redislabs
home of redis

WHITE PAPER

Managing Transactions in Redis^{e}

Roshan Kumar, Senior Product Marketing Manager, Redis Labs

CONTENTS

Executive Summary

Businesses, big and small, are gearing up to manage data at an unprecedented scale. Their challenge increases exponentially when managing data that require a permanent proof of record, especially when they arrive in great volume and at high speed.

Transactional databases, such as relational databases, are designed to assure data consistency. They are implemented with a rigid schema and strict ACID compliance, and are thus limited to moderate throughput and high latencies. NoSQL databases defy this rigidity. They differentiate themselves by being flexible on both schema and data consistency. Their flexibility enables solutions to manage high volumes of data, faster. However, as the responsibility of data consistency moves to the application layer, developing and maintaining applications that rely on high-volume transactional data becomes more complex and expensive.

Traditionally, solutions followed a monolithic architecture wherein all the modules and components employed a common, central data store to manage all the data. The new microservices approach breaks this paradigm. It parses every solution into multiple, independent microservices and assigns the responsibility of data management to each microservice. Each microservice employs its own data store specific to the type of data it handles and what it does with it. Microservices managing high-speed transactions require the flexibility of NoSQL databases, the fast performance of in-memory databases, and the ACID compliance of transactional databases.

Microservices-based architectures drive some new requirements for transaction handling. When the data is shared between multiple microservices, the idea of transactions becomes a bit blurred at the solution level. Depending on the type of data being handled by the transactions, microservices may ease some restrictions on consistency at the database layer, but will eventually manage them at the application layer. Following the microservices architecture requires databases to be tunable for data consistency and durability while supporting atomicity and isolation. Most NoSQL databases are specialized for certain use cases and fall short of offering tunable capabilities for consistency and durability. Redis, on the other hand, offers both tunable consistency and durability.

Redis is a high-performance, in-memory database benchmarked to deliver, using fewest resources. Redis offers the following ACID controls to support high-speed transactions:

- Atomicity: Provided with all Redis commands, with Lua Scripts and MULTI-EXEC blocks.
- Consistency: Tunable consistency through the WAIT command
- Isolation: Optimistic locking through the WATCH command
- Durability with replication and disk persistence as a protection against data loss under failures

Redis^{e}, the enterprise edition of Redis database offered by Redis Labs, supports clustering, instantaneous replication, data backup, automatic recovery, and strong data consistency, making it ideal for enterprise level, mission-critical solutions. Redis^{e} also allows you to fine tune persistence and replication in order to meet your performance and data consistency requirements.

This whitepaper outlines the Redis tools used when handling transactions, illustrates various scenarios using sample programs, and details how Redis^{e} can help you scale and increase performance with transactional workloads.

Introduction

Many existing software solutions use relational databases as the primary source of record keeping, mainly due to the assured data consistency that databases offer via the ACID properties of their transactions. Offloading the responsibility of managing transactions to the databases simplifies the architecture at the application layer. However, this comes at the cost of performance. The databases themselves spend significant effort when enforcing the data constraints that maintain strict ACID compliance for all operations without any tunable flexibility.

A few NoSQL databases have tried avoiding transactions altogether. They assert that not all applications require the same level of data consistency and durability, and if needed they could be enforced in the application layer itself. By relaxing their data consistency guarantees, these NoSQL databases have been able to support the high data volume and velocity that traditional transaction-oriented databases could not handle. Such databases work well for applications in which data availability takes a priority over consistency.

Now the big question is, "How can applications process data arriving at high volume and velocity with very low latency, while offering a reasonable assurance of data consistency?" Solutions that deal with payment transfers, order processing, workflow management, title recording, etc. fall under this category. This is where the need to support high-speed transactions comes into the picture. The databases that support high-speed transactions not only have a capacity for processing thousands if not millions of transactions per second, but also offer a guarantee that the data is consistent and durable.

