95-702 Distributed Systems for ISM

Project 2

Assigned: Friday, September 20, 2019

Due: Friday October 4, 11:59pm

Learning Objectives: To understand and work with UDP, RPC, TCP, and Digital Signatures

Project 1 established very precise project naming conventions, the use of a submission folder, and the proper submission format. Use those same conventions for Project 2. Submit screenshots for each task. Internal program names are of your own choosing. Choose good names. The course rubric will be applied to a subset of these five tasks. There are five separate and distinct tasks in Project 2.

In all of what follows, we are concerned with designing servers to handle one client at a time. We are not exploring important issues surrounding multiple, simultaneous visitors. If you write a multi-threaded server to handle several visitors at once, that is great but is not required. It gains no additional credit.

In addition, for all of what follows, we are assuming that the server is run before the client is run. If you want to handle the case where the client is run first, without a running server, that is great but will receive no additional credit.

In general, if these requirements do not explicitly ask for feature “x”, then you are not required to provide feature “x”.

(1) Make the following modifications to EchoServerUDP.java and EchoClientUDP.java found at http://www.andrew.cmu.edu/course/95-702/examples/sockets/.
   (a) Change the client's “arg[0]” to a hardcoded "localhost".
   (b) Document the client and the server. Describe what each line of code does.
   (c) Add a line at the top of the client so that it announces, by printing a message, "Client Running" at start up.
   (d) Add a line at the top of the server so that it announces "Server Running" at start up.
   (e) Make additional modifications in the server code so that the request data is copied to an array with the correct number of bytes. Use this array of bytes to build a String of the correct size. Without these modifications, trailing zero bytes may be displayed on each visit. Upon each visit, your server will display the request arriving from the client.
   (f) If the client enters the command "quit!", both the client and the server will halt execution. When the client enters "quit!" it sends "quit!" to the sever but does not wait for any reply.
   (g) Add a line in the client so that it announces when it is quitting.

(h) Add a line in the server so that it announces when it is quitting. The server only quits when it is told to do so by the client.
(i) Note, in the remaining tasks, we do not provide the client with the ability to stop the server. The servers are left running - forever.

(2) Modify and rename EchoServerUDP.java and EchoClientUDP.java.
(a) The server will hold an integer value sum, initialized to 0, and will receive requests from the client - each of which includes a value to be added to the sum. Upon each request, the server will return the new sum as a response to the client. On the server side console, upon each visit by the client, the new sum will be displayed.
(b) Separate concerns on the client. On the client, all of the communication code will be placed in a method named "add". In other words, the main method of the client will have no code related to interacting with a server. Instead, the main routine will simply call a local method named "add". The "add" method will not perform any addition, instead, it will request that the server perform the addition. The "add" method will encapsulate or hide all communication with the server. It is within the "add" method where we actually work with sockets. This is a variation of what is called a "proxy design". The "add" method is serving as a proxy for the server.
(c) Write a client that sends 1000 messages to your server in order to compute the sum 1+2+3+..+1000. Since we are using a proxy design, your client side main routine will call its local "add" method 1000 times. The "add" method actually sends the message to the server. Display the final result to the user on the client side. There is no need to display partial sums on the client side. If you were to run the client a second time, it would be working with the sum that was left on the server by the first client. That is, the server is still alive and is available for use.

(3) Modify your work in Task 2 so that the client may request either an "add" or "subtract" or "view" operation be performed by the server. In addition, each request will pass along an integer ID. Thus, the client will form a packet with the following values: ID, operation (add or subtract or view), and value (if the operation is other than view). The server will carry out the correct computation (add or subtract or view) using the ID found in each request. The client will be menu driven and will repeatedly ask the user for the user ID, operation, and value (if not a view request). When the operation is "view", the value held on the server is returned. When the operation is "add" or "subtract" the server performs the operation and simply returns "OK". During execution, the client will display each returned value from the server to the user. This returned value will be either "OK" or a value (if a view request was made). If the server receives an ID that it has not seen before, that ID will be associated with the sum of 0.

On the server, you will need to map each ID to the value of a sum. Different ID's may be presented and each will have its own sum. The server is given no prior knowledge of what ID's will be transmitted to it by the client. You may only assume that ID's are positive integers.

The client side menu will provide an option to exit the client. This has no impact on the server.

Use a proxy design to encapsulate the communication code.

(4) This is almost the same task as Task 3. The only difference is you will use TCP rather than UDP. Make modifications to EchoServerTCP.java and EchoClientTCP.java found at http://www.andrew.cmu.edu/course/95-702/examples/sockets/.

Use a proxy design to encapsulate the communication code.

(5) Make the following modifications to your work in Task 4.

(a) Each time the client runs, it will create new RSA public and private keys. See RSAExample.java on the schedule for guidance. After it creates these keys, it interacts with the server.

(b) The client's ID will be formed by taking the least significant 20 bytes of the hash of the client's public key. Note: an RSA public key is the pair e and n. Prior to hashing, you might decide to combine these two integers with concatenation. Unlike in (4), we are no longer prompting the user to enter the ID – the ID is computed in the client code. It is derived from the public key.

(c) As before, the client will be interactive and menu driven. It will transmit add or subtract or view requests to the server, along with the ID computed in (b) and an option to exit.

(d) The client will also transmit its public key with each request. Again, note that this key is a combination of e and n. These values will be transmitted in the clear and will be used by the server.

(e) Finally, the client will sign each request. So, by using its private key (d and n), the client will encrypt the hash of the message it sends to the server. The signature will be added to each request. It is very important that the big integer created with the hash (before signing) is positive. See BabySign and BabyVerify for details.

(f) The server will make two checks before servicing any client request. First, does the public key (included with each request) hash to the ID (also provided with each request)? Second, is the request properly signed? If both of these are true, the request is carried out on behalf of the client. The server will add, subtract or view. Otherwise, the server returns the message "Error in request".

(g) See BabyVerify.java and BabySign.java on the course schedule. You need to understand that code before computing a signature. Your solution, however, will not use the short message approach as exemplified there. We are not using any Java crypto API.

(h) We will use SHA-256 for our hash function h(). To clarify further:
The client will send the id: last20BytesOf(h(e+n)), the public key: e and n in the clear, the operation (add, view, or subtract), the operand, and the signature E(h(all prior tokens),d). The signature is thus an encrypted hash. It is encrypted using d and n - the client's private key. E represents standard RSA encryption. The function h(e+n) is the hash of e concatenated with n.

During one client session, the ID will always be the same. If the client quits and restarts, it will have a new ID and operate on a new sum. The server is left running and survives client restarts.

Use a proxy design to encapsulate the communication code.