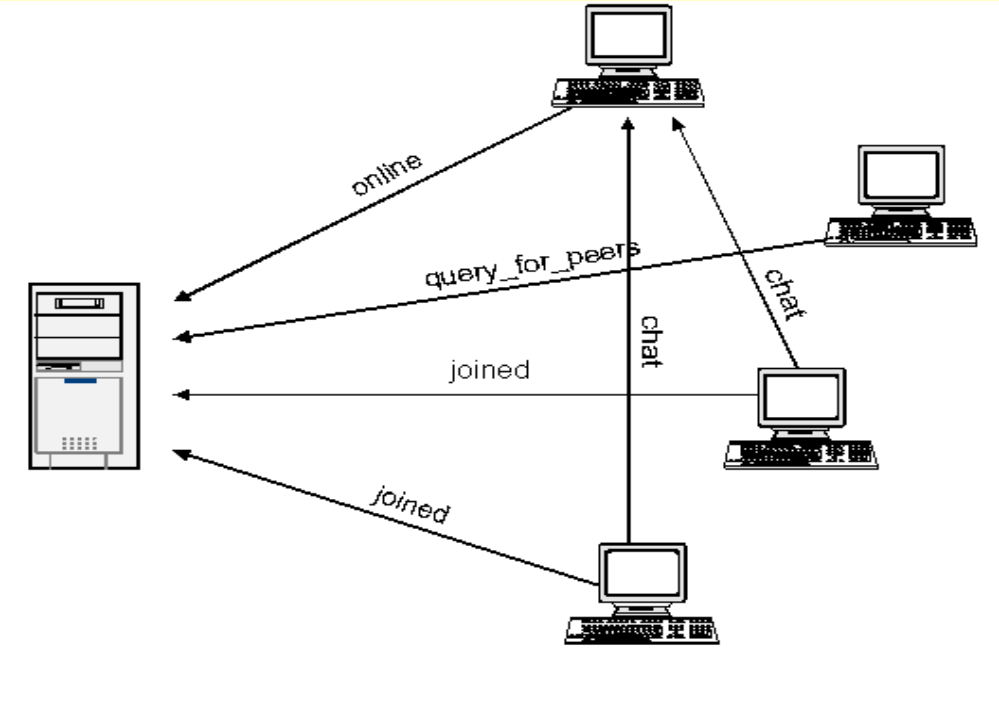
CPS 706 Course Project : A P2P chatroom application

In this course project, we will develop a peer to peer chatroom application that uses a centralized peer directory. Project can be implemented in Java. Directory entry includes user's nickname (e.g. Mark or Nick), user's host name, host's IP address, the protocol port of the user's chatting server, popularity rating of the user's chatting server, and the nickname of the chatroom's owner where user is currently chatting. For the sake of simplicity (in order to avoid issues about graphical user interface) we will assume that one user can join one chatroom at the time and that user who has joined somebody's chatroom can not run his/her own chatroom at that time. However, the user should be able to change her/his role in time. The architecture of the application is shown in the figure below where the labels on the arrow lines are only used to represent some sample actions in the system.



We are going to implement a centralized directory server, a P2P chat-client and a P2P chat-server. The P2P client has a double-task: it acts as the client of the directory server in order to obtain information about online peers; it may also act as the client of a P2P transient chat-server in order to exchange messages with peers. Therefore, this project consists of two major parts: implementation of the communication protocol between the directory server and a P2P chat-client, and implementation of the communication protocol between a P2P client and a transient P2P chat-server.

Directory server and its interaction with P2P client

The centralized directory server maintains a directory of online users willing to chat. It should contain entries which list user's nickname, the host name of user's computer,

host's IP address, the protocol port of the user's transient server for online chatting, the rating of chat-server run by this user and the nickname of the chatroom where user is currently chatting (if any). When a user, say Bob, wants to advertise that he is online and willing to chat, he will send a complete directory (without the popularity field) entry to the directory server using an "*Online*" message. This message will be sent using UDP. The directory server should acknowledge the "Online" message. A user can send his/her data to the directory server in the format of the header lines similar to HTTP request message shown below.

