Faculty of Computer and
Information Sciences

IRC Server / Client

Prepared for: Computer Science Department, Information Sciences Department

October 30, 2011

Faculty of Computer and
Information Sciences

Table of Contents

Summary

What is IRC

IRC is a global, distributed, real-time chat system that operates over the Internet. An IRC network consists of a set of interconnected servers. Once users are connected to an IRC server, they can converse with other users connected to any server in the IRC network. IRC provides for group communication, via named channels, as well as personal communication through "private" messages. In our project we will only support private messages.

Using IRC

To build an IRC server it is best to see how it works. Download **mIRC** (http://www.mirc.com/get.html) and connect to the **allnetworks** server. Join any channel in this server and communicate together and with other people on the server.

Objectives

The following are the objectives of this project:

1. Gain experience in developing concurrent network applications.
2. Learn how to read an RFC document and understand it's method of protocol definition.
3. Learn how to utilize your knowledge of socket programming in implementing a global standard. IRC is one of the popular application layer protocols (along with HTTP, FTP and DNS), and you are implementing that protocol.
4. Learn how to implement routing using a protocol similar to OSPF.
5. Learn design patterns used in large scale applications

Overview

How IRC Works

An IRC network is composed of a set of nodes interconnected by virtual links in an arbitrary topology. Each node runs a process that we will call a routing daemon. Each routing daemon maintains a list of IRC users available to the system. The figure below shows a sample IRC network composed of 5 nodes. The solid lines represent virtual links between the nodes. Each node publishes a set of users (i.e., the nicks of the IRC clients connected to it) to the system. The dotted lines connect the nodes to their user sets.

The usage model is the following: If Bob wants to contact Alice, the IRC server on the left first must find the route or path from it to the node on the right. Then, it must forward Bob's message to each node along the path (the dashed line in the figure) until it reaches the IRC server at Alice's node, which can then send the message to the client Alice.

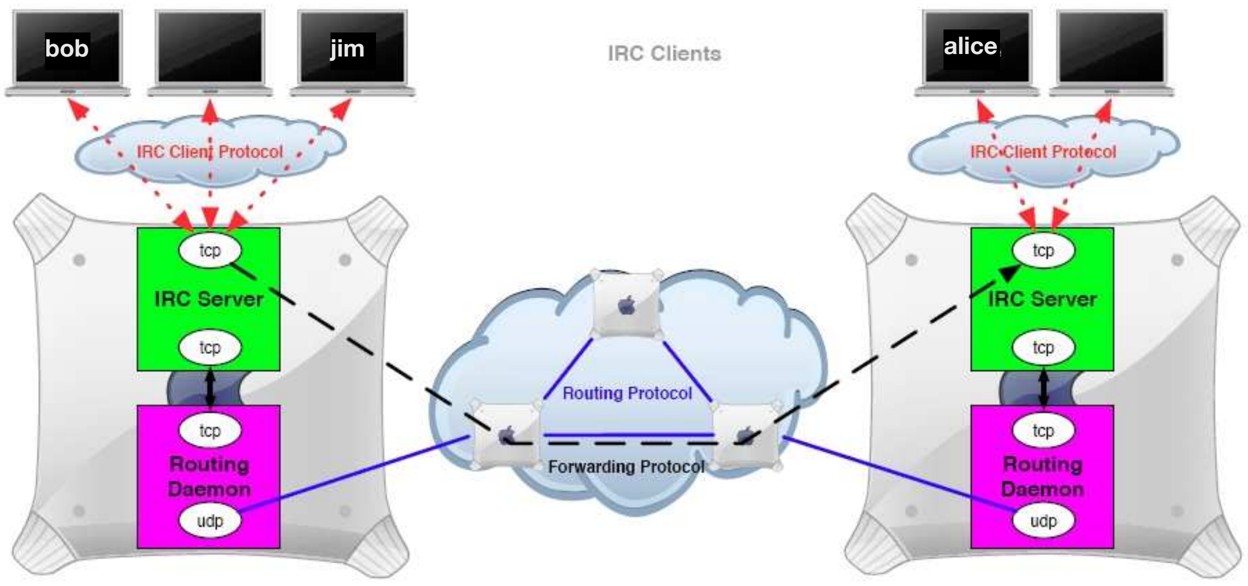


Figure 1: Operation of a normal IRC network.

Commands and replies

IRC is based on a line-based structure with the client sending single-line messages to the server, receiving replies to those messages and receiving copies of some messages sent by other clients.

