Assignment 5

1 Introduction

The assignment requires building a distributed key-value storage service using the Replicated State Machine (RSM) model, where all state changes are coordinated through a Raft consensus log. The system consists of:

- **Clients**: Issue Get, Put, and Append operations via RPC
- **KV Servers**: Each server embeds a Raft peer and maintains a replicated state machine
- **Raft Layer**: Provides consensus on the order of operations

The key challenges include handling duplicate requests, managing leader changes, and ensuring linearizability under concurrent access and network failures.

2 Architecture Design

2.1 System Overview

The architecture follows a layered design:

1. **Client Layer**: Sends operations with unique identifiers
2. **KV Server Layer**: Processes RPCs and applies committed operations
3. **Raft Layer**: Provides total ordering of operations through log replication

2.2 Client-Server Interaction

The interaction workflow is as follows:

1. Client sends RPC (Get/Put/Append) to a server
2. If server is leader:
    - Server calls `rf.Start(op)` to append operation to Raft log
    - Server waits for operation to be committed via `applyCh`
    - Server applies operation to state machine
    - Server responds to client with result
3. If server is not leader or operation fails:
    - Server responds with `WrongLeader = true`
    - Client retries with next server

2.3 Op Structure Design

The `Op` struct encapsulates all information needed for an operation:

```
type Op struct {
    Operation string // "Get", "Put", or "Append"
    Key       string
    Value     string
    ClientId  int64  // Unique client identifier
    RequestId int    // Monotonically increasing request ID
}
```

Listing 1: Op Structure Definition

Key design decisions:

- **All fields capitalized**: Required for Go's RPC marshalling
- **ClientId**: Generated randomly at client initialization using cryptographically secure random number generator
- **RequestId**: Incremented for each new operation, enables duplicate detection

2.4 Server State Management

The KV server maintains the following state:

```
type RaftKV struct {
    mu                sync.Mutex
    rf                *raft.Raft
    applyCh           chan raft.ApplyMsg

    kvStore           map[string]string  // Key-value store
    lastApplied       map[int64]int      // ClientId -> RequestId
    notifyCh          map[int]chan Op    // Log index -> notification
        channel
    lastAppliedIndex int                // Last applied log index
}
```

Listing 2: RaftKV Structure

3 Duplicate Detection Strategy

3.1 Exactly-Once Semantics

To ensure exactly-once semantics for Put and Append operations, the system implements:

1. **Client-side**: Each client generates a unique `ClientId` at initialization and maintains a monotonically increasing `RequestId`
2. **Server-side**: The `lastApplied` map tracks the highest `RequestId` successfully applied for each client:

```
lastReq, exists := kv.lastApplied[op.ClientId]
if !exists || op.RequestId > lastReq {
    // Apply operation
    switch op.Operation {
    case "Put":
        kv.kvStore[op.Key] = op.Value
```

```
    case "Append":
        kv.kvStore[op.Key] += op.Value
    case "Get":
        // Read-only, no state change
    }
    kv.lastApplied[op.ClientId] = op.RequestId
}
```

Listing 3: Duplicate Detection Logic

3.2 Handling Retries

When a client retries due to timeout or leader change:

- If the operation was already applied, the server skips re-execution but still validates the request
- The notification mechanism ensures the correct response is sent even for duplicate requests

3.3 Memory Management

The `lastApplied` map grows over time. In a production system, this would require:

- Periodic garbage collection of old client entries
- Client-side acknowledgment mechanism to inform servers which requests can be forgotten
- For this assignment, we rely on the fact that tests use a bounded number of clients

4 Consistency Guarantees

4.1 Linearizability

The system ensures linearizability through:

1. **Total Ordering**: All operations pass through Raft log, establishing a global order
2. **State Machine Replication**: All servers apply operations in the same order
3. **Leader-based Reads**: Even Get operations go through Raft log, preventing stale reads

4.2 Preventing Stale Reads

Key design choice: **All Get() requests are submitted to Raft log**

- This prevents returning stale data from a partitioned or outdated server
- Trade-off: Lower read performance but stronger consistency
- Alternative (not implemented): Lease-based read optimization from Raft paper Section 8

