HomeWork-3

Mohan Srinivas Naga Satya Pothukuchi.

NUID: 001821918

Exercise 4.13

Define a class whose instances represent remote object references. It should contain information similar to that shown in Figure 4.13 and should provide access methods needed by higher-level protocols (see request-reply in Chapter 5, for example). Explain how each of the access methods will be used by that protocol. Give a justification for the type chosen for the instance variable containing information about the interface of the remote object.

The "*DataPacket*" class specified below acts as a Remote Object Reference within the distributed system.

METHOD Definitions:

getPacketType()
This method acts as a getter of PacketType variable. In the DataPacket class, the packet type variable represents an enumeration of operations. These operations represent the different kind of requests and responses that can exist in a server-client communication.

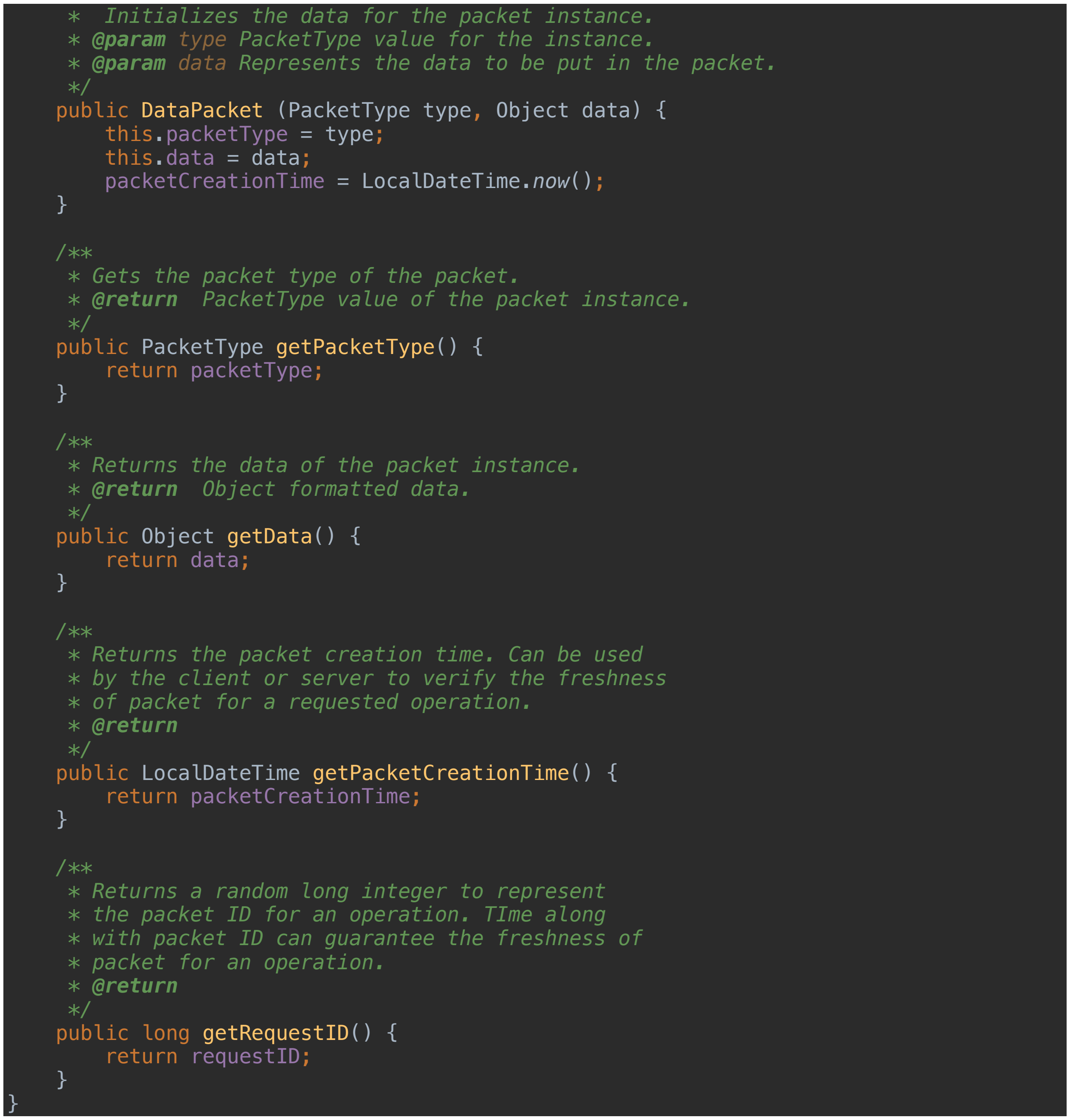
getData()
This method acts as a getter of data variable. This variable is cast into generic Object type, and when the request packet is received by either server or client, based on the packet type, the data variable is cast into its respective type. When the type-checking fails once the operations proceeds, then the server or client throws as casting exception and continues listening for new messages.

