PubSub Server/Client

By Aparna Mahadevan, Ming-Hong Yang, Eric Kuha

The goal was to create a Publish/Subscribe server and client in C using Linux rpcgen. The rpcgen utility creates a set of source code files from an interface defined in the file **communicate.x** From there, we have two main components:

Server

Aside from the main interface, there are two main files.

communicate_server.c
This implements the interface for RPC calls. It handles all calls that come from clients. A client registers with the service by submitting its IP address and port number and then a series of subscriptions as strings. These are stored in a simple linked list of “SubNodes” containing their current state. Whenever a client publishes an “article”, the server creates a separate thread (to hopefully avoid race conditions) which iterates over the list and sends the article to any client that is subscribed to the article’s tags. This is conducted through a UDP socket that is set up during the server’s initialization.

communciate_svc.c
This file contains the main method. It is here that the entire RPC service is initialized as well as a thread which registers the server on the course registry-server. This thread waits for “heartbeat” pings from the registry server and returns them to remind the server that it is alive.

Client

There is one file that does almost all of the work.

communicate_client.c
This program starts three threads when it is initialized. First, is a ping thread which periodically pings the server to ensure that it is alive and able to send and receive messages. Second, is the UDP thread which waits for messages from the server in the form of articles which this client is subscribed to. When it receives one, it simply prints it to the console.

Finally, the RPC thread presents a simple text-based user interface so that the user can subscribe/unsubscribe to articles and publish articles to the service. It has an option to quit, terminating the connection to the server, joining and killing all threads, and closing the program.

Design Decisions

The way subscriptions are handled is that if a client subscribes to Science & UMN, then it will get sent all articles that are Science related, or all articles that are UMN related. The reason for this is that it takes less time to iterate through a client's subscriptions and find the first one to match one of the fields in the article, as opposed to iterating through all the subscriptions and making sure all the fields of the article are in the client's subscription list. The latter scenario is also more unlikely compared to the former, especially when there are a lot of clients, and a lot of combinations of fields. Unsubscribing is handled in a similar way - a client unsubscribing from Science & UMN will have those fields removed from its list.

Whenever an article is published by the client, the "publishing" method iterates over all the clients. For each client, it further iterates through its subscription list, and if one or more of the subscriptions matches one or more of the fields in the article, it sends the article to that client via UDP. The sockaddr_in struct is stored in that client's SubNode structure. The alternative method we thought of for publishing was to have a hashmap of key-value pairs where keys were the fields, and values were a list of clients subscribed to those fields. We would iterate through the fields in the article, and for each one, look up the corresponding key in the hashmap and iterate over the linked-list of clients that is the value. However this was not done for 2 reasons. 1) A hashmap is not easy to implement in C and 2) the collisions in the hash-map would have resulted in an O(n) lookup [this is likely, since there are many clients]. Therefore, iterating through the clients would also result in the same time complexity, O(n), for each field. Simply iterating through the clients (instead of a hashmap) also reduces the space requirement because then an additional separate data structure is not necessary to store the mapping between subscriptions and clients.

Google PubSub

The other requirement of this project was to implement a pubsub server with an existing framework, namely the Google PubSub API. There are some interesting differences between these two ways of implementing this project.

On the one hand, Google PubSub is far, far quicker to set up. We coded it in Python which is simple and straightforward. This uses the google cloud sdk and and the API defined by it is robust and functional. All you have to do is call functions. It's just feeding inputs into a little black box and the outputs come out exactly as you expect them to.

There is a downside to this, however. You have to spend some time learning the API first, which is not a problem, but no matter what you do, unless you want to really put in the hours (as we had to with rpcgen), you never really know what's going on under the hood. Sure, you have an application that works, but you don't really know *how* it works. It also leaves you highly dependent on Google’s infrastructure and services. It requires you to have a Google account and Google security credentials for authentication. This locks you into a software ecosystem that is as closed as it is easy to use.

Then again, that comes with its own benefits too. Google’s security is robust. It’s software generally works well.

The only file included is the **client.py**. It presents a simple text-based UI which requires a server be set up. It’ll work with whatever server it can currently find (I understand it does this by way of environment variables). We used the pubsub emulator locally rather than incur charges on Google’s servers. A number of methods are implemented allowing clients to join the service (by starting the application), create topics, subscribe to those topics, and receive messages when a message is published to one. It’s fairly straightforward and could be polished up a lot, but the basic functionality is all there.

How to run the program/ Test Cases

Instructions for running the program are in README.md.
The following shows a simple test case for testing our program. We use two clients to demo the subscription and publication. The sequence and the input are shown as follows.

C1: subscribe Business;;;
C2: publish Business;;;xxx
C2: publish Sports;;;xxx

C1 will only receive Business;;;xxx as a result. Followed by:

C1: unsubscribe Business;;;
C2: publish Business;;;xxxx

C1 will not receive Business;;;xxxx since it is unsubscribed.

C1: subscribe ;;;food
C1: subscribe ;;;;
C2: publish food;;;
C2: publish ;;;;

C1 cannot subscribe to either of these articles as they are of illegal format. **C2** also cannot publish those articles for the same reason.

In addition, the server responds to the heartbeats from the registry server, for which there are no testcases.