Phase 1: Standalone Server

Overview

In this phase, you only need to implement a standalone IRC server. You can assume that there is only one IRC server and all clients are connected to the server. So referring to Figure 1, we only want to support communication between Bob and Jim.

Definitions

- **nodeID**: unique identifier that identifies an IRC server, or node.
- **destination**: IRC nickname as a null terminated character string. As per the IRC RFC, destinations will be at most 9 characters long and may not contain spaces.
- **IRC port**: The TCP port on the IRC server that talks to clients.

Basics

Messages

Servers and clients send each other messages which may or may not generate a reply. If the message contains a valid command, as described in later sections, the client should expect a reply.

Each IRC message may consist of two main parts: the command, and the command parameters . The command must either be a valid IRC command or a three (3) digit number represented in ASCII text. IRC messages are always lines of characters terminated with a CR-LF (Carriage Return - Line Feed) pair, and these messages shall not exceed 512 characters in length, counting all characters including the trailing CR-LF. Thus, there are 510 characters maximum allowed for the command and its parameters.

Numeric Replies

Most of the messages sent to the server generate a reply of some sort. The most common reply is the numeric reply, used for both errors and normal replies. The numeric reply must be sent as one message consisting of the sender prefix, the three digit numeric, and the target of the reply. In all other respects, a numeric reply is just like a normal message, except that the keyword is made up of 3 numeric digits rather than a string of letters. See the appendix for a list of replies and examples.

Connection Registration

For a user to initiate the connection with the server, the user must initiate two commands: NICK, and USER. These commands are interchangeable, so whichever is used first doesn't affect the registration process.

USER Command

The USER message is used at the beginning of a connection to specify the username, hostname, servername and realname of a new user. If it succeeds it will respond with "OK".

Format

Parameters: <username> <hostname> <servername> <realname>

Parameters

- username: The username of the user attempting to connect to the server.
- hostname: The hostname (machine name) of the user connecting to the server. This parameter is ignored by the server, because the server should obtain the hostname via reverse lookup.
- servername: The name of the server.
- realname: The real name of the user. Should be prefixed with a colon (:) because it could contain spaces.

Responses

ERR_NEEDMOREPARAMS

ERR_ALREADYREGISTRED: If a client attempts to send the USER command twice, it will receive this error message.

Example

USER guest tolmoon tolsun :Ronnie Reagan

--> User registering themselves with a username of "guest" and real name "Ronnie Reagan".

NICK Command

NICK message is used to give user a nickname or change the previous one. The <hopcount> parameter is only used by servers to indicate how far away a nick is from its home server. A local connection has a hopcount of 0. If supplied by a client, it must be ignored. If it succeeds it will respond with “OK”.

Format

Parameters: <nickname>

Parameters

- nickname: The desired nickname of the user.

Responses

ERR_NONICKNAMEGIVEN

ERR_NICKNAMEINUSE: If the server receives, from a directly connected client, a NICK indentical to the NICK of an existing local local user, the server may issue an ERR_NICKNAMEINUSE to the local client, and drop the NICK command.

Example

NICK Wiz

--> Change nickname to Wiz

QUIT Command

A client session is ended with a quit message. The server must close the connection to a client which sends a QUIT message. If a "Quit Message" is given, this will be sent instead of the default message, the nickname. If it succeeds it will respond with “OK”.

Format

Parameters: [<quit message>]

Behavior

If, for some other reason, a client connection is closed without the client issuing a QUIT command (e.g. client dies and EOF occurs on socket), the server must still notify other users of the client's departure. The servers is required to fill in the quit message with some sort of message reflecting the nature of the event which caused the departure.

Example

QUIT :Gone to have lunch

--> Quit and send this message to users in my channels. The colon indicates a parameter containing spaces.

Sending Messages

PRIVMSG Command

Send messages to users. The target is one or more nicknames. If it succeeds it will respond with “OK”.

Format

Parameters: <target>{,<target>} <text to be sent>

Behavior

Our server should send the message to the target(s) specified in the command. See the example below.

Responses

ERR_NORECIPIENT

ERR_NOTEXTTOSEND

ERR_NOSUCHNICK

Example

The server will receive this from Adam:

PRIVMSG Angel :I’m sending a message!

The server will send this to Angel:

:Adam PRIVMSG :I’m sending a message!

Behavior with other commands

For all other commands, your server must return ERR_UNKNOWNCOMMAND. If you are unable to implement one of the above commands (perhaps you ran out of time), your server must return the error code ERR_UNKNOWNCOMMAND, rather than failing silently, or in some other manner.