getPacketCreationTime()
The packet creation time method in this class returns the time the packet is created. This value is initialized whenever the constructor of a class is called, and new instance is initialized. This values when used by the client or server to receive responses or requests will help the operation maintain a fresh copy of the packet for the request type.

getRequestID()

The requestID is a variable of Long datatype designed to be selected by the client and initialized. This is a random number generated by client and the server uses the same in the response to a request by the client.

```java
package main.java.com.northeastern.Utils;

import java.io.Serializable;
import java.time.LocalDateTime;

/**
 * Represents the packet being transported over the
 * network.
 * Each instance will have the ability to distinguish
 * between the packet type and the pack data as a serialized
 * object.
 */

public class DataPacket implements Serializable {

    //Enum for packet types.
    public enum PacketType {
        PUT,
        DELETE,
        GET,
        DATA,
        SUCCESS,
        KEYS,
        EXIT;

        public String toString() {
            return name();
        }
    }

    // Members of enum.
    private PacketType packetType;

    //Data sent over as an generic type.
    //Based on the packet type, the data
    //variable is casted to it's needed
    //type.
    private Object data;

    //Provides the time of packet generation.
    private LocalDateTime packetCreationTime;

    //Provides the ID of the packet.
    //It is unique for each packet
    //and the client responsible generates
    //the requestID.
    private long requestID;

    /**
     * Constructor for the instance which
     *  Initializes the PacketType for the packet instance.
```



```
     *  Initializes the data for the packet instance.
     * @param type PacketType value for the instance.
     * @param data Represents the data to be put in the packet.
     */
    public DataPacket (PacketType type, Object data) {
        this.packetType = type;
        this.data = data;
        packetCreationTime = LocalDateTime.now();
    }

    /**
     * Gets the packet type of the packet.
     * @return  PacketType value of the packet instance.
     */
    public PacketType getPacketType() {
        return packetType;
    }

    /**
     * Returns the data of the packet instance.
     * @return  Object formatted data.
     */
    public Object getData() {
        return data;
    }

    /**
     * Returns the packet creation time. Can be used
     * by the client or server to verify the freshness
     * of packet for a requested operation.
     * @return
     */
    public LocalDateTime getPacketCreationTime() {
        return packetCreationTime;
    }

    /**
     * Returns a random long integer to represent
     * the packet ID for an operation. TIme along
     * with packet ID can guarantee the freshness of
     * packet for an operation.
     * @return
     */
    public long getRequestID() {
        return requestID;
    }
}
```

As a utility class, this can be shared among the server and client instances. So, for every incoming request or response, the variable type of the remote object for server or client instances can be “***DataPacket***” rather than any other generic data type.

Exercise 4.15

Outline the design of a scheme that uses message retransmissions with IP multicast to overcome the problem of dropped messages. Your scheme should take the following points into account:

- There may be multiple senders.
- Generally, only a small proportion of messages are dropped.
- Recipients may not necessarily send a message within any particular time limit.

Assume that messages that are not dropped arrive in sender order.

The following assumptions are made while designing a schema to handle dropped messages and providing the capabilities for message retransmission.

- Client is part of the multicast group.
- Client has disk space to store the dropped packets sequence number and sender IP Address.
- Server provides the size of the message, sequence number of starting packet and size of packet for one transmission before starting to transmit the message.

