Implementation of a Fault-Tolerant Key-Value Server Using RPC: A Redis Clone Comprehensive Report

Technical Report

December 31, 2024

1 Introduction

This report details the implementation of a fault-tolerant key-value server that replicates core Redis functionality. The system emphasizes reliability and data consistency while providing a comprehensive set of data structures and operations through a Remote Procedure Call (RPC) interface.

2 System Architecture

2.1 Core Components

The system consists of four main components:

1. **RPC Server**
   - Handles client connections and method invocations
   - Uses thread pools for concurrent client handling
   - Manages method registration and execution
2. **RPC Client**
   - Provides interface for client-server communication
   - Handles connection management
   - Implements JSON-based protocol
3. **Redis Clone Implementation**

- Core key-value store functionality
- Thread-safe operations
- Data persistence mechanisms

4. **Client Interface**
    - Command-line interface for user interaction
    - Support for all Redis-like commands
    - Error handling and validation

2.2 Communication Protocol

- JSON-based RPC protocol
- Request Format: `(method_name, args, kwargs)`
- Response Format: JSON-encoded return value
- Transport: TCP/IP sockets with fixed buffer size (1024 bytes)

3 Data Structures and Operations

3.1 Key-Value Operations

Basic operations supported:

- `SET key value [EX seconds]`: Store key-value pair with optional expiry
- `GET key`: Retrieve value
- `DELETE key`: Remove key-value pair
- `KEYS`: List all keys
- `FLUSHALL`: Clear all data
- `APPEND key value`: Append to existing string value
- `EXISTS key`: Check key existence

3.2 Time-To-Live (TTL) Operations

- `EXPIRE key seconds`: Set key expiration time
- `TTL key`: Get remaining time
  - Returns -2 if key doesn't exist
  - Returns -1 if key has no expiration
- `PERSIST key`: Remove expiration from key

3.3 Hash Operations

- `HSET hash_key field value`: Set field in hash
- `HGET hash_key field`: Get field value
- `HDEL hash_key field`: Delete field
- `HGETALL hash_key`: Get all fields and values
- `HDELALL hash_key`: Delete entire hash

3.4 Sorted Set Operations

- `ZSET key score value`: Add element with score
- `ZRANGE key start stop`: Get range (low to high)
- `ZREVRANGE key start stop`: Get range (high to low)
- `ZRANK key value`: Get element rank
- `ZDELVALUE key value`: Remove element
- `ZDELKEY key`: Delete entire sorted set
- `ZGETALL key`: Get all elements with scores

3.5 List Operations

- `LPUSH/RPUSH key value [value ...]`: Add elements
- `LPOP/RPOP key`: Remove and return element
- `LRANGE key start stop`: Get range of elements

- `LLEN key`: Get list length
- `DELPUSH key`: Clear and reinitialize list

4 Fault Tolerance Mechanisms

4.1 Thread Safety

- Reentrant locks (RLock) for data store operations
- Separate locks for list operations
- Atomic operations for data modifications
- Thread-safe method registration and invocation

4.2 Data Persistence

- Periodic snapshots to disk (default 30-second interval)
- Background thread for snapshot management
- JSON-based snapshot format:

```
snapshot_data = {
    'data': self.data_store,
    'expiry': self.expiry_times,
    'sorted_sets': self.sorted_sets
}
```

4.3 Error Handling

- Comprehensive exception handling for all operations
- Logging system with operation tracking
- Graceful handling of network failures
- Type checking and validation
- Automatic cleanup of expired keys

5 Implementation Details

5.1 Thread Management

```
# Server thread pool
Thread(target=self.__handle__, args=[client, address]).
    start()

# Background threads
self.cleanup_thread = threading.Thread(
    target=self._cleanup_expired_keys,
    daemon=True
)
self.snapshot_thread = threading.Thread(
    target=self._periodic_snapshot,
    daemon=True
)
```

5.2 Data Store Structure

```
class FaultTolerantRedisClone:
    def __init__(self):
        self.data_store: Dict[str, Any] = {}
        self.sorted_sets: Dict[str, List[tuple]] = {}
        self.expiry_times: Dict[str, float] = {}
        self.lock = threading.RLock()
        self.lock_list = threading.Lock()
```

6 Performance Considerations

6.1 Memory Management

- In-memory storage with disk persistence
- Efficient string operations
- Memory-conscious data structures
- Background cleanup of expired keys

6.2 Network Efficiency

- JSON serialization for data transport
- Fixed buffer size management
- Connection pooling for multiple clients

7 Reliability Features

7.1 Data Integrity

- Atomic operations for data consistency
- Snapshot verification on load
- Transaction logging
- Type validation

7.2 Recovery Mechanisms

- Automatic snapshot recovery
- Connection failure handling
- Error state recovery
- Expired key cleanup

8 Client Interface Example

```
# Basic operations
Redis Clone Client
Type your commands (e.g., 'SET key value') or type 'EXIT'
    to quit.
> SET mykey distributed
OK
> GET mykey
distributed
> DEL mykey
distributed
>GET mykey
None
```

```
# Hash operations
> HSET student name kious
OK
> HSET student age 19
OK
> HGETALL student
{'name': 'kious', 'age': '19'}

# Sorted set operations
> ZSET scores 100 alice
1
> ZSET scores 20 bob
1
> ZSET scores 50 kious
1
> ZRANGE scores 0 3
['bob', 'kious', 'alice']
>EXIST
Goodbye!
```

9 Future Improvements

Potential enhancements include:

- Data replication for high availability
- Support for additional complex data types
- Transaction support
- Incremental backup system
- Connection pooling optimization
- Cluster support

10 Conclusion

Fault-tolerant key-value server implementations provide a powerful and efficient solution to key-value storage needs. Integration of comprehensive data structure support tree safety Database stability and error handling

This makes it suitable for production applications that require Redis-like applications. Although not as versatile as Redis itself, it provides a solid foundation for space-hungry distributed systems. Reliable storage and access to data