Programming Assignment 2: A Mail User Agent in Java

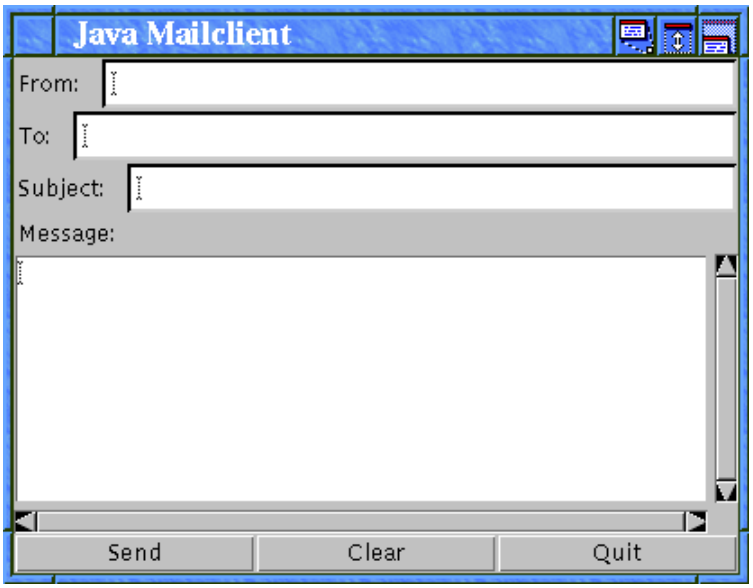
Requirements:

- A single *.java file is to be submitted with the submit command. The java file should include all the classes needed to run the mail client.
- The file should have the appropriate header (see `/home/jonesro/cs460/submitHeaders/eMailClientHeader.txt`) and use a good programming style. [The following has been provided for your use: `/home/jonesro/cs460/labHandouts/MailClient.java`]
- After compiling your java file, it should execute by typing: java MailClient.
- Is able to send to someUserID@jordan.byui.edu and someUserID@aus213l4.byui.edu when executed on a lab machine. The "someUserID" will be a netID associated with someone that has a valid login in the Linux Lab. [Note: The address is "aus213" followed by a small "L" <ell> then the number 4.] (More information about sending to these two Linux Lab systems is given later.)
- Provide an error message to the console and continue (don't die) when handling a bad host address (not connecting to an SMTP server) and an unknown user (SMTP error code 550).
- The client should print out statements indicating status, something like:

```
GUI preparing to send mail ...
Client received reply: 220 jordan.byui.edu ESMTP Sendmail 8.13.1/8.13.1; Fri, 27 Apr 2007 00:14:53 -0600
Client sending handshake command: HELO 157.201.194.226
Client received reply: 250 jordan.byui.edu Hello aus206l1 [157.201.194.237], pleased to meet you
Client sending mail from: MAIL FROM: <jonesro@aus213l222.byui.edu>
Client received reply: 250 2.1.0 <jonesro@aus213l222.byui.edu>... Sender ok
Client sending rcpt to: RCPT TO: <jonesro@jordan.byui.edu>
Client received reply: 250 2.1.5 <jonesro@jordan.byui.edu>... Recipient ok
Client sending DATA flag: DATA
Client received reply: 354 Enter mail, end with "." on a line by itself
Client sending message body: From: jonesro@aus213l222.byui.edu
Client received reply: 250 2.0.0 l3R6ErIb011728 Message accepted for delivery
Client received reply: 221 2.0.0 jordan.byui.edu closing connection
Mail sent successfully!
```

Instructions:

In this lab you will implement a mail user agent (MUA) that sends mail to remote hosts. Your task is to program the SMTP interaction between MUA and the remote SMTP server. The client provides a graphical user interface containing fields for entering the sender and recipient addresses, the subject of the message and the message itself. Here's what the user interface looks like:



With this interface, when you want to send a mail, you must fill in an address for both the sender and the recipient, i.e., `user@someschool.edu`, not just simply `user`. You can send mail to only one recipient. When you have finished composing your mail, press *Send* to send it.