Design Outline:

- When there are multiple senders, each packet is associated with the IP Address of the sender. This helps the receiver in the multi-cast group to identify the messages coming in from different senders.
- Senders store the packets sent to the receiver. This is because, as the receiver drops the packet only in small proportions, sending acknowledgments for each and every packet delivered successfully adds lot of overhead. Hence, sending negative acknowledgments to the sender decreases the overhead.
- For doing so, the sender only stores the packet until the end of transmission to receive any negative acknowledgments from the receiver.
- If the sender doesn’t receive any negative acknowledgments, the sender discards or flushes the storage of all the packets it has sent.
- But when the sender receives the negative acknowledgments, it stores the packets based on the sequence number. Once it receives list of dropped packets, it sends a unicast message to the specific receiver in the group.

Sender:

```
//Arguments from the command line provide
    //the message and address of receiver.
    public static void main(String[] args) {
        parseAndVerifyArguments();

        //Create the instance, socket and initializes the
        //buffer and packet.
        Sender senderObject = new Sender();

        //Send information to the IP Multicast group.
        int messageLength = senderObject.getMessage().length();

        //Random number to start the sequence to sending
        //the message.
        double sequenceNumber = Math.random();
        String outMessage
        = "LEN: " + messageLength
            + "SEQ_NUM: " + sequenceNumber
            + "PACKETSIZE: " + packetSize;

        //Send the information packet to the IP Multicast.
        //This will serve as the pointer for identifying
        //any missed packets.
        senderObject.senderPacket(outMessage.getBytes());

        //When the entire message can be sent in one
        //packet. And Save the packet sent in the HashMap
        byte[] data = senderObject.getMessage().getBytes();
        if (packetSize > messageLength) {
            senderObject.senderPacket(data);
            senderObject.getStorageMap.add(sequenceNumber, data);
        } else {
            //When the entire message can be sent in one packet,
            //split the packet and associate a sequence number
            //with each packet.
            for (int index = 0; index <messageLength;) {
                byte[] sendMessage =
Arrays.copyOfRange(senderObject.getMessage().getBytes(),
                                                        index,
                                                        index+packetSize);
                outMessage = "SEQ_NUM: " + sequenceNumber + "DATA: " + sendMessage;
                data = outMessage.getBytes();
                senderObject.senderPacket(data);
                senderObject.getStorageMap().add(sequenceNumber, data);
                index = index + packetSize;
                sequenceNumber = sequenceNumber++;
            }
        }

        //Sender can receive the list of dropped packets or success reply
        //from the receiver to confirm the message has been delivered successfully.
        //This reply will consist of unicast address of the receiver to respon
        //back successfully.
        receiveRequestAndSendMissedPackets()
    }
```

Receiver:

```
    //Arguments from the command line provide
    //the port number to bind the socket.
    public static void main(String[] args) {
        parseAndVerifyArguments();

        //Create the instance, socket and initializes the
        //buffer and packet.
        Receiver receiverObject = new Receiver();

        do {
        //Random number to start the sequence to sending
        //the message.
        double previousSequenceNumber;
        double currentSequenceNumber;

        //The client will receive the length of the
        //message, size of each packet and starting
        //sequence number.
        receiverObject.receive();
        previousSequenceNumber = receiverObject.getSequenceNumber();
        int packetSize = receiverObject.getPacketLength();
        int messageSize = receiverObject.getMessage().length();
        int numOfPacketsExpected = messageSize / packetSize;

        //Estimate the number of iterations needed to receive the message.
        if (numOfPacketsExpected < 1) {
            //Receives the entire packet in one iteration.
            receiverObject.receive();
        } else {
            //Receives the packets in multiple iterations and stores
            //the missed packets.
            numOfPacketsExpected = Math.ceil(numOfPacketsExpected);
            for (int index = 0; index < numOfPacketsExpected; index++) {
                receiverObject.receive();
                currentSequenceNumber = receiverObject.getSequenceNumber();
                if (currentSequenceNumber != previousSequenceNumber + 1) {

receiverObject.getMissedPacketsMap().add(receiverObject.getServerAddress(),
currentSequenceNumber-1);
                }

                previousSequenceNumber = currentSequenceNumber;
            }
        }

        //Once all the packets are received, the receiver checks for any missed
        //packets from the senders, and makes a request for retransmission.
        sendRetransmissionRequest(receiverObject.getMissedPacketsMap());
        } while (receiverObject.getMissedPacketsMap().size() == 0);
    }
```

Exercise 4.18

Revisit the Internet architecture as introduced in Chapter 3 (see Figures 3.12 and 3.14). What impact does the introduction of overlay networks have on this architecture, and in particular on the programmer's conceptual view of the Internet?

