CSCi 5105: Introduction to Distributed Computing
Spring 2020
Instructor: Jon Weissman
Project 1: PubSub
Due: Feb 19 12pm (noon)

1. Overview

In this project, you will implement a simple publish subscribe system (PubSub). You will use two forms of communication: basic messaging 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 communication protocols. You may assume that UDP communication is reliable. This lab also 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. It is assumed that you are familiar with basic IP communication (e.g. IP addresses and port).

2. Project Details

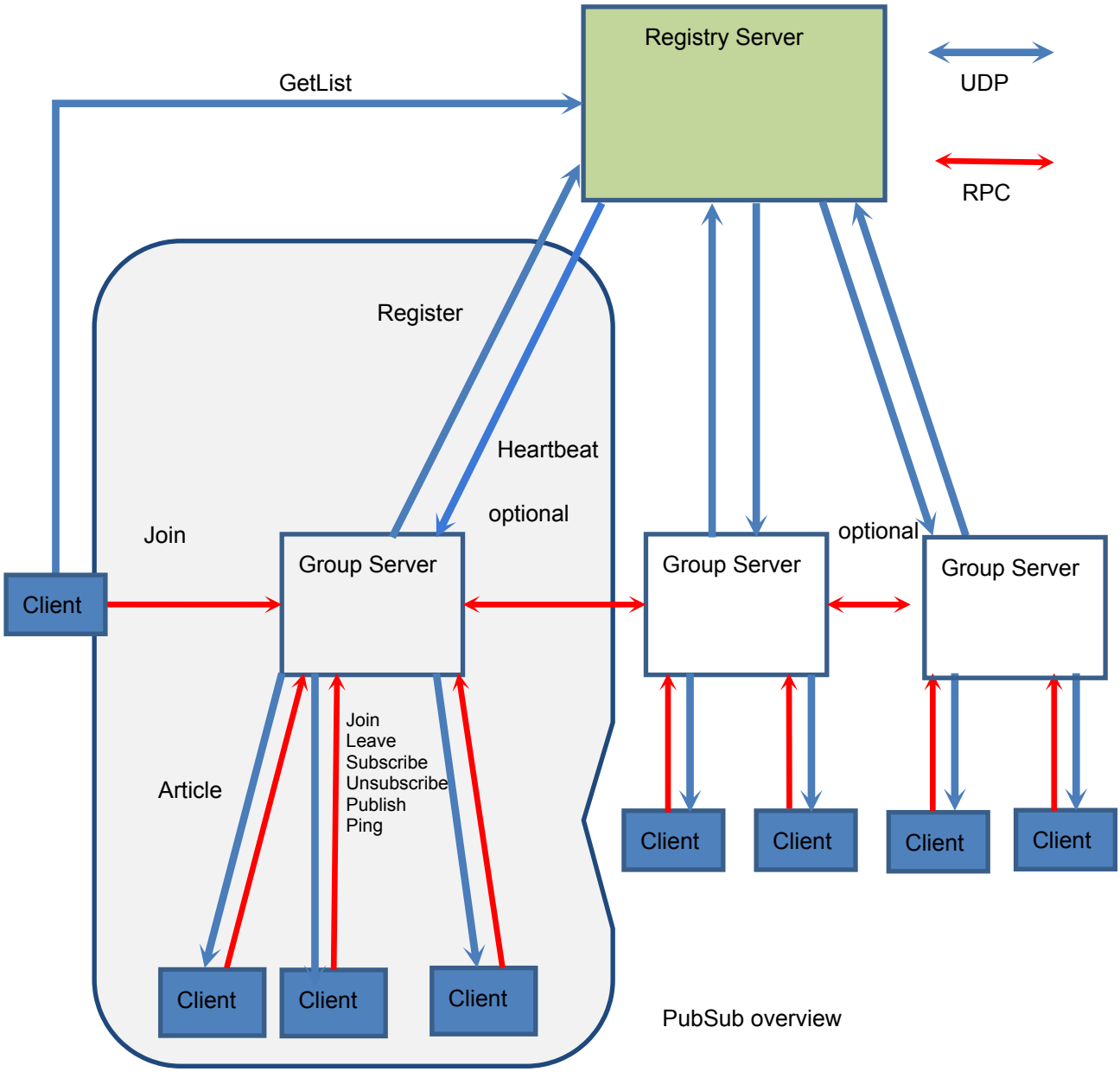
The project will be comprised of: **clients**, **group servers**, and a **registry-server**. You will program a client and group server. The registry server code will be provided. The registry server is assumed to be a known IP address and port (you need to establish them on your own). It will communicate via low-level UDP and be neutral to your group server language and RPC/RMI implementation. The registry stores information about your group server(s). To obtain the existing group server list, you may call `GetList`, via UDP. Note that, you are not supposed to modify the registry server, if you need to do that, describe your modification in the delivery document.