4.3 Leader Change Handling

The system handles leader changes gracefully:

1. RPC handlers detect leadership loss via timeout or term change
2. Notification channels use non-blocking sends to avoid deadlock
3. Clients retry operations until they succeed at the current leader

```
select {
case appliedOp := <-ch:
    // Verify this is our operation
    if appliedOp.ClientId == op.ClientId &&
       appliedOp.RequestId == op.RequestId {
        reply.WrongLeader = false
        reply.Err = OK
    } else {
        // Different op committed (leadership lost)
        reply.WrongLeader = true
    }
case <-time.After(1000 * time.Millisecond):
    reply.WrongLeader = true
}
```

Listing 4: Leader Change Detection

5 Concurrency and Synchronization

5.1 Avoiding Deadlocks

Critical design decisions to prevent deadlocks:

1. **Separate goroutine for apply loop**: The `applyLoop()` runs independently, continuously reading from `applyCh`
2. **Non-blocking notification**: Use buffered channels and select with default case:

```
if ch, ok := kv.notifyCh[msg.Index]; ok {
    select {
    case ch <- op:
    default:
    }
}
```

3. **Lock ordering**: Always acquire `kv.mu` before accessing shared state, release before blocking operations

5.2 Race Condition Prevention

- All access to `kvStore`, `lastApplied`, and `notifyCh` is protected by `kv.mu`
- Channel creation and deletion are synchronized
- Notification channels are created before starting Raft operation to avoid missing notifications

6 Client Optimization

6.1 Leader Caching

The client caches the last known leader:

```
type Clerk struct {
    servers   []*labrpc.ClientEnd
    clientId  int64
    requestId int
    leaderId  int  // Cached leader ID
}
```

Listing 5: Client Leader Cache

Benefits:

- Reduces latency for subsequent operations
- Minimizes unnecessary RPC attempts to non-leaders
- Simple round-robin fallback on failure

6.2 Request ID Management

```
func (ck *Clerk) Get(key string) string {
    ck.requestId++
    args := GetArgs{
        Key:       key,
        ClientId:  ck.clientId,
        RequestId: ck.requestId,
    }
    // ... retry loop
}
```

Listing 6: Request ID Increment

7 Testing Results

The implementation successfully passes all 12 test cases:

7.1 Performance Observations

- **Total test time**: 211 seconds for all 12 tests
- **Average per test**: 17.6 seconds
- Tests involving partitions and unreliable networks take longer (20-28s)
- Simple tests complete quickly (1-2s for TestUnreliableOneKey)

8 Key Implementation Insights

8.1 Design Principles

1. **Separation of Concerns**: Raft handles consensus, KV server handles state machine logic

Test Case	Points	Status
TestBasic	7	PASS
TestConcurrent	7	PASS
TestUnreliable	7	PASS
TestUnreliableOneKey	7	PASS
TestOnePartition	7	PASS
TestManyPartitionsOneClient	7	PASS
TestManyPartitionsManyClients	7	PASS
TestPersistOneClient	7	PASS
TestPersistConcurrent	8	PASS
TestPersistConcurrentUnreliable	8	PASS
TestPersistPartition	9	PASS
TestPersistPartitionUnreliable	9	PASS
Total	**90**	**90/90**

Table 1: Test Results Summary

2. **Simplicity**: Avoid premature optimization; focus on correctness first
3. **Idempotency**: Design operations to be safely retryable

8.2 Go-Specific Considerations

- Channel-based communication between layers
- Goroutines for concurrent RPC handling and apply loop
- Mutex for protecting shared state
- Capitalized field names for RPC marshalling

9 Conclusion

This assignment demonstrates the practical implementation of a fault-tolerant distributed system using the Replicated State Machine model. The key achievements include:

- **Strong Consistency**: Linearizable operations through Raft log
- **Exactly-Once Semantics**: Duplicate detection prevents repeated execution
- **Fault Tolerance**: Handles network partitions, server failures, and message loss
- **Correctness**: Passes all test cases including stress tests

The implementation provides a solid foundation for understanding distributed consensus and could be extended with features like:

