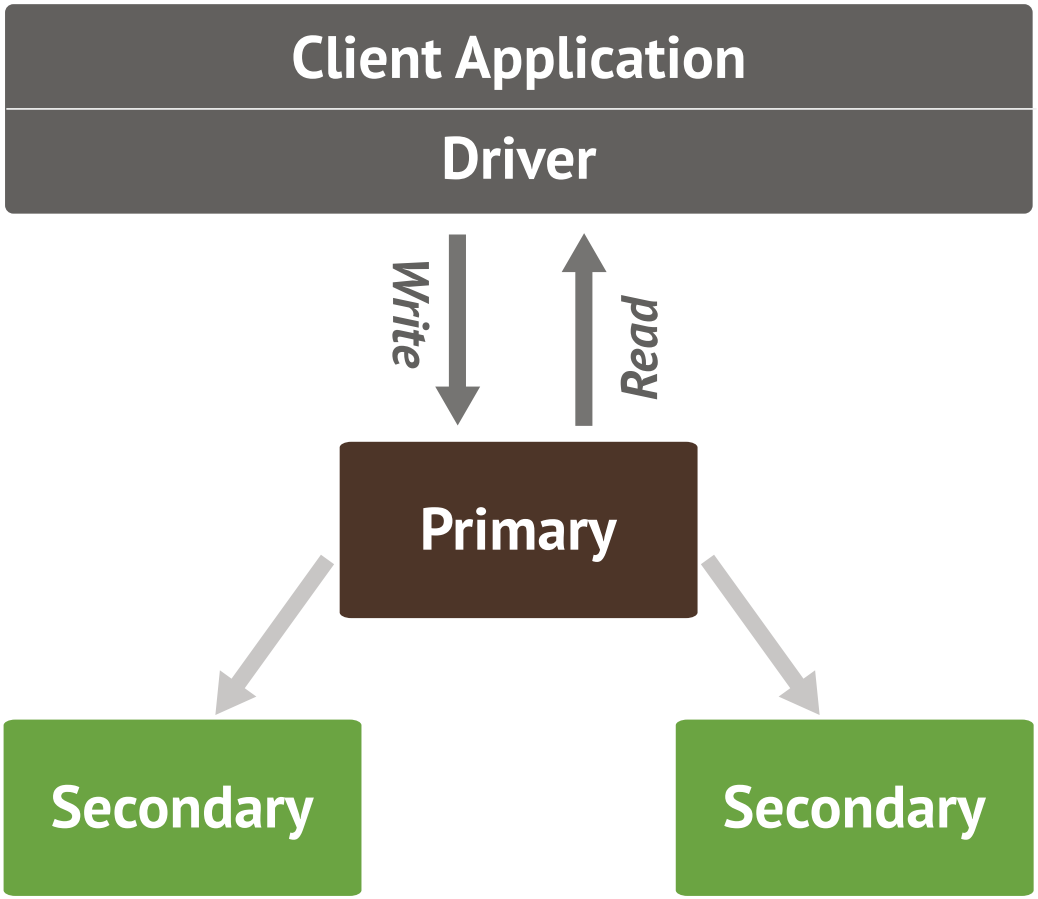
Initial sync details

0. Add _initialSyncFlag to minValid collection to tell us to restart initial sync if we crash in the middle of this procedure
1. Record start time.
2. Clone.
3. Set minValid1 to sync target's latest op time.
4. Apply ops from start to minValid1, fetching missing docs as needed.
5. Set minValid2 to sync target's latest op time.
6. Apply ops from minValid1 to minValid2.
7. Build indexes.
8. Set minValid3 to sync target's latest op time.
9. Apply ops from minValid2 to minValid3.
10. Cleanup minValid collection: remove _initialSyncFlag field, set ts to minValid3 OpTime

