Networking Labs

ECE

ING4 (SI)

2013/2014

Chat Application

Salim NAHLE

Session Requirements:

- Please install Eclipse CDT or Juno on your machine.
- Please read the whole topic before starting the implementation
- You are supposed to work in groups of 2 persons (same groups shall be maintained through the whole semester)
- Please do not forget to save your project on an external storage device. You may need to work at home
- Moreover you need to continue working on the application in a second lab session
- The deadline of submitting the project is 31 March 2014.
- Expected documents:
    - PDF report (5 to 10 pages explaining basic design aspects)
    - The application output (separate client and server jars in addition to the java libraries used, all zipped in one file).
- The report and the application output shall be posted on the campus page.

Chat Application

The objective of the last two lab sessions is to create a networking **chat** application. The application relies on client-server architecture. It enables users to connect and create different chat rooms and to exchange messages on these rooms. The server is the central entity that maintains the basic information of the network like:

1. Connected users
2. Created chat rooms
3. Connected users on each chat room.

Communication layers

The defined protocol in this project is applicative protocol. It shall be built on lower-level communication protocols. The lower communication layers, however, shall be transparent to the application. For this purpose, we recommend using **JGroups [1]** to maintain and control the communications between the clients and the server. JGroups is a powerful toolkit for reliable messaging. It can be used to create clusters whose nodes can send messages to each other. The main features [1] include:

- Cluster creation and deletion. Cluster nodes can be spread across LANs or WANs
- Joining and leaving of clusters
- Membership detection and notification about joined/left/crashed cluster nodes
- Detection and removal of crashed nodes
- Sending and receiving of node-to-cluster messages (point-to-multipoint)
- Sending and receiving of node-to-node messages (point-to-point)

Application Description

Client side:

- A client is identified in the whole application (server and chat rooms) by its name. The client application shall ask the user to provide his name when the application is launched. This shall be done prior to any communication with the server. The name is unique and shall be maintained during the life time of the application. Moreover it shall be displayed as a title of the running application (**JFrame** title for example).
- A client can connect to several chat rooms at a time (Three at most)
- A client can request the server to create chat rooms
- The client application shall display all the chat rooms to which the user is connected. On each room, the connected users and the exchanged messages shall be displayed. For each message, the sender name shall be also displayed. (Think of using a **JScrollPane** and a **JPanel** for each room).
- A user can join and leave a chat room anytime he wants.
- When a user leaves a chat room, the server shall be informed.
- When the user leaves definitely, the server shall be informed.

Server side:

- The server creates the chat rooms based on users' requests.
- The server application displays connected users as well as created chat rooms and the connected users on each chat room
- The server shall send permanent updates to report the available chat rooms and the

connected users on each chat room. These updates may be triggered by:
    - A new user joining a chat room
    - A user leaving a chat room
    - Creation of a new chat room
    - Deletion of an existing chat room.
- The server application shall provide a means for administrators to delete or create chat rooms.

What to do?

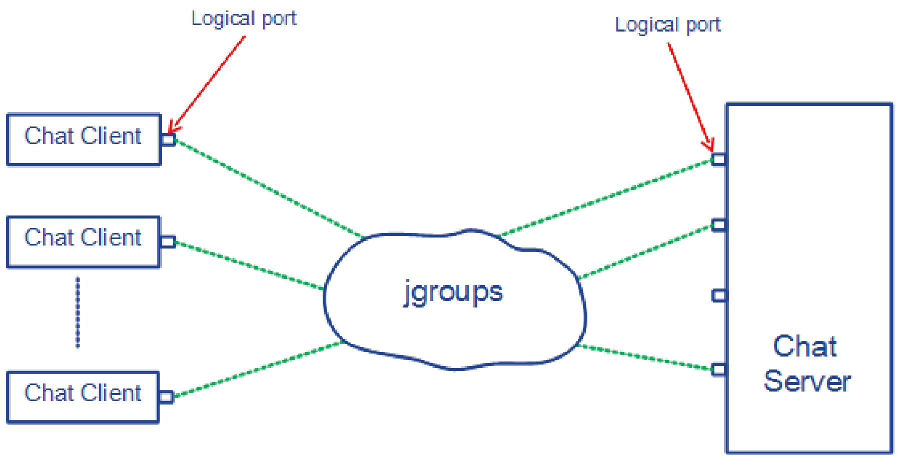
Prior to starting implementation, you shall define the application protocol as well as the message format (header and data fields) to be used.
For instance, the following message types shall be distinguished:

