CmpE 160 - Introduction to Object Oriented Programming Project 2 - Messaging Program

Deadline: 14.05.2020 - 23:55
For your questions on this project, please contact with your student teaching assistant Ece

Sarkın via ecesarkin@gmail.com **1. Introduction**

The main objective of this project is implementing a basic messaging program. This program consists of users, the server, messages, and mailboxes. Users can add each other as friends, they can send/receive messages via mailboxes, and the server acts as the coordinator and long-term storage of the messages.

You are already provided with the general structure of the project. Within this context, there are 3 packages and 7 classes. The details of the classes, functionalities to be implemented and overall operations of the project with the corresponding format are presented in the following sections.

2. Packages and classes

2.1. Package: executable

The `Main.java` class is placed in this package.
The Main should be implemented in a way that it should read the corresponding parameters and events from `input.txt` and log the results to `output.txt` .

2.2. Package: elements

3 main concepts of this project reside in this package: `User` , `Server` and `M essage` classes. The java files of these classes are currently empty, you need to implement these classes according to the information provided in this document.

2.2.1. User.java

Each user object created in the program has an ID, an inbox, an outbox, and a list of friends. The IDs of the users are determined by the Main method as consecutive numbers, starting from 0. A user can send another user a message. However, there is a prerequisite of receiving a message: the receiver and sender side should be in the friends list of each other. One can send a message even though the receiver is not in the friends list. In that case, the message is stored in the server but the receiver side cannot fetch it. In this program, a user can add or remove a friend. As soon as they become friends, the receiver becomes able to read that message.

The fields and signature of the methods that should be implemented for the `User` class are as follows:

int id ;
Inbox inbox ;
Outbox outbox ; ArrayList<User> friends ; User(**int** id):
void addFriend(User other)

also add the user to the other user's friends list. **void** removeFriend(User other)

a constructor that takes ID number as a parameter
a method for adding a friend to the friends list. It should

a method for removing a friend from the friends list. It should also remove the user from the other user's friends list.

boolean isFriendsWith(User other) It should return true if the user and the other user are friends, false otherwise.

void sendMessage(User receiver , String body , **int** time , Server server)
A new message should be created and added to the user's

sent list which is in his/her outbox.

2.2.2. Message.java :

All messages in this program should be instances of this class. Each message should have an ID, a body (text), a sender, a receiver and various time stamps. The IDs of the messages are determined by the order they are created as consecutive integer values. The first message sent in the program has the ID of 0, the second message sent has the ID of 1 and so on. IDs should be unique, that is, if two messages have the same ID number, it means they are the same message. Also, messages should be comparable with each other regarding their body length.

The necessary fields and signature of the methods to be implemented for the `Message` class are as follows:

static int *numOfMessages* = 0; **int** id ;
String body ;
User sender ;

User receiver ;
int timeStampSent ;

int timeStampRead ;
int timeStampReceived ;

number of total messages in the program.

Message(User sender , User receiver , String body , Server server , **int** time)
3 setter methods for timeStampSent, timeStampReceived and timeStampRead

int getId()
String getBody()
int compareTo(Message o)

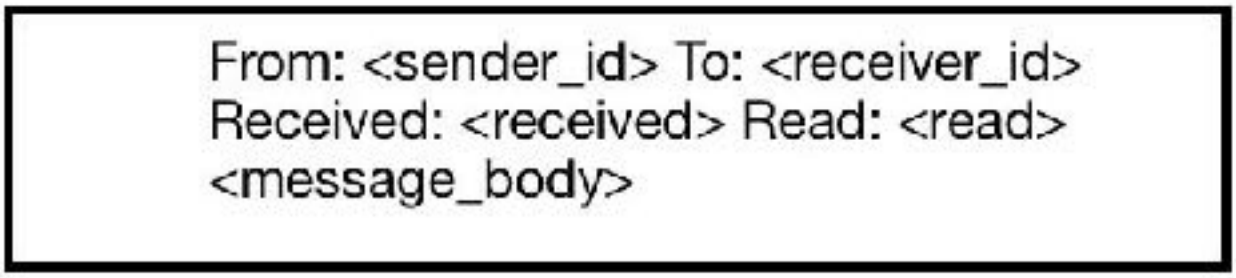
a getter method for ID field
a getter method for the body field
if this message is longer than the other message, return a

positive number. Else if the other message is longer, return a negative number. Return 0 if both messages have the same number of characters.