Phase 2: Routing Daemon

Overview

The routing daemon will be a separate program from your IRC server. Its purpose is to maintain the routing state of the network (e.g., build the routing tables or discover the routes to destinations). When the IRC server wants to send a message to a remote user, it will ask the routing daemon how to get there and then send the message itself. In other words, the routing daemon does the routing and the IRC server does the forwarding. Refer to Figure 1 to understand where the routing daemon is topologically positioned.

In your implementation, the routing daemon will communicate with other routing daemons (on other nodes) over a UDP socket to exchange routing state. It will talk to the IRC server that is on the same node as it via a local TCP socket. The IRC server will talk to other IRC servers via the TCP socket that it also uses to communicate with clients. It will simply use special server commands. This high level design is shown in the two large IRC server nodes in Figure 1.

In order to find out about the network topology, each routing daemon will receive a list of neighboring nodes when it starts. In this project, you can assume that no new nodes or links will ever be added to the topology after starting, but nodes and links can fail (i.e., crash or go down) during operation (and may recover after failing).

Definitions

- **node**: An IRC server and routing daemon pair running together that is part of the larger network. In the real world, a node would refer to a single computer, but we can run multiple “virtual” nodes on the same computer since they can each run on different ports. Each node is identified by its nodeId.
- **neighbor**: Node 1 is a neighbor of node 2 if there is a virtual link between 1 and 2. Each node obtains a list of its neighbors’ nodeIDs and their routing and forwarding ports at startup.
- **IRC port**: The TCP port on the IRC server that talks to clients and other IRC servers.
- **forwarding port**: Same as IRC port.
- **routing port**: The UDP port on the routing daemon used to exchange routing information with other routing daemons.
- **local port**: The TCP port on the routing daemon that is used to exchange information between it and the local IRC server. For example, when the IRC server wants to find out the route to remote user, it queries the routing daemon on this port. The socket open for listening will be on the routing daemon. The IRC server will connect to that open socket using the local port.
- **OSPF**: The shortest path link state algorithm that inspires the (much simpler) algorithm you will implement
- **routing table**: The data structure used to store the “next hops” that packet should take used in OSPF.

Link-State Routing Protocol

Basic Operation

You will implement a link-state routing protocol similar to OSPF, which is described in further details in the OSPF RFC (refer to the references section). Note, however, that your protocol is greatly simplified compared to the actual OSPF specification. As described in the references, OSPF works by having each router maintain an identical database describing the network's topology. From this database, a routing table is calculated by constructing a shortest-path tree. Each routing update contains the node's list of neighbors and users. Upon receiving a routing update, a node updates its routing table with the "best" routes to each destination. In addition, each routing daemon must remove entries from its routing table when they have not been updated for a long time. The routing daemon will have a loop that looks similar the following:

```
while (1) {
        /* each iteration of this loop is "cycle" */
        wait_for_event(event);
        if (event == INCOMING_ADVERTISEMENT) {
                process_incoming_advertisements_from_neighbor();
        }
        else if (event == IT_IS_TIME_TO_ADVERTISE_ROUTES)
        {
                advertise_all_routes_to_all_neighbors();
                check_for_down_neighbors();
                expire_old_routes();
                delete_very_old_routes();
        }
}
```

Let's walk through each step. First, our routing daemon A waits for an event. If the event is an incoming link-state advertisement (LSA), it receives the advertisement and updates its routing table if the LSA is new or has a higher sequence number than the previous entries. So, if the routing advertisement is from a new router B or has a higher sequence number than the previously observed advertisement from router B, our router A will flood the new announcement to all of its neighbors except the one from which the announcement was received, and will then update its own routing tables.

If the event indicates a predefined period of time has elapsed and it is time to advertise the routes, then the router advertises all of its users and links to its direct neighbors. If the routing daemon has not received any such advertisements from a particular neighbor for a number of advertisements, the routing daemon should consider that neighbor down and mark it as down so it no longer attempts to send it LSAs.

If B receives an LSA announcement from A with a lower sequence number than it has previously seen (which can happen, for example, if A reboots), B should echo the prior LSA back to A. When A receives its own announcement back with a higher sequence number, it will increment its transmitted serial number to exceed that of the older LSAs.

Each routing announcement should contain a full state announcement from the router – all of its neighbors and all of its users. This is an inefficient way to manage the announcements, but it greatly simplifies the design and implementation of the routing protocol to make it more tractable for a 6 week project. Each time your router originates a new LSA, it should increment the serial number it uses. When a router receives an updated LSA, it recomputes its local routing table. The LSAs received from each of the peer nodes tell the router a link in the complete router graph. When a router has received all of the LSAs for the network, it knows the complete graph. Generating the user routing table is simply a matter of running a shortest-paths algorithm over this graph.

