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

Technical Report

December 30, 2024

1 Introduction

This implementation creates a simplified Redis clone with a focus on fault tolerance and reliability, while maintaining the core functionality of a key-value store.

2 System Architecture

2.1 Core Components

The system consists of four main components:

1. **RPC Server**: Handles client connections and method invocations
2. **RPC Client**: Provides the interface for client-server communication
3. **Redis Clone**: Implements the key-value store functionality
4. **Client Interface**: Provides a command-line interface for user interaction

2.2 Communication Protocol

The RPC implementation uses a JSON-based protocol for method invocation:

- **Request Format**: (method_name, args, kwargs)
- **Response Format**: JSON-encoded return value
- **Transport**: TCP/IP sockets

3 Fault Tolerance Mechanisms

3.1 Thread Safety

The implementation ensures thread safety through:

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

3.2 Data Persistence

Data persistence is achieved through:

- Periodic snapshots to disk
- Background thread for snapshot management
- Automatic recovery from snapshot on startup

3.3 Error Handling

Comprehensive error handling includes:

- Exception handling for all operations
- Logging system for operation tracking
- Graceful handling of network failures
- Type checking and validation

4 Implementation Details

4.1 Key-Value Store Operations

The system implements the following Redis-like commands:

- `SET key value`: Store a key-value pair
- `GET key`: Retrieve a value by key
- `DELETE key`: Remove a key-value pair

- `KEYS`: List all keys
- `FLUSHALL`: Clear all data
- `APPEND key value`: Append to string values

4.2 Thread Management

- Server uses a thread pool for handling client connections
- Background thread for periodic snapshots
- Thread synchronization using reentrant locks

5 Code Structure

5.1 Server Implementation

```
class FaultTolerantRedisClone:
    def __init__(self):
        self.data_store = {}
        self.lock = threading.RLock()
        self.snapshot_interval = 60
        # ... initialization code
```

5.2 RPC Layer

```
class RPCServer:
    def __handle__(self, client, address):
        # Handle client requests
        while True:
            try:
                functionName, args, kwargs = json.loads(
                    client.recv(SIZE).decode())
                response = self._methods[functionName](*args, **kwargs)
                client.sendall(json.dumps(response).encode())
            except:
                break
```

6 Performance Considerations

6.1 Memory Management

- In-memory storage with disk persistence
- Efficient string operations
- Memory-conscious data structures

6.2 Network Efficiency

- JSON serialization for data transport
- Efficient socket buffer management
- Connection pooling for multiple clients

7 Reliability Features

7.1 Data Integrity

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

7.2 Recovery Mechanisms

- Automatic snapshot recovery
- Connection failure handling
- Error state recovery

8 Future Improvements

Potential enhancements include:

- Data replication for high availability
- Support for complex data types
- Transaction support

- Incremental backup system
- Connection pooling optimization

9 Conclusion

The implemented fault-tolerant key-value server provides a reliable and efficient solution for basic key-value storage needs. The combination of thread safety, data persistence, and comprehensive error handling makes it suitable for production use cases requiring basic Redis-like functionality.