CSCI 5105

Introduction to Distributed Systems

Spring - 2020

Design Document

PubSub System

Date: 02/19/2020

Team:

Soumya Agrawal (agraw184)
Garima Mehta (mehta250)
Bhaargav Sriraman (srira048)

TABLE OF CONTENTS

Project Implementation:

Our implementation consists of two packages: Client and Server. The github repo link: https://github.umn.edu/agraw184/pubsub

Client:

The client-side functionality is implemented in the Client package, with two files, Client.java and PublishedClient.java. Client.java handles the RPC connection to the server, where it joins a server, subscribes, unsubscribes and/or publishes an article, and leaves a server. This also handles the connection to the Registry Server to get the list of active servers for the client to connect. PublishedClient.java runs on a separate thread that handles the UDP connection between the client and the server. It receives the published article from the server.

Server:

The server-side functionality is written in the Server package, with Communicate.java (interface), CommunicateImpl.java which has the core functionality of the publisher-subscriber system, CommunicateHelper.java for the helper functions to validate requests, and connect to the Registry Server, GroupServer.java, which registers itself to the registry server and helps in propagating the publisher-subscriber system by using the client's request, GroupServerHeartbeat.java, which handles the connection with the Registry Server. If the registry server does not receive a heartbeat from the group server for 5 seconds, it deregisters the server automatically.

The system built is multithreaded, wherein there are separate threads for the client to server RPC connection, registry server to server UDP connection, server to client UDP connection, and a thread between registry server and client, which handles the sending and receiving of messages.

How to run:

To run the project, there is a file called **artifacts.zip** which contains all the jar files required. There is one jar file called **pubsub_srv.jar,** which runs the Group Server instance. It can be run on several machines.
Then there are multiple client jars, each of which has a defined role, done specifically for testing different scenarios (more details in the Clients section). The filenames are **pubsub_clnt.jar, pubsub_clnt1.jar, pubsub_clnt2.jar, pubsub_clnt3.jar.** We strongly recommend running these jars in the Keller CSE labs.

Now, the first step is to ensure that the RMI registry (Registry Server) is up and running on one of the machines.
Next step would be to run the above-mentioned jars, firstly, run the pubsub_srv.jar, using the following command:
java -jar pubsub_srv.jar <*public ip addr of registry server*>
Similarly, run the clients jar using the command:
java -jar pubsub_clnt.jar <*public ip addr of registry server*>
java -jar pubsub_clnt1.jar <*public ip addr of registry server*>
java -jar pubsub_clnt2.jar <*public ip addr of registry server*>
java -jar pubsub_clnt3.jar <*public ip addr of registry server*>

You can see the heartbeat transfer with messages on the console to ensure it is running smoothly. Also, logs of each client and server are printed on their console, which is self-explanatory.

Note:
The project was developed in our local machines using IntelliJ. While developing, we used the local IPs of the server and remote server to make UDP and RMI calls. But these IPs do not work in the CSE lab machines. We are looking up "http://checkip.amazonaws.com/" to generate our public IP. If you wish to run the files in the local machine, localhost needs to be passed rather than public IP.

About the Classes

1. GroupServer

The GroupServer does the following:

a. Register to the remote server using *CommunicateHelper.udpToRemoteServer*.
   The IP of the remote server is passed as an argument. It the sends a UDP message of the format "Register;RMI;IP;Port;BindingName;Port for RMI" to register itself in the remote server.
   Note that Public IP has to be used if hosted in the CSE lab machines and local IP has to be used if hosted in personal local machines.

b. It then binds the object of CommunicateImpl to the registry.
   Note that *createRegistry* command created a registry in the local address. But since we need to host in the public address, we need to change the system property to do so. If hosted in personal local machines, we should skip this step.

c. If there is a timeout on the server, it can deregister after n second.
This part of the code is currently commented out. In order to test it, please uncomment and set the value for the scheduler. It is initially set to 60 seconds.

2. CommunicationImpl

a. Ping()

It prints "I am alive" in the console of the server. Always returns true. It is used periodically by the client to see if the server is still running and alive. The fault tolerance technique will be explained in the client section.

b. Join()

The client joins the server using this function which accepts the **IP of the client and the port in which it expects the server to respond**, (when publishing UDP messages).
This updates the *clientList* of the server with the client IP and the port number. If the number of clients in it exceeds MAX_CLIENTS, it will not let the client join the server. Returns **true** if successful.

c. Leave()

The client uses this function to leave from the server. It deletes the entry of the client in both, *clientList* and *clientSubscriptionList*. Return **true** once removed.

d. Subscribe()

The client uses this to subscribe to a topic. First, the server checks if it is a valid request or not. If it is, it adds to the *clientSubscriptionList* which is a *HashMap<String, ArrayList<Strings>>*. The map is indexed based on the client IP and has a list of all of the client's subscription.

e. Unsubscribe()

The client uses this function to unsubscribe to a topic. I**t is assumed that the client can only unsubscribe to the same topics that it has subscribed to before**. For eg if it subscribed to *"Sports;;UMN"* then it can't unsubscribe from *";;;UMN"*.
The server removes the corresponding entry from the *clientSubscriptionList.*

f. Publish()

Once a client publishes an article which is tagged with certain topics, the server first finds a list of clients that fall under that category.
For eg: if the published article has the tags *"Lifestyle;ABC;UMN"*, then a client subscribing to *"Lifestyle;;;"* should also get the message.
Once the list of clients is obtained, the server sends a UDP message to all of them to their corresponding IP and the port number.

3. Clients

a. Clients first contact the registry server to get a list of servers that are up.
Note that the registry server only returns a list of IPs and Ports. In order to work around it, we assumed that the binding name is always "server.comm"
The default port that the clients listen to is 1098. If the port needs to be changed, it needs to be updated in both PublisherClient and the corresponding join RMI call.

b. The client pings occasionally to see if the server is still running. If not, it gets the list from the remote server again and connects to a server which is running.

c. There are 4 clients that are created for testing purposes: Client, TestClient1, TestClient2 and TestClient3. The clients when run sequentially subscribe, unsubscribe and publish articles for the other clients to receive articles hence fulfilling the pubsub functionality. We have tested, not limited to this list,
   i. The fault tolerance technique which happens when the server fails - with and without other servers running.
   ii. UDP communication, remoteServer - Client, Server-Client, remoteServer - Server.
   iii. RMI connection to the server.
   iv. Publish with multiple clients.
   v. Publish with a client who is subscribed to a subset of the tags.
   vi. Invalid subscriptions and unsubscription.
   vii. Join when MAX_CLIENTS is reached.
   viii. When the article is longer than 120 bytes, it is trimmed
   ix. Join when no server.

d. PublishedClient is a class that receives UDP messages from the server at 1098 port.

Things to do

1. The server needs to implement a heartbeat from its side to see if its clients are still active. Currently, the clientList is only updated if the client sends a leave request.
2. More Null checks and exception handling. The code is highly likely to fail if unexpected characters or objects are sent.
3. Port number can be parametrized in PublishedClient for flexibility.