SPL - 221 Assignment 3

Published on: 20/12/2021 Due date: 9/1/2022

Responsible TA's: Ella Avrahamov , Ameer Abu Ganeem

1. General Description

In this assignment you will implement a simple social network server and client. The communication between the server and the client(s) will be performed using a binary communication protocol. A registered user will be able to follow other users and post messages. Please read the entire document before starting.

The implementation of the server will be based on the **Thread-Per-Client (TPC)** and **Reactor** servers taught in class.
The servers, as seen in class, only support pull notifications.
Any time the server receives a message from a client it can replay back to the client itself. But what if we want to send messages between clients, or broadcast an announcement to a group of clients? We would like the server to send those messages directly to the client without receiving a request to do so. this behaviour is called push notifications.
The first part of the assignment will be to replace some of the current interfaces with new interfaces that will allow such a case.
Note that this part changes the servers pattern and **must not know** the specific protocol it is running. The current server pattern also works that way (Generics and interfaces).

Once the server implementation has been extended you will have to implement an example protocol. The BGS (Ben Gurion Social) Protocol will emulate a simple social network.
Users need to register to the service. Once registered, they will be able to post messages and follow other users. It is a binary protocol that uses pre defined message length for different commands. The commands are defined by an opcode, a short number at the start of each message. For each command, a different length of data needs to be read according to it's specifications. In the following sections we will define the specifications of the commands supported by the BGS protocol.

Unlike real social network you will not work with real databases. You will need to save data (Users, Passwords, Messages, ect...). You only need to save information from the time the server starts and keep it in memory until the server closes.

1.1 Establishing a client/server connection

Upon connecting, a client must identify himself to the service. A new client will issue a Register command with the requested user name and password. A registered client can then login using the Login command. Once the command is sent, the server will reply on the validity of the username and password. Once a user is logged in successfully, he can submit other commands. The register and login commands are stated in the following section. Note that the register command will not perform aotumatic login (you will need to call login after it).

1.2 Supported Commands

The BGS protocol supports various commands needed in order to share posts and messages. There are two types of commands, Server-to-Client and Client-to-Server. The commands begin with 2 bytes (short) to describe the opcode. The rest of the message will be defined specifically for each command as such:

2 bytes	Length defined by command
Opcode	…

We supplied functions that encode \ decode between 2 bytes and short for both java and C++ in the assignment page.

The BGS protocol supports 11 types of messages:

- 1-8 are Client-to-Server messages
- 9-11 are Server-to-Client messages

Opcode	**Operation**
1	Register request (REGISTER)
2	Login request (LOGIN)
3	Logout request (LOGOUT)
4	Follow / Unfollow request (FOLLOW)
5	Post request (POST)
6	PM request (PM)
7	Logged in States request (LOGSTAT)
8	Stats request (STAT)
9	Notification (NOTIFICATION)
10	Ack (ACK)
11	Error (ERROR)
12	Block (BLOCK)

NOTE: Every message should be ended with the symbol ';', you can assume that none of the strings inside the message contains this symbol.

REGISTER Messages:

2 bytes	string	1 byte	string	1 byte	string	1 byte
Opcode	Username	0	Password	0	birthday	0

Messages that appear only in a Client-to-Server communication.

A REGISTER message is used to register a user in the service. If the username is already registerd in the server, an ERROR message is returned. If successful an ACK message will be sent in return. Both string parameters are a sequence of bytes in UTF-8 terminated by a zero byte (also known as the '\0' char).

Parameters:

- Opcode: 1.
- Username: The username to register in the server.
- Password: The password for the current username (used to log in to the server).
- Birthday: the birthday of the user (should be in the format DD-MM-YYYY)

Command initiation:

• This command is initiated by entering the following text in the client command line interface:
REGISTER <Username> <Password> <Birthday>

LOGIN Messages:

Messages have the following format:

2 bytes	string	1 byte	string	1 byte	1 byte
Opcode	Username	0	Password	0	Captcha

Messages that appear only in a Client-to-Server communication.
A LOGIN message is used to login a user into the server. If the user doesn't exist or the password doesn't match the one entered for the username, sends an ERROR message. An ERROR message should also appear if the current client has already succesfully logged in.
An ERROR message should appear also if the captcha byte is 0.
Both string parameters are a sequence of bytes in UTF-8 terminated by a zero byte.