An overlay network is a virtual network consisting of nodes and virtual links, which sits on top of an underlying network (such as an IP network). Overlays are layers, but layers that exist outside the standard architecture (such as the TCP/IP stack) and exploit the resultant degrees of freedom.

Overlay networks have the following advantages:

- They enable new network services to be defined without requiring changes to the underlying network, a crucial point given the level of standardization in this area and the difficulties of amending underlying router functionality.
- They encourage experimentation with network services and the customization of services to particular classes of application.
- Multiple overlays can be defined and can coexist, with the end result being a more open and extensible network architecture.

The impacts the overlays introduce are an extra level of indirection which effects the performance of the network and they add to the complexity of network services when compared, for example, to the relatively simple architecture of TCP/IP networks.

In developer's perspective, overlay developers are free to redefine the core elements of a network. These including

- the mode of addressing,
- the protocols employed and
- the approach to routing.

These changes often are tailored towards the need of the application and greatly diverge from the norm or definitions of a standard IP network.

Exercise 5.6

Describe the ways in which the request-reply protocol masks the heterogeneity of operating systems and of computer networks.

A request-reply protocol can be defined as a scheme in which provides delivery guarantees using the send and receive operations defined in each service and invokes the operations on another system using remote object references. The protocol can be described based on a trio of communication primitives: *doOperation, getRequest* and *sendReply.*

The *doOperation* method is used by clients to invoke remote operations. Its arguments specify the remote server and which operation to invoke, together with additional information (arguments) required by the operation. Its result is a byte array containing the reply. It is assumed that the client calling doOperation marshals the arguments into an array of bytes and unmarshals the results from the array of bytes that is returned.

The protocol provides masking of operating systems in a distributed network using Remote References. The first argument of doOperation is an instance of the class RemoteRef, which represents references for remote servers. This class provides methods for getting the Internet address and port of the associated server. The doOperation method sends a request message to the server whose Internet address and port are specified in the remote reference given as an argument. After sending the request message, doOperation invokes receive to get a reply message, from which it extracts the result and returns it to the caller. The caller of doOperation is blocked until the server performs the requested operation and transmits a reply message to the client process.

In doing so, the underlying implementation of method is completely depedent on the client receiving the request. This common understanding of remote object references extracts the implementation details and allows multiple systems of different structure interact with each other in a distributed network.

Similarly, the request-reply protocol doesn't rely on acknowledgements of each packet received. Rather, they depend on the response to the request as an acknowledgment. In doing so, they completely eliminate the overhead caused by connection oriented. Another advantage the request-reply protocol provides is, irrespective of the underlying network used to establish communication, the protocol works in a similar fashion and does not reply on the properties of communicating channel. This helps in abstracting the networks used to communicate and provides cross communication between system which are on different networks as well.

Exercise 5.11

An Election interface provides two remote methods:

- vote: This method has two parameters through which the client supplies the name of a candidate (a string) and the 'voter's number' (an integer used to ensure each user votes once only). The voter's numbers are allocated sparsely from the range of integers to make them hard to guess.
- result: This method has two parameters through which the server supplies the client with the name of a candidate and the number of votes for that candidate.

Which of the parameters of these two procedures are input and which are output parameters?

The specification of a procedure in the interface of a module in a distributed program describes the parameters as input or output, or sometimes both.

- Input parameters are passed to the remote server by sending the values of the arguments in the request message and then supplying them as arguments to the operation to be executed in the server.
- Output parameters are returned in the reply message and are used as the result of the call or to replace the values of the corresponding variables in the calling environment.

From the above description, the method **"vote"** takes in the two parameters passed in by the client. These values are then used by the server to perform an operation. So, as the values are not returned back to the client, the parameters to this method are **"Input Parameters"**

Also, the method **"result"** takes in two parameters, which are unspecified, but the result of the parameters is transmitted back to the client as name of the candidate. As these are sent back to the client, and the client or calling environment has name of the candidate in its environment, these parameters are classified as **"output parameters".**

Exercise 5.16

Outline an implementation for the Election service that ensures that its records remain consistent when it is accessed concurrently by multiple clients.

