CSCi 5105: Introduction to Distributed Computing

Spring 2018

Instructor: Jon Weissman

Project 1: PubSub

Due: Feb 14 12pm (noon)

1. Overview

In this project, you will implement a simple publish subscribe system (PubSub). You will use two forms of communication: basic messages using UDP and an RPC system of your choice, either Java RMI or Linux RPC. For this reason, you may code the project in the language of your choice, though we recommend the latter. *As a third option, you are welcome to try out Apache thrift using a language of your choice (e.g. Java). We have included links to it on the additional reading list, but you are on your own with thrift.*

Your PubSub system will allow the publishing of simple formatted "articles". In this lab, you will learn about distributed computing and asynchronous communication protocols. You may assume that UDP communication is reliable. The second task is to implement a similar design using google cloud's pubsub API in a language of your choice and compare the result with your from-scratch implementation qualitatively. This lab contains a number of **optional** features that bring no extra credit, but may challenge you. This lab has a number of details to work through so get started ASAP.

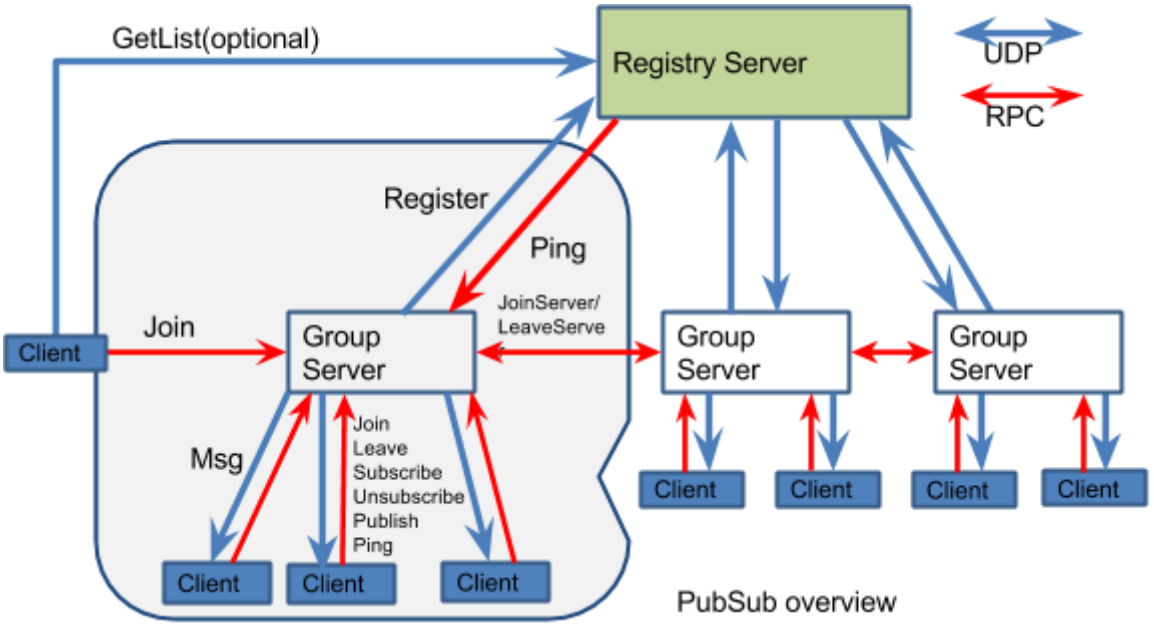
2. Project Details

The project will be comprised of: **clients**, **servers**, and a **registry-server**. You will program a client and server. Your client communicates to its server to `Subscribe` and `Publish` articles. You will deploy a server (your group server) and a set of clients. Clients communicate to their server by means of RPC/RMI to publish and subscribe. The server determines the matching set of clients for a published article, and then propagates the article to each client via UDP. Thus, you may want to multithread your client: one thread for RPCs and the other for receiving subscribed articles. The client must first `Join` to their local server when it starts up. The server will allow at most MAXCLIENT clients at any one time. A client may leave a server at any time.