Parameters:

- Opcode: 2.
- Username: The username to log in the server.
- Password: The password for the current username (used to log in to the server).
- Captcha: is to simulate captcha (must be 1, for successful login)

Command initiation:

• This command is initiated by entering the following text in the client command line interface:
LOGIN <Username> <Password> <Captcha>

LOGOUT Messages:

Messages have the following format:

2 bytes
Opcode

Messages that appear only in a Client-to-Server communication. Informs the server on client disconnection. Client may terminate only after reciving ACK message in replay. If no user is logged in, sends an ERROR message.

Parameters:

• Opcode: 3.

Command initiation:

• This command is initiated by entering the following text in the client command line interface:
LOGOUT
• Once the ACK command is received in the client, it must terminate itself.

FOLLOW Messages:

Messages have the following format:

2 bytes	1 byte	String	1 byte
Opcode	Follow/Unfollow	UserName	0

Messages that appear only in a Client-to-Server communication. A FOLLOW message allows a user to add/remove other user to/from his follow list.

If the FOLLOW command failed an ERROR message will be sent back to the client.

The user must be logged in, otherwise an ERROR message will be sent.

For a follow command to succeed, a user on the list must not already be on the following list of the logged in user. (The opposite also applys for the unfollow command).
The ack for this command will contain the user name of the followed/unfollowed user.
The ACK message will have the following form:
ACK-Opcode FOLLOW-Opcode <username>

2 bytes	2 bytes	string	1 byte
ACK Opcode	FOLLOW Opcode	UserName	0

Parameters:

• Opcode: 4.

• Follow/Unfollow: This parameter has a value of 0 when a user wants to follow, otherwise it has a value of 1(Unfollow).

• UserName: The requested user name to follow/unfollow.

Command initiation:

• This command is initiated by entering the following texts in the client command line interface: FOLLOW <0/1 (Follow/Unfollow)> <UserName>

POST Messages:

Messages have the following format:

2 bytes	String	1 byte
Opcode	Content	0

Messages that appear only in a Client-to-Server communication. A post message allows a user to share messages with other users.

The Content parameter is a sequence of bytes in UTF-8.

All posts should be saved to a data structure in the server, along with PM messages. A post message will be sent to users who are listed with a "@username" inside the message (if username is registered in the system) and to users following the user who posted the message.

In order to send a POST message the user must be logged in, otherwise an ERROR message will be returned to the client.

Parameters:

• Opcode: 5.

• Content: The content of the message a user wants to post. The message may contain @<username> in order to send it to specific users other then those following the poster.

Command initiation:

• This command is initiated by entering the following texts in the client command line interface:
POST <PostMsg>

PM Messages:

Messages have the following format:

2 bytes	string	1 byte	string	1 byte	String	1 byte
Opcode	UserName	0	Content	0	Sending date and time	0

Messages that appear only in a Client-to-Server communication.

PM message is used to sent private messages to another user.

All string parameters are a sequence of bytes in UTF-8 terminated by a zero byte.

In order to send a PM message the sending user must be logged in, otherwise an ERROR message will be returned to the client.

If the reciepient user isn't registered an ERROR message will be returned to the client.
@<username> isn't applicable for private messages.

If the user is not following the reciepient user, ERROR message will be returned to the client.

All PM messages should be saved to a data structure in the application, along with post messages.
Before showing and saving the message, the server should filter the message from words provided in the server, and every filtered word should be replaced with '<filtered>'.
The data structure which includes the words need to be filtered, is hard coded in the server.
For example: if the server has a list of words ['war', 'Trump'], and the message
Which some client had send was: 'Trump is planning to declare a war on the republic of Lala-land'
Then the message that will be saved in the server is: '<filtered> is planning to declare a <filtered> on the republic of Lala-land'

Parameters:

- Opcode: 6.
- UserName: The user to send the message to.
- Content: The content of the message the logged in user wants to send to the other user.

- Sending date and time: the date and time when the message has sent (Format: DD-MM-YYYY HH:MM)

Command initiation:

• This command is initiated by entering the following texts in the client command line interface:
PM <Username> <Content>

LOGSTAT Message:

The message has the following format:

2 bytes
Opcode

Messages that appear only in a Client-to-Server communication.

A LOGSTAT message is used to recieve data on a logged in users (the age of every user , number of posts every user posted, number of every user's followers, number of users the user is following). In order to send a LOGSTAT message the user must be logged in, otherwise an ERROR message will be returned to the client.

