MESSAGE DSITRIBUTION SERVER

Duțu Alin Călin

CONTENTS

1 INTRODUCTION

1.1 Context

The message distribution server is a network application designed to facilitate the transmission of messages between a server and multiple clients over a network. These apps are using two fundamental communication protocols:

- TCP, known for its reliable, ordered, and error-checked data delivery
- UDP, which provides faster but less reliable communication

These applications are essential for various use cases, such as real-time messaging, gaming, live streaming, and any scenario requiring efficient data exchange between networked devices.

1.2 Objectives

This program aims to create an application using the client-server model to handle message distribution between different client types. The following objectives are covered upon implementing the app:

- Understand the mechanism behind UDP and TCP connections
- Understand the multiplexing mechanism
- Define and use a data protocol over UDP
- Develop a client-server app using sockets

2 PROPOSED SOLUTION

This project aims to develop multiple components of a message distribution app over a predefined transport protocol to ensure message transfers and interpret information from the upcoming messages in the provided format.

2.1 Architecture

Looking from a general point of view, there can be 3 entity categories that perform different actions and interactions to ensure message distribution:

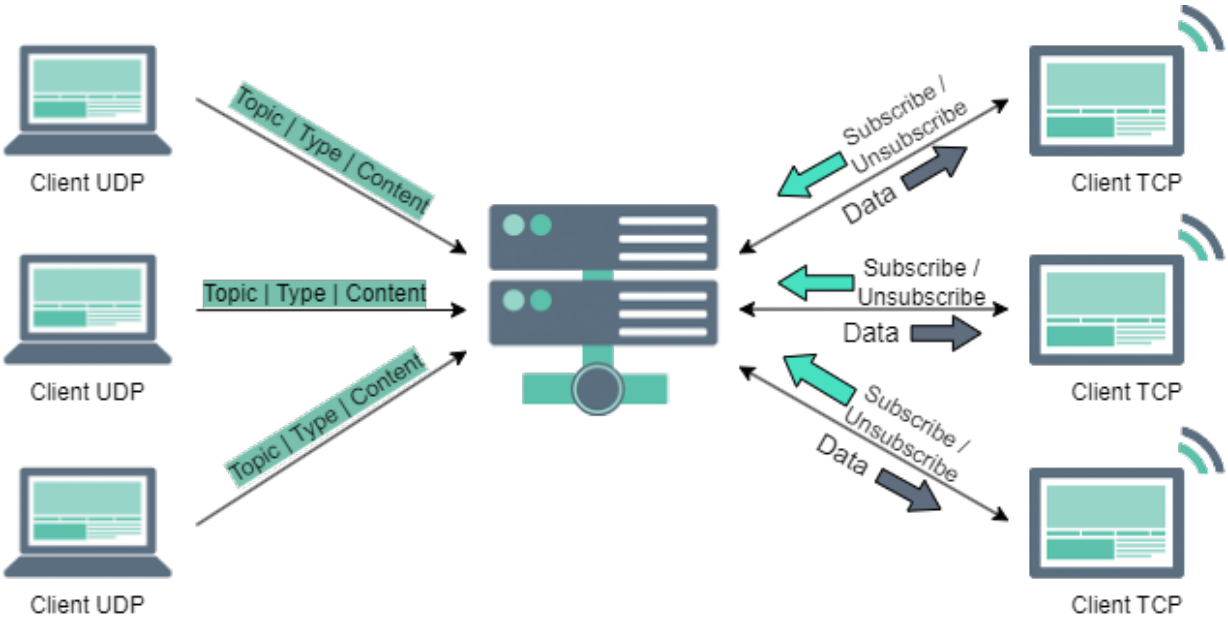

Figure 1: Application architecture [1]

The UDP clients, provided by the PCOM team[2], will publish messages for specific topics to the server respecting the format provided by the predefined protocol.

The server, which runs using a program developed in C language, is the message broker. It listens and establishes connections with the UDP and TCP clients, receives and processes

[1] `https://www.flaticon.com/icons`. Icons created by RawPixel
[2] Communication Protocols team @ Polytechnic University of Bucharest

messages from UDP clients, manages TCP client subscriptions on message topics provided by the UDP clients, and sends new data to the TCP clients when the topics get a new update.

The TCP client is another C program that accepts commands from the keyboard and sends requests to the server depending on the command type. The client aims to subscribe to a topic of interest and receive message updates regarding the topic from the broker.

2.2 Communication Protocol

The communication between the UDP clients and the broker server will be made using the UDP transport protocol and will respect the following format:

	Topic	**Data Type**	**Content**
Dimension	50 bytes	1 byte	Maximum 1500 bytes
Format	A char array of 50 bytes that ends with \0 or datagram ending	An unsigned int used to specify the content data type	Variable-based data type

Table 1: UDP Client message format

This message format transmits updates on specific topics provided by the UDP clients.