- Connection establishment (sent by client)
- Connection acknowledgment (sent by server)
- Chat rooms information request (sent by client)
- Available chat rooms and connected users (sent by server)
- Chat room creation request (sent by client)
- Chat room creation acknowledgment (sent by server)
- Chat room joining request (sent by client)
- Chat room leaving request (sent by client)
- User Logging out (sent by client)
- User data in a chat room (sent by the client). In each data message, the user name, the cluster name and a data field are needed at least.
- Server closing down (sent by server)

For each message type, the important control and possible data/information fields shall be defined.

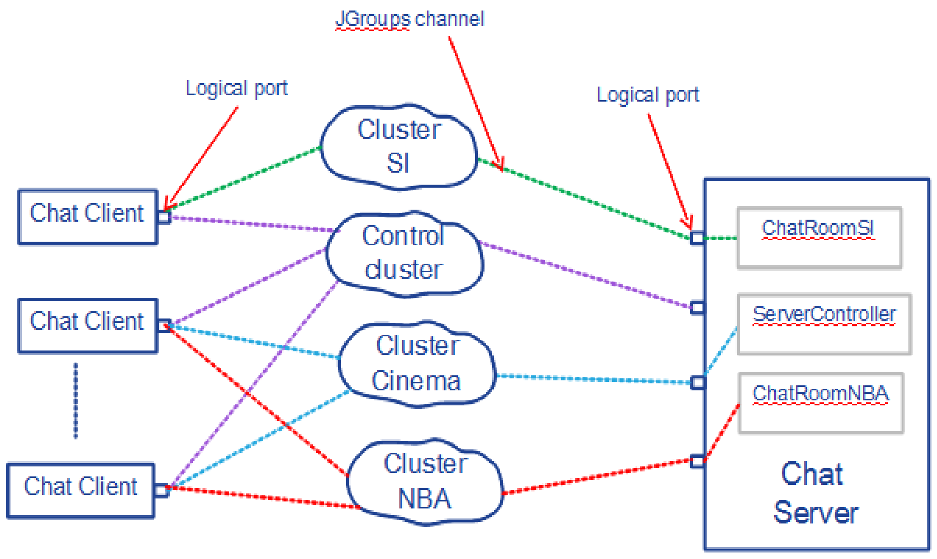
Application design with jgroups

JGroups can be used to ensure communications between clients and the server as depicted in the figure. A cluster (or many) can be created as shown in the figure below. Each client and the server create virtual connection (class **JChannel**) onto the cluster. On this channel, they can send and receive information sent on the cluster by implementing the interface **Receiver**.