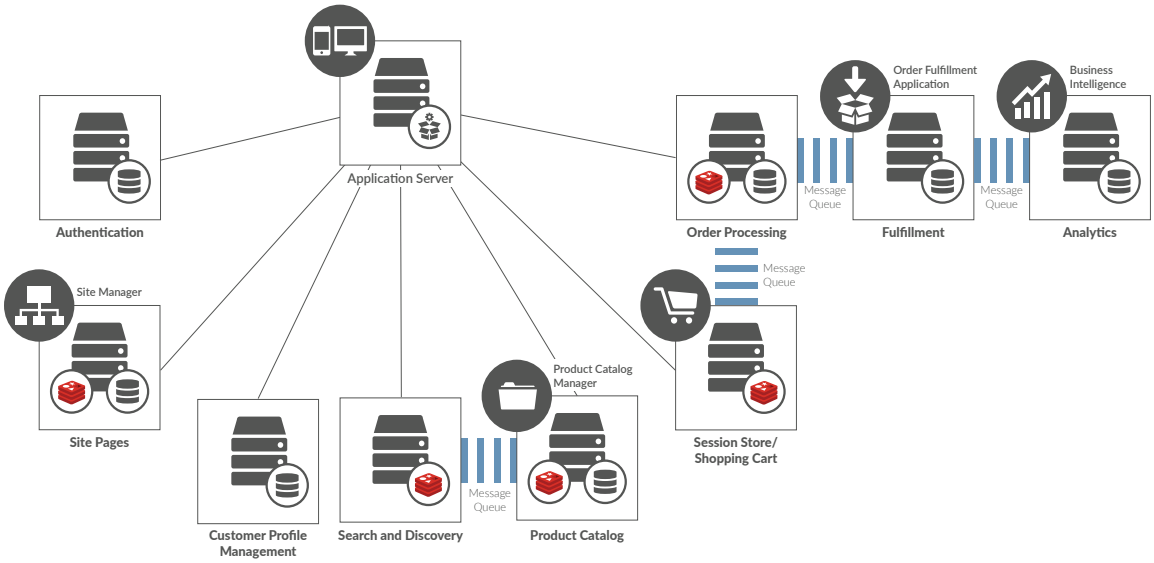

Figure 1. Microservices Architecture of a sample e-Commerce solution

In reality, not all components of a solution have the same requirements for data durability, consistency, availability, and latency. For example, an e-Commerce application may implement independent components or microservices, as shown in Figure 1. Therefore, microservices select their own data store. A caching service, for example, may be satisfied with low durability guarantees as the primary data store can always be consulted for the actual value. However, a payment processing service may require strict atomicity, consistency, isolation, and durability.

Redis and Redis[e] Building Blocks for High-speed Transactions

Redis, a high-speed, in-memory database platform is uniquely positioned to offer transaction support with sub millisecond latency for data arriving at high volume and velocity. Redis equips its applications with ACID controls to suit varying performance needs. Redis is popular in use cases ranging from job queue management to session store, high-speed financial transactions to real-time analytics, and more. Redis[e] is the enterprise edition of Redis offered by Redis Labs, the home of Redis. Redis[e] enhances open-source Redis with "always-on" availability; stable and predictable high performance; ACID support in a clustered environment; effortless scaling and the ability to run on Flash memory used as a RAM extender.

The following section will discuss the building blocks of Redis transactions, and how they can be used by applications to meet their transactional needs.

1. MULTI, EXEC Commands

The MULTI command initiates a transaction block in Redis. Redis command execution engine, being single threaded, does not interleave multiple concurrent commands or transaction block executions. This ensures that all commands entered between MULTI and EXEC are serialized and executed as if they were a single command, thus providing control over the atomicity of the execution. Figure 2 shows a simple example of a few commands encapsulated within the MULTI-EXEC block.

Example:

```
> MULTI
OK
> HMSET tx:18321 uid abcdef sku 94723 quantity 1 price 18.29 paymentmethod default
QUEUED
> HINCRBY account:abcdef orders 1
QUEUED
> LPUSH neworders tx:18321
QUEUED
> EXEC
1) OK
2) (integer) 1
3) (integer) 1
```

Figure 2. Sample execution of an e-Commerce transaction using Redis commands

Here's the sample Java program snippet that executes the same MULTI/EXEC block shown above:

```
Jedis jedis = conn.getJedis();

HashMap txIn = new HashMap();
txIn.put("uid", uid);
txIn.put("sku", sku);
txIn.put("quantity", quantity);
txIn.put("price", price);
txIn.put("paymentmethod", paymentMethod);

Transaction tx = jedis.multi();
tx.hmset("tx:"+txId, txIn);
tx.hincrBy("account:"+uid, "orders", 1);
tx.lpush("neworders", "tx:"+txId);

List<Object> out = tx.exec();
```

Figure 3. A sample Java program executing an e-Commerce transaction

2. WATCH Command – Optimistic Locking

WATCH allows Redis transactions to implement optimistic locking with Check and Set (CAS). Doing so will detect concurrent writes.