Reliable Flooding

OSPF is based upon reliable flooding of link-state advertisements to ensure that every node has an identical copy of the routing state database. After the flooding process, every node should know the exact network topology. When a new LSA arrives at a router, it checks to see if the sequence number on the LSA is higher than it has seen before. If so, the router reliably transmits the message to each of its peers except the one from which the message arrived. The flooding is made reliable by the use of acknowledgement packets from the neighbors. When router A floods an LSA to router B, router B responds with an "LSA Ack." If router A does not receive such an acknowledge from its neighbor within a certain amount of time, router A will retransmit the LSA to B.

With the information contained in the LSAs, each server should be able to deliver messages from one user to another without much trouble.

Protocol Specification

The table below shows the routing update message format, with the size of each field in bytes in parenthesis.

Version (1), TTL (1), Type (2)
Sender nodeID (4)
Sequence number (4)
Number of Link Entries (4)
Number of User Entries (4)
Link Entries (Variable)
User Entries (Variable)

- **Version** – the protocol version, always set to 1
- **TTL** – the time to live of the LSA. It is decremented each hop during flooding, and is initially set to 32.
- **Type** – Advertisement packets should be type 0 and Acknowledgement packets should be type 1.
- **Sender nodeID** – The nodeID of the sender of the message, not the immediate sender.
- **Sequence number** – The sequence number given to each message
- **Num link entries** – The number of link table entries in the message.
- **Num user entries** – The numbers of users announced in the message
- **Link entries** – Each link entry contains the nodeID of a node that is directly connected to the sender. This field is 4 bytes.
- **User entries** – Each user entry contains the name of the destination user as a null terminated string.

An acknowledgement packet looks very similar to an announcement packet, but it does not contain any entries.It contains the sender nodeID and sequence number of the original announcement, so that the peer knows that the LSA has been reliably received.

Protocol Requirements

Your implementation of OSPF should have the following features:

- Given a particular network configuration, the routing tables at all nodes should converge so that forwarding will take place on the paths with shortest length.
- Remove the LSAs for a neighbor if it hasn't given any updates for some period of time.
- If a node or link goes down (e.g., routing daemon crashes, or link between them no longer works and drops all messages), your routing tables in the network should re-converge to reflect the new network graph. You shouldn't have to do anything more to make sure this happens, as the above protocol already ensures it.

Local Server-Daemon Protocol

This section describes the mini-protocol that an IRC Server uses to talk to the local routing daemon on the same node. It is important that you follow these specifications carefully because we will test your routing daemon independently of your IRC server!

The routing daemon listens on the local port when it starts up to service route lookup requests. When the IRC server on the same node starts up, it connects to the local port of the routing daemon. The IRC server will communicate with the routing daemon using a simple protocol. This is a line-based protocol like the IRC-protocol itself. Each request and response pair looks like this:

command arguments . . .

results . . .

Where **command** is the name of the request, **arguments . . .** is a space-separated list of arguments to the command, and **results . . .** is a space-separated list of results returned. All requests and responses are terminated with a newline character (\n) and are case sensitive, but some responses have multiple lines. If any command fails to process, the routing daemon will respond with “ERROR”.

Routing Daemon Commands

ADDUSER

Request	ADDUSER nick
Response	OK
Description	This request is issued when a new user is registered with the IRC server. The user's nick is added to the routing daemon's list of local users so that other nodes can find the user.

Examples	req: ADDUSER bob resp: OK req: ADDUSER alice resp: OK

REMOVEUSER

Request	REMOVEUSER nick
Response	OK
Description	This request is issued when a local user leaves the IRC server. The user's nick is removed from the routing daemon's list of local destinations so that other nodes will know that they can no longer reach the user there.
Examples	req: REMOVEUSER bob resp: OK req: REMOVEUSER alice resp: OK

NEXTHOP

Request	NEXTHOP nick
Response	OK nodeID distance NONE
Description	This request is used to find nodeID of the next hop to use if we want to forward a message to the user nick. It should return OK if the routing table has a valid next hop for the nick along with the distance to that destination, and NONE otherwise (e.g., if the destination's distance is not known or user does not exist).
Examples	req: NEXTHOP bob resp: OK 2 5 req: NEXTHOP alice resp: OK 3 2 req: NEXTHOP baduser resp: NONE