Your client will first contact your Registry Server to get a list of existing group servers. The client calls `Join` on one of the group servers and then may call `Subscribe` and `Publish` on the server. Clients communicate to their server by means of RPC/RMI. The group 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, but it is assumed they notify the server first. You can decide whether to cache articles at the server or not. If you do, you need to describe how you would garbage-collect them.

Your group server begins by registering itself with a registry server. You may deploy one or more group servers (for fault tolerance). You will use UDP to `Register` to the registry server, passing to it (IP, Port,

and RPC or RMI). Your server may also `Deregister` if it wishes to go "off-line". The main functionality of the server is to support PubSub with multiple clients, including: handling client's requests (like join, subscribe etc.) and sending the published articles to matched clients. As an **optional** feature: if you have multiple group servers, you may want to propagate to the server state (e.g. either subscriptions, and/or published articles, etc). **But not required.**

To check whether the group server is up, the client will Ping 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, in which case the client knows that the server is down. Then the client should ask the registry server for a list of existing servers and re-join to one of them. Note that we don't require recovering the client's subscription information if the server is down. On the registry server side, it will also send a 'heartbeat' string to the server via UDP and wait for a response. If there is no response, then it will assume that the server is down and remove it from the existing server list.



3. Implementation Details

3.1. The Article String Format (by RPC/RMI)

An article is a simple formatted string with 4 fields separated by ";": type, originator, org, and contents. The type can be one of the following categories: <*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 standardized, the rest could be 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 group 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 (IP, Port, Article): 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 (and is registered to the registry server). 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. Thus, clients have to check for a UDP message to receive the new article. 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 → 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 (RPC/RMI), the group 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 UDP)

The register server will periodically sending a string "heartbeat" to joined servers via UDP. Your group server needs to return the same string is received from the register server. If your server does not respond to this request within 5 seconds, your server will be removed from the registry server. Thus, your group server should have a UDP thread that awaits such requests from the registry server.

We are providing the registry server. If there are any communication/functionality problems about the registry server, please don't hesitate to contact TA (wuxx1354@umn.edu).

4. 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).

5. Your Test Cases and Possible Issues

Note: the use of threads may require you use locks as needed. You should also develop your own test cases for all functionality: join/leave, subscribe/unsubscribe, and publish/receive articles. You must be able to show

how your client can fail-over to another group server (as said before, it is not required that any previous state be maintained). Note: PubSub is expected to work normally when deployed across different machines as well as when deployed on a single machine. Also, there are many failure modes relating to illegal article formats, and system calls. Try to catch as many of these as possible. **Finally, you must be able to run the registry server, clients and servers on the CSE machines.**

6. Deliverables

a. Design document, not to exceed 4 pages. It must include:
   - Instructions explaining **how to compile and run** your code, including command line syntax, configuration file particulars, and user input interface.
   - Design details: a high-level description about your PubSub system, then followed by the description about files, functions and notable implementation information.

b. A section on testing containing: a list of cases attempted (it could be negative cases) and their results, and known limitations (existing race conditions or deadlocks).

c. Source code, makefiles and/or a script to start the system, and cselabs compiled executables

d. Pledge: your team will sign a pledge that no one sought out any on-line solutions, e.g. github, for portions of this lab. Violations will lead to course failure.

7. 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

b. 50% - The implementation of your server/client: [10%] client/server contact with the registry server; [10%] client side; [10%] server side; [10%] failure handling; [10%] subscription matching and others

c. 40% - Pass all the tests for your PubSub implementation

8. 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. Apache Thrift example :

https://thrift.apache.org

Appendix A (IDL Files)

For RPC:

```
<communicate.x> - IDL file for RPCGen
program COMMUNICATE_PROG {
      version COMMUNICATE_VERSION  {
          bool Join (string IP, int Port) = 1;
          bool Leave (string IP, int Port) = 2;
          bool Subscribe(string IP, int Port, string Article) = 3;
          bool Unsubscribe (string IP, int Port, string Article) = 4;
          bool Publish (string Article, string IP, int Port) = 5;
          bool Ping () = 6;
     } = X;
} = 0xXXXXXXXX;
```

For java RMI:

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

public interface Communicate extends Remote {
     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 Ping () throws RemoteException;
}
```