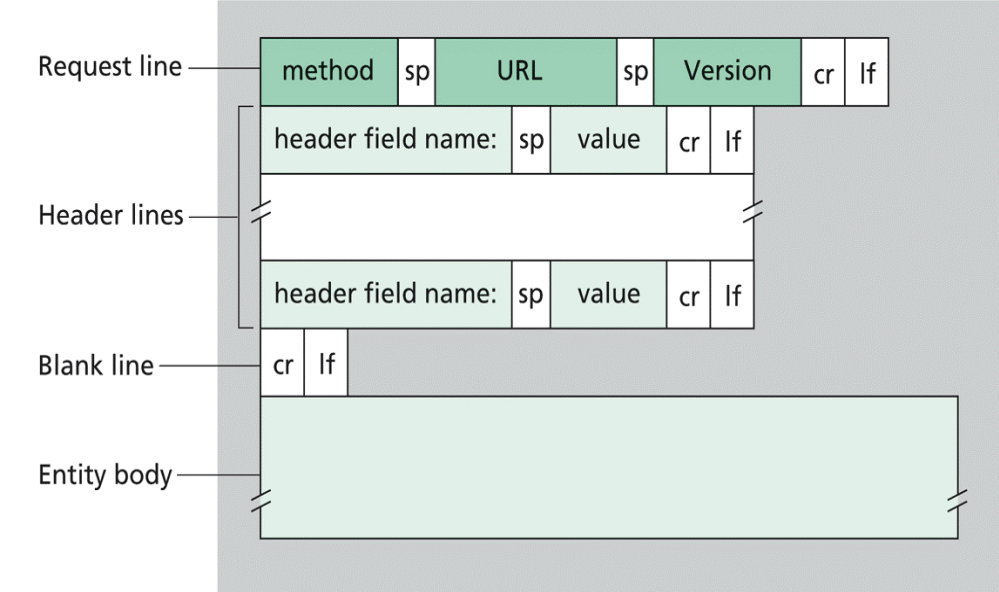


Figure 2.8 ♦ General format of a request message

Users who only want to see the list of online users, and existing chat-rooms will send "*query for peers*" message to the directory server. Server will return the list of records where each record contains the user's nickname, user's hostname, user's IP address, user server's protocol port, the popularity, and the nickname of the chatroom(s) where user is active (if any). The popularity P can be calculated as the number of users who joined particular chat-room divided by total number of online users in the period of 1 hour. After obtaining the information from the server, the user can join one of the existing chat-rooms by making direct TCP connection to target chat-room's server.

When user connects to the target chat-room server, the user should inform the directory server that she/he has joined the chat-room by using the "*Joined*" message which contains the nickname of the user whose chatroom is joined. The server should acknowledge the joined message and insert the nickname of the chatroom and current time in the user's record. When a client wants to exit from the chatroom, he/she should send an "*exit chatroom*" message upon receiving which the server will delete the chatroom nickname form the user's record, and send an acknowledgement to the client. Argument of the "exit chatroom" is the nickname of the chatroom and current

time. Directory server will calculate the current popularity of the chat-room server as n/N where n is current number of users in the chatroom and N is the total number of online users in the period of one hour.

When user wishes to go offline she/he will send message "*Offline*" which should result in removing of user's record from the directory server (we will also remove the rating to keep the things simple).

All messages sent to the directory server can resemble the HTTP request message. For example you can have "Online", "query for peers", "Joined", "exit chatroom" and "Offline" in the method field. In the so called URL and version fields you can put client's host name and protocol port address. The directory server should be multi-threaded and should handle concurrent writings to the media file directory (For synchronized file access among threads, you may want to take a look at "synchronizing threads" in the Java tutorial).

When the directory server replies to a P2P chat-client, it will use the format similar to the HTTP response message. The format of the HTTP response message is shown in the figure below.