USERTABLE

Request	USERTABLE
Response	OK size

Description	If this request is issued, the routing daemon should respond with OK, the size or number of entries in the routing table, and a multi-line response with its entire user table in the following format: nickname next-hop distance nickname next-hop distance nickname next-hop distance ... Where nick is the nickname, next-hop is the nodeID of the next hop, and distance is the current distance value for that destination. You should not include local nicknames in this list. The order of entries does not matter. Your IRC Server will probably not need to use this command. We will use this to test your routing daemon. This would be similar to calling NEXTHOP on every user on the server.
Examples	req: USERTABLE resp: OK 3 bob 2 2 alice 3 1 jim 3 2

Phase 3: Server/Daemon Integration

Overview

Now that we have covered the IRC server, the routing protocols, and the server-daemon protocol, the only major issue remaining is how to extend your IRC Server to use the routing daemon so it can send messages to users on remote IRC Servers.

Remember that the PRIVMSG command targets nicknames. The IRC server must first determine if there is a local user with that nickname. If not, then it should try to locate the user on a remote IRC Server (using the routing daemon) and, if found, forward the message to that IRC Server which will then send it to the target. If the target is not found, then you should send the user an ERR_NOSUCHNICK error.

Requirements

Your extensions to the IRC server should have the following features:

- Connect to the routing daemon's local port when it starts up. You can assume the routing daemon will be started first.
- When a new user is registered with the IRC server, it should add the user's nick to the routing daemon's list of users using the ADDUSER request.
- When a user leaves the IRC server, it should remove the user's nick from the routing daemon's list of users using the REMOVEUSER request.
- If a user changes his or her nick, remove the old nick and add the new one to the routing daemon.
- When a PRIVMSG is sent to a nick that we don't know locally, the IRC Server should ask the routing daemon to find it, if possible, and then the IRC Server would forward the message to that user. The remote IRC server receiving the message should send it to the target user the same way it would send any other PRIVMSG to him or her.
- If the target is not found, then you should send the user an ERR_NOSUCHNICK error.
- The PRIVMSG command should support multiple targets; i.e., the PRIVMSG command may have a comma-separated list of target users (nicknames) that should all be sent the message.
- If the routing daemon dies or you cannot communicate with it, your IRC server may exit.

Message Forwarding

Once the IRC Server has found the next hop or route to a remote nickname, it must forward the message to the remote IRC Server. Here are a couple things to keep in mind:

- When using OSPF, you can only obtain the next hop from the routing daemon. Hence, each IRC server along the path will have to query its routing daemon to figure out where to send the packet next.
- When using OSPF, while forwarding, a node or virtual link may go down (or the target user may leave). In this circumstance, you can just drop the message. You do not have to inform the user that the message was dropped.
- IRC Servers and virtual links may go down and come back up. If you detect that your neighbor is down (i.e., the socket is closed), you should check to see if they have come back up at least once every 3 seconds. In fact, when the network first starts up, since only one server will come up at a time, all its neighbors will appear to be down at first.
- You should not have IRC Servers communicate if they are not neighbors.
- Your forwarding protocol should not be “flood every message to every IRC server on the network.”
- You should make modifications to the routing daemon. We may test your IRC Server on our own routing daemon.

Server-to-Server Commands

SERVER

Request	SERVER NodeID
Response	OK
Description	Once server A starts, it will load it’s neighbor list from the configuration file (see the next section). Then it will attempt to communicate with every neighbor by sending a SERVER message, containing the node Id of server A. Once this message is received by server B, it will be able to receive and send messages to server A without a problem.
Examples	req: SERVER 2 resp: OK

SPRIVMSG

Request	SPRIVMSG <source> <target> :<message>
Response	OK
Description	This message is used by the IRC Server to forward a private message to another server. If the receiving server fails to find the target nickname in its database, it will ask its routing daemon. If the routing daemon finds the next hop, the message will be forwarded again. If the routing daemon fails to find the next hop for the target nickname, the message will be dropped.
Examples	req: SPRIVMSG adam angel :Hello ! resp: OK

Implementation Details

Phase 1: Standalone IRC Server

Server

Your server should be able to support multiple clients concurrently. While the server is waiting for a client to send the next command, it should be able to handle inputs from other clients. Also, your server should not hang up if a client sends only a partial command or an erroneous command. This means the server should not fail or crash if a malicious client is present. This will be tested for with our test cases. As a rule with all networking applications, be very lenient with what you receive, but very strict with what you send. So you should be flexible in what you receive to prevent it from crashing your server.