The Java program snippet below demonstrates usage of the WATCH command by reading the value stored in key "totalFundsRaised", updating it and saving the new value inside the transaction.

```
int fundsRaised = Integer.parseInt(jedis.get("totalFundsRaised"));

fundsRaised = fundsRaised + pledgeAmount;

Transaction tx = jedis.multi();
tx.set("totalFundsRaised",Integer.toString(fundsRaised));
List<Object> out = tx.exec();

System.out.println("Total funds raised:"+ jedis.get("totalFundsRaised"));
```

Figure 4. A sample code snippet that sets the value of totalFundsRaised

The picture below shows two Java threads executing the code snippet in parallel. Even though Redis assures atomic execution within a MULTI-EXEC block, the write operation is acting on the previously retrieved totalFundsRaised value outside the MULTI-EXEC block. Concurrent execution of the block results in incorrect serialization of both events and incorrect totalFundsRaised value at the end of the execution (a dirty-write).

Figure 5. Two threads reading and writing to the same key. Thread 2 writes over the data written by Thread 1

The WATCH command is used by programs to implement optimistic locking, which prevents dirty writes. Here's a sample program that applies the WATCH command. The SET command by the second thread fails in this case because the first thread had already modified the value of "totalFundsRaised."

```
jedis.watch("totalFundsRaised");
int fundsRaised = Integer.parseInt(jedis.get("totalFundsRaised"));

fundsRaised = fundsRaised + pledgeAmount;

Transaction tx = jedis.multi();
tx.set("totalFundsRaised",Integer.toString(fundsRaised));
List<Object> out = tx.exec();

System.out.println("Total funds raised:"+ jedis.get("totalFundsRaised"));
```

Figure 6. A sample Java code demonstrating the WATCH command

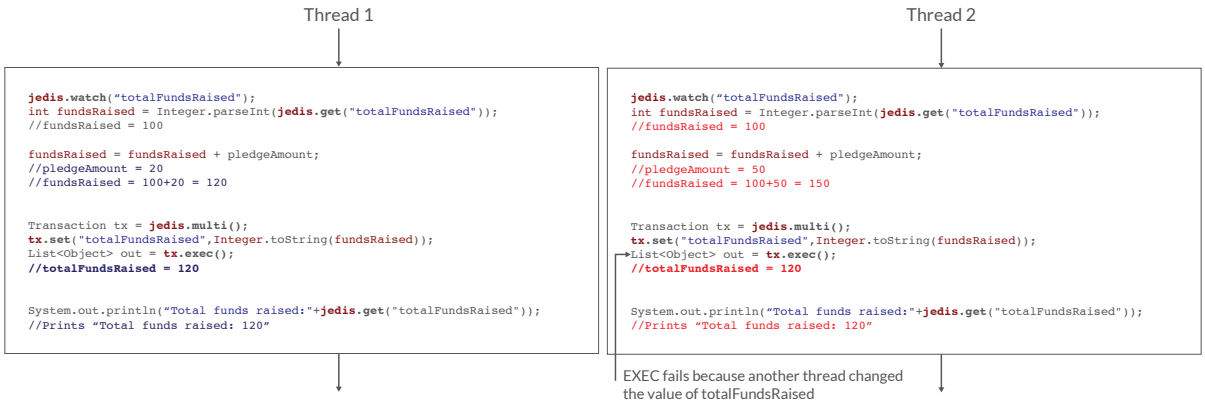

Figure 7. Two threads writing to the same key. WATCH command forces the second update to fail

A NOTE OF CAUTION

A question frequently raised is, "If MULTI/EXEC promises serializability, then why can't we insert commands to read and update variables in a MULTI/EXEC block [as shown below]?"

```
jedis.watch("totalFundsRaised");

Transaction tx = jedis.multi();

int fundsRaised = Integer.parseInt(jedis.get("totalFundsRaised"));
fundsRaised = fundsRaised + pledgeAmount;

tx.set("totalFundsRaised",Integer.toString(fundsRaised));
List<Object> out = tx.exec();

System.out.println("New value of totalFundsRaised:"+ jedis.get("totalFundsRaised"));
```

Figure 8. Incorrect implementation inside MULTI/EXEC block

The program above is wrong and throws an exception. Please note that all the commands after the MULTI command are queued for execution, and are not executed until the EXEC command. Therefore, tx.get("totalFundsRaised") in the above code snippet returns "Queued", and not the value of totalFundsRaised.

