Mathias Morache Module 4 Programming Report

Status:
My programs all work without issue.

Directions:
Assumptions: A current version of JDK is installed and PATH variables are set to run JVM via the command line. If the client and server are on separate machines, then firewall rules / port forwarding has been configured to allow UDP traffic.

1. In the compressed folder, you will find a plain text document called *commands*. The document contains a list of javac commands to compile each of the 14 Java programs necessary to run the client and server applications.
2. Copy the block of commands to the clipboard.
3. Open a command prompt to the directory containing the .java files.
4. Paste the clipboard contents into the command prompt of the machine(s) on which you wish to run the applications. If running the client and server on separate machines, complete this action on both machines. The Server and Client both create Request and Response objects so each will need all the supporting code.
5. On the machine you wish to run the server, enter the following command: `java RecvUDP <port>` The server will listen on the port equal to the argument value plus my GID, which is 1. For example, if you entered 10010, the server will listen on port 10011.
6. On the machine you wish to run the client (or in a second command prompt window if running locally), enter the following command, replacing *host* with the hostname or IP of the server and replacing *port* with the value given to the server plus 1 to account for the GID: `java SendUDP <host> <port>`
7. Follow the prompts to enter a bitwise calculation.
8. The program will return the result of your calculation along with message statistics, then prompt you for another. This will continue indefinitely until you use the escape sequence. Depending on the OS, this is generally *Ctrl + C*.

Report:
This project reinforced two key concepts for me:

1. Be methodical
2. Understand what the code does before making changes

Adapting the provided *Friend* program files into versions required for *Request* and *Response* depended upon multiple pieces of code working together nicely. As such, it was easy to overlook small details. For example, to accommodate the TML of a bitwise NOT request necessitates overloading the constructor of a Request object. In turn, this means making sure that the encoder and the decoder both remain compatible with the changes, as well as making sure any method calls go to the correct constructor.

With a total of 14 programs interacting, this project made me think of the definition of *complex vs. complicated*. It's complicated in the sense that there are a lot of pieces. However, these all interact together in predictable ways. Therefore, they are not complex. The key is to understand their relationships. To this end, I found it helpful to draw out a diagram for reference (attached on the next page). Early on, this was particularly useful.

I also added outputs which I think the typical Facebook user would find convenient. An example of the client output is below:

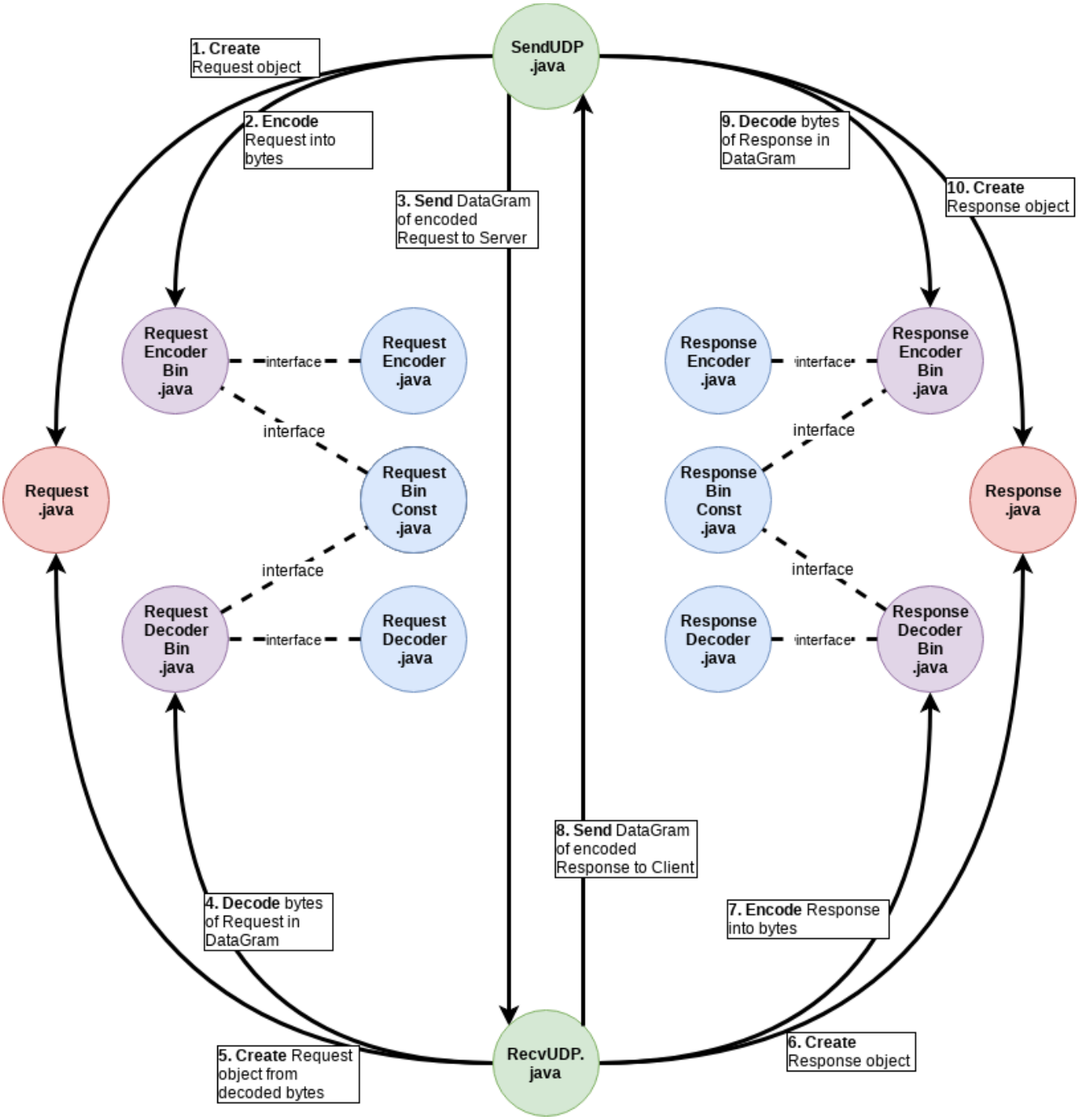
```
Sent Binary-Encoded Request      : 0x08 0x03 0x05 0x02 0x02 0xDF 0x00 0x02
Received Binary-Encoded Response : 0x07 0x03 0x00 0x00 0x00 0x0B 0x7C

Total Message Length of Request  : 8 Bytes
Total Message Length of Response : 7 Bytes
Request Identification Number    : 3
Error Code                       : 0 (Request has no errors)
Result                           : 735 << 2 = 2940
Elapsed time                     : 4.443913 milliseconds

Press 'Enter' to continue
```

I tested my programs by running both the client and server on a local machine and by running them on separate machines in my LAN. Although the minimum times were similar in each scenario, running on the LAN had an average of triple and a maximum of quintuple the time of running locally. The variances do not fall within a discernible pattern either, such as the initial call being the slowest. Regardless, these times are much higher than I would anticipate, which makes me suspect that my measurement method lacks precision, my program runs inefficiently, or my LAN has a routing issue.

	Minimum (ms)	Average (ms)	Maximum (ms)	Number of Tests
Client & Server across LAN	4.631828	172.686782608696	474.4703	23
Client & Server on *localhost*	1.864401	67.7444355185185	97.18329	27


SendUDP .java
1. Create Request object
2. Encode Request into bytes
3. Send DataGram of encoded Request to Server
9. Decode bytes of Response in DataGram
10. Create Response object
Request .java
Request Encoder Bin .java
interface
Request Encoder .java
interface
Request Bin Const .java
interface
Request Decoder Bin .java
interface
Request Decoder .java
Response Encoder .java
interface
Response Encoder Bin .java
interface
Response Bin Const .java
interface
Response Decoder .java
interface
Response Decoder Bin .java
Response .java
8. Send DataGram of encoded Response to Client
4. Decode bytes of Request in DataGram
7. Encode Response into bytes
5. Create Request object from decoded bytes
RecvUDP. java
6. Create Response object