Azzolini Riccardo

Socket – UDP e gestione di client multipli

1 Socket UDP

UDP (User Datagram Protocol) permette alle applicazioni di inviare e ricevere datagrammi. Un **datagramma** è definito come un messaggio indipendente e auto-contenuto spedito sulla rete, il cui arrivo, tempo di arrivo, e contenuto non sono garantiti.

Java fornisce sostanzialmente tre classi a supporto del protocollo UDP, contenute nel package `java.net`:

- `DatagramPacket` rappresenta un datagramma;
- `DatagramSocket` è un socket che consente l'invio e la ricezione di `DatagramPacket`;
- `MulticastSocket` viene utilizzato per spedire un messaggio indirizzato a molteplici riceventi.

2 Esempio dimostrativo di UDP

Per dimostrare l'uso di socket UDP in Java, si realizza un server che, in un ciclo infinito, riceve una stringa da un client, la trasforma in maiuscolo, e la rimanda al mittente.

2.1 Server

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

public class UDPServer {
    private static final int SERVER_PORT = 9876;

    public static void main(String[] args) throws IOException {
        try (
            DatagramSocket serverSocket =
                new DatagramSocket(SERVER_PORT)
        ) {
```

```
            byte[] receiveData = new byte[1024];

            while (true) {
                DatagramPacket receivePacket =
                    new DatagramPacket(receiveData, receiveData.length);
                serverSocket.receive(receivePacket);
                String sentence = new String(
                    receivePacket.getData(), 0, receivePacket.getLength()
                );
                System.out.println("RECEIVED: " + sentence);

                String capitalizedSentence = sentence.toUpperCase();

                InetAddress clientAddr = receivePacket.getAddress();
                int clientPort = receivePacket.getPort();
                byte[] sendData = capitalizedSentence.getBytes();
                DatagramPacket sendPacket = new DatagramPacket(
                    sendData, sendData.length, clientAddr, clientPort
                );
                serverSocket.send(sendPacket);
            }
        }
    }
}
```

1. Per prima cosa, il server crea il socket sul quale ricevere le richieste, che qui è di tipo `DatagramSocket` (a differenza del `ServerSocket` usato per TCP).
2. Viene creato un array di byte, `receiveData`, che farà da buffer in cui inserire i dati ricevuti.
3. Nel ciclo, viene creato un `DatagramPacket` associato al buffer `receiveData` (che viene indicato passando al costruttore un riferimento all'array e la sua lunghezza).
4. Si chiama il metodo `serverSocket.receive`, che mette il server in attesa dell'arrivo di un pacchetto (sulla porta specificata nel costruttore della `DatagramSocket`).
5. Una volta ricevuto un pacchetto, si usa il metodo `getData()` del `DatagramPacket` per estrarne il contenuto,[1] che viene poi convertito in una stringa.
6. La stringa visualizzata a schermo e trasformata in maiuscolo.
7. Dal `DatagramPacket` vengono anche estratti l'indirizzo IP e il numero di porta del client.

[1] Il metodo `getData()` restituisce semplicemente un riferimento al buffer associato al `DatagramPacket`, che in questo caso è `receiveData`, quindi sarebbe equivalente accedere direttamente a tale variabile.

8. Si crea un `DatagramPacket` che contiene la versione maiuscola della stringa, ed è indirizzato al client (mediante i dati estratti al passo precedente).
9. Il nuovo pacchetto viene mandato al client, usando il metodo `serverSocket.send`.

2.2 Client

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

public class UDPClient {
    private static final int SERVER_PORT = 9876;

    public static void main(String[] args) throws IOException {
        System.out.println("Enter a string: ");
        String sentence;
        try (
            BufferedReader inFromUser = new BufferedReader(
                new InputStreamReader(System.in)
            )
        ) {
            sentence = inFromUser.readLine();
        }

        try (DatagramSocket clientSocket = new DatagramSocket()) {
            InetAddress serverAddr = InetAddress.getByName(null);
            byte[] sendData = sentence.getBytes();
            DatagramPacket sendPacket = new DatagramPacket(
                sendData, sendData.length, serverAddr, 9876
            );
            clientSocket.send(sendPacket);

            byte[] receiveData = new byte[1024];
            DatagramPacket receivePacket =
                new DatagramPacket(receiveData, receiveData.length);
            clientSocket.receive(receivePacket);
            String modifiedSentence = new String(
                receivePacket.getData(), 0, receivePacket.getLength()
            );
            System.out.println("FROM SERVER: " + modifiedSentence);
        }
    }
}
```

Il codice del client è in gran parte simile a quello del server. Dal punto di vista delle operazioni di gestione dei socket, l'unica differenza sostanziale è che il client deve conoscere l'indirizzo IP e il numero di porta del server, mentre il server ricava indirizzo e porta dei client dai datagrammi che riceve.

Un esempio di esecuzione del client è:

```
Enter a string:
prova
FROM SERVER: PROVA
```

3 Esempio: trasmissione di un'immagine con UDP

Nell'esempio che segue, si usa UDP per inviare un'immagine dal server al client, scomponendola in tanti datagrammi da 1024 byte ciascuno.

Osservazione: Questo esempio potrebbe essere esteso per inviare, ad esempio, i frame di un video.