In JAVA programming language, when a client tries to invoke a remote reference object in a server, which shares resources between multiple client, it is crucial that the resources needs to be consistent between multiple remote object reference calls.

To achieve this consistency, the server can implement a "Synchronized" method(s), in which any resource whose state needs to be consistent among multiple calls be used. In doing so, only one copy of that function exists in the stack of the program, there by maintaining consistency with the data.

In the implementation outlines below

- Server has a "**ClientHandler**" class which takes in client requests by creating a new instance for any incoming request. This client handler class the capability to access the input and output streams and invoke any other procedures to get the job done.
- In doing so, the Client invokes a procedure in the Utility class for which only one instance is created at the Server. This instance passed or shared between all the instances of Clienthandler class.
- Using this instance, the synchronized method doesn't allow another instance of ClientHandler to access the variables at the same time.

```java
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;

/**
 * This class is used to represent a server
 * which takes in requests from multiple clients
 * and yet, maintains a consistent data storage
 * between simulataneous client calls.
 */
public class Server {

    Server() {
        //Create a socket and instantiate any other
        //variables.
    }

    //Takes in the arguments to parse the port number
    //and necessary information to initialize the server.
    public static void main(String[] args) {
        parserAndVerifyArguments();

        try {
            ExecuteRPC rpc = new ExecuteRPC();
            while (1) {
                Socket clientSocket = socket.accept();
                Thread t = new ClientHandler(clientSocket, rpc);
                t.start();
            }
        } catch (IOException e) {
            LOGGER.severe(e.getMessage());
        }
    }
}

/**
 * Class which handles the client logic for
 * an incoming requests.
 */
class ClientHandler extends Thread {
    //Socket for the client.
    Socket socket;

    //Utility class instance.
    ExecuteRPC executeRPC;

    ClientHandler(Socket socket, ExecuteRPC rpc) {
        //Initialize client socket for reading
        //and communicating back to the server

        //Initialzie the executeRPC instance as well.
    }

    public void run() {
        //The parameters are parsed from the input stream.
        executeRPC.execute(voterID, customerName);
    }
}

/**
 * Executes the remote procedure call based on the input values.
 */
class ExecuteRPC {

    //Constructor for intializing the streams.
    ExecuteRPC() {}

    /**
     * The synchronizes keyword makes sure the
     * the method is loaded onto the stack in the
     * memory only one instance at time.
     *
     * Doing so, any resources, that are shared between
     * multiple threads can maintain a common and stale state.
    */
    synchronized public String execute(int voterID, String customerName) {
        customerMap.put(voterID, customerName);
    }
}
```

Exercise 5.22

A client makes remote method invocations to a server. The client takes 5 milliseconds to compute the arguments for each request, and the server takes 10 milliseconds to process each request. The local operating system processing time for each send or receive operation is 0.5 milliseconds, and the network time to transmit each request or reply message is 3 milliseconds. Marshalling or unmarshalling takes 0.5 milliseconds per message.

Calculate the time taken by the client to generate and return from two requests:

- if it is single-threaded
- if it has two threads that can make requests concurrently on a single processor.

You can ignore context-switching times. Is there a need for asynchronous invocation if the client and server processes are threaded?

CASE 1:

Time Taken by Client for one request
= Time Taken by Client to compute arguments
    + time taken for marshalling
    + send operation time requirement
    + time taken to transmit message
    + receive operation by server
    + time taken for unmarshalling
    + server processing time
    + time take by server for marshalling response
    + send operation time requirement
    + time taken for transmitting response
    + time taken to receive message
    + time taken for unmarshalling

= 5ms + 0.5ms + 0.5ms + 3ms + 0.5ms + 0.5ms + 10ms + 0.5ms + 0.5ms + 3ms + 0.5ms + 0.5ms

= 25ms

Total time taken by client for two requests = 25ms * 2
= 50ms

CASE 2:

Time taken by Client for one request = 25ms.
Time taken by Client for second request processed parallely
= time take by client for one request – server processing time
= 25ms – 10ms
= 15ms

Total time take by client for two requests in a multi-threaded environment = 40ms.