- Log compaction and snapshotting
- Client session management
- Read-only query optimization
- Multi-datacenter deployment

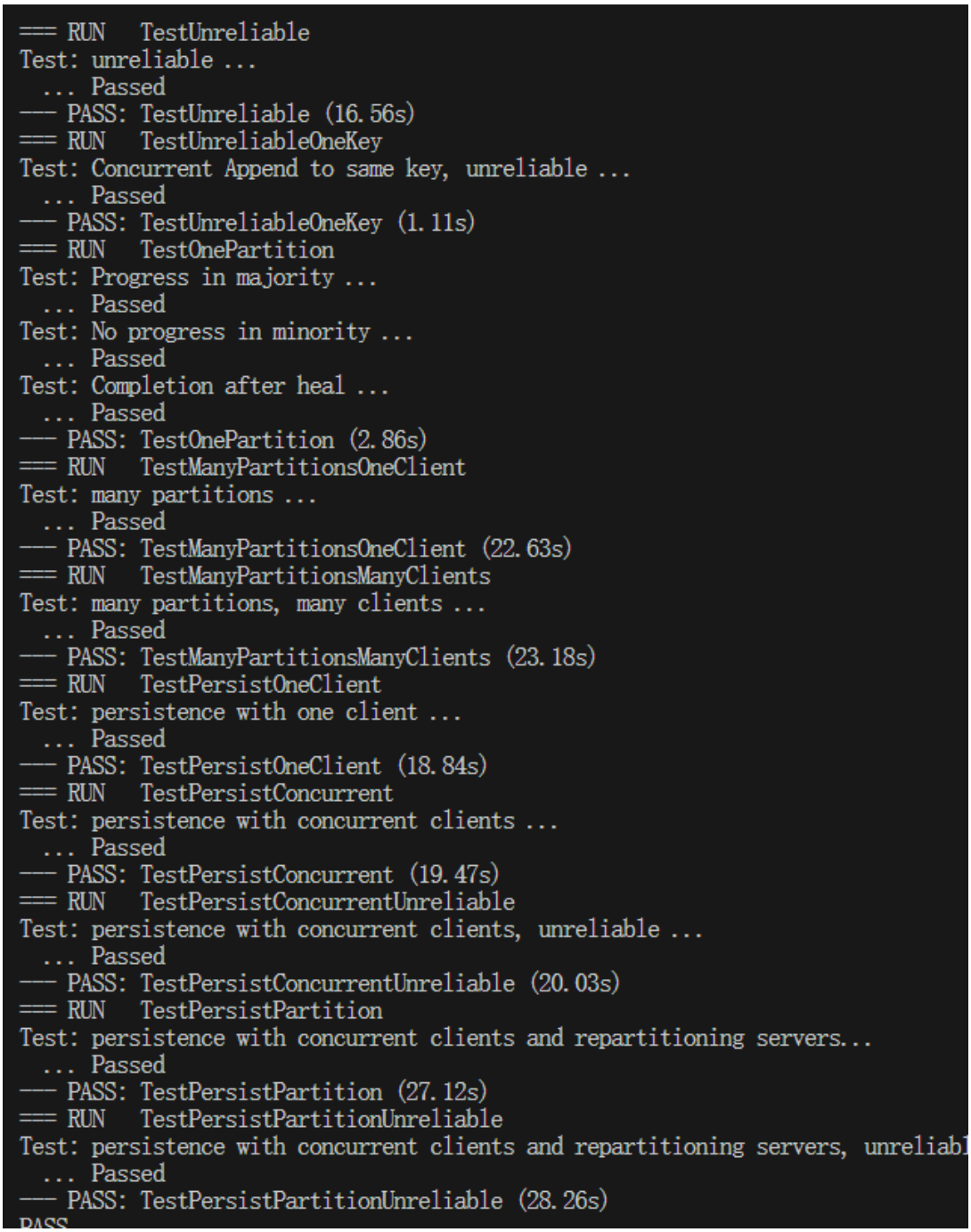

Figure 1: Complete test execution showing all test cases passing. The implementation handles concurrent clients, network partitions, message loss, and server restarts while maintaining consistency and exactly-once semantics.