The server should be executed from the command prompt. It should receive as an argument the port number it listens to.

```
C:\>server.exe 9000
```

Client

The client used for testing will be IRCClient (see the Testing section below). However, you could implement a graphical client to simplify the command sending process.

Phase 2: Routing Daemon

Your routing daemon must take the following command line arguments in any order.

```
C:\>routed.exe -i nodeID -c config_file [options]
```

-i integer: NodeID. Sets the nodeID for this process.

-c filename: Config file. Specifies the name of the configuration file that contains the information about the neighbor nodes. The format of this file is described below.

It should also recognize the following optional switches:

-a seconds: Advertisement cycle time. The length of time between each advertisement cycle. Defaults to 30.

-n seconds: Neighbor timeout. The elapsed time after which we declare a neighbor to be down if we have not received updates from it. You may assume that this value is a multiple of advertisement cycle time. Defaults to 120.

-r seconds: Retransmission timeout. The elapsed time after which a peer will attempt to retransmit an LSA to a neighbor if it has not yet received an acknowledgement for that LSA. This value is an integral number of seconds. Defaults to 3.

-t seconds: LSA timeout. The elapsed time after which we expire an LSA if we have not received updates for it. You may assume that this value is a multiple of advertisement cycle time. Defaults to 120.

Configuration File

This file describes the neighborhood of a node. The neighborhood of a node 1 is composed by node 1 itself and all the nodes n that are directly connected to 1. For example, in the below figure, the neighborhood of node 1 is {1, 2, 3}. The format of the configuration file very simple, and we will supply you with code to parse it. The file contains a series of entries, one entry per line. Each line has the following format:

nodeID hostname routing-port local-port IRC-port

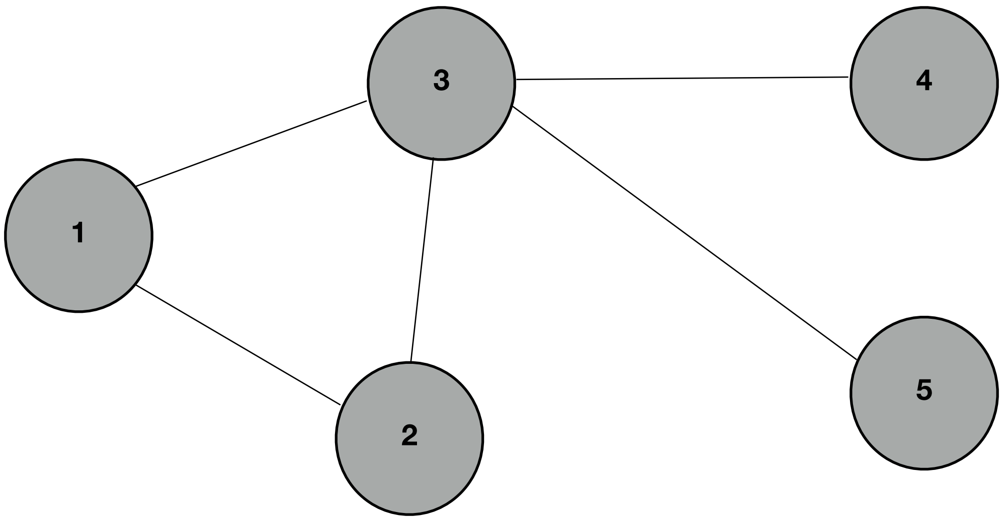
nodeID: Assigns an identifier to each node.

hostname: The name or IP address of the machine where the neighbor node is running.

local-port: The TCP port on which the routing daemon should listen for the local IRC server.

routing-port: The port where the neighbor node listens for routing messages.

IRC-port: The TCP port on which the IRC server listens for clients and other IRC servers.



Node 2	Node 5
2 localhost 20203 20204 20205	3 srv3.cis.asu.edu.eg 20206 20207 20208
1 srv1.cis.asu.edu.eg 20200 20201 20202	5 localhost 20209 20210 20211
3 srv3.cis.asu.edu.eg 20206 20207 20208	

How does a node find out which ports it should use as routing, IRC, and local ports? When reading the configuration file if an entry's nodeID matches the node's nodeID of the node (passed in on the command line), then the node uses the specified port numbers to route and forward packets. The above figure contains a sample configuration files corresponding to node 2 and node 5 for the network shown. Notice that the file for node 2 contains information about node 2 itself. Node 2 uses this information to configure itself.