If user is not registered, an error message will be returned.

The returned ACK message will contain (for every single username) user's age, number of posts a user posted (not including PM's), number of followers, number of users the user is following in the optional section of the ACK message.

Example:
ACK-Opcode LOGSTAT-Opcode <Age><NumPosts> <NumFollowers> <NumFollowing>
ACK-Opcode LOGSTAT-Opcode <Age><NumPosts> <NumFollowers> <NumFollowing>
:

2 bytes	2 bytes	2 bytes	2 bytes	2 bytes	2 bytes
ACK Opcode	LOGSTAT Opcode	Age	NumPosts	NumFollowers	NumFollowing

Parameters:

• Opcode: 7.

Command initiation:

• This command is initiated by entering the following texts in the client command line interface:
LOGSTAT

STAT Messages:

Messages have the following format:

2 bytes	String	1 byte
Opcode	List of usernames	0

A STAT message is used to recieve data on a certain users (the age of every user , number of posts every user posted, number of every user's followers, number of users the user is following).

The 'List of usernames' parameter are a sequence of bytes in UTF-8 terminated by a zero byte, **With the following format ' usernam1|username2|username3|…' for simplicity, you can assume that 'username' doesn't contain the symbol '|'.**

Messages that appear only in a Client-to-Server communication.

In order to send a STAT message the user must be logged in, otherwise an ERROR message will be returned to the client.

If user is not registered, an error message will be returned.

The returned ACK message will contain (for every single username) user's age, number of posts a user posted (not including PM's), number of followers, number of users the user is following in the optional section of the ACK message.

Example:
ACK-Opcode STAT-Opcode <Age><NumPosts> <NumFollowers> <NumFollowing>
ACK-Opcode STAT-Opcode <Age><NumPosts> <NumFollowers> <NumFollowing>
:

2 bytes	2 bytes	2 bytes	2 bytes	2 bytes	2 bytes
ACK Opcode	STAT Opcode	Age	NumPosts	NumFollowers	NumFollowing

Parameters:

- Opcode: 8.
- List of usernames: The list of users whose stats will be returned to the client.

Command initiation:

- This command is initiated by entering the following texts in the client command line interface:

STAT <UserNames_list>

NOTIFICATION Messages:

Messages have the following format:

2 bytes	1 byte	string	1 byte	string	1 byte
Opcode	NotificationType - PM/Public	PostingUser	0	Content	0

Messages that appear only in a Server-to-Client communication. This message will be sent from the server for any PM sent to the user, post sent by someone the user is following, or a post that contained @<MyUsername> in the content of a message.

Both string parameters are a sequence of bytes in UTF-8 terminated by a zero byte.
A user will recive any POST/PM notification sent after follow (for users the current user is following) that he didn't see. I.e. wasn't logged in when the other user posted/sent the message. (Clue: for each user, save timestamp of last message recieved from each of the other users / timestamp of the follow command)

Parameters:

- Opcode: 9.
- NotificationType : indicates whether the message is a PM message (0) or a public message (post) (1).
- PostingUser: The user who poster/sent the message.
- Content: The message that was posted/sent.

Client screen output:

- Any NOTIFICATION message received in client sould be written to the screen:

NOTIFICATION <"PM"/"Public"> <PostingUser> <Content>

ACK Messages:

Messages have the following format:

2 bytes	2 bytes	-
Opcode	Message Opcode	Optional

Messages that appear only in a Server-to-Client communication.

ACK Messages are used to acknowledge different Messages. Each ACK contains the message number for which the ack was sent. In the optional section there will be additional data for some of the Messages (if a message uses the optional section it will be specified under the message description).

All Messages that appear in a Client-to-Server communication require an ack/error message in response.

Parameters:

- Opcode: 10.
- Message Opcode: The message opcode the ACK was sent for.
- Optional: changes for each message.

Client screen output:

- Any ACK message received in client sould be written to the screen:

ACK <Message Opcode> <Optional>

- Printing of the optional part:
  - Multi-parameter optional sections should be split by space and printed by order of the ack response
  - Short should be printed as numbers
  - String lists should be seperated by a space

ERROR Messages:

Messages have the following format:

2 bytes	2 bytes
Opcode	Message Opcode

Messages that appear only in a Server-to-Client communication.