Your server begins by registering with a registry-server than we will deploy at a well-known address that we will provide. You will use UDP for `Register` passing to it (IP, Port, and RPC or RMI). Server may also `Deregister` if it wishes to go "off-line". The registry stores information about all group servers. To obtain this list, you may call `GetList`, also via UDP. Your server may choose to scale out by joining another group's server and call `JoinServer` via RPC/RMI **(optional)**. To do this, your server can get the list of available group servers that speak its RMI/RPC protocol. The destination server may refuse if it is over the MAXCLIENT threshold. If server A joins to server B, then any article that is `Published` to A, should be then `Published` to B (from A), **and** vice-versa. Thus, your server must also act as a client, and may need to

be multithreaded. It may be easiest to propagate each article (to both clients and joined servers) as it is published but this is up to you. Once an article is propagated it may be garbage-collected at the server. **Optionally** you could decide to store articles for clients that `Join` later. Be mindful of the need to protect any data-structures in either the client or server due to the presence of multiple threads. One note**: you should prevent duplicate article propagation that could arise from actions of servers that are themselves joined.**

To check whether server is up, the clients **and** registry-server will send heartbeat messages (Ping) to the group server(s) periodically via RPC. The ping interval for your client is up to you. A failed RPC/RMI is detected and an error is returned by the RPC/RMI system. Decide how you will determine if the server is down. As an **optional** feature, would be to allow your client to `Join` another server upon failure. The workflow of the system is depicted below (**optional** functionality is also shown, e.g. the scale out option).



3. Implementation Details

An article is a simple formatted string with 4 fields separated by ";": type, originator, org, and contents. The type can be one of following category: <*Sports, Lifestyle, Entertainment, Business, Technology, Science, Politics, Health*>. You may assume an article is a fixed length string of length MAXSTRING (120 bytes), any padding can be done at the end of the contents. The contents field is mandatory (for Publish) and at least 1 of the first 3 fields must be present (for Subscribe) and no contents field is allowed for Subscribe. Note that only the *type* is standardize, the rest are arbitrary strings.

"Sports;; contents" → (OK for Publish only)

";Someone;UMN; contents" → (OK for Publish only)

"Science;Someone;UMN;contents" → (OK for Publish only)

"Science;;UMN;" → (OK for Subscribe – Science AND UMN only)

";;;contents" → No first three fields. (**Impossible**)

Think about how your server will *efficiently* match published articles to subscriptions.

3.1. Client → Group Server API (by RPC/RMI)

The operations which a client should be able to perform with the server are below.

a. Join (IP, Port):Register to a group server (provide client IP, Port for subsequent UDP communications)
b. Leave(IP, Port): Leave from a group server
c. Subscribe (IP, Port, Article): Request a subscription to group server
d. Unsubscribe (IP, Port, Article): Request an unsubscribe to group server
e. Publish (Article, IP, Port): Sends a new article to the server
f. Ping(): Check whether the server is up

The client should know the location of its group server (IP, port, any RPC/RMI names/interfaces) when it starts up. The client can then choose to `Subscribe` and `Publish`. Note that `Publish`, `{Un}Subscribe` should fail if the Article is not in a legal format.

3.2. Server → Client (By UDP)

The server will propagate the article to each matched client using UDP unicast. So clients have check for a UDP message to receive the new article from the server. You may want to devote a blocking thread in the client to receive subscribed articles. As stated above, the article format will be [**"type;originator;org;contents"].**

3.3. Server ↔ Server (by RPC) (optional)

The operations (by RPC) which the server should be able to perform with other servers.

a. JoinServer (IP, ProgID, ProgVers): Once the server receives the list of servers from registry server, it can join to other servers to receive articles from them.
b. LeaveServer (IP, ProgID, ProgVers)
c. PublishServer(Article, IP, Port)

To enable servers to communicate they have to implement common IDL for RPC and interface for RMI. The ProgID and Version of IDL file can be different from each group.