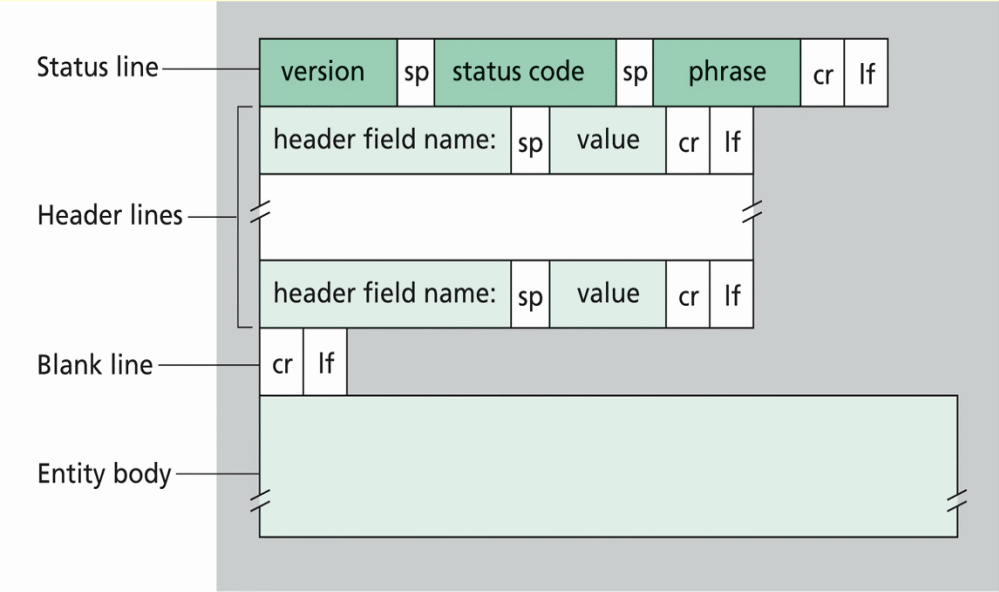


Figure 2.9 ♦ General format of a response message

The version field can be omitted. A status code and a status phrase are used to acknowledge the various request messages sent by clients. At least two status codes (and corresponding phrases) should be implemented. These are 200 (OK) and 400 (ERROR). Other status codes may be defined (please describe them in your system documentation). When acknowledging a "query for peers" request, directory entries in the query result can be transmitted using the header lines. Each header line contains an

entry for one online user, including, his/her nickname, his/her hostname, IP address, protocol port, rating and joined chatroom.

Protocol requirements between P2P chat-client and directory server

For messages exchange between the directory server and a P2P client, we assume that the maximum transfer unit (MTU) size of 80 bytes. Therefore you need to segment longer messages into smaller packets. Make sure that packets can be re-assembled into original message at the destination. For messages sent to the directory server, each packet should contain a first line which indicates the host name and address of the sending client, this way the directory server can recognize which client this packet was sent from.

In order to implement reliability we will need acknowledgements and time-outs for individual packets. Each packet has to be acknowledged which means that an "ACK" message has to be implemented. This also means that packets need to have sequence numbers. Examples of communication scenarios are given below:

P2P client			Directory s
Online, block#1	---------->		
		<----------	ACK, bloc
Online, block#2	---------->		
		<----------	ACK, bloc
etc.			

P2P client			Directory s
query for peers	---------->		
		<----------	data, bloc
ACK, block#1	---------->		
		<----------	data, bloc
ACK, block#2	---------->		
etc.			

The protocol handles lost packets by having the sender of the data packets use a timeout with retransmission. If a data packet is lost, the sender eventually times out and retransmits the packet. If an acknowledgment packet is lost, the sender of the data packet still times out and retransmits the data packet. In this case the receiver of the data packets notes from the block sequence number in the data packet that it is a duplicate, so it ignores the duplicate of the data packet and retransmits the acknowledgement.

There is a subtle problem in the above specification of the protocol. If both the sender and receiver use a timeout with retransmission, and both retransmit whatever they last sent, a condition termed the *sorcerer's apprentice syndrome* results (see RFC 1123). Actually, with the above scheme, when a time-out occurs (say because the ACK was delayed in the network), every further data packet and acknowledgement packet will be sent twice. The correction to this syndrome is for the sender never to retransmit a data packet if it receives a duplicate acknowledgement.

In order to determine the time-out interval you should use the round-trip time estimation procedure outlined in the section 3.5.3 of your text book. Please use parameters alpha and beta from the book and also assume that initial EstimatedRTT=100ms.

P2P chat-client and its Interaction with P2P chat-server

A P2P client is a client of both the directory server and the P2P server. The P2P client shall provide a text-based user interface for a user to interact with the directory server. With this interface, commands such as "Online", "query for peers", "Joined", "exit chatroom" and "Offline" are issued, responses returned from the directory server are displayed. A user can then select a chatroom server from a list of P2P chat-servers, and connect to it.

We do not enforce a particular format on how you maintain database of user entries at the directory server. Similarly, we do not insist on a particular format for the query results of a directory listing request. However, you should document your design choice in the system documentation, and clearly describe how to use your user interface to issue commands or to select a P2P chat-server in the user documentation.

Between a P2P client and a P2P server, we will use simple message exchange. A P2P chat-server must be able to handle multiple simultaneous users in parallel. This means that the P2P server is multi-threaded and launches a separate thread for each connected chat-client. In the main thread, the server listens to a fixed port. When it receives a TCP connection request, it creates a new TCP socket and services the request in a separate thread.

Port number assignment

The same port number should be used for both the directory server and the P2P server (because they use different protocols, namely UDP and TCP respectively, this would

not be a problem even when the two servers are running on the same host). Each group will be assigned a unique port number upon our receiving your group information. Before then, you are free to use one that is not widely used on the Internet. Please refer to section "About port numbers" below.

Other aspects

The basic ingredients for your code for TCP and UDP are given in sections 2.7 and 2.8 in the textbook respectively. Both client and server codes for a simple capitalization service are given. Note that we do not insist on graphical interfaces in your programs; text-based interfaces are fine. Both directory server and P2P server programs should have all received messages printed on the standard output display.

