SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING
ADDIS ABABA INSTITUTE OF TECHNOLOGY, ADDIS ABABA UNIVERSITY

Topic: Consistency with Sockets, RPC, and Message Passing

Objective: The goal of this lab is to familiarize you with consistency

Activity 1: Eventual Consistency with Message Passing

Objective: Simulate eventual consistency where replicas propagate updates using message-passing over TCP sockets.

Step-by-Step Instructions:

replica_eventual.go

```
package main

import (
	"bufio"
	"fmt"
	"net"
	"os"
	"strings"
	"sync"
	"time"
)

type Replica struct {
	data  map[string]string
	mu    sync.Mutex
	peers []string // List of peer addresses
}

func (r *Replica) Update(key, value string) {
	r.mu.Lock()
	defer r.mu.Unlock()
	r.data[key] = value
}
```

```
func (r *Replica) propagateUpdates(key, value string) {
	for _, peer := range r.peers {
		go func(peer string) {
			conn, err := net.Dial("tcp", peer)
			if err != nil {
				fmt.Println("Error connecting to peer:", peer,
err)
				return
			}
			defer conn.Close()
			fmt.Fprintf(conn, "%s:%s\n", key, value)
		}(peer)
	}
}

func handleConnection(conn net.Conn, replica *Replica) {
	defer conn.Close()
	reader := bufio.NewReader(conn)
	for {
		message, err := reader.ReadString('\n')
		if err != nil {
			break
		}
		parts := strings.Split(strings.TrimSpace(message), ":")
		if len(parts) == 2 {
			replica.Update(parts[0], parts[1])
		}
	}
}

func main() {
	if len(os.Args) < 3 {
		fmt.Println("Usage: go run replica_eventual.go
<machine_ip:port> <peer1_ip:port> [<peer2_ip:port>...]")
		return
	}

	// Parse command-line arguments
	machineAddr := os.Args[1]
	peers := os.Args[2:]

	// Initialize the replica
	replica := &Replica{
```

```
        data:  make(map[string]string),
        peers: peers,
    }

    // Start the server
    listener, err := net.Listen("tcp", machineAddr)
    if err != nil {
        panic(err)
    }
    defer listener.Close()
    fmt.Printf("Replica listening on %s\n", machineAddr)

    go func() {
        for {
            conn, err := listener.Accept()
            if err != nil {
                continue
            }
            go handleConnection(conn, replica)
        }
    }()

    // Simulate an update
    replica.Update("key1", "value1")
    replica.propagateUpdates("key1", "value1")

    time.Sleep(5 * time.Second) // Wait for updates to propagate
    replica.mu.Lock()
    fmt.Println("Replica Data:", replica.data)
    replica.mu.Unlock()
}
```

Instructions for Testing:

1. Start multiple replicas on different ports and machines (replace <IP:PORT> with actual values):

   go run replica_eventual.go localhost:8000 localhost:8001 localhost:8002 go run replica_eventual.go localhost:8001 localhost:8000 localhost:8002 go run replica_eventual.go localhost:8002 localhost:8000 localhost:8001

2. Simulate a client connecting to any replica to observe eventual consistency.

Exercise 1: Observe Convergence Time

1. Modify the delay in the propagateUpdates function to simulate different network conditions.
2. Measure how long it takes for replicas to converge to the same state after an update.

Hint:

- Use Go's time.Now() and time.Since() to log propagation times.