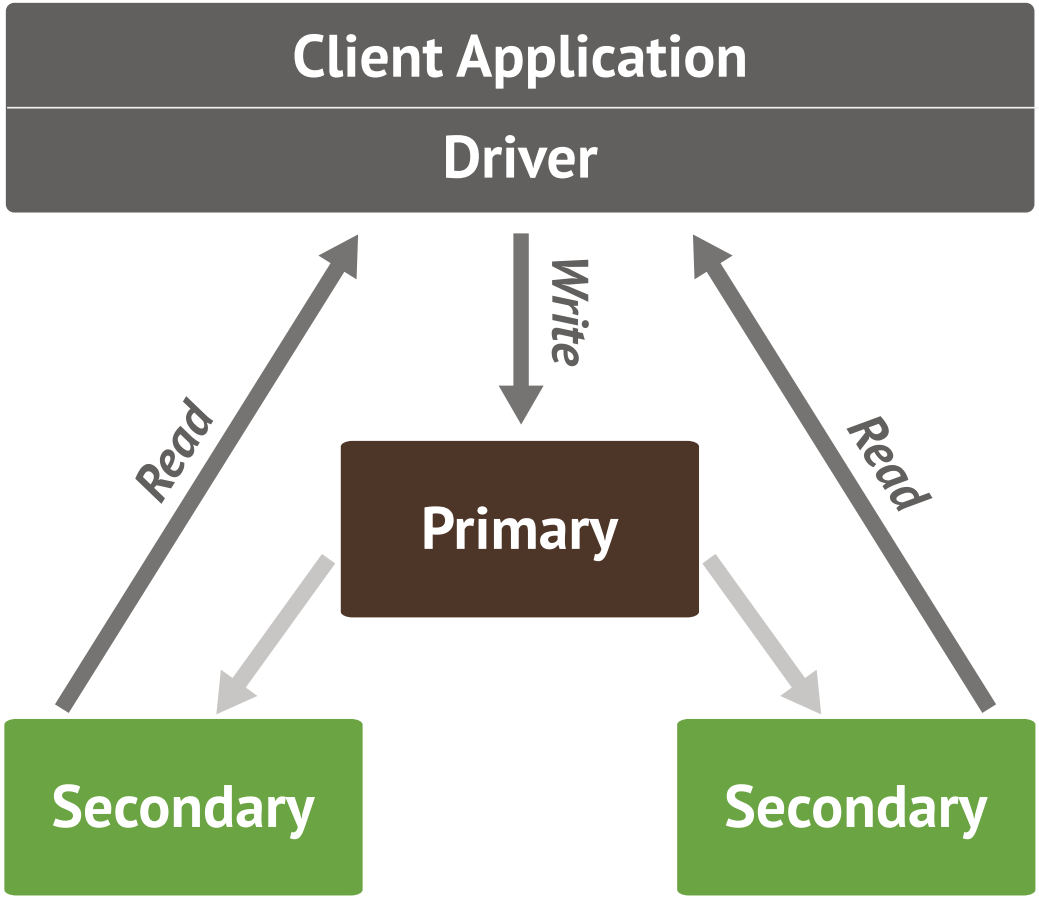
Developing with replica sets

- Consistency
- Write preferences
- Read preferences

Strong Consistency



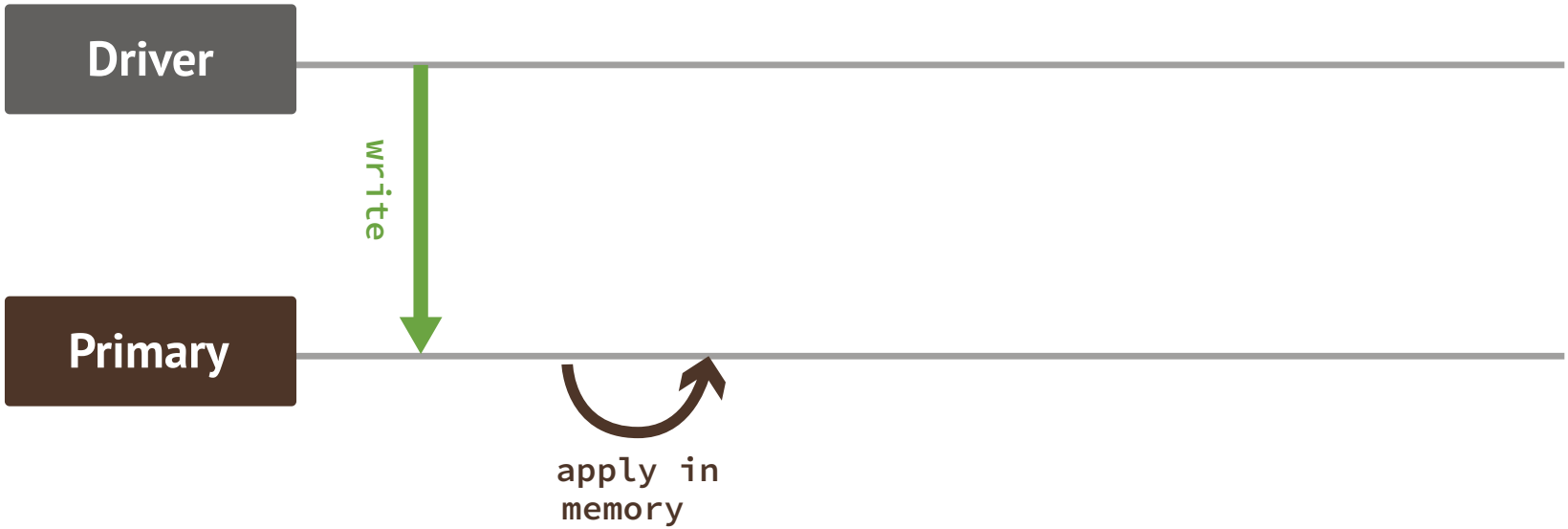
