RPC-based chat

# Overview

In this exercise, we implemented a client-server RPC-based chat using rpcgen, in C. Following is the diagram from which we based our approach :

Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement

RPC, or remote procedure call, is a client-server driven technology that allows for an easier implementation of distributed systems. It does so by offering a way for the developers to implement a procedure that aims to be called from another process, and such without having to write specifically the details of the remote interaction.

## Outcome of the project :

To run the program, you need to enter the directory daniel.strens\_matteo.firenze\_ex3/ then compile the sources with ./COMPILEALL.sh and finally start the different processes, starting with ./chatapp\_server on a terminal, and ./chatapp\_client localhost <name> on other terminals.

Une image contenant capture d’écran, texte, logiciel, Logiciel multimédia

Description générée automatiquement

The different clients can now chat !  
Une image contenant texte, logiciel, Logiciel multimédia, Logiciel de graphisme

Description générée automatiquement

## Overview of the files in the project

Une image contenant texte, capture d’écran, Police

Description générée automatiquementHere are the different files contained in this project :

chatapp.x : contains the definition of the program.

chatapp.h : header file generated by rpcgen from chatapp.x. contains the declarations of the data types and functions defined in chatapp.x

chatapp\_xdr.c : XDR routines file generated by rpcgen

chatapp\_svc.c : server-side implementation of the procedures defined in chatapp.x

chatapp\_server : binary created from chatapp\_svc.c and chatapp\_xdr.c

chatapp\_clnt.c : client stub generated from chatapp.x. Contains the client-side implementation of the procedures.

chatapp\_client.c : Contains the main logic for the client, manages the CLI and contains the main function.

chatapp\_client : binary created from chatapp\_client.c, chatapp\_clnt.c and and chatapp\_xdr.c

chat\_history.txt : Contains the chat history

# Developing the chatapp.

## chatapp.x

Here is the code inside the chatapp.x file :   
Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement

The most important part of this file is the program CHATAPP {} section. This defines a remote program named CHATAPP with a version VER1. The program has two procedures: get\_chat\_history and send\_message. get\_chat\_history takes no arguments and returns a chat\_history structure. send\_message takes a message structure as an argument and returns an integer. The procedures are identified by the numbers following the equals signs. The program CHATAPP is identified by the hexadecimal number 0x3a3afeeb.

This file is written in the XDR (External Data Representation) language. It’s used by the rpcgen program to generate client and server stubs for remote procedure calls. The stubs handle the details of network communication, allowing the programmer to focus on the logic of the application.

From this file, we can generate all the necessary ones with the command : rpcgen -C chatapp.x

# Server-side

## chatapp\_svc.c

This file contains the server-side implementation of the procedures : get\_chat\_history and send\_message.

Those procedures manage the history of the chat, which is contained in the file chat\_history.txt.

Une image contenant texte, capture d’écran, Police

Description générée automatiquementHere is what the chat\_history looks like :

Une image contenant texte, capture d’écran, Police

Description générée automatiquementHere are the implementations of the procedures :

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquementUne image contenant texte, capture d’écran, Police

Description générée automatiquement

## chatapp\_server

This file is the binary created with the command : gcc -o chatapp\_server chatapp\_svc.c chatapp\_xdr.c -lnsl -lpthread -lncurses -ltirpc -I /usr/include/tirpc/

It should be executed without argument.

# Client-side

## chatapp\_clnt

This is the client stubs file generated by rpcgen. It contains the client-side implementation of the procedures defined in chatapp.x

## chatapp\_client.c

This is the file containing the logic for the client side of the program.

It is divided in two threads :

* The main thread is responsible for listening to the inputs of the user on the terminal. When a line is entered, it sends a message to the server using the following function:  
  Une image contenant texte, capture d’écran, Police, logiciel

  Description générée automatiquement
* The second thread is responsible for running the function that updates the chat, and it does so every second. Here is the function that updates the chat :  
  Une image contenant texte, Police, logiciel, capture d’écran

  Description générée automatiquement

## Point about the local and remote procedures

As shown in the previous point, the methods used by the client to call the remote procedures are send\_message\_1 and get\_chat\_history\_1. Here are the headers of the functions :

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

These functions are defined automatically by rpcgen to call the remote procedure. This means that, as developers, we can use them in our code as if they were local procedures. The remote access being abstracted from us. This greatly simplifies working with remote procedures and distributed systems.

# Limitations

The limitations of our RPC-based chat application include:

1. **Scalability**: Our current implementation stores all messages in memory and in a single file. This could become a problem if the chat history grows very large.
2. **Concurrency**: Our server reads from and writes to the chat history file for each RPC. If multiple clients are sending messages at the same time, this could lead to race conditions.
3. **Error Handling**: Our current implementation does not handle errors robustly. For example, it does not always check if malloc returns NULL, which could lead to a segmentation fault.
4. **Memory Management**: Our server allocates memory for each message but never frees it. This could lead to a memory leak if the server runs for a long time.

As for RPC itself, its main limitation is that in the traditional RPC model, the client is the one that initiates requests and the server responds to those requests. This means that in the context of a chat application, the client must actively request updates from the server to get the latest messages. This is not the case with, for example, websockets, where the communication is made in both directions.

# Conclusion

Our RPC-based chat application project has been a valuable learning experience. We’ve successfully implemented a distributed system using RPC, allowing clients to send messages and retrieve chat history from a server.

In terms of RPC, we’ve learned about its strengths, such as abstraction of network programming details and the ability to create distributed applications and we’ve also noted some of its limitations.

Overall, this project has provided us with a solid foundation in RPC and distributed systems.