For this lab, your program may require that the domain part of the recipient's address *has to be* the canonical name of the SMTP server handling incoming mail. For example, if the user was sending mail to address `user@someschool.edu` and the SMTP server of `someschool.edu` is `smtp.somechool.edu`, you could require the user to use the address `user@smtp.someschool.edu` in the To-field. Requiring this will allow your program to handle sending to one of two SMTP servers in the Linux Lab. In the Linux Lab, a DNS query for MX records does not provide the name of the Linux lab server (*jordan)* that you should be sending your mail to for this lab. In other words, trying to have your program handle a user sending mail to yourID@byui.edu is not going to work in the Linux Lab (unless when you see byui.edu you hardcode your program to send to jordan). The 'To:' address will need to be yourID@jordan.byui.edu or yourID@aus213l4.byui.edu (the canonical names for the SMTP servers in the Linux Lab).

> If you want to take on the added challenge of revising your mail client to do DNS queries for MX records, so that it is more useful (can use domain names and not just canonical names) outside of the Linux Lab, you might be interested in DNSmessage.class and MXLookup.class that you will find in the /home/jonesro/cs460/labHandouts directory. In the same directory you will find other files related to these two class files. DNSmessage.class was written by Ketsle Alexander; see http://www.geocities.com/SunsetStrip/Underground/6834/nslookup.html#api. MXLookup.class was written by Nathan Kingsley and Michael Owens as students at BYU-Idaho.

When you send to someUserID@jordan.byui.edu the Linux Lab server jordan forwards the email to the BYU-Idaho email system using the hidden .forward file in the user's home directory. The .forward file is set up so that mail received by jordan goes to a student's BYU-Idaho email account. You may change your .forward file to send your mail, such as test messages, to a different email address.

When you send to someUserID@aus213l4.byui.edu, system number 4 in the Linux Lab will receive the mail and will try to forward it using the .forward file but will be unable to do so. The BYU-Idaho email system will only accept mail from jordan. An SMTP server has been setup on AUS213L4 so that you could use it for testing, but to do so you need to remove (or move out the way) your .forward file. If you do this, you may send mail to AUS213L4 and then you will need to log in to AUS213L4 (157.201.194.204) to see your mail there. One program you can use to see your mail that has been sent to this system, even with just a terminal window up, is `mutt`. To see if the mail arrived on AUS213L4, you could also just open up /var/spool/mail/yourUserID with an editor.

The Code

The program consists of four classes:

MailClient	The user interface
Message	Mail message
Envelope	SMTP envelope around the Message
SMTPConnection	Connection to the SMTP server

You will need to complete the code in the `SMTPConnection` class so that in the end you will have a program that is capable of sending mail to any recipient. The code for the `SMTPConnection` class, and the other three classes, is provided below.

Most of the places where you need to complete the code have been marked with the comments `/* Fill in */`. Each of the places requires one or more lines of code.

The `MailClient` class provides the user interface and calls the other classes as needed. When you press *Send*, the `MailClient` class constructs a `Message` class object to hold the mail message. The `Message` object holds the actual message headers and body. Then the `MailClient` object builds the SMTP envelope using the `Envelope` class. This class holds the SMTP sender and recipient information, the SMTP server of the recipient's domain, and the `Message` object. Then the `MailClient` object creates the `SMTPConnection` object which opens a connection to the SMTP server and the `MailClient` object sends the message over the connection. The sending of the mail happens in three phases:

1. The `MailClient` object creates the `SMTPConnection` object and opens the connection to the SMTP server.
2. The `MailClient` object sends the message using the function `SMTPConnection.send()`.
3. The `MailClient` object closes the SMTP connection.

The `Message` class contains the function `isValid()` which is used to check the addresses of the sender and recipient to make sure that there is only one address and that the address contains the @-sign. The provided code does not do any other error checking.

Reply Codes

For the basic interaction of sending one message, you will only need to implement a part of SMTP. Section 2.4 of the text provides a more complete description of SMTP, but in this lab you need only to implement the commands in the following table.

Command	**Reply Code**
DATA	354
HELO	250
MAIL FROM	250
QUIT	221
RCPT TO	250

The above table also lists the accepted reply codes for each of the SMTP commands you need to implement. For simplicity, you can assume that any other reply from the server indicates a fatal error and abort the sending of the message. In reality, SMTP distinguishes between transient (reply codes 4xx) and permanent (reply codes 5xx) errors, and the sender is allowed to repeat commands that yielded in a transient error. See Appendix E of RFC 821 for more details.

In addition, when you open a connection to the server, it will reply with the code 220.

Note: RFC 821 allows the code 251 as a response to a RCPT TO-command to indicate that the recipient is not a local user. You may want to verify manually with the `telnet` command what your local SMTP server replies.