Delayed Consistency



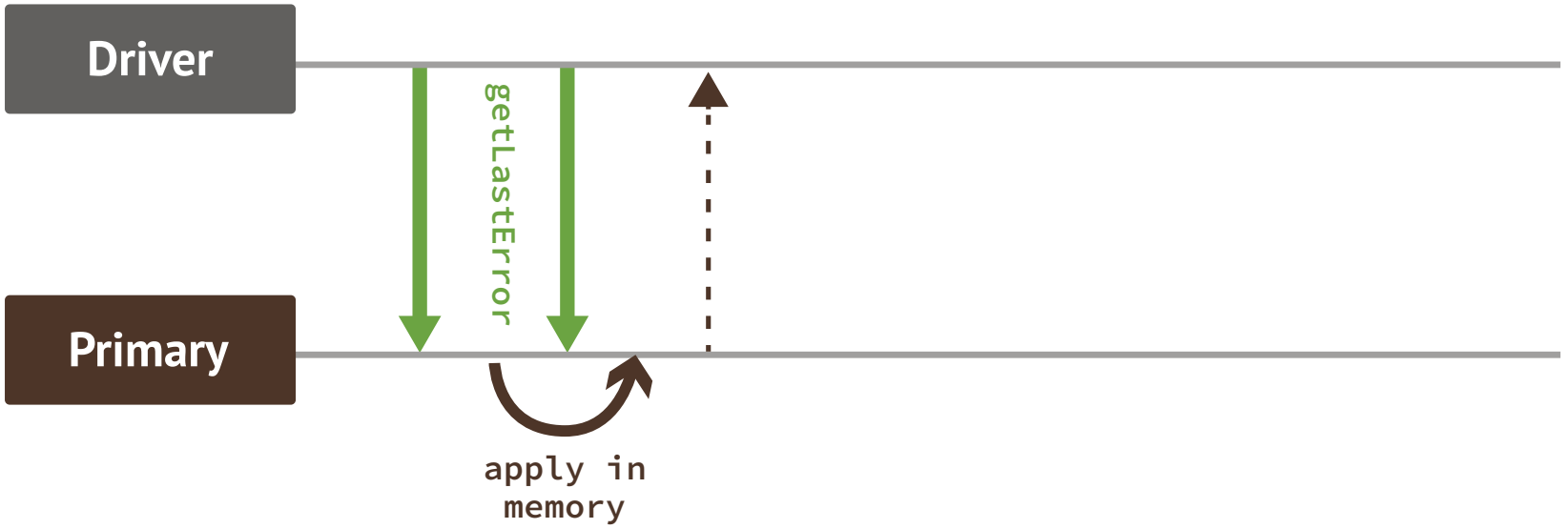
Write Concern

- Network acknowledgement
- Wait for error
- Wait for journal sync
- Wait for replication