Phase 3: Server/Router Integration

The integrated server version runs as follows:

```
C:\>server.exe nodeID config_file
```

nodeID: The nodeID of the node.

config_file: The configuration file name. The same format as the configuration file for the routing daemon.

Testing the entire network

This is how we will start your IRC network.

First, we start each routing daemon with the commands:

```
C:\>routed.exe -i 0 -c node0.conf ...

C:\>routed.exe -i 1 -c node1.conf ...

C:\>routed.exe -i 2 -c node2.conf ...
```

Each routing daemon will be started with its own configuration file to find out about its neighbors (described above) and its nodeID. In addition, we will pass it certain arguments to set the timer values.

Next, we will start each IRC server at each node:

```
C:\>server.exe 0 node0.conf

C:\>server.exe 1 node1.conf

C:\>server.exe 2 node2.conf
```

Grading

The project will be graded out of 100 points. These will be scaled into your practical grade.

Phase 1: 40 points

Successful implementation of phase 1 will earn you 30 points. This means that if the IRC server passes the test cases for it you will get the full grade.

Phase 2: 30 points

Test cases for this phase will isolate the routing daemon and test its capability to apply the routing mechanism in different topologies.

Phase 3: 20 points

In this phase, a complete IRC network will be tested to validate you have successfully integrated the IRC server with the client.

Style: 10 points

Poor design, documentation, or code structure will probably reduce your grade by making it hard for you to produce a working program and hard for the grader to understand it; egregious failures in these areas will cause your grade to be lowered even if your implementation performs adequately.

Deliveries and Policies

Delivery Plan

Project deliveries will be scheduled according to the following table:

Phase 1	10/12/2011 11:59 pm
Phase 2	24/12/2011 11:59 pm
Phase 3	Practical Exam Week

Phase 1 and 2 will be delivered using an automated delivery system. You will download the test application, connect it to your IRC Server/Routing Daemon, and let it run test cases. The application will grade your project based on the test cases, and upload the results to our servers. You will need to point the application to your source code, and enter a secret key provided for your team. Phase 3 will be delivered in the practical exam week, and the code and design decisions of **all three phases** will be discussed with the team members.

Policies

- You will work in teams of maximum **5** members. You will lose **10%** of your grade for every **12 hours** of delay.
- You are allowed to use the template code we provide, yet you are not obliged to use it. You can structure your code however you chose, yet be aware that 10% of the final grade goes to the style and structure of your code. So a well-designed project will not only benefit you in successfully completing all 3 phases, but it will earn you a valuable 10%.

Honor Code

This project will apply **strict** honor codes to your work. Violation of our honor code will result in you receiving **ZERO** in the practical grade part of the course. Following is a list of **COMPLETELY PROHIBITED** collaboration between teams:

1. Copying solutions from others. In particular, do not ask anyone to provide a copy of his or her solution or, conversely, give a solution to another student who requests it. Similarly, do not discuss algorithmic strategies to such an extent that you and your collaborator submit exactly the same solution. Use of solutions posted to websites, such as at other universities, is prohibited. We use a sophisticated tool that cross-checks every submitted project against every other project submitted this year. It catches common code, even if comments and variable names are changed. In fact, in order to "fool" it, you have to change so much code that it would be quicker to do the assignment yourself.
2. Studying another student's code. Do not read another solution submission whether in electronic or printed form, even to "check you’re going the right way."
3. Debugging code for someone else. When debugging code it is easy to inadvertently copy code or algorithmic solutions. It is acceptable to describe a problem and ask for advice on a way to track down the bug.

Appendix

Numeric Responses

The following table presents all the needed numeric responses made by the server, the description and format of each numeric response.

Numeric Response	Description	Format
ERR_NEEDMOREPARAMS	Returned to indicate to the client not enough parameters are provided.	461 <command> : Need more params
ERR_ALREADYREGISTRED	Returned by the server to a client sending the USER command after being already registered.	462 : You are already registered
ERR_NONICKNAMEGIVEN	Returned when a nickname parameter isn't found.	431 :No nickname given
ERR_NICKNAMEINUSE	Returned when a NICK message attempts to select an already used nick	433 <nick> :Nickname is already in use
ERR_NORECIPIENT	No recipient provided to the command	411 :No recipient given (<command>)
ERR_NOTEXTTOSEND	No text provided to send	412 :No text to send
ERR_NOSUCHNICK	Used to indicate the nickname parameter isn’t registered.	401 <nickname> :No such nick/channel
ERR_UNKNOWNCOMMAND	Returned to a registered client to indicate that the command sent is unknown by the server.	421 <command> :Unknown command

Design Patterns

Following are the design patterns used in the template code.