boolean equals(Object o) String toString()

This method should return a String in this form:

Each line should start with a tab. <received> or <read> should be empty strings if the message has not been received/read yet.



```
From: <sender_id> To: <receiver_id>
Received: <received> Read: <read>
<message_body>
```

2.2.3. Server.java :

There is only one server throughout the program. This server functions as the mechanism where all non-received messages are stored in a first come, first served (FCFS) manner. It has a finite capacity set at the beginning of the execution. This capacity is the third argument specified in the input file and its unit is the number of characters. If the sum of lengths of the messages stored in the server exceeds the capacity value, all messages in the server are deleted. Detailed information about this issue is available in Section 4.

The fields and methods to be implemented in `Server.java` :
long capacity capacity of the server
long currentSize This number should be kept updated when

messages in the server are deleted or added. Total number of the characters of messages' bodies in the msgs queue.

Queue msgs
Server(**long** capacity)
void checkServerLoad(PrintStream printer) **long** getCurrentSize()
void flush()

2.3. Package: boxes 2.3.1. Box.java :

A queue where non received messages are stored. The constructor with a parameter

prints the warnings about the capacity
A getter method for the current size field empties the queue

`Box.java` is the parent class of `Inbox.java` and `Outbox.java`. Every Box should have an owner of type User.

User owner

2.3.2. Inbox.java :

Both read and unread messages sent to a user are stored in the user's Inbox. An inbox has two main functionalities: receiving messages from the server, and reading messages. However there are various signatures of `readMessages` method due to different event types parsed from the input file.

● `Inbox.java` should be a child of `Box.java`.

- ● There should be two fields for read and unread messages of the inbox, `unread`
  should be a stack and `read` should be a FCFS queue.

- ● When the `receiveMessages` method is called, it should fetch from the server those
  messages that are sent from the owner's friends to the owner. These messages' `timeStampReceived` should get updated and they should be pushed to the `unread` stack. The method shouldn't return anything.
  ○ The messages received at the same time have the same `timeStampReceived`. But they are pushed to the stack one by one. So, the last sent message should be on top of the stack (first message to be read).

- ● There are 3 different methods for reading messages. Two of them return an integer and the last one is a void method. If the number of messages read successfully is 0

or 1, the non-void methods should return 1. Else, those methods should return the number of messages read successfully.

The fields and methods to be implemented in `Inbox.java` : Stack unread Queue read

Inbox(User user)

void receiveMessages(Server server , **int** time) receives messages from the adds to the `unread` stack. This method also changes `timeStampReceived` with the parameter `time` .

server,

int readMessages(**int** num , **int** time) This method is for reading a certain amount of messages from the `unread` stack. If the `num` parameter is 0, then all messages in `unread` should be read. If the number of messages in `unread` is less than read, still all messages should be read.

int readMessages(User sender , **int** time) This method is for reading a specified sender's messages.

void readMessage(**int** msgId , **int** time) When this method is called, the message with the ID number `msgId` should be read, if it exists.

● `readMessages` and `readMessage` methods pop messages in the `unread` , read them (stamps their `timeStampRead`) and then transfer them to the `read` queue.
● Unlike the `receiveMessages` method, reading each message takes 1 time. For

example, suppose there are 3 messages in the stack and we should read all at time tick t. The message at the top of the stack should be read first and it's `timeStampRead` should be t. The second message to be read should have a `timeStampRead` value of t+1. And the third message should be read at `timeStampRead` = t+2. Then, the method should return 3. The next event occuring after this reading should be at time tick t+3, not t+1.

2.3.3. Outbox.java :

The `Outbox.java` represents an outbox of the owner. It stores all messages a user has sent, in other words, all messages constructed with the `sender` field as the `owner` of this outbox.

- ● This class is a child class of `Box.java` .

- ● Even if a message is never added to the server's queue due to capacity overload,
  this message should be added to the `sent` queue in the user's outbox. The necessary fields and methods for `Outbox.java`
  Queue sent ;
  Outbox(User owner) A constructor with the parameter User owner

3. Input & Output

The first row of the input file specifies the number of users, number of queries, and capacity of the server : A B C
The next B lines are the queries. There are 10 types of events exactly.