Hints

Most of the code you will need to fill in is similar to the code you wrote in the WebServer lab. You may want to use the code you have written there to help you.

To make it easier to debug your program, do not, at first, include the code that opens the socket, but use the following definitions for `fromServer` and `toServer`. This way, your program sends the commands to the terminal. Acting as the SMTP server, you will need to give the correct reply codes by typing them in on the keyboard. When your program works, add the code to open the socket to the server.

```
fromServer = new BufferedReader(new InputStreamReader(System.in));
toServer = new DataOutputStream(System.out);
```

The lines for opening and closing the socket, i.e., the lines `connection = ...` in the constructor and the line `connection.close()` in function `close()`, have been commented out by default.

Start by completing the function `parseReply()`. You will need this function in many places. In the function `parseReply()`, you should use the `StringTokenizer`-class for parsing the reply strings. You can convert a string to an integer as follows:

```
int i = Integer.parseInt(argv[0]);
```

In the function `sendCommand()`, you should use the function `writeBytes()` to write the commands to the server. The advantage of using `writeBytes()` instead of `write()` is that the former automatically converts the strings to bytes which is what the server expects. Do not forget to terminate each command with the string CRLF.

You can throw exceptions like this:

```
throw new Exception();
```

You do not need to worry about details, since the exceptions in this lab are only used to signal an error, not to give detailed information about what went wrong.

Optional Exercises

You may want to try the following optional exercises to make your program more sophisticated.. For these exercises, you will need to modify also the other classes (MailClient, Message, and Envelope).

- **Verify sender address**. Java's System-class contains information about the username and the InetAddress-class contains methods for finding the name of the local host. Use these to construct the sender address for the Envelope instead of using the user-supplied value in the From-header.
- **Additional headers**. The generated mails have only four header fields, From, To, Subject, and Date. Add other header fields from RFC 822, e.g., Message-ID, Keywords. Check RFC 822 for the definitions of the different fields.
- **Multiple recipients**. Currently the program only allows sending mail to a single recipient. Modify the user interface to include a Cc-field and modify the program to send mail to both recipients. For a more challenging exercise, modify the program to send mail to an arbitrary number of recipients.
- **More error checking**. The provided code assumes that all errors that occur during the SMTP connection are fatal. Add code to distinguish between fatal and non-fatal errors and add a mechanism for signaling them to the user. Check the RFC to see what the different reply codes mean. This exercise may require large modifications to the `send()`, `sendCommand()`, and `parseReply()` functions.

SMTPConnection.java

This is the code for the SMTPConncetion class that you will need to complete. The complete code for the other three classes is given below.

```
import java.net.*;
import java.io.*;
import java.util.*;

/**
 * Open an SMTP connection to a remote machine and send one mail.
 *
 */
public class SMTPConnection {
    /* The socket to the server */
    private Socket connection;

    /* Streams for reading and writing the socket */
    private BufferedReader fromServer;
    private DataOutputStream toServer;

    private static final int SMTP_PORT = 25;
    private static final String CRLF = "\r\n";

    /* Are we connected? Used in close() to determine what to do. */
    private boolean isConnected = false;

    /* Create an SMTPConnection object. Create the socket and the
       associated streams. Initialize SMTP connection. */
    public SMTPConnection(Envelope envelope) throws IOException {
```

```
        // connection = /* Fill in */;
        fromServer = /* Fill in */;
        toServer =   /* Fill in */;

        /* Fill in */
        /* Read a line from server and check that the reply code is 220.
           If not, throw an IOException. */
        /* Fill in */

        /* SMTP handshake. We need the name of the local machine.
           Send the appropriate SMTP handshake command. */
        String localhost = /* Fill in */;
        sendCommand( /* Fill in */ );

        isConnected = true;
    }

    /* Send the message. Write the correct SMTP-commands in the
       correct order. No checking for errors, just throw them to the
       caller. */
    public void send(Envelope envelope) throws IOException {
        /* Fill in */
        /* Send all the necessary commands to send a message. Call
           sendCommand() to do the dirty work. Do _not_ catch the
           exception thrown from sendCommand(). */
        /* Fill in */
    }

    /* Close the connection. First, terminate on SMTP level, then
       close the socket. */
    public void close() {
        isConnected = false;
        try {
            sendCommand( /* Fill in */ );
            // connection.close();
        } catch (IOException e) {
            System.out.println("Unable to close connection: " + e);
            isConnected = true;
        }
    }

    /* Send an SMTP command to the server. Check that the reply code
       is what is is supposed to be according to RFC 821. */
    private void sendCommand(String command, int rc) throws IOException

{
        /* Fill in */
        /* Write command to server and read reply from server. */
        /* Fill in */

        /* Fill in */
        /* Check that the server's reply code is the same as the
           parameter rc. If not, throw an IOException. */
        /* Fill in */
    }

    /* Parse the reply line from the server. Returns the reply code. */
    private int parseReply(String reply) {
        /* Fill in */
    }

    /* Destructor. Closes the connection if something bad happens. */
    protected void finalize() throws Throwable {
        if(isConnected) {
            close();
```

```
        }
        super.finalize();
    }
}
```