An ERROR message may be the acknowledgment of any other type of message. In case of error, an error message should be sent.

Parameters:

• Opcode: 11.

• Message Opcode: The message opcode the ERROR was sent for.

Error Notification:

• Any error message received in client should be written to screen:

ERROR <Message Opcode>

BLOCK Message:

The message has the following format:

2 bytes	String	1 byte
Opcode	username	0

Messages that appear only in a Client-to-Server communication.

Message send to block a specific user.

Blocked user can't follow, PM, or show any information about the user who blocked him.

Once user blocking acknowledged, both users (the blocking and the blocked) stop following each other.

If 'username' doesn't exist, and ERROR message will be sent back.

Parameters:

- Opcode: 12
- Username: the username to block

Command initiation:

- This command is initiated by entering the following texts in the client command line interface: **BLOCK <UserNames>**

2 Implementation Details:

2.1 General Guidelines

• The server should be written in Java. The client should be written in C++ with BOOST. Both should be tested on Linux installed at CS computer labs.

• You must use maven as your build tool for the server and MakeFile for the c++ client.

• The same coding standards expected in the course and previous assignments are expected here.

2.2 Server

You will have to implement a single protocol, supporting both the Thread-Per-Client and Reactor server patterns presented in class. Code seen in class for both servers is included in the assignment wiki page. You are also provided with 3 new or changed interfaces:

• Connections – This interface should map a unique ID for each active client connected to the server. The implementation of Connections is part of the server pattern and not part of the protocol. It has 3 functions that you must implement (You may add more if needed):

- boolean send(int connId, T msg) – sends a message T to client represented by the given connId
- void broadcast(T msg) – sends a message T to all active clients. This includes clients that has not yet completed log-in by the BGS protocol. Remember, Connections belongs to the server pattern implemenration, not the protocol!.
- void disconnect(int connId) – removes active client connId from map.

• ConnectionHandler - A function was added to the existing interface. o Void send(T msg) – sends msg T to the client. Should be used by send and broadcast in the Connections implementation.

• BidiMessagingProtocol – This interface replacesthe MessagingProtocol interface. It exists to support peer 2 peer messaging via the Connections interface. It contains 2 functions:

- void start(int connectionId, Connections connections) – initiate the protocol with the active connections structure of the server and saves the owner client's connection id.

- void process(T message) – As in MessagingProtocol, processes a given message. Unlike MessagingProtocol, responses are sent via the connections object send function. Left to you, are the following tasks:
    1. Implement Connections to hold a list of the new ConnectionHandler interface for each active client. Use it to implement the interface functions. Notice that given a connections implementation, any protocol should run. This means that you keep your implementation of Connections on T. public class ConnectionsImpl<T> implements Connections<T> {…}.
    2. Refactor the Thread-Per-Client server to support the new interfaces. The ConnectionHandler should implement the new interface. Add calls for the new Connections<T> interface. Notice that the ConnectionHandler<T> should now work with the BidiMessagingProtocol<T> interface instead of MessagingProtocol.
    3. Refactor the **Reactor** server to support the new interfaces. The ConnectionHandler should implement the new interface. Add calls for the new Connections<T> interface. Notice that the ConnectionHandler<T> should now work with the BidiMessagingProtocol<T> interface instead of MessagingProtocol<T>.
    4. **Tasks 1 to 3 MUST not be specific for the protocol implementation**. Implement the new BidiMessagingProtocol and MessageEncoderDecoder to support the BGS protocl as described in section 1.2. You will also need to define messages(<T> in the interfaces). You may add more classes as neccesery to implement the protocol (shared protocol data ect…).

Leading questions:

- Which classes and interfaces are part of the Server pattern and which are part of the Protocol implementation?
- When and how do I register a new connection handler to the **Connections** interface implementation?
- When do I call **start** to initiate the connections list? **Start** must end before any call to **Process** occurs. What are the implications on the reactor? (Note: start cannot be called by the main reactor thread and must run before the first)
- How do you collect a message? Are all message types collected the same way?

Tips:

- You can test tasks 1 – 3 by fixing one of the examples in the impl folder in the supplied spl-net.zip to work with the new interfaces (easiest is the echo example)
- You can complete tasks 1 and 2, proceed to 4 and return to the reactor code later. Thread per client implementation will be enough for testing purposes.
- The BGS protocol will require a shared object between client protocol implementation. You can transfer it in the constructor using the protocol factory. (as seen in NewsFeedServerMain.java in the examples). This means you will also need to consider synchronization from multiple clients working on the data structures at the same time.