Write Unacknowledged

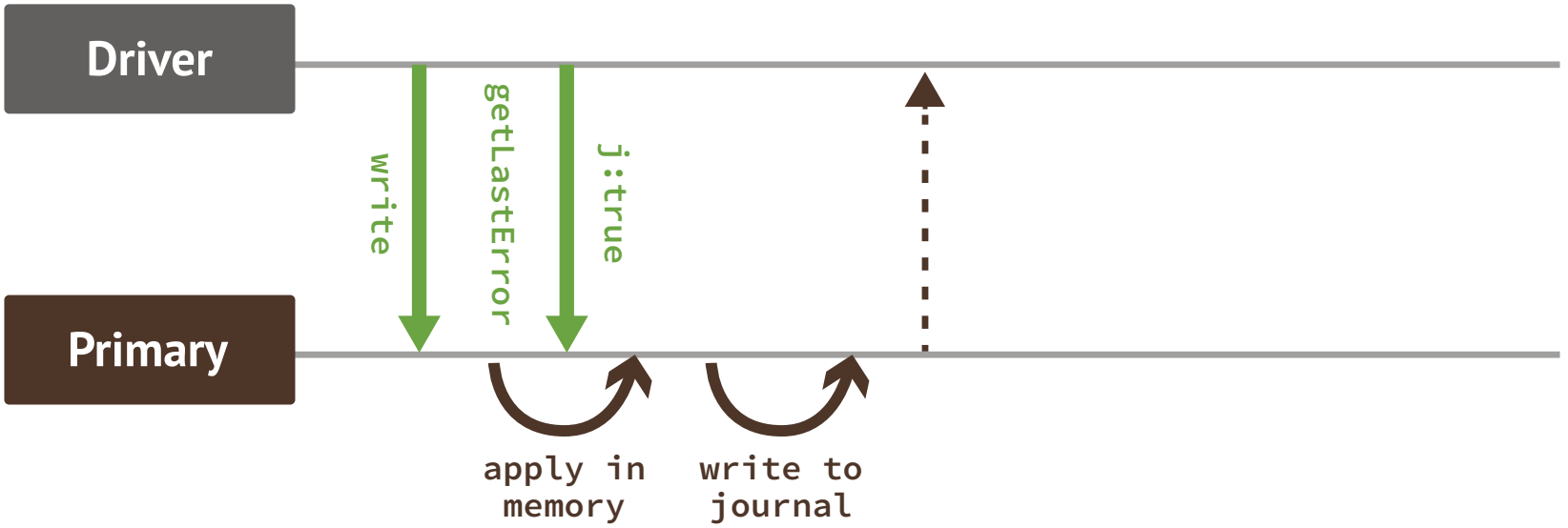


Write acknowledged



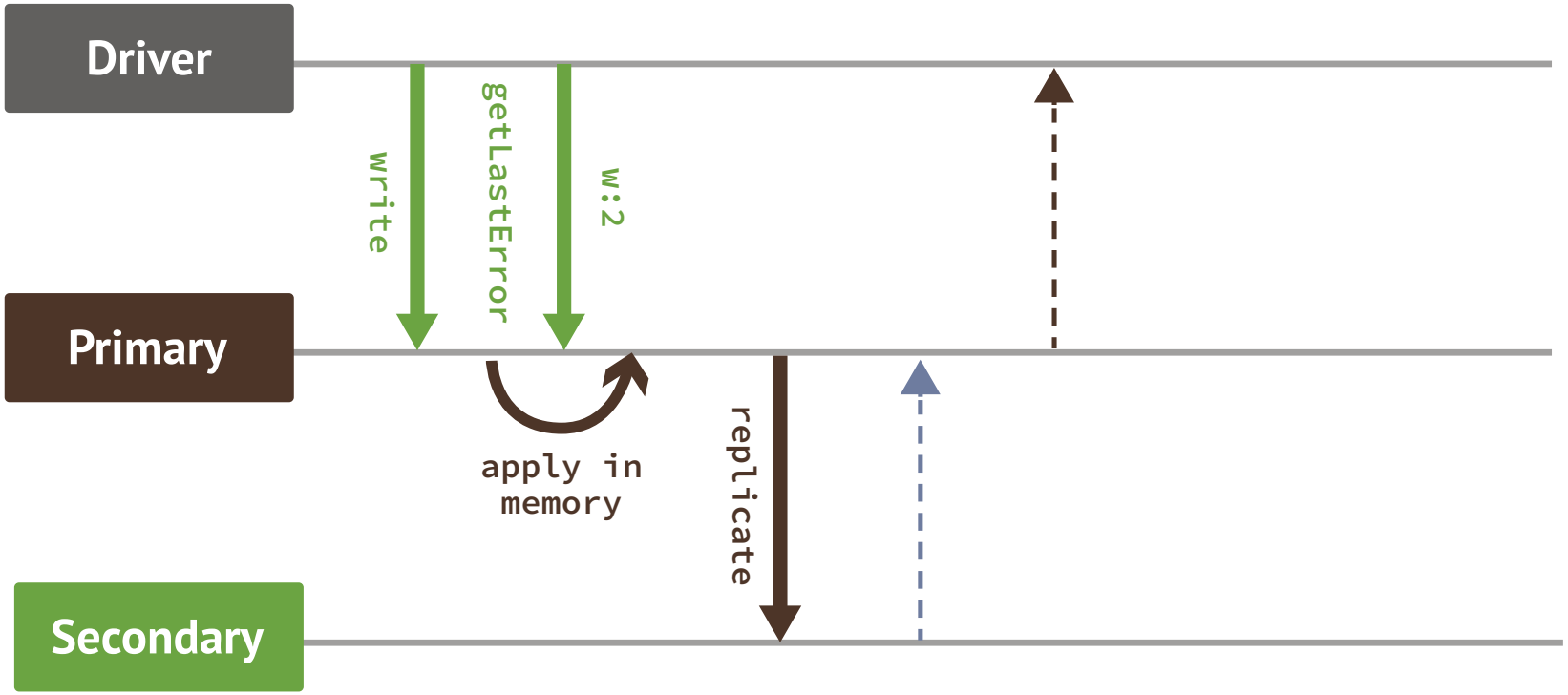
Write with a journal sync

- Journal is written to disk before it returns



Write with acknowledged replication

- Written to at least another node, in memory



Writing to a majority of nodes

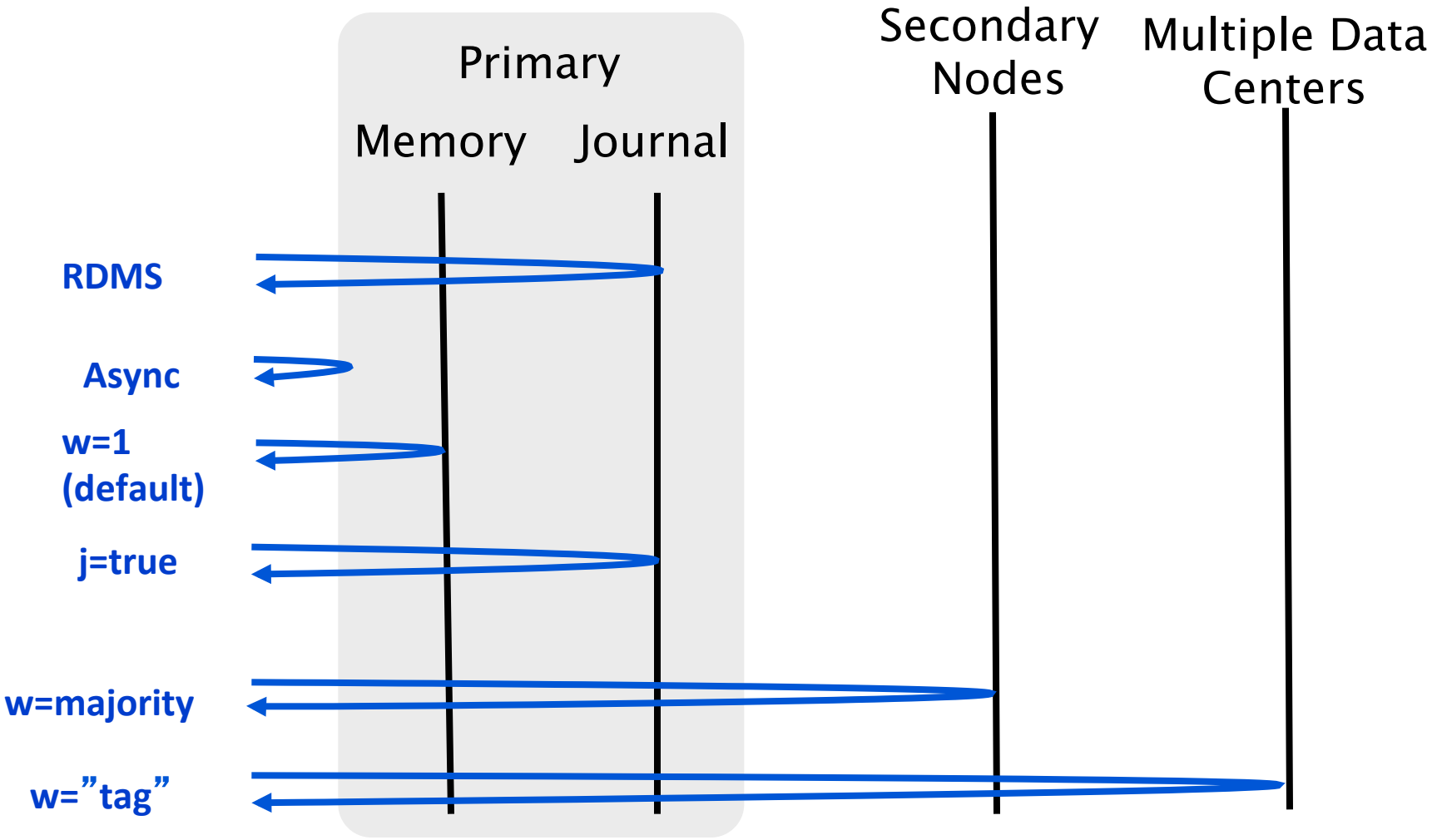
- The way to ensure data is not getting rolled back
- Primary does not wait on secondaries to do its write to the disk, making it available to other requests
- Only when secondaries have the write (may still not be persisted) that the Primary will return the status to the driver.
- Write majority is different than election majority
  - 2.6 => majority of replica set members
  - 3.0 => majority of voting members

Read Preferences

MongoDB drivers support five read preference modes.

Read Preference Mode	Description
`primary`	Default mode. All operations read from the current replica set primary.
`primaryPreferred`	In most situations, operations read from the primary but if it is unavailable, operations read from secondary members.
`secondary`	All operations read from the secondary members of the replica set.
`secondaryPreferred`	In most situations, operations read from secondary members but if no secondary members are available, operations read from the primary.
`nearest`	Operations read from member of the replica set with the least network latency, irrespective of the member's type.

Durability


Primary
Memory
Journal
Secondary Nodes
Multiple Data Centers
RDMS
Async
w=1 (default)
j=true
w=majority
w="tag"