3.4. Server → RegistryServer (by UDP)

The operations which the server should be able to perform with the register server are below.

a. Register (string): no return
b. Deregister (string): no return
c. GetList (string): returns list of active servers + contact information as a string

Depending on the RPC format, server should use different arguments for the UDP message. You will use a single string in both cases.

For RPC:

["Register;RPC;IP;Port;ProgramID;Version"]
["Deregister;RPC;IP;Port"]
["GetList;RPC;IP;Port"]

For java RMI:

["Register;RMI;IP;Port;BindingName;Port for RMI"]
["Deregister;RMI;IP;Port"]
["GetList;RMI;IP;Port"]

For GetList, the registry will return the list of other servers in both cases as a UDP message. This format will also be a string.

For RPC:

["IP;ProgramID;Version;IP;ProgramID;Version;IP;ProgramID;Version… and so on"]

For java RMI:

["IP;BindingName;Port;IP;BindingName;Port;IP;BindingName;Port … and so on"]

You can assume that the length of the returned server list will not exceed *1024 bytes*.

3.5. Registry Server → Server (by RPC)

a. Ping (): Registry server will keep sending heartbeat message to your server.

The register server will send a string "heartbeat" to joined servers. So your server needs to return the same string as your server received to a register server. If your server does not response to this ping within 5 seconds, your server will be removed from registry server.

The registry server will be on **dio.cs.umn.edu ("128.101.35.147") with port 5105**. Please note that the host and port can be changed. If there are any communication problems between your server and registry server, please don't hesitate to contact with TA (ramkr004@umn.edu). And we encourage you to use the forum to let other students know your server is ready to communicate each other.

You can implement any further functions as much as you want but only need to comment about it on a document. Be creative, and try to do something fun!

4. Google Cloud PubSub

Implementing PubSub from scratch was a lot work. For comparison, try out this public service and implement a scaled-down version of our PubSub interface.

Please install the google cloud pubsub server emulator and example client using the guide in the appendix A.

The PubSub API can be used to create a 'topic', which is identified by a string conveyed to the server when the topic is created. Clients can subscribe or unsubscribe to a topic, and also publish strings to it. Any string published to a topic is received by all subscribers.

Your client should be able to connect to the server, create a new `topic', and subscribe/unsubscribe/publish to that topic. Write a client which gives the user the options *Join, Leave, CreateTopic, Subscribe, Unsubscribe and Publish*, as described below -

Join : Connect to the PubSub server emulator using a project id. (similar to the code under the example code's comment "Creates a Client"). The project-id can be hard-coded.

CreateTopic : Creates a new topic using a user-defined string..

Subscribe : Subscribes to a the topic specified by the user. Should fail if the topic does not exist.

Unsubscribe : Unsubscribes from a particular topic. Should fail if topic does not exist.

Publish : Publishes a user-defined string to a particular topic. This string should be received by all subscribers.

A single server (i.e. the PubSub emulator) needs to service all the client requests. There is no requirement for running a registry server or multiple servers, because the servers are provided as a black box whose internal functionality is hidden. Also note that the client APIs hide the details of communication so you don't need to worry about the underlying technology used. Compare the time and effort spent on this implementation with the time and effort spent on the original implementation.

5. Project Group

All students should work in groups of size 2 or 3. If you have difficulty in finding a partner, we encourage you to use the forum to find your partner(s).