MailClient.java

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;

/**
 * A simple mail client with a GUI for sending mail.
 */
public class MailClient extends Frame {
    /* The stuff for the GUI. */
    private Button btSend = new Button("Send");
    private Button btClear = new Button("Clear");
    private Button btQuit = new Button("Quit");
    private Label fromLabel = new Label("From:");
    private TextField fromField = new TextField("", 40);
    private Label toLabel = new Label("To:");
    private TextField toField = new TextField("", 40);
    private Label subjectLabel = new Label("Subject:");
    private TextField subjectField = new TextField("", 40);
    private Label messageLabel = new Label("Message:");
    private TextArea messageText = new TextArea(10, 40);

    /**
     * Create a new MailClient window with fields for entering all
     * the relevant information (From, To, Subject, and message).
     */
    public MailClient() {
        super("Java Mailclient");

        /* Create panels for holding the fields. To make it look nice,
           create an extra panel for holding all the child panels. */
        Panel fromPanel = new Panel(new BorderLayout());
        Panel toPanel = new Panel(new BorderLayout());
        Panel subjectPanel = new Panel(new BorderLayout());
        Panel messagePanel = new Panel(new BorderLayout());
        fromPanel.add(fromLabel, BorderLayout.WEST);
        fromPanel.add(fromField, BorderLayout.CENTER);
        toPanel.add(toLabel, BorderLayout.WEST);
        toPanel.add(toField, BorderLayout.CENTER);
        subjectPanel.add(subjectLabel, BorderLayout.WEST);
        subjectPanel.add(subjectField, BorderLayout.CENTER);
        messagePanel.add(messageLabel, BorderLayout.NORTH);
        messagePanel.add(messageText, BorderLayout.CENTER);
        Panel fieldPanel = new Panel(new GridLayout(0, 1));
        fieldPanel.add(fromPanel);
        fieldPanel.add(toPanel);
        fieldPanel.add(subjectPanel);

        /* Create a panel for the buttons and add listeners to the
           buttons. */
        Panel buttonPanel = new Panel(new GridLayout(1, 0));
        btSend.addActionListener(new SendListener());
        btClear.addActionListener(new ClearListener());
        btQuit.addActionListener(new QuitListener());
        buttonPanel.add(btSend);
        buttonPanel.add(btClear);
```

```
    buttonPanel.add(btQuit);

    /* Add, pack, and show. */
    add(fieldPanel, BorderLayout.NORTH);
    add(messagePanel, BorderLayout.CENTER);
    add(buttonPanel, BorderLayout.SOUTH);
    pack();
    show();
}

static public void main(String argv[]) {
    new MailClient();
}

/* Handler for the Send-button. */
class SendListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        System.out.println("Sending mail");

        /* First, check that we have the sender and recipient. */
        if((fromField.getText()).equals("")) {
            System.out.println("Need sender!");
            return;
        }
        if((toField.getText()).equals("")) {
            System.out.println("Need recipient!");
            return;
        }

        /* Create the message */
        Message mailMessage = new Message(fromField.getText(),
                                          toField.getText(),
                                          subjectField.getText(),
                                          messageText.getText());

        /* Check that the message is valid, i.e., sender and
           recipient addresses look ok. */
        if(!mailMessage.isValid()) {
            return;
        }

        /* Create the envelope, open the connection and try to
           Send the message. */
        Envelope envelope = new Envelope(mailMessage);
        try {
            SMTPConnection connection = new SMTPConnection(envelope);
            connection.send(envelope);
            connection.close();
        } catch (IOException error) {
            System.out.println("Sending failed: " + error);
            return;
        }
        System.out.println("Mail sent successfully!");
    }
}