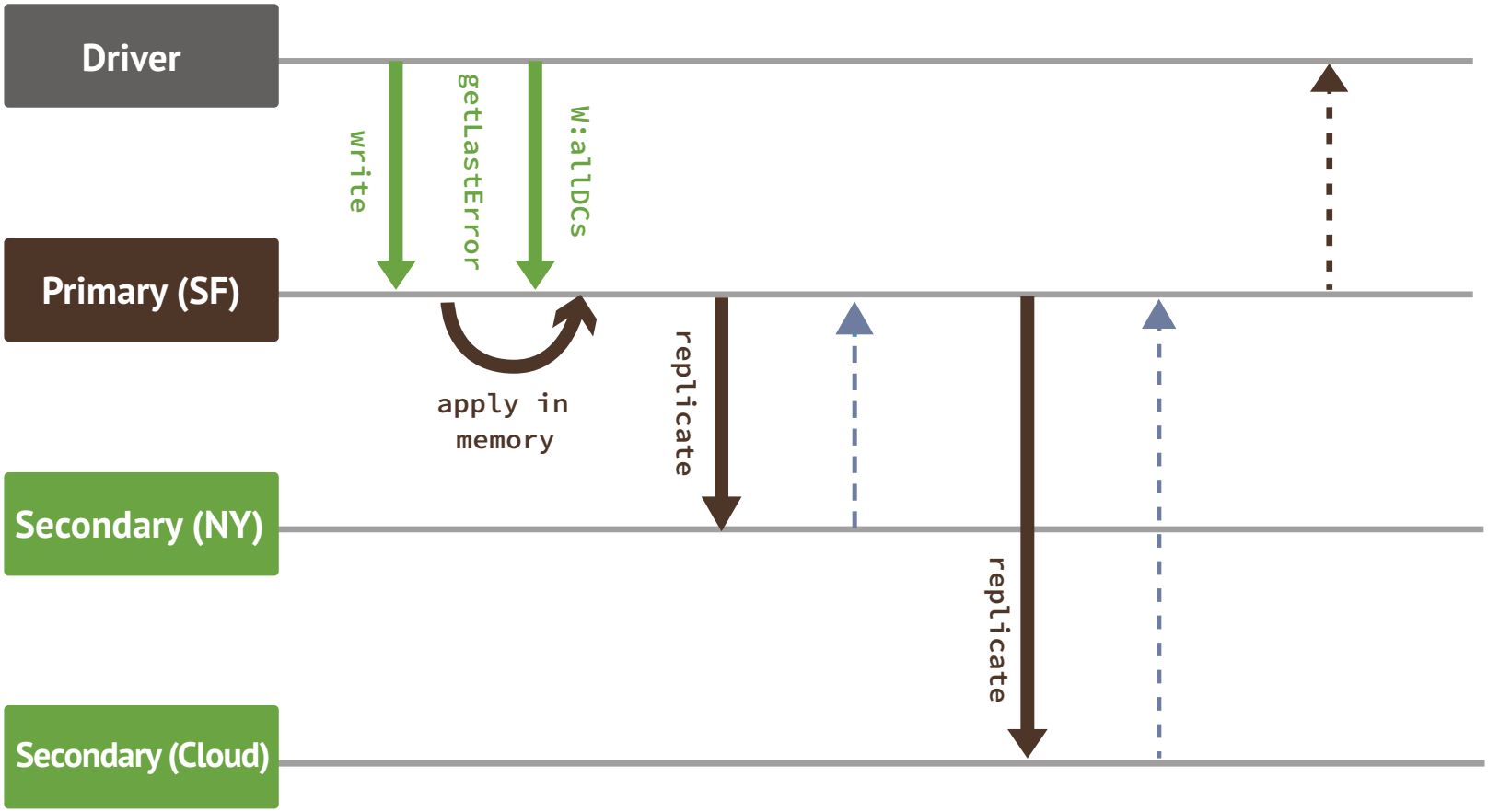
Tagging

- Control where data is written to, and read from
- Each member can have one or more tags
  - tags: {dc: "ny"}
  - tags: {dc: "ny",
    subnet: "192.168",
    rack: "row3rk7"}
- Replica set defines rules for write concerns
- Rules can change without changing app code

Tagging Example

```
{
   _id : "mySet",
   members : [
      {_id : 0, host : "A", tags : {"dc": "ny"}},
      {_id : 1, host : "B", tags : {"dc": "ny"}},
      {_id : 2, host : "C", tags : {"dc": "sf"}},
      {_id : 3, host : "D", tags : {"dc": "sf"}},
      {_id : 4, host : "E", tags : {"dc": "cloud"}}],
   settings : {
      getLastErrorModes : {
         allDCs :  {"dc" : 3},
         someDCs : {"dc" : 2}} }
}
> db.blogs.insert({...})
> db.runCommand({getLastError : 1, w : "someDCs"})
```