Testing run commands:

- Reactor server:
  mvn exec:java -Dexec.mainClass="bgu.spl.net.impl.BGSServer.ReactorMain" -Dexec.args="<port> <Num of threads> "

- Thread per client server:
  mvn exec:java -Dexec.mainClass="bgu.spl.net.impl.BGSServer.TPCMain" -Dexec.args="<port>"

The **server** directory should contain a **pom.xml** file and the **src** directory. Compilation will be done from the server folder using:

mvn compile

2.3 Client

An echo client is provided, but its a single threaded client. While it is blocking on stdin (read from keyboard) it does not read messages from the socket. You should improve the client so that it will run 2 threads. One should read from keyboard while the other should read from socket. The client should receive the server's IP and PORT as arguments. You may assume a network disconnection does not happen (like disconnecting the network cable). You may also assume legel input via keyboard.

The client should recive commands using the standard input. Commands are defined in section 1.2 under command initiation sub sections. You will need to translate from keyboard command to network messages and the other way around to fit the specifications.

Notice that the client should close itself upon reception of an **ACK** message in response of an outgoing **LOGOUT** command.

The **Client** directory should contain a src, include and bin subdirectories and a Makefile as shown in class. The output executable for the client is named **BGSclient** and should reside in the bin folder after calling **make**.

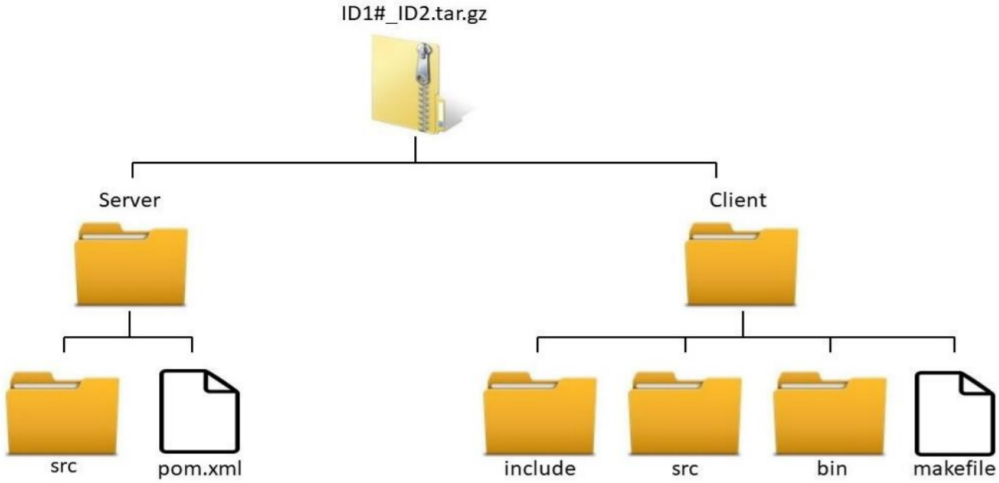
Testing run commands: **BGSclient <ip> <port>**

3 Submission instruction

- Submission is done only in pairs. If you do not have a pair, find one. You need explicit authorization from the course staff to submit without a pair. You cannot submit in a group larger than two.
- You must submit one .tar.gz file with all your code. The file should be named "ID#1_ID#2.tar.gz". Note: We require you to use a .tar.gz file. Files such as .rar, .zip, .bz, or anything else which is not a .tar.gz file will not be accepted and your grade will suffer.
- Extension requests are to be sent to majeed. Your request email must include the following information:
  - Your name and your partners name.
  - Your id and your partners id.
  - Explanation regarding the reason of the extension request.
  - Offcial certification for your illness or army drafting.

  Requests without a compelling reason will not be accepted

- The submitted file should contain a Client directory and a Server directory (Their content was explained in the implementation section).



4 Examples

The following section contains examples of commands running on client. It assumes that the software opened a socket properly and a connection has been initiated. We use "CLIENT#No<"and"CLIENT#NO>" to annotate client #No terminal input (keyboard) \ output (screen print). The order of commands matches order of reception in server. Server and client actions are explained in between. Note that the examples do not show the actual structure of the network messages, just the input \ output on the client terminal. The translation should be done according to specifications in section 1.2.

