Publish Subscribe system

To implement the Publisher Subscriber system in this project we used Java Remote Method Invocation(RMI). We have a few coupled interacting components which we would discuss now:-

1. ***CONSTANTS file*** :- All the shared system variables are put in CONSTANTS file so as it can be updated at one point instead of changing the code all over the place. This includes server host, registry host, RMI registry port, registry server port. This will be shared by both clients and server.
2. ***RMI Interface*** :- RMI interface is implemented in class PubSub which will be shared by both server and client programs.
3. ***RMI Registry*** :- We are creating RMI registry at server startup using port 1099 defined in CONSTANTS file and assuming the RMI registry and server are running on same machine. [*This obviously can be scaled to run on different machines but for simplicity we have made such an assumption.*]
4. ***HeartBeatListener*** :- This is a class for UDP server which will get heartbeat message from registry server and print it on standard output.
5. ***Server*** class is the driver program of server which start and stops the RMI server. During start it will create a RMI registry, start RMI server and register it with registry server. It will also start the Heartbeat listener for the server. During stop it will deregister the server and close the RMI server and driver program.
6. ***PubSubImpl*** is the RMI server implementation which will have all the methods for Join, Leave, Subscribe, Unsubscribe, Publish, Ping, etc. It uses classes ClientInfo and SubscriptionManager as defined below.
7. ***ClientInfo*** is the class to store information about the clients which join and leave the system.
8. ***SusbcriptionManager*** is class which store data about active clients and subscription info for all the clients. It has methods to add client used by Join, remove client used by Leave, subscribe, unsubscribed which are all called from PubSubImpl. It also has a special function to get all the clients info which qualifies for a publish. For example if there is a publish for “Sports;XYZ;ABC;Contents”. We need to find all clients which may have subscribed for 8 combinations of first three values. After we get this list we can send messages in parallel.
9. ***ClientHeartbeatCall*** is the class used by client to send periodic messages to server via Ping.
10. ***Listener*** is the class which act as the UDP server for client to get all the published messages.
11. Client programs and driver programs to test the whole system.

System Implementation

1. ClientInfo store is IP and port of the client and implements equals and hashCode function to be used elsewhere.
2. SubscriptionManager maintains a HashSet for active clients which is updated on Join and Leave by synchronized calls holding lock for very less period of time. The rationale is that a time very less number of clients will be leaving and joining with respect to subscribing and publishing. It also has a ConcurrentHashMap with subscription as key and HashSet of clients which subscribe to it. This data structure in addition to being scalable as only a part of concurrent hash map is locked for update and reads can happen in parallel, also provides an easy to way to compute all the clients for which a message is to be send as described in point 8 of section above. It also have a method to check article validity for publish, subscribe and unsubscribe. While adding the client the addClient method will check the number of active clients against the max clients allowed.
3. PubSubImpl class is the one which binds all the server components and implements the RMI interface. It used SubscriptionManager as utility to do all the work described in above point.

4. PubSub system maintains the subscription information even if the client leaves the server. So that if the same client joins back it starts receiving articles based on earlier subscriptions.

Test Cases attempted

(All the test cases described below are present in the clients and driver programs. Comments are included in the code)

1. Checking whether the article is in legal format.
2. Checking whether contents is not present while subscribing / unsubscribing.
3. Checking whether contents is present while publishing.
4. Checking whether at least one filter is present while subscribing / unsubscribing / publishing.
5. Checking whether the duplicates messages are not received when the client is subscribed to both specific and general filters using the same type/sender/org.
6. Checking whether the clients are receiving the subscribed articles even if they leave and join the server in between.
7. Checking whether the clients which is not currently joined is not able to do any subscribe/unsubscribe/publish operations.
8. Checking whether the message is not received when the client unsubscribes.
9. Checking whether the client doesn't receive the messages after leaving the server and starts receiving the messages based on previous subscriptions as soon as it is joined.
10. Making sure that client ends as soon as it ping (heartbeat) fails.

Project implemented PubSub versus Google PubSub

- ***Subscription methodology :***

The subscriber needs to keep track of the object (returned when subscribing) which executes the callback function whenever it receives any new request. This object is required for unsubscribing. This is different from our own PubSub where we don't need to keep track of any object. Whenever we want to unsubscribe, we need to just send the channel name and the server will handle the rest.

Google PubSub :

Google PubSub has an extra layer called subscription layer which connects with the topics. So, the client should create a subscription object on a topic using the topic name and unique subscription id. All the clients using the same subscription object are treated as a single subscription group.

Project implemented PubSub :

The client directly uses the topic names to subscribe and each client is treated as a individual subscriber. No subscription groups are formed.

- ***Delivery semantics :***

Google pubsub:

It guarantees delivery to all the attached subscriptions groups. But all the clients in a single subscription group are not guaranteed to receive all the messages published on that subscription. The google pubsub retries the delivery of messages to the subscriptions until at least one client receives it and as soon a message in delivered to one of the clients in the subscription group, it deletes the message.

Project implemented Pubsub:

We use at most once delivery semantics. The message may/ may not be delivered depending on the UDP messages. No retries are made.

- ***Scalability*** :

Google PubSub scales horizontally depending the load factors. Whereas, our implementation doesn't scale after threshold if there the number of subscriptions/clients increases. We need more servers and join them to distribute workload.

- ***Message storage :***

Google PubSub stores the messages until is delivered at least to one client in each subscription group. Whereas, our PubSub doesn't store the messages. As soon as the messages are received, they are sent to all the subscribed clients.

Apart from major differences mentioned above, some of the basic differences are

- The client doesn't wait if the server goes down. The client stops as soon as the server doesn't respond to the heart beats.
- Google PubSub was easy to set up as everything was already built. Whereas in the case of own implementation, we had to take care synchronization, race conditions, efficiency and several other factors. Basically google PubSub was done in few hours while it took few days to build the own system.
- We don't need to write any server side code. We only need to write code for client (publish/subscribe). This means that we don't have to do a lot of things including:
    - Not writing the code for matching a newly published article with its subscribers.
    - Making sure the message gets delivered (at least once semantic) to the subscriber.

Scripts to build the project:

There are two make files one client and one in server folder.

Steps:

1. cd Client
2. make all
3. cd ../Server
4. make all

This will build all the classes and copy the stubs appropriately. Currently the local test registry server entry is added in CONTANTS class. To test with main registry server, change host and port in CONSTANTS class in both client and server folders and then build the system again.

Steps to run Server:

1. cd Server
2. java Server

To stop the server just type in "Stop" in command line interface.

Steps to run client and test cases:

1. cd Client

    To run simple tests : java SimpleTestCaseDriver

    To run stress tests : java StressTestDriver

Please be advised that all the clients have thread sleep used in them for test out for more uncertainty.