Wait for Replication (Tagging)



isMaster()

2.6


```
{
        "setName" : "training",
        "setVersion" : 3,
        "ismaster" : true,
        "secondary" : false,
        "hosts" : [
                "Daniels-MacBook-Pro-2.local:28000",
                "Daniels-MacBook-Pro-2.local:28003",
                "Daniels-MacBook-Pro-2.local:28002"
        ],
        "primary" : "Daniels-MacBook-Pro-2.local:28000",
        "me" : "Daniels-MacBook-Pro-2.local:28000",
        "maxBsonObjectSize" : 16777216,
        "maxMessageSizeBytes" : 48000000,
        "maxWriteBatchSize" : 1000,
        "localTime" : ISODate("2015-02-25T15:07:35.840Z"),
        "maxWireVersion" : 2,
        "minWireVersion" : 0,
        "ok" : 1
}
```


3.0


```
{
        "setName" : "training_30",
        "setVersion" : 3,
        "ismaster" : true,
        "secondary" : false,
        "hosts" : [
                "Daniels-MacBook-Pro-2.local:28011",
                "Daniels-MacBook-Pro-2.local:28012",
                "Daniels-MacBook-Pro-2.local:28013"
        ],
        "primary" : "Daniels-MacBook-Pro-2.local:28011",
        "me" : "Daniels-MacBook-Pro-2.local:28011",
        "electionId" : ObjectId("54ee121b5720d1650ccf97e8"),
        "maxBsonObjectSize" : 16777216,
        "maxMessageSizeBytes" : 48000000,
        "maxWriteBatchSize" : 1000,
        "localTime" : ISODate("2015-02-25T18:20:00.792Z"),
        "maxWireVersion" : 3,
        "minWireVersion" : 0,
        "ok" : 1
}
```

Rollback

- Old primary comes back up with oplogs the new primary does not have
- find the merge point with timestamp and “h” value
- ‘undo’ the extra operations
  - insert => delete document
  - delete => fetch document and insert
  - update => fetch document and update
- Max of 300MB of Oplog can be rollbacked

Assymetrical network partition

- Description:
  - Node A sees B
  - Node B does not see A
- Leads to difficult situations to debug

Doing Maintenance

- No downtime
- Rolling upgrade
  - start with secondaries
  - primary last
  - examples:
    - upgrading the MongoDB version
    - adding indexes
    - resizing the oplog
    - adding security features
  - MMS Automation to the rescue to do it
- Fixing issues
  - Resync node => probably the most used action on critical issues

Re-sync a node

- Remove dbpath, restart mongod
- From backup
  - Considerations: storage engine, local database
- rsync

Sync operations

- rs.syncFrom()
  - Allow to sync from a different node
  - Wrapper on replSetSyncFrom()
- Initial sync from a secondary
  - 30 sec window between the node being added and the start of the sync

SERVER-17975

- Two primaries at the same time
  - Writes to the old primary would be wrong
    - W=majority would prevent them
  - Reads to old primary may be stale if newer writes happened on the new primary
- … also mention of:
  - Commits can be visible on the master before a slave receives the oplog entry. Therefore visible commits can be rolled back regardless of the write concern.
- References
  - https://groups.google.com/forum/#!topic/mongodb-user/1Q0PRJply-I (Asya's response)
  - https://docs.google.com/document/d/1wbQNTa_A0nBtMOTgwHL9KQP8V8TZuIKgyX-jBs95BsE/edit

Doing Backups with mongodump

Cool tips from Asya

- Getting the list of available parameters in the shell
  - db.adminCommand({getParameter:'*'})
- Setting log level for a component
  - db.adminCommand({setParameter:1,logLevel:0})
  - db.adminCommand({setParameter: 1,'logComponentVerbosity':{replication:5}})

Exercise

- Why are we bringing out the replica set members one at a time to build indexes, when doing a rolling upgrade?

References

- http://www.kchodorow.com/blog/2010/10/12/replication-internals/
- http://docs.mongodb.org/manual/core/replica-set-elections/
- RAFT protocol: In Search of an Understandable Consensus Algorithm - by Diego Ongaro