6. Test Cases and Possible Issues

You should also develop your own test cases for all the components of the architecture, and provide documentation that explains how to run each of your test cases, including the expected results. Also tell us about any possible deadlocks or race conditions. It is ok to have potential deadlocks. *You may wish to defer the use of* `JoinServer` *until you are certain your server and theirs works! Otherwise, bugs will propagate!*

Note that all these services are expected to work normally when deployed across different machines as well as

when deployed over a single machine. Also, there are many failure modes relating to the content of messages, parameters, and system calls. Try to catch as many of these as possible. **Finally, you should run your clients and servers within the CSE firewall – as outside access particularly through UDP may be blocked.**

7. Deliverables

a. Design document describing each component. Not to exceed 3 pages. Compare your from-scratch PubSub to the Google PubSub in terms of development effort, ease-of-use, etc.
b. Instructions explaining how to run each component and how to use the service as a whole, including command line syntax, configuration file particulars, and user input interface
c. Testing description, including a list of cases attempted (which must include negative cases) and the results
d. Source code, makefiles and/or a script to start the system, and executables

8. Grading

The grade for this assignment will include the following components:

a. 10% - The document you submit – This includes a detailed description of the system design and operation along with the test cases used for the system (must include negative cases)
b. 60% - The functionality and correctness of your own server and clients
c. 20% - The functionality and correctness of communication with other servers and registry server
d. 10% - The functionality and correctness of the PubSub implementation.

9. Resources

a. RPC tutorial: http://www.cs.cf.ac.uk/Dave/C/node34.html
b. RMI example:
   http://www.asjava.com/distributed-java/java-rmi-example-just-get-starting/
c. Introduction to ONC RPC: http://www.pk.org/rutgers/hw/rpc/index.html
d. ONC+ Dev Guide(handy RPC reference): http://docs.sun.com/app/docs/doc/816-1435
e. Java RMI documentation:
   http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp
f. Introduction to Java RMI:
   http://www.javacoffeebreak.com/articles/javarmi/javarmi.html
g. RMI Tutorial: http://java.sun.com/docs/books/tutorial/rmi/index.html
h. UDP example (C, C++):
   http://www.binarytides.com/programming-udp-sockets-c-linux/
i. UDP example (JAVA) :
   http://www.cs.rpi.edu/courses/fall02/netprog/notes/javaudp/javaudp.pdf
j. Thrift example :

https://thrift.apache.org

Appendix A - Google Cloud PubSub :

Google Cloud PubSub provides servers and a set of client APIs which can be leveraged to build the clients and customize the PubSub system to meet user requirements. Your task is to build a similar but simpler PubSub system as described earlier using said client APIs and a `fake' PubSub server (because the real one costs money to use).

For this section, you need a *google account and a credit card on file* with it. The credit card *will not be charged* because we do not use the cloud infrastructure, instead, we use a local server emulator.

Building the Server Emulator

Please use the following steps to setup a **local install of the PubSub emulator**, on any CSELabs machine :

A. Google cloud can be installed using the following sequence of commands :

```
wget https://dl.google.com/dl/cloudsdk/channels/rapid/downloads/google-cloud-sdk-186.0.0-linux-x86_64.tar.gz
tar -xvf google-cloud-sdk-186.0.0-linux-x86_64.tar.gz
cd google-cloud-sdk/bin
./install.sh
```

B. Following the google cloud install, you may install the fake pubsub server using the following sequence of commands :

```
gcloud components install pubsub-emulator
gcloud components update
```

The emulator can be started using the command

```
gcloud beta emulators pubsub start --host-port=maximus.cs.umn.edu:46839
```

The host system is maximus.cs.umn.edu and the port is 46389 in this case.

You may use any other CSELABS machine to host the server emulator.

Building the Client

The client can be built using any language of your choice. The following description uses *golang* to build the client, because the code is concise. However, feel free to choose any other language to build the client if you wish. The following steps require a bash terminal. Type `bash' to enter the bash terminal.