3. Lua Scripts

Lua scripts are Redis' equivalent of "stored procedures." Redis ensures atomic execution of Lua scripts. Using Lua scripts, applications could execute complex transactions with zero latency. Here's a sample Lua script that takes two keys as an input and increments them:

```
-- read input arguments
local uid = ARGV[1]
local sku = ARGV[2]
local quantity = ARGV[3]
local paymentmethod = ARGV[4]
local price = redis.call("GET","price:"..sku)
local txid = redis.call("INCR","txid")

-- insert and update transaction data
redis.call("HMSET","tx:"..txid,
             "uid", uid,
             "sku", sku,
             "quantity", quantity,
             "price", price,
             "paymentmethod", paymentmethod);

redis.call("HINCRBY", "account:"..uid, "orders", 1)

-- push transaction to message queue
redis.call("LPUSH", "neworders", "tx:"..txid)

-- notify consumers of message queue
redis.call("PUBLISH", "neworderlisteners", "tx:"..txid)
```

Figure 9. A sample Lua script running the e-Commerce transaction

You can run the script in two ways:

```
Jedis jedis = conn.getJedis();
String script = loadScriptFromFile(); //assuming this loads the script
jedis.eval(script, 4, "abcdef", "94723", "1", "default");
```

1. Load the script and pass it to Redis every time for execution:

Figure 10. A sample use of EVAL command

```
Jedis jedis = conn.getJedis();
String script = loadScriptFromFile(); //assuming this loads the script
String sha = jedis.scriptLoad(script);
.
.
.
.
jedis.evalSha(sha, 4, "abcdef", "94723", "1", "default");
```

2. Pre-compile the code and execute it later by passing arguments

Figure 11. A sample use of SCRIPTLOAD and EVALSHA commands

4. Replication and Tunable Consistency

Redis^{e}, Redis Labs' enterprise offering of Redis offers high availability and durability capabilities via low latency replication. Memory based replication protocol in Redis^{e} can provide faster durability compared slower storage subsystems.

When replication is enabled, Redis^{e} maintains multiple copies of the same dataset. Replication can be set up between nodes within the same data-center, or across data-centers but in the same region (i.e. "multi-az" - multiple availability zones in the cloud), or across racks in the same data-center (for on-premises scenarios). If the master server fails, the slave automatically takes over as the master.

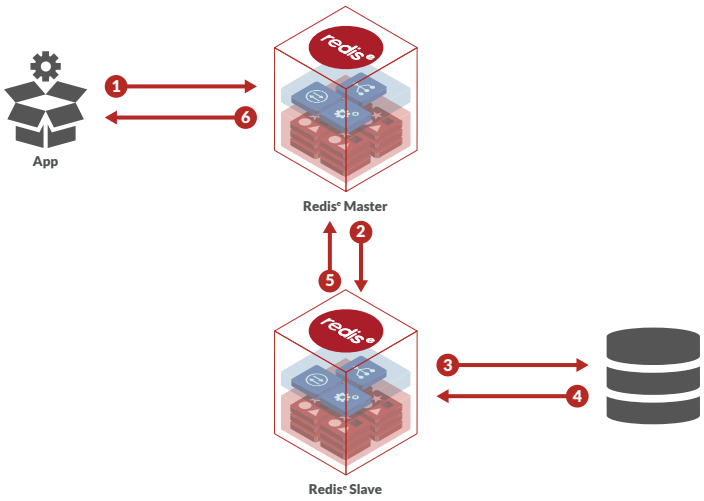

Figure 12. Asynchronous replication with master-slave setup. Only the slave is configured to persist data

With the WAIT command, applications can request acknowledgements for updates after multiple replicas complete the operation, or after multiple replicas complete the operation and its persistence to disk. With the Redis^{e} platform, WAIT command also operate across multiple keys that reside in multiple shard or nodes transparently.