JGroups is a JAVA toolkit for reliable multicast

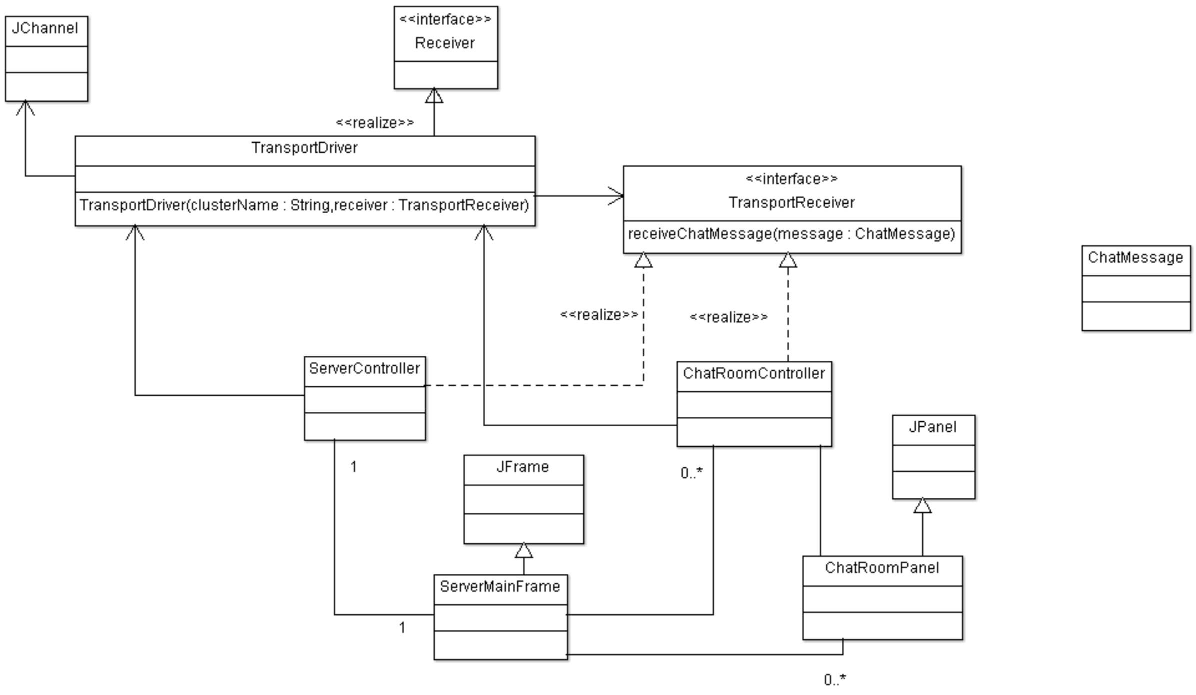
In order to provide a flexible and efficient design, it is recommended to create a jgroups cluster per chat room. The cluster name holds the name of the room it self (it is the name provided by the user or the server otherwise). This design solution can simplify information treatment as all data can be sent in multicast. All the users (members) of a chat room shall display data messages of all other users. Nevertheless, the initial connection establishment as well as all management messages and update are communicated between the clients and the server on the CONTROL cluster. **The control cluster name is known to all clients by default. It is called chatControlCluster (final static attribute).** The server name, however, is sent by the server during connection establishment.



