Mini RPC Framework

Team Members:
- Helly Gandhi
- Kush Gandhi
- Ruslan Radetskiy
- Aditya Prakash

1. Project Overview

This project implements a Mini Remote Procedure Call (RPC) Framework in C. The objective is to allow a client program to invoke functions that execute on a remote server as if they were local calls. The framework abstracts low-level networking details and provides a structured mechanism for message serialization, dynamic function resolution, and concurrent request handling.

The system follows a modular client–server architecture. Server-side application functions are loaded dynamically at runtime using shared libraries. Demo programs are provided to demonstrate and validate the complete workflow of the RPC framework.

2. System Architecture

The framework follows a client–server architecture with a modular design.

Main Components

- Protocol Layer (protocol.c/.h)
  - Defines message headers
  - Handles serialization and deserialization of primitive data
  - Sends and receives messages over sockets

- Message Handling (message_handler.c/.h)
  - Converts RPC requests and responses to/from byte buffers
  - Encodes function name and parameters

- Server Core (server.c/.h)
  - Initializes TCP socket
  - Accepts client connections
  - Uses one thread per client
  - Provides send/receive wrappers

- RPC Server (rpc_server.c/.h)
  - Routes incoming RPC requests
  - Looks up functions by name
  - Executes functions dynamically
  - Returns results or error messages

- Dynamic Function Loader (dl_handler.c/.h)
  - Uses dlopen() and dlsym() to load functions at runtime
  - Maintains a registry of function names and pointers

- Client Core (client.c/.h)
  - Manages client socket connection
  - Sends requests and receives responses

- RPC Client (rpc_client.c/.h)
  - Provides high-level RPC call interface
  - Handles request creation and response parsing

3. Dynamic Function Loading

A key feature of the framework is runtime function loading using shared libraries.

- The server loads a shared object (libexample.so) at startup
- Functions are registered by name
- Clients can invoke these functions without recompiling the server

This design separates the RPC framework from application logic, making the system extensible.

4. Communication Flow

Client Request Flow
- Client connects to the server
- Client serializes function name and parameters
- Request is sent over TCP
- Client waits for response
- Response is deserialized and returned

Server Processing Flow
- Server accepts client connection
- Server spawns a worker thread
- Request is received and deserialized
- Function is looked up in registry
- Function is executed
- Result is serialized and sent back

5. Concurrency Model

- The server uses one thread per client
- Threads are detached to avoid memory leaks
- Mutexes protect shared resources (function registry)
- This model is simple and suitable for moderate workloads

6. Error Handling

The framework handles common error cases:
- Invalid function name
- Missing shared library
- Client disconnection
- Socket failures
- Serialization errors

Errors are returned to the client as structured RPC responses.

7. Build and Run Instructions

Requirements
- GCC or Clang
- POSIX-compatible OS (Linux / macOS)
- pthread support

Build the Project

make clean
make

Build the Shared Library (Required for Demo)

gcc -shared -fPIC -o bin/libexample.so src/example_functions.c

Run the Server (Terminal 1)

make run-server

Run the Client (Terminal 2)

make run-client

8. Usage Guidelines

- The server must be running before the client
- The shared library must exist before starting the server
- Multiple clients can connect sequentially or concurrently
- Requests for non-existent functions return an error response

9. Project Structure

```
RPC-framework/
src/
 ├── client.c            # Low-level client socket operations
 ├── rpc_client.c        # RPC client implementation
 ├── demo_client.c       # Demo client for testing RPC calls
 ├── server.c            # Low-level server socket and threading logic
 ├── rpc_server.c        # RPC server logic and RPC dispatch
 ├── demo_server.c       # Demo server application
 ├── protocol.c          # Network protocol implementation
 ├── message_handler.c   # Message serialization/deserialization
 ├── dl_handler.c        # Dynamic library loading and function registry
 ├── example_functions.c # Example RPC-callable functions
└── message_test.c       # Unit tests for message handling
include/
 ├── client.h            # Client-side socket interface
 ├── server.h            # Server-side socket interface
 ├── rpc_client.h        # RPC client API
 ├── rpc_server.h        # RPC server API
 ├── protocol.h          # Message protocol definitions
 ├── message_handler.h   # Serialization/deserialization interfaces
└── dl_handler.h         # Dynamic loader and registry interfaces
bin/
 ├── rpc_server
 ├── rpc_client
└── libexample.so
 ├── Makefile
└── README.md
```

10. Testing

Testing was performed using:

- Demo client and server programs
- Invalid function calls
- Multiple sequential client connections
- Manual stress testing with repeated calls

11. Team Contributions

Helly Gandhi

- Implemented all client-side components
- client.c, rpc_client.c, demo_client.c
- Client RPC logic, request handling, and testing

Kush Gandhi

- Implemented all server-side components
- server.c, rpc_server.c, demo_server.c
- Multi-threaded server logic and RPC dispatch handling

Ruslan Radetskiy

- Implemented message serialization and deserialization
- message_handler.c/.h, dl_handler.c/.h
- Created message_test.c for validating message handling

Aditya Prakash

- Designed and implemented the communication protocol
- protocol.c/.h
- Created the Makefile and project README

12. Limitations

- Only basic data types supported
- Blocking I/O model
- Thread-per-client limits scalability
- No authentication or encryption

13. Conclusion

This project successfully demonstrates the implementation of a functional RPC framework using low-level operating system primitives. The framework highlights practical use of sockets, threads, synchronization, and dynamic linking. While simplified, it provides a solid foundation for understanding distributed systems and RPC mechanisms.