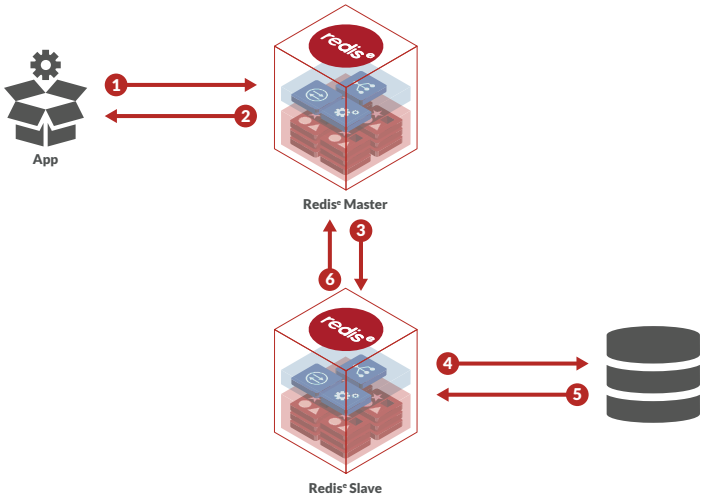

Figure 13. Synchronous replication with the WAIT command in a master-slave setup

Here is a LUA sample code snippet demonstrating the WAIT command:

```
-- read input arguments
local uid = ARGV[1]
local sku = ARGV[2]
local quantity = ARGV[3]
local paymentmethod = ARGV[4]
local price = redis.call("GET","price:"..sku)
local txid = redis.call("INCR","txid")

-- insert and update transaction data
redis.call("HMSET","tx:"..txid,
             "uid", uid,
             "sku", sku,
             "quantity", quantity,
             "price", price,
             "paymentmethod", paymentmethod);

redis.call("HINCRBY", "account:"..uid, "orders", 1)

-- push transaction to message queue
redis.call("LPUSH", "neworders", "tx:"..txid)

-- notify consumers of message queue
redis.call("PUBLISH", "neworderlisteners", "tx:"..txid)

redis.call("WAIT", 0)
```

Figure 14. The sample Lua script executing a WAIT command after the write operations. WAIT 0 means that the client will WAIT until the operation is succeeded

5. Disk Persistence – AOF, Snapshot

Redis[e] offers a rich set of features that enables you to fine tune the level of persistence, striking a balance between durability and performance while rendering the dataset durable and recoverable from disk under machine restarts. Redis provides two basic methods of persistence:

- AOF (Append only File)
- Snapshots

In AOF, Redis appends all 'write' operations to a file. This file could be used to reconstruct the dataset to the latest recorded write event. Snapshotting dumps Redis dataset to a file, which can be later used for recovery. While snapshots are efficient for backup and recovery, AOF files are best suited for durability.

With Redis[e] you get additional options for durability:

- AOF with fsync for every write
- AOF with fsync every second

The first technique promises the highest degree of durability while the second gives better performance.

Conclusion

Redis Labs' Redis[e] combines the power of open-source Redis with enterprise level high availability, durability, and consistency capabilities, making it a reliable database for mission-critical, high-speed transactions. Redis[e] satisfies the ACID needs of mission-critical services in the following ways:

Atomicity: Redis supports atomic execution of multi-command operations via MULTI/EXEC command blocks and LUA scripts.

Consistency: Redis[e] offers tunable consistency through the WAIT command to help applications relax or tighten consistency rules for best performance.

Isolation: Redis provides optimistic concurrency control with the WATCH command. Applications can simply implement check and set (CAS) with ease.

Durability: Durability in Redis[e] is achieved through replication and disk persistence. Redis[e] allows further tuning of disk persistence for best protection against failures.

The beauty of the Redis[e] platform is that it not only satisfies the most stringent data consistency, durability and reliability requirements, but also allows you to fine tune and relax those capabilities to optimize performance and user experience. This is ideal for microservices that deploy their own data stores.

Microservice	Atomicity	Consistency	Isolation	Durability	Redis Commands	Redis[e] Configuration for Persistence	Redis[e] Configuration for Replication
Cache Server	Low	Low	Low	Low	None	No persistence	Asynchronous in-memory replication
Session Store	Low	Low	Low	Low	None	No persistence	Asynchronous in-memory replication
Product Catalog Database	Medium	Medium	High	Medium	MULTI/EXEC	AOF every second on slave	Asynchronous in-memory replication
Search and Discovery	Medium	Medium	Medium	Medium	MULTI/EXEC	AOF every second on master	Asynchronous Disk based replication
Email Notifier	High	High	Medium	Medium	MULTI/EXEC, WATCH	AOF every second on slave	Asynchronous in-memory replication
Order Processing Database	High	High	High	High	MULTI/EXEC/ Lua, WATCH, WAIT	AOF every write on slave	Synchronous in-memory replication
Payments Transactions*	High	High	High	High	MULTI/EXEC/ Lua, WATCH, WAIT	AOF every write on slave	Synchronous in-memory replication

**Some microservices such as Payment Transactions that require a high degree of atomicity, must handle system failures that may occur in the middle of transactions and manage data rollbacks at the application layer*

Figure 15. Possible Redis commands and Redis[e] configuration in different microservices

700 E El Camino Real, Suite 250
Mountain View, CA 94040
(415) 930-9666
redislabs.com