The TCP clients are required to send Subscribe/Unsubscribe requests to their topics of interest to receive updates from the broker. The confirmation of a successful command comes as an output message to the client after the message exchange with the broker ends.

When clients are subscribed to a topic, every time a UDP client sends an update to the topic the TCP client has subscribed to, the TCP client will receive it as well. Every update received by the TCP client will be shown in the program's output in the following format:

```
<UDP_CLIENT_IP>:<UDP_CLIENT_PORT> - <TOPIC> - <DATA_TYPE> - <CONTENT>
```

Value	**Data type**	**Content format**
0	INT	| Number sign [3] (`byte`) | Number (`uint_32t`) |
1	SHORT_REAL	Number modulus that was multiplied by 100 (`uint16_t`)
2	FLOAT	| Number sign [3] (`byte`) | Integer and fractional parts of the number concatenated and in modulus (`uint32_t`) | Fractional part's number of digits (`uint_8t`) |
3	STRING	Char array (1500 chars max) delimited by datagram ending or \0

Table 2: TCP Client output format

[3] The number sign will be 0 for positive numbers and 1 for negative numbers

3 IMPLEMENTATION DETAILS

The implementation mainly consists of 2 applications implemented in C language, one for the server and one for the TCP client.

3.1 Server

The server that serves as a broker in the program has the following functionalities:

- new_client - creates a new client
- add_client - Registers the newly created client in the list of clients
- $get\_client\_by\_id$ - Searches for a client based on id
- $get\_client\_by\_socket$ - Searches for a client based on his connection socket
- $add\_subscriber$ - Adds a client to the subscriber's list of a given topic
- $delete\_subscriber$ - Removes a client from the subscriber's list of a given topic
- add_topic - Creates a new topic
- get_topic - Returns a topic from the list of topics

At initialization, the server sets up all its internal structures and then it creates UDP and TCP sockets for setting up connections with new clients, making sure to register every client in its internal structures.

Besides the main functionality, the server can also read commands from the keyboard. However, the only command available in this implementation is the exit command which is sufficient for the project scope. The exit command sends connection-closing signals to all clients, closes all the sockets to ensure connections are completely closed on the client side, and finishes the execution of the program itself.

There are 5 main events that trigger the server to take action: a connection request from a UDP client, a message received from a UDP client, a connection request from a TCP client, a message received from a TCP client, and client disconnection.

A connection request from a UDP client starts the process of setting up all the parameters and structures that register the UDP client on the server.

When the server receives a message from a UDP client, the information will be interpreted in a *UDP_msg* structure and registered on the specific topic. If the requested topic doesn't exist, it will be created and then the request will be processed. Next, the server computes a new message based on the information type and sends it to the subscribed TCP clients.

A connection request from a TCP server is going to, firstly, trigger checks for connection collisions based on the client ID and determine whether the client connects for the first time, reconnects, or creates a connection collision which determines a refused connection. Any new connection is logged in the program's output. An important note should be made when a client reconnects as he will be resubscribed to his previously subscribed topics. Moreover, if the client has opted for the Store-and-forward option, an option for each topic subscription, he will receive all the missed messages.

When a TCP client message is received, it means it is a subscribe/unsubscribe request, so the server extracts message data into a textitTCP_sub_msg structure and processes it to execute the desired action. Additionally, if the broker doesn't find any topic with the specified name, it will be created on the spot.

When a client disconnects, the server closes the connection and makes the necessary changes in the client's saved data to ensure continuity when he reconnects. This action is going to be logged in the program's output.

3.2 TCP client

The TCP client at initialization synchronizes with the broker through a handshake-like protocol that establishes a connection, followed by sending the client ID to the broker.

The TCP client supports 3 commands that can be sent using the keyboard:

- The *exit* command closes the socket and the program stops
- The *subscribe* command creates a new *TCP_sub_msg* message with the command, topic name, and the Store-and-Forward field (SF)
- The *unsubscribe* creates a similar request as the *subscribe* command, but the unsubscribe command set

.

Any created request is going to be sent to the broker. There are different events that trigger a broker's response and based on that, a broker's response can be interpreted in 3 ways:

- *Exit command* - It will close the socket and the program
- *Store and forward message* - It will write a log in the program's output for every message the server saves.
- *Normal message* - Based on the data type it will write specific logs in the program's output

4 CONCLUSIONS

In conclusion, this project proposes a message distribution server designed to facilitate the transmission of messages between a server and multiple clients over a network using the basic transport protocols used in networking, the UDP and TCP protocols, to understand the main features and the flows on practical examples that resembles the MQTT protocol[1], a standard used in IOT industry.

[1]MQTT - The Standard for IoT Messaging. https://mqtt.org/. Latest access: 25.06.2024