- ➔ **0: sending a message:**
  `0 <sender_id> <receiver_id> <message_body>` Example:
  `0 1 4 Hi there!`
  User#1 sends user#4 the message "Hi there!".

- ➔ **1: recieve messages:**
  `1 <reciever_id>`
  Example: `1 25`
  User#25 receives all the messages that are sent to him/her from the server.

- ➔ **2: read a certain amount of messages:**
  `2 <receiver_id> <numberOfMessages>`

● Note: If the number of messages is 0, then read all.

- ➔ **21: read all the messages from a sender** `21 <receiver_id> <sender_id>`

- ➔ **22: read a specific message**
  `22 <receiver_id> <message_id>`

- ➔ **3: add a friend**
  `3 <id1> <id2>`
  Make User#<id1> and User#<id2> friends. In case they are already

friends, do nothing.

- ➔ **4: remove a friend**
  `4 <id1> <id2>`
  Remove each other from their friends lists. In case they are already not friends, do nothing.

- ➔ **5: flush server** `5`
  Deletes all messages from the queue of the server

- ➔ **6: print the current size of the server** `6`
  Note: The output should be:
  ```
  Current load of the server is <currentSize>
  characters.
  ```

- ➔ **61: print the last message a user has read**

4.

●

● ●

●

● ●

● ●

●

```
61 <user_id>
```

The corresponding output should be:

```
From: <sender_id> to: <user_id>
Received: <time_stamp1> Read: <time_stamp2>
<message_body>
```

Important Notes

The method signatures and field names should be exactly the same with the ones specified in this document.
You can implement and utilize additional helper methods of your choice.

Please keep in mind that providing the necessary accessibility and visibility is important: you should not implement everything as public, even though the necessary functionality is implemented. The usage of appropriate access modifiers and other Java keywords (super, final, static etc.) play an important role in this project since there will be partial credit specifically for the software design.

There will be partial credit for the code documentation. You need to document your code in Javadoc style including the class implementations, method definitions (including the parameters and return if available etc.), and field declarations. You are not going to submit a documentation file generated by Javadoc.

You may use Java's `Queue` and `Stack` classes as well as you may implement
yours.
Each line in the input file represents an activity to occur during the execution. The processing time of an event starts as soon as the previous event is completed. For example, if an event starts at time t and has ended by the time t+5, the following event in the input file starts at t+5, not t+1. However, only reading messages activity can consume more than 1 time.
Do not make assumptions about the numbers and the size of the input.
Only friends can receive messages from each other. However, a message can be sent to the server even if the users are not friends at that time.
Warnings about the server occupancy rate: Warnings are made only once when 50% or 80% is achieved. As stated, the overall capacity and the allocated capacity of the server is represented through the size of the messages as the number of characters in total. Whenever the utilization of the server exceeds 50% (i.e at least 50% of the overall server capacity gets full), your program should print out a warning message. However, this is just a warning message, your program should not execute an additional operation by itself. Second warning message is printed out when the server becomes 80% full. When no space is available in the server and all of its capacity is allocated for the messages, another informative message is printed out and all of the messages in the server are deleted. A corresponding use case is depicted in the table below. At time t, it is calculated that only 49% of the server capacity is utilized for storing the messages. After an event is executed at time t+1, it is seen that the server occupancy rate reaches 50%. Therefore, a warning message is printed out. Then, the following events continue increasing the allocated capacity of the server. As a result of the activity occurred at time t+4, the server occupancy rate exceeds 80%. In this direction, the corresponding warning message is printed

out once more. Please keep in mind that the server occupancy rate does not have to be exactly 50% or 80% to make your program print out the warning messages. If there is an activity that results in exceeding these values (at time t+4, rate increases to 83% from 76%), the same output should be provided. At time t+5, the current size of the utilized server capacity is decreased to 65% after a message or set of messages is received, but it is still higher than 50%. Thus, another warning message is printed out to inform that the current size is still resulting in a warning even though it is less than the previous time slot. At time t+7, the server gets full, which means that there is no room for an incoming message. The messages causing the server to get full should also be deleted but they are stored in the outbox of the sender. All messages are automatically deleted as soon as the server has no remaining capacity. When that is the case, another message is provided as the output as presented in the table.