Program 1b – a multi-threaded Encryption Server (25 points)

Due Date: Oct. 9 (Wednesday) 11:59pm

Program Submission: submit a runnable .jar file and the source code (*.java) to Canvas.

Program Description

Write a **multi-threaded Server** program (no GUI) to provide encryption service to multiple clients. The objective is to learn how to use **ServerSocket** and Java **Thread**. The server will be "listening" on port **5520** for the connection requests from the clients. You can run the server and the clients on the same machine, such as your laptop. In this case, the server will be a "**localhost**" with the IP Address **127.0.0.1**. You can use the client program from Program 1a to test your server. Once a connection request is accepted, the server creates a new thread to handle the connection and keeps listening on port 5520 for additional connection requests. That is, your server must be able to handle multiple connections simultaneously. Please note that, **if you are using iLab machines**, it is possible that someone is using the port number 5520 already. In this case, you need to use a different port number to run your server. For example, the next available port number 5521, 5522, …, etc.

Program Requirement

1. The server program must implement the Polyalphabetic encryption algorithm described below to encrypt the messages sent (cleartext) by the clients and send the encrypted messages (ciphertext) back to the clients.

Caesar cipher (textbook page #598 ~ #600)

All cryptographic algorithms involve substituting one thing for another. Message in original form is called **plaintext**, and encrypted message is known as **ciphertext**. Your Server implements an old symmetric key algorithm known as the **Caesar cipher** (a cipher is a method for encrypting data.) and provide the service of encryption. Caesar cipher would work by taking each letter in the plaintext message and substituting the letter that is ***n*** letters later (allowing wraparound; that is, having the letter ***z*** followed by the letter ***a***) in the alphabet. For example, if ***n* = 3**, then the letter ***a*** in plaintext becomes ***d*** in ciphertext (and the letter ***A*** becomes ***D***.)

plaintext	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
ciphertext	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c

Polyalphabetic encryption is to use a repeating pattern of multiple ciphers for encryption. Your Server must use **2 Caesar ciphers**, C_1 and C_2, with the repeating pattern $C_1\ C_2\ C_2\ C_1\ C_2$ for the encryption. For example, if $C_1 = 5$, $C_2 = 19$, the plaintext message "**Nice to meet you!**" results in the ciphertext "**Sbvj mt fxjm dhn!**". NOTE: the Server encrypts alphabet letters only.

plaintext	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
$C_1 = 5$	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e
$C_2 = 19$	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s

2. The server must continuously encrypt the messages received until a "**quit**" message is received. In this case, the server sends a "**Good Bye!**" message back to the client and close the connection (socket.)
3. Your server must not crash under any situations and must work with your client program from Program 1a.
4. You must try-catch all exceptions and output appropriate error messages to the console.

5. Your server must include the Server class and the ServerThread class described below.

- **Server class** should have exactly two public methods: main() and run() (you may have other private methods). The main() method of the Server should simply create a new Server object and call the run() method on it. Exceptions generated from this class must be caught and the error messages must be displayed on the Server console. (Please refer to page #3 of the Mini Java Socket Tutorial on Canvas)
  In the run() method, do the following:
  a. Create a new ServerSocket listening on port 5520
  b. Create a new PrintWriter object with a new FileOutputStream that writes to a log file called **prog1b.log**, which should be stored in your project folder. Make sure the second argument to the PrintWriter constructor is **true**. Every ServerThread will write connection information to the same log file. Each time your Server comes up, you can either overwrite or append to **prog1b.log**.
  c. Have an infinite while loop that
     i. Listens to connection requests using .accept() method, and once a connection is accepted, create a new instance of Socket to handle the connection.
     ii. Creates a new instance of ServerThread to handle the connection. When you're creating a new ServerThread object, use the socket object and the log file as arguments. So each ServerThread can communicate with the Client socket and write to the same log file.
     iii. Calls the start() method of the ServerThread object to start the thread.

- **ServerThread** class should extend the Thread class and should NOT be designated as public.
  a. Declare one Socket object, two PrintWriters (one for the socket I/O, one for the log file) object, and one BufferedReader object, outside of all methods. So they are visible in ServerThread class.
  b. Define a constructor with the interface:
     ```
     public ServerThread (Socket  clientSock,  PrintWriter  logfile)
     ```
  c. Since ServerThread extends Thread, it MUST have a run() method, which is the method that eventually gets called after you call start() on a Thread. The run() method should do the following:
     i. Write a line to the log file indicating that a connection was received. You should write the *date/time* the connection was received, the remote *IP address* and the *remote port* of the connection. You may find the following methods useful for your log file: Date().toString(), getInetAddress() and getPort() of the Socket class.
     ii. Have a while loop deals with receiving/sending the messages and the encryption. You can either write a method or a class for encryption and include it in the loop. Your Server should loop until a "quit" message is received.
     iii. When the Client disconnects (quit), you should print an appropriate message to the log file or console saying the connection on some port number is closed. **DO NOT close the PrintWriter for the log file.** Just close the Client socket and return from the run() method. Doing so effectively kills the ServerThread.
     iv. All output from ServerThread must go to the log file. No output may be sent to System.out.

- **BONUS** – you will get a **3-point EXTRA CREDIT** if you implement the polyalphabetic algorithm with an individual class rather than a method in the ServerThread class. The **PolyAlphabet class** should implement the polyalphabetic encryption algorithm using 2 Caesar ciphers C_1 and C_2 with the repeating pattern $C_1\ C_2\ C_2\ C_1\ C_2$. You must use this class in the ServerThread class for encryption in order to get the extra credit.

6. Your Server.java and ServerThread.java should look something like these:

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

class Server
{
   public static void main(String argv[])
   {
      ...
   }
   public void run()
   {
      ...
   }
}

class ServerThread extends Thread
{
   ...
   public ServerThread (Socket clientSock, PrintWriter logfile) //constructor
   {
      ...
   }
   public void run()
   {
      ...
   }
}
```

7. Log file **entries** may look something like these: (only if you choose to implement a log file)

```
Got a connection: Fri Sep 05 13:40:16 CDT 2014  /137.104.121.250  Port: 9312
Connection closed. Port: 9312
Got a connection: Mon Sep 15 13:32:41 CDT 2014  /137.104.121.234  Port: 11132
ServerThread Exception: java.net.SocketException: Connection reset
```

8. You can run and test your Server and Client on one machine. To do so, bring up your server first, and run your clients with "localhost" (127.0.0.1) and port 5520 (use the next available port number if you are using iLab machines.) Test cases are provided on Canvas for you to test your programs.

9. Grading

Exceptions/Violations	Each Offense	Max Off
Program not running	25	25
Cannot handle multiple connections	10	10
Encryption logic errors	1	3
Improper logs in the log file	0.5	1
Missing the log file	2	2
Improper handling try/catch exceptions	0.5	2