Singleton

The Singleton pattern restricts the instantiation of a class to one object.This is useful when exactly one object is needed to coordinate actions across the system.

We use this pattern in the ServerBackend, which should only be created once. The server backend stores the users and the sessions, and so it doesn't make sense to be able to create more than one object of the server backend.

Command Pattern

The command pattern is a design pattern in which an object is used to represent and encapsulate all the information needed to perform a certain command.

This pattern is very useful for us in a networks application. Since the IRC protocol is basically a set of commands, the pattern enables us to easily add more commands as we go. A command is a class that inherits from the IRCCommandBase and holds all the parameters of a command. A command handler is a class that inherits from CommandHandlerBase, and it is responsible for executing the command.

Factory Method

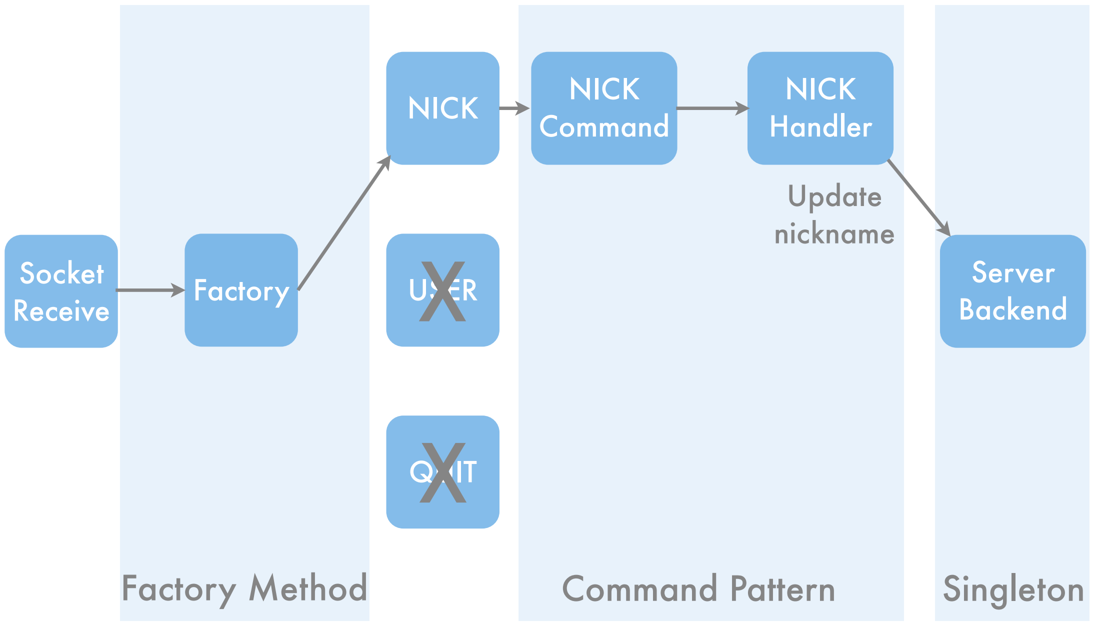
The Factory Method pattern deals with the problem of creating objects (products) without specifying the exact class of object that will be created.

This pattern is useful to us because we receive the command as a string, and we don't know what Command object we should create, so we have to parse the command. The factory method is responsible for reading the message, and returning a suitable command object based on that message. The method returns the CommandBase object so it could be used with any type of command. Here is sample pseudocode:

```
CommandBase GetCommand(message)
{
    type = GetType(message);
    if (type == NICKCommand)
        return new NICKCommand();
    ...
}
```

How the Patterns Fit Together

The following figure demonstrates a simple flow of execution in the IRC Server, and how each design pattern we used fits into that flow.



References

The Carnegie-Mellon University annotated IRC RFC reference is a highly recommended read. It will give you a fuller understanding of the IRC protocol and its inner-workings.

http://www.cs.cmu.edu/~srini/15-441/S10/project1/rfc.html

Other IRC references:

http://www.cs.cmu.edu/~srini/15-441/S10/assignments.html

http://en.wikipedia.org/wiki/Internet_Relay_Chat

Design Patterns:

Singleton: http://www.dofactory.com/Patterns/PatternSingleton.aspx

Command Pattern: http://www.dofactory.com/Patterns/PatternCommand.aspx

Factory Method: http://www.dofactory.com/Patterns/PatternFactory.aspx