Activity 2: Strong Consistency with Message Passing and RPC

Objective:

Enforce strong consistency using **message passing** for broadcasting updates and **RPC** for acknowledgments, with replicas accepting machine IP and port as input.

replica_strong.go

```
package main

import (
	"fmt"
	"net"
	"net/rpc"
	"os"
	"sync"
)

type Replica struct {
	data    map[string]string
	mu      sync.Mutex
	peers   []string // List of peer addresses
	ackLock sync.Mutex
	acks    map[string]int // Track acknowledgments
}

type Args struct {
```

```
	Key   string
	Value string
}

func (r *Replica) Update(args *Args, reply *bool) error {
	r.mu.Lock()
	defer r.mu.Unlock()
	r.data[args.Key] = args.Value
	*reply = true
	return nil
}

func (r *Replica) propagateUpdates(key, value string) {
	r.ackLock.Lock()
	r.acks[key] = 0
	r.ackLock.Unlock()

	for _, peer := range r.peers {
		go func(peer string) {
			client, err := rpc.Dial("tcp", peer)
			if err != nil {
				fmt.Println("Error connecting to peer:", peer,
err)
				return
			}
			defer client.Close()

			args := &Args{Key: key, Value: value}
			var reply bool
			err = client.Call("Replica.Update", args, &reply)
			if err == nil && reply {
				r.ackLock.Lock()
				r.acks[key]++
				r.ackLock.Unlock()
			}
		}(peer)
	}
}

func (r *Replica) waitForAcknowledgments(key string, required int) {
	for {
		r.ackLock.Lock()
		if r.acks[key] >= required {
```

```
                        r.ackLock.Unlock()
                        break
                }
                r.ackLock.Unlock()
        }
}

func main() {
        if len(os.Args) < 3 {
                fmt.Println("Usage: go run replica_strong.go
<machine_ip:port> <peer1_ip:port> [<peer2_ip:port>...]")
                return
        }

        // Parse command-line arguments
        machineAddr := os.Args[1]
        peers := os.Args[2:]

        // Initialize the replica
        replica := &Replica{
                data:  make(map[string]string),
                peers: peers,
                acks:  make(map[string]int),
        }

        rpc.Register(replica)

        // Start the RPC server
        listener, err := net.Listen("tcp", machineAddr)
        if err != nil {
                panic(err)
        }
        defer listener.Close()
        fmt.Printf("Replica RPC Server listening on %s\n", machineAddr)

        go func() {
                for {
                        conn, err := listener.Accept()
                        if err != nil {
                                continue
                        }
                        go rpc.ServeConn(conn)
                }
```

```
    }()

    // Simulate a strong consistency update
    key, value := "key1", "value1"
    replica.Update(&Args{Key: key, Value: value}, nil)
    replica.propagateUpdates(key, value)
    replica.waitForAcknowledgments(key, len(replica.peers))

    fmt.Println("Update committed after receiving acknowledgments")
}
```

Instructions for Testing:
1. Start multiple replicas on different ports (e.g., localhost:8000, localhost:8001, localhost:8002):

```
go run replica_strong.go localhost:8000 localhost:8001
localhost:8002
go run replica_strong.go localhost:8001 localhost:8000
localhost:8002
go run replica_strong.go localhost:8002 localhost:8000
localhost:8001
```

2. Observe the output to ensure updates are applied after acknowledgment.

Exercise 1: Add Logging

1. Add detailed logs to show:
    - When an update is initiated.
    - Which replicas have acknowledged the update.
    - When the update is finally committed.
2. Analyze the sequence of events.

Hint:

- Use fmt.Printf or Go's log package for structured logging.

Exercise 2: Add Quorum-Based Acknowledgments

1. Modify the code to implement quorum-based consistency:
    - Define a quorum Q (e.g., majority of replicas).

- Commit an update after receiving acknowledgments from Q replicas instead of all replicas.

2. Experiment with different quorum sizes and observe the impact on consistency and performance.

Hint:

- Replace `len(replica.peers)` in `waitForAcknowledgments` with a configurable Q.

