Case Study- Java RMI

1. Basic Structure

```
// Remote Interface
import java.rmi.*;
public interface Calculator extends Remote {
     int add(int a, int b) throws RemoteException;
     int subtract(int a, int b) throws RemoteException;
 }
```

2. Implementation

```
// Remote Implementation
import java.rmi.server.UnicastRemoteObject;
public class CalculatorImpl extends UnicastRemoteObject implements Calculator {
public CalculatorImpl() throws RemoteException {
  super();
  }
public int add(int a, int b) throws RemoteException {
     System.out.println("Adding " + a + " and " + b);
     return a + b;
     }
 public int subtract(int a, int b) throws RemoteException {
   System.out.println("Subtracting " + b + " from " + a);
    return a - b;
    }
 }
```

3. Server Setup:

```
// RMI Server import
java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
    public class CalculatorServer {
            public static void main(String[] args) {
                 try {
                 // Create and export the remote object
                    Calculator calculator = new CalculatorImpl();
                    // Create and start RMI registry on port 1099
                    Registry registry = LocateRegistry.createRegistry(1099);
                    // Bind the remote object to a name
                    registry.bind("CalculatorService", calculator);
                    System.out.println("Calculator Server is ready");
                    } catch (Exception e) {
                           System.err.println("Server exception: " +
e.toString());
                           e.printStackTrace();
                           }
                      }
                }
```

4. Client Setup:

```
// RMI Client
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
public class CalculatorClient {
 public static void main(String[] args) {
  try { // Get registry
    Registry registry = LocateRegistry.getRegistry("localhost",  1099);
    // Look up the remote object
    Calculator calculator = (Calculator) registry.lookup("CalculatorService");
    // Invoke remote methods
    System.out.println("3 + 5 = " + calculator.add(3, 5));
    System.out.println("10 - 4 = " + calculator.subtract(10, 4));
    } catch (Exception e) {
    System.err.println("Client exception: " + e.toString());
e.printStackTrace();
    }
   }
 }
```

4. Advaneced Features Example:

```
// Enhanced Calculator Interface with callbacks
public interface AdvancedCalculator extends Remote {
  void calculateAsync(int a, int b, CalculationCallback callback) throws
RemoteException;
  }
  // Callback Interface
  public interface CalculationCallback extends Remote {
  void onResult(int result) throws RemoteException;
  void onError(String error) throws RemoteException;
  }
  // Implementation with thread pool
  public class AdvancedCalculatorImpl extends UnicastRemoteObject implements
AdvancedCalculator {
  private ExecutorService executorService;
  public AdvancedCalculatorImpl() throws RemoteException {
   super();
   executorService = Executors.newFixedThreadPool(10);
   }
   public void calculateAsync(int a, int b, CalculationCallback callback) throws
RemoteException {
   executorService.submit(() -> { try { Thread.sleep(1000);
   // Simulate long calculation
   callback.onResult(a + b);
   } catch (Exception e) {
   try {
   callback.onError(e.getMessage());
    } catch (RemoteException re) {
    re.printStackTrace();
    }
    }
    });
   }
  }
```

6. Security Implementation:

```
// Security Manager Setup
public class SecureCalculatorServer {
    public static void main(String[] args) {
       if (System.getSecurityManager() == null) {
           System.setSecurityManager(new SecurityManager());
      } try {
       Calculator calculator = new CalculatorImpl();
        Registry registry = LocateRegistry.createRegistry(1099);
        registry.bind("SecureCalculatorService", calculator);
         System.out.println("Secure Calculator Server is ready");
     } catch (Exception e) {
        System.err.println("Server exception: " + e.toString());
       e.printStackTrace();
      }
    }
  }
  // security.policy file
  grant {
    permission java.security.AllPermission;
  };
```

7. Exception Handling:

```
 public class RobustCalculatorImpl extends UnicastRemoteObject implements
Calculator {
    public int add(int a, int b) throws RemoteException {
       try {
       // Input validation
       if (a > Integer.MAX_VALUE - b) {
       throw new ArithmeticException("Integer overflow");
        }
        // Logging
        Logger.getLogger("CalculatorService").log( Level.INFO, "Adding numbers:
" + a + ", " + b );
        return a + b;
     } catch (Exception e) {
       Logger.getLogger("CalculatorService").log( Level.SEVERE, "Error in add
operation", e );
       throw new RemoteException("Calculation failed", e);
       }
    }
  }
```

8. Monitoring and Management:

```
public class MonitoredCalculatorImpl extends UnicastRemoteObject implements
Calculator {
   private AtomicInteger operationCount = new AtomicInteger(0);
   private Map<String, Long> operationTimes = new ConcurrentHashMap<>();
   public int add(int a, int b) throws RemoteException {
   long startTime = System.currentTimeMillis();
   try {
    int result = a + b;
    operationCount.incrementAndGet();
    operationTimes.put( "add_" + System.currentTimeMillis(),
System.currentTimeMillis() - startTime );
    return result;
    } catch (Exception e) {
     throw new RemoteException("Add operation failed", e);
     }
    }
     public Map<String, Object> getStatistics() throws RemoteException
{  Map<String, Object> stats = new HashMap<>();
     stats.put("totalOperations", operationCount.get());
     stats.put("operationTimes", operationTimes);
     return stats;
     }
    }
```

9. Client-side Load Balancing:

```
public class LoadBalancedCalculatorClient {
  private List<Calculator> calculators;
  private AtomicInteger currentServer = new AtomicInteger(0);
  public LoadBalancedCalculatorClient(String[] hosts) throws Exception {
    calculators = new ArrayList<>();
    for (String host : hosts) {
     Registry registry = LocateRegistry.getRegistry(host, 1099);
     Calculator calculator = (Calculator) registry.lookup("CalculatorService");
     calculators.add(calculator);
     }
   }
     public int add(int a, int b) throws RemoteException {
      int serverIndex = currentServer.getAndIncrement() % calculators.size();
      return calculators.get(serverIndex).add(a, b);
      }
      }
```

This case study demonstrates various aspects of Java RMI implementation, including basic setup, advanced features, security, exception handling, monitoring, and load balancing. It provides a practical example of how to build distributed applications using Java RMI.