Project Logistics

Grouping

- You need to form a group of 3 (or 2) students to implement the course project
- We will collect the project submission on a group basis. Please elect a student representative within your group. The group representative will inform the TAs by email (a) the group composition (b) group representative information. The group representative is in charge of submitting the project hand-in's. Deadline for submission of group composition is 16. March at 8pm. Failure to register your group by 16. March at 8pm will results in project mark reduction of 10% per day.
- After registering your group you will get the block of protocol ports for your demo.

Due Date

- Your final project submission is due 29. March at 8pm . It shall contain two parts: Java programs, and a project documentation. There shall be three programs for Directory Server, P2P client, and P2P server respectively. Your documentation shall include the following:
- (a) user documentation: describes how to compile and run your programs, and (b) system documentation: describes data structures, design choices, and important algorithms used.

Hand-in

- Procedure for project source files and documentation files will be announced two weeks before the submission due date.
- Project documentation shall be in .pdf,.doc or .txt format.
- Students will be requested to give demo to the TAs on 30. March between 9am - 4pm in labs which will be announced in week of March 23.
-

- The schedule of demos will be made in the week of March 23, and students should sign up at schedule sheets at the door of room ENG 261.
- Demos will be conducted in extended lab slots on 30. March and will account for 14 points.
- Any late submissions will incur 25% mark deduction per day. This means that projects submitted between 8:01 pm on 29. March and 8pm on 30. March will get mark reduced by 25% and so on.
- In case of dispute and split of the group members, each part of the old group will be graded as a new group with full project requirements.

Plagiarism:

The project must be done independently by the project group. Copying from other groups is strictly prohibited. Students involved in plagiarism will get zero mark for the course project. Please put your code in secure place to prevent unauthorized plagiarism.

About port numbers :

At any given time, multiple processes can use either UDP or TCP (don't worry we'll learn TCP and UDP in detail in the weeks to come). Both TCP and UDP use 16-bit integer *port numbers* to differentiate between these processes. Both TCP and UDP define a group of well known ports to identify well-known services. For example, every TCP/IP implementation that supports FTP assigns well-known port of 21 (decimal) to the FTP server. TFTP servers, for the Trivial File Transfer Protocol, are assigned the UDP port of 69.

Clients on the other hand, use *ephemeral ports*, that is short-lived ports. These port numbers are normally assigned automatically by TCP or UDP to the client. Clients normally do not care about the value of the ephemeral port; the client just needs to be certain that the ephemeral port is unique on the client host. The TCP and UDP codes guarantee this uniqueness.

RFC 1700 (Request For Comments - this is the standard name for Internet documents) contains the list of port number assignments from the *Internet Assigned Authority (IANA)*. However, usually the file Port number is more up-to-date than RFC 1700. For registries also see Registries. The port numbers are divided into three ranges:

1. The *well-known ports*: 0 through 1023. These port numbers are controlled and assigned by IANA. When possible the same port is assigned to a given server for both TCP and UDP. For example, port 80 is assigned for a Web server, for both protocols, even though all implementations currently use only TCP.
2. The *registered ports:* 1024 through 49151. These are not controlled by the IANA, but the IANA registers and lists the uses of these ports as a convenience to the

community. When possible the same port is assigned to a given service for both TCP and UDP. For example, ports 6000 through 6063 are assigned for an X Window server for both protocols, even though all implementations currently use only TCP. The upper limit of 49151 for these ports is new, as RFC 1700 lists the upper range as 65535 (FYI only).

3. The *dynamic or private* ports, 49152 through 65535. The IANA says nothing about these ports. These are what we call *ephemeral* ports.

Some interesting points:

- Unix systems have the concept of a reserved port, which is any port less than 1024. These ports can be assigned to a socket by a super user process. All the IANA well-known ports are reserved ports; hence the server allocating this port (such as the FTP server) must have super user privileges when it starts.
- Historically, Berkeley-derived implementations have allocated ephemeral ports in the range 1024-5000. This was fine in the early 1980s, when server hosts were not capable of handling more than 3977 clients at any given time. Therefore some systems allocate ephemeral ports differently to provide more ephemeral ports. For example, Solaris allocates ephemeral ports in the range 32768-65535.
- There are few clients (not servers) that require a reserved port as part of the client-server authentication: the *rlogin* and *rsh* clients are the common examples. These clients call the library function resvport to create a TCP socket and assign an unused port in the range 513-1023 to the socket. This function normally tries to bind port 1023 and if it fails, tries to bind 1022, and so on, until it either succeeds or fails on port 513.

Note that during program development, you as the client-server application programmer must be responsible to avoid conflicts in choosing protocol port for your server application!