A. Firstly, install a local copy of Go using the following steps :

```
wget https://dl.google.com/go/go1.9.3.linux-amd64.tar.gz
tar -xvf go1.9.3.linux-amd64.tar.gz
mkdir gomaster && mv go gomaster
cd gomaster
```

```
export GOROOT=`pwd`/go
export PATH=$PATH:$GOROOT/bin
mkdir gopath
export GOPATH=`pwd`/gopath
```

B. Next, download the Go PubSub libraries using the following command :

```
go get -u cloud.google.com/go/pubsub
```

C. Navigate to the gomaster folder and create a folder `client' using the following command :

```
mkdir  gopath/client
cd gopath/client
```

D. Create a file `client.go` and copy and save the example client code into it :

```
// Sample pubsub-quickstart creates a Google Cloud Pub/Sub topic.
package main
import (
        "fmt"
        "log"
        // Imports the Google Cloud Pub/Sub client package.
        "cloud.google.com/go/pubsub"
        "golang.org/x/net/context"
)
func main() {
        ctx := context.Background()
        // Sets your Google Cloud Platform project ID.
        projectID := "YOUR_PROJECT_ID"
        // Creates a client.
        client, err := pubsub.NewClient(ctx, projectID)
        if err != nil {
                log.Fatalf("Failed to create client: %v", err)
        }
        // Sets the name for the new topic.
        topicName := "my-new-topic"
        // Creates the new topic.
        topic, err := client.CreateTopic(ctx, topicName)
        if err != nil {
                log.Fatalf("Failed to create topic: %v", err)
```

```
        }
        fmt.Printf("Topic %v created.\n", topic)
}
```

The `projectID` variable can be set to a string of your choice. Compile the program using the command `go build client.go`

E. Next, setup authentication for your client. Go to the link `https://cloud.google.com/docs/authentication/getting-started` and follow the steps under "Creating a service account". The result of these steps is to generate a file containing a JSON string, which needs to be copied to the machine on which you intend to execute the client. This string will authorize the client to use the PubSub APIs.

F. Lastly, export the following environment variables so that the client knows where to look for the JSON string and also the IP address of the server emulator :

```
export GOOGLE_APPLICATION_CREDENTIALS=<path-to-json-file-from-step-E>
export PUBSUB_EMULATOR_HOST=<host-ip-address> (eg : maximus.cs.umn.edu:46389)
export PUBSUB_PROJECT_ID=<project-id-string-from-step-D>
```

Run the server emulator and client example and verify that the `topic was successfully created' message is displayed.

Next Steps

Note: The PubSub API is provided for many languages, you may choose one that you prefer for your implementation. The Golang API documentation is found at `https://godoc.org/cloud.google.com/go/pubsub`

All the steps in sections `Running the Server Emulator', `Building the Client', and `Next Steps' are based on the documentation available on Google Cloud as on Jan-27-2018 at `https://cloud.google.com/pubsub/docs/`

Appendix B (IDL Files) - (some of these are **optional** as described above)

For RPC:

<communicate.x**>** **-** IDL file for RPCGen

```
program COMMUNICATE_PROG {
        version COMMUNICATE_VERSION
        {
            bool JoinServer (string IP, int ProgID, int ProgVers) = 1;
            bool LeaveServer (string IP, int ProgID, int ProgVars) = 2;
            bool Join (string IP, int Port) = 3;
```

```
            bool Leave (string IP, int Port) = 4;
            bool Subscribe(string IP, int Port, string Article) = 5;
            bool Unsubscribe (string IP, int Port, string Article) = 6;
            bool Publish (string Article, string IP, int Port) = 7;
           bool PublishServer (string Article, string IP, int Port) = 8;
            bool Ping () = 9;
      } = X;
} = 0xXXXXXXXX;
```

For java RMI: (some of these are **optional** as described above)

```
<Communicate.java> - Define a remote interface
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Communicate extends Remote
{
      boolean JoinServer (String IP, int Port) throws RemoteException;
      boolean LeaveServer (String IP, int Port) throws RemoteException;
      boolean Join (String IP, int Port) throws RemoteException;
      boolean Leave (String IP, int Port) throws RemoteException;
      boolean Subscribe(String IP, int Port, String Article) throws
RemoteException;
      boolean Unsubscribe (String IP, int Port, String Article) throws
RemoteException;
      boolean Publish (String Article, String IP, int Port) throws
RemoteException;
      boolean PublishServer (String Article, String IP, int Port) throws
RemoteException;
      boolean Ping () throws RemoteException;
}
```