Activity 3: Numerical Consistency with Message Passing

replica_numerical.go

```
package main

import (
	"fmt"
	"math"
	"net"
	"strings"
	"sync"
)

type Replica struct {
	value float64
	mu    sync.Mutex
	peers []string // List of peer replica addresses
}
```

```
func (r *Replica) Update(newValue, delta float64) bool {
	r.mu.Lock()
	defer r.mu.Unlock()
	if math.Abs(newValue-r.value) <= delta {
		r.value = newValue
		return true
	}
	return false
}

func (r *Replica) propagateUpdates(delta float64) {
	for _, peer := range r.peers {
		go func(peer string) {
			conn, err := net.Dial("tcp", peer)
			if err != nil {
				fmt.Println("Error connecting to peer:", peer,
err)
				return
			}
			defer conn.Close()

			r.mu.Lock()
			message := fmt.Sprintf("%.2f\n", r.value)
			r.mu.Unlock()
			conn.Write([]byte(message))
		}(peer)
```

```
        }
}

func handleConnection(conn net.Conn, replica *Replica, delta float64) {
        defer conn.Close()
        for {
                buffer := make([]byte, 1024)
                n, err := conn.Read(buffer)
                if err != nil {
                        break
                }
                newValue := strings.TrimSpace(string(buffer[:n]))
                var value float64
                fmt.Sscanf(newValue, "%f", &value)
                replica.Update(value, delta)
        }
}

func main() {
    if len(os.Args) < 3 {
                fmt.Println("Usage: go run replica_numerical .go
<machine_ip:port> <peer1_ip:port> [<peer2_ip:port>...]")
                return
        }

      peers := os.Args[2:]

        replica := &Replica{
```

```
        value: 10.0,
        peers: peers,
}

delta := 5.0
listener, err := net.Listen("tcp", ":8000")
if err != nil {
        panic(err)
}
defer listener.Close()

fmt.Println("Replica listening on port 8000")

go func() {
        for {
                conn, err := listener.Accept()
                if err != nil {
                        continue
                }
                go handleConnection(conn, replica, delta)
        }
}()

// Simulate an update
replica.value = 12.0
replica.propagateUpdates(delta)
```

```
        fmt.Println("Replica Value:", replica.value)
}
```

Exercise 1: Support Dynamic δ

1. Modify the code to accept δ as a command-line argument.
2. Experiment with different δ values to observe how the system behaves.

Hint:

- Parse δ from os.Args.

Objective: Use message passing to maintain numerical deviation between replicas within a threshold.

1. **Handle RPC Errors Gracefully**.
2. Implement **timeouts** for long-running RPC calls to avoid blocking clients.

Client Code with Timeout Handling:

```
package main

import (
  "fmt"
  "log"
  "net/rpc"
  "time"
)

// Args holds the arguments for arithmetic operations
type Args struct {
  A, B int
}

func main() {
  client, err := rpc.Dial("tcp", "localhost:1234")
  if err != nil {
     log.Fatal("Error connecting to RPC server:", err)
  }

  args := Args{A: 10, B: 0} // Division by zero to trigger an error
  var reply int
```

```
  // Call RPC method with a timeout
  call := client.Go("Calculator.Divide", &args, &reply, nil)
  select {
  case <-call.Done:
    if call.Error != nil {
      log.Println("RPC error:", call.Error)
    } else {
      fmt.Printf("Result: %d\n", reply)
    }
  case <-time.After(2 * time.Second):
    log.Println("RPC call timed out")
  }
}
```

Activity 4: Exercise – Extending the RPC Calculator for Persistent State

Objective:

In this exercise, students will:

1. Extend the calculator to **maintain state** (e.g., store the result of the last operation).
2. Implement **a method to retrieve the last result** from the server.

Exercise Steps:

1. Add a **stateful method** to store the last result on the server.

```
type Calculator struct {
  lastResult int
  mu       sync.Mutex
}

func (c *Calculator) GetLastResult(args *Args, reply *int) error {
  c.mu.Lock()
  defer c.mu.Unlock()
  *reply = c.lastResult
  return nil
}
```

2. Modify the **client** to call GetLastResult after performing operations.

Submission Requirements:

1. **Code Submission**:
    - Submit code for all **activity that addresses the exercise** .
2. **Testing Evidence**:
    - Provide **screenshots** showing multiple clients interacting with the server.