/* Clear the fields on the GUI. */
class ClearListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Clearing fields");
        fromField.setText("");
        toField.setText("");
        subjectField.setText("");
        messageText.setText("");
    }
```

```
    }

    /* Quit. */
    class QuitListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            System.exit(0);
        }
    }
}
```

Message.java

```
import java.util.*;
import java.text.*;

/**
 * Mail message.
 */
public class Message {
    /* The headers and the body of the message. */
    public String Headers;
    public String Body;

    /* Sender and recipient. With these, we don't need to extract
       them from the headers. */
    private String From;
    private String To;

    /* To make it look nicer */
    private static final String CRLF = "\r\n";

    /* Create the message object by inserting the required headers
       from RFC 822 (From, To, Date). */
    public Message(String from, String to, String subject, String text)

{
        /* Remove whitespace */
        From = from.trim();
        To = to.trim();
        Headers = "From: " + From + CRLF;
        Headers += "To: " + To + CRLF;
        Headers += "Subject: " + subject.trim() + CRLF;

        /* A close approximation of the required format. Unfortunately
           only GMT. */
        SimpleDateFormat format =
            new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss 'GMT'");
        String dateString = format.format(new Date());
        Headers += "Date: " + dateString + CRLF;
        Body = text;
    }

    /* Two functions to access the sender and recipient. */
    public String getFrom() {
        return From;
    }

    public String getTo() {
        return To;
    }

    /* Check whether the message is valid. In other words, check that
```

```
       both sender and recipient contain only one @-sign. */
    public boolean isValid() {
        int fromat = From.indexOf('@');
        int toat = To.indexOf('@');

        if(fromat < 1 || (From.length() - fromat) <= 1) {
            System.out.println("Sender address is invalid");
            return false;
        }
        if(toat < 1 || (To.length() - toat) <= 1) {
            System.out.println("Recipient address is invalid");
            return false;
        }
        if(fromat != From.lastIndexOf('@')) {
            System.out.println("Sender address is invalid");
            return false;
        }
        if(toat != To.lastIndexOf('@')) {
            System.out.println("Recipient address is invalid");
            return false;
        }
        return true;
    }

    /* For printing the message. */
    public String toString() {
        String res;

        res = Headers + CRLF;
        res += Body;
        return res;
    }
}
```

Envelope.java

```
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * SMTP envelope for one mail message.
 */
public class Envelope {
    /* SMTP-sender of the message (in this case, contents of

From-header. */
    public String Sender;

    /* SMTP-recipient, or contents of To-header. */
    public String Recipient;

    /* Target MX-host */
    public String DestHost;
    public InetAddress DestAddr;

    /* The actual message */
    public Message Message;

    /* Create the envelope. */
    public Envelope(Message message) {
        /* Get sender and recipient. */
        Sender = message.getFrom();
```

```
        Recipient = message.getTo();

        /* Get message. We must escape the message to make sure that
           there are no single periods on a line. This would mess up
           sending the mail. */
        Message = escapeMessage(message);

        /* Get the hostname part of the recipient. It should be the
           name of the MX-host for the recipient's domain. */
        int atsign = Recipient.lastIndexOf('@');
        DestHost = Recipient.substring(atsign + 1);

        /* Map the name into an IP-address */
        try {
            DestAddr = InetAddress.getByName(DestHost);
        } catch (UnknownHostException e) {
            System.out.println("Unknown host: " + DestHost);
            System.out.println(e);
            return;
        }
        return;
    }

    /* Escape the message by doubling all periods at the beginning of
       a line. */
    private Message escapeMessage(Message message) {
        String escapedBody = "";
        String token;
        StringTokenizer parser = new StringTokenizer(message.Body,

"\n", true);

        while(parser.hasMoreTokens()) {
            token = parser.nextToken();
            if(token.startsWith(".")) {
                token = "." + token;
            }
            escapedBody += token;
        }
        message.Body = escapedBody;
        return message;
    }

    /* For printing the envelope. Only for debug. */
    public String toString() {
        String res = "Sender: " + Sender + '\n';
        res += "Recipient: " + Recipient + '\n';
        res += "MX-host: " + DestHost + ", address: " + DestAddr + '\n';
        res += "Message:" + '\n';
        res += Message.toString();

        return res;
    }
}
```