4.1 Registeration and login

Server assumptions for example:

- Server currently has 1 registered user named "Morty" with password "a123"

```
CLIENT#1< LOGIN Morty a321
CLIENT#1> ERROR 2
(Failed because of wrong password)
CLIENT#1< LOGIN Rick a123 1
CLIENT#1> ERROR 2
(Failed because username Rick isn't registered)
CLIENT#1< LOGIN Morty a123 0
CLIENT#1> ERROR 2
(Failed because of captcha 0)
```

CLIENT#1< LOGIN Morty a123 1
CLIENT#1> ACK 2

CLIENT#2< LOGIN Morty a123 1
CLIENT#2> ERROR 2
(Failed because Morty is already logged-in)
CLIENT#2< REGISTER Rick pain 12-10-1951
CLIENT#2> ACK 1

CLIENT#1< LOGOUT
CLIENT#1> ACK 3
(client 1 closes)
CLIENT#2< LOGOUT
CLIENT#2> ERROR 3
(client 2 did not login)

4.2 Following and posting / PM

Server assumptions for example:

- Server currently has 3 registered users:
  - "Morty" with password "a123"
  - "Rick" with password "pain"
  - "Bird-person" with password "Gubba"
- Followings:
  - Morty follows Rick and Bird-person
  - Rick follows Bird-person

CLIENT#1< LOGIN Morty a123 1
CLIENT#1> ACK 2
CLIENT#1< FOLLOW 0 Rick
CLIENT#1> ERROR 4
(Tried to follow users that he already follows, since both failed an error returned)

CLIENT#2< LOGIN Bird-person Gubba 1
CLIENT#2> ACK 2
CLIENT#2< POST Gubba nub nub doo rah kah
CLIENT#2> ACK 5
CLIENT#1> NOTIFICATION Public Bird-person Gubba nub nub doo rah kah
(Morty follows bird-person and is online so he gets the message pushed)

CLIENT#3< LOGIN Rick pain 1
CLIENT#3> ACK 2
CLIENT#3> NOTIFICATION Public Bird-person Gubba nub nub doo rah kah
(Rick follows Bird-person, now that he logged-in he receives messages he missed)

CLIENT#3< PM Bird-person why aren't you following me?
CLIENT#3> ACK 6
CLIENT#2> NOTIFICATION PM Rick why aren't you following me?
(Bird-person is online and was sent a PM, it is pushed right away to him)
CLIENT#3< POST wubba lubba dub dub @Bird-person is not following me
CLIENT#3> ACK 5
CLIENT#1> NOTIFICATION Public Rick wubba lubba dub dub @Bird-person is not following me
CLIENT#2> NOTIFICATION Public Rick wubba lubba dub dub @Bird-person is not following me
(Bird-person receives rick's latest post because his @username appears in it)

CLIENT#2< FOLLOW 0 Mortneey
CLIENT#2> ACK 4 1 Rick
(Bird-person failed to follow Morty because he misspelled his name)

4.3 STAT and unfollow

Server assumptions for example:

- Server currently has 3 registered users:
  - "Morty" with password "a123". Registered first
  - "Rick" with password "pain". Registered second
  - "Bird-person" with password "Gubba". Registered third
- Followings:
  - Morty follows Rick and Bird-person
  - Rick follows Bird-person
- Messages:
  - Morty sent 2 posts and 3 PMs
  - Rick sent 4 posts and 2 PMs
  - Bird-person sent 1 post and 1 PM

CLIENT#1< LOGIN Morty a123 1
CLIENT#1> ACK 2
CLIENT#1< STAT Bird-person
CLIENT#1> ACK 8 47 1 2 0

CLIENT#1< POST @bird-person I will not follow you any more, you are not social at all
CLIENT#1> ACK 5
(not one receives the post because bird person isn't logged-in and no one follows Morty, when bird person logs in he should get it)

CLIENT#1< FOLLOW 1 1 Bird-person
CLIENT#1> ACK 4 1 Bird-person
(Morty no longer follows bird-person and will not see his posts that do not contain @Morty in them from now on)

CLIENT#1< STAT Bird-personaaaa
CLIENT#1> ERROR 8
(No such user Bird-personaaaa)

CLIENT#1< LOGOUT
CLIENT#1> ACK 3
(client 1 closes)