A cluster is a jgroup group with a set of members

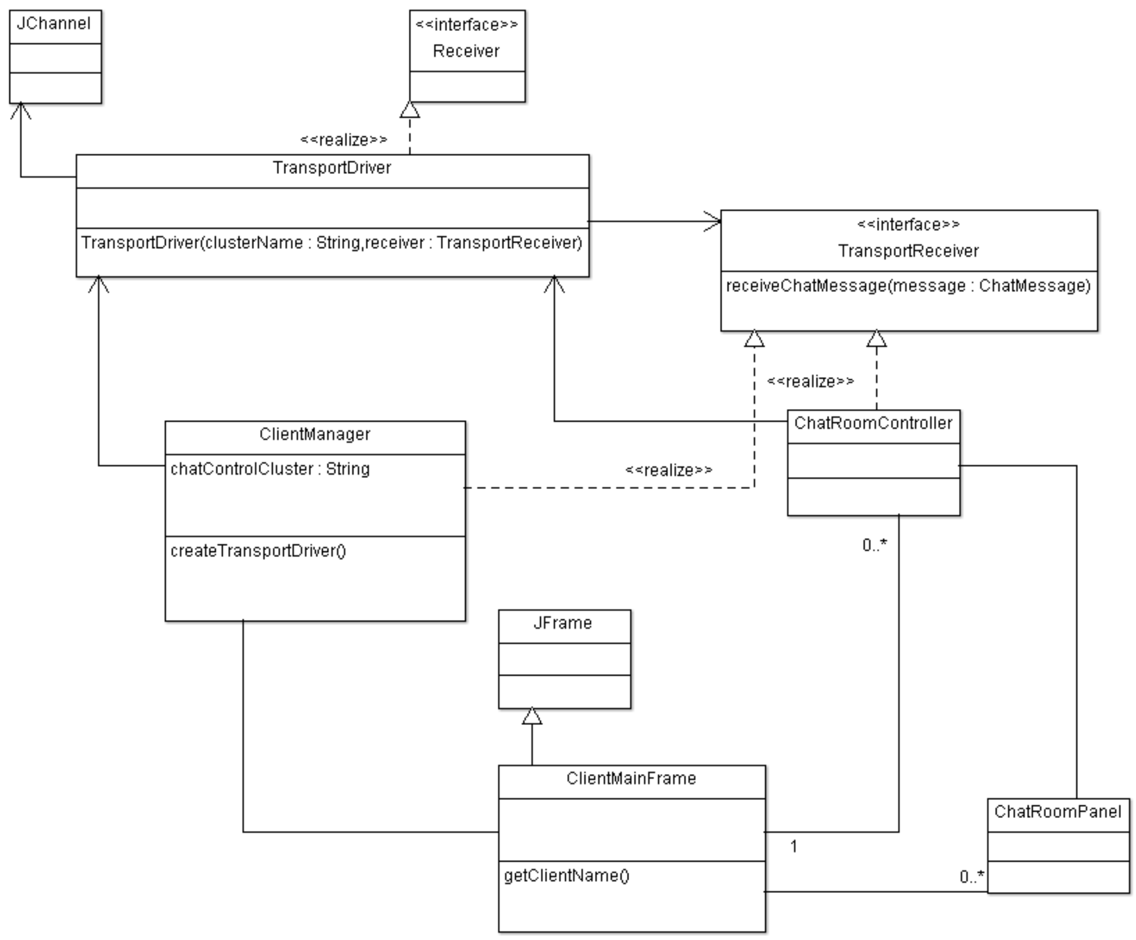
Class diagram of Chat server

The following class diagram is a possible implementation of the chat server. Please feel free to design the chat server as you want. This class diagram assumes using **jgroups**. In fact, the interface **Receiver** and the class **JChannel** are provided by jgroups. Note also that the classes **JPanel** and **JFrame** are provided by **javax.swing**.
The class **<u>TransportDriver</u>** and the service interface **<u>TransportReceiver</u>** simulate the interface with the transport layer that is provided by jgroups (you shall create them).



Class diagram of Chat Client

The following class diagram is a possible implementation of the chat client. Please feel free to design the client application as you want. This class diagram assumes using **jgroups**.
The class **TransportDriver** and the service interface **TransportReceiver** simulate the interface with the transport layer that is provided by jgroups.



Appendix A: TransportDriver constructor example

```
public  TransportDriver( final String clusterName, TransportReceiver transportReceiver) {

    logger.debug("Forcing to IpV4 stack"); \\think of using log4j for logging important information
    System.setProperty("java.net.preferIPv4Stack", "true");
    System.setProperty("jgroups.bind_addr", "127.0.0.1" );

    this.clusterName = clusterName;
    this.transportReceiver = transportReceiver;

    try{
          drvChannel = new JChannel();
            UDP protocol = (UDP)drvChannel.getProtocolStack().getTransport();
            if (getPort(clusterName) !=0)
                   protocol.setMulticastPort(getPort(clusterName));
            else
                  JOptionPane.showMessageDialog(null, "can't find a port, default is used");
          drvChannel.connect(clusterName);
          drvChannel.setReceiver(this);
          drvChannel.setDiscardOwnMessages(true);
    }
    catch (Exception e){
             logger.error("<<class Abstract Channel Driver>>can not create channel ");
    }

}
```

Appendix B: Cluster modification in jgroups example

Any modification in jgroups cluster members is reported by using the callback '**viewAccepted**' of the interface **Receiver.** You can implement this callback to exploit the information.

```
public void viewAccepted(View view) {
    //#[ operation viewAccepted(View)
          // Define your implementation here.
    //#]
}
```

Example

```
public void viewAccepted(View view) {
    Vector <Address> vect = new Vector <Address>(view.getMembers());
    ChatMessage chatMsg = new ChatMessage(vect);
    // Fill in message fields as appropriate
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
        public void run() {
          if (drvChannel.getAddress()!=null)
                      transportReceiver.receiveChatMessage(chatMsg);
        }
    });
}
```

References:

[1] http://www.jgroups.org/