3.1 Server

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

public class UDPImageServer {
    private static final int SERVER_PORT = 9876;

    public static void main(String[] args) throws IOException {
        try (
            DatagramSocket serverSocket =
                new DatagramSocket(SERVER_PORT)
        ) {
            while (true) {
                DatagramPacket receivePacket = receive(serverSocket);
                InetAddress clientAddr = receivePacket.getAddress();
                int clientPort = receivePacket.getPort();
                send(serverSocket, clientAddr, clientPort);
            }
        }
    }

    private static DatagramPacket receive(
```

```
        DatagramSocket serverSocket
    ) throws IOException {
        byte[] receiveData = new byte[1024];
        DatagramPacket receivePacket =
            new DatagramPacket(receiveData, receiveData.length);
        System.out.println("Waiting");
        serverSocket.receive(receivePacket);
        String sentence = new String(
            receivePacket.getData(), 0, receivePacket.getLength()
        );
        System.out.println("RECEIVED: " + sentence);
        return receivePacket;
    }

    private static void send(
        DatagramSocket serverSocket,
        InetAddress clientAddr,
        int clientPort
    ) throws IOException {
        File file = new File("image.png");
        System.out.println("Starting to send file: " + file);
        try (FileInputStream fis = new FileInputStream(file)) {
            byte[] sendData = new byte[1024];
            while (true) {
                int size = fis.read(sendData);
                if (size < 0) break;
                DatagramPacket sendPacket = new DatagramPacket(
                    sendData, size, clientAddr, clientPort
                );
                serverSocket.send(sendPacket);
            }
        }

        String eof = "END_FILE";
        byte[] sendEOFData = eof.getBytes();
        System.out.println("Sending: " + eof);
        DatagramPacket sendEOFPacket = new DatagramPacket(
            sendEOFData, sendEOFData.length, clientAddr, clientPort
        );
        serverSocket.send(sendEOFPacket);
    }
}
```

Dopo aver creato, come al solito, il proprio `DatagramSocket`, il server si mette in attesa

di ricevere una richiesta, cioè un datagramma, da un client. Tale datagramma conterrà un qualche messaggio (in questo esempio non importa) e, soprattutto, l'indirizzo IP e il numero di porta del client.

Successivamente, il file immagine da spedire viene aperto, e letto a blocchi di (al massimo) 1024 byte. Ciascuno di questi blocchi viene inviato non appena è stato letto, usando i dati del mittente estratti dal `DatagramPacket`.

Una volta raggiunta la fine del file, si manda un pacchetto contenente solo la stringa `"END_FILE"`, per informare il client che la trasmissione è completa.

Infine, il server si rimette in attesa di un'altra richiesta, in un ciclo infinito.

3.2 Client

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

public class UDPImageClient {
    private static final int SERVER_PORT = 9876;

    public static void main(String[] args) throws IOException {
        try (DatagramSocket clientSocket = new DatagramSocket()) {
            send(clientSocket);
            receive(clientSocket);
        }
        System.out.println("CLIENT: finished");
    }

    private static void send(DatagramSocket clientSocket)
            throws IOException {
        InetAddress serverAddr = InetAddress.getByName("localhost");
        byte[] sendData = "START".getBytes();
        DatagramPacket sendPacket = new DatagramPacket(
            sendData, sendData.length, serverAddr, SERVER_PORT
        );
        clientSocket.send(sendPacket);
    }

    private static void receive(DatagramSocket clientSocket)
            throws IOException {
        File file = new File("client-image.png");
        try (FileOutputStream fos = new FileOutputStream(file)) {
            byte[] receiveData = new byte[1024];
```

```
            while (true) {
                DatagramPacket receivePacket =
                    new DatagramPacket(receiveData, receiveData.length);
                clientSocket.receive(receivePacket);
                String sentence = new String(
                    receivePacket.getData(), 0, receivePacket.getLength()
                );
                if (sentence.equals("END_FILE")) {
                    System.out.println(
                        "RECEIVED FROM SERVER: " + sentence
                    );
                    break;
                }
                fos.write(
                    receivePacket.getData(), 0, receivePacket.getLength()
                );
            }
        }
    }
}
```

Il client crea un `DatagramSocket`, e lo usa innanzitutto per mandare al server un pacchetto contenente la stringa `"START"`.

Sapendo che, in risposta, il server inizierà a inviare i pacchetti dell'immagine, il client prepara file di destinazione, e poi inizia un ciclo in cui riceve ciascun pacchetto dal server e ne scrive i contenuti nel file. IL ciclo termina quando il server invia la stringa `"END_FILE"`, che segnala il completamento della trasmissione.

3.3 Uso di UDP per quest'applicazione

Usare UDP per quest'applicazione (in generale, per applicazioni di trasferimento di file) non è una buona idea: se i pacchetti arrivassero in un ordine diverso da quello di invio, o se alcuni di essi si perdessero, il file "ricostruito" dal client non avrebbe lo stesso contenuto di quello inviato dal server. Peggio ancora, si potrebbe perdere proprio il pacchetto che segnala la fine del file: allora, il client rimarrebbe indefinitamente in attesa.

Invece, UDP è adatto (ad esempio) al trasferimento di video nell'ambito dei servizi di streaming: in questo caso, se un frame si perde, è meglio saltarlo piuttosto che attendere che il server lo rimandi, perché intanto lo streaming deve continuare.

4 Server che gestiscono più di un client

Gli esempi di server visti finora sono in grado di gestire un solo client alla volta. Invece, nella maggior parte dei casi, un server deve poter gestire diversi client in parallelo. Ciò è possibile mediante il multithreading:

1. Si crea un unico `ServerSocket`.
2. Si attende una nuova connessione usando `accept()`.
3. Quando un client si connette, cioè quando `accept()` restituisce un `Socket` connesso, si crea un nuovo thread, che ha il compito di "servire" unicamente quel particolare client:
   - al thread viene passato il `Socket` connesso al client;
   - quando ha finito di servire il suo client, il thread termina.
4. Subito dopo aver creato il thread, il server si mette in attesa di un altro client con una nuova chiamata `accept()` (tornando al punto 2).

Osservazione: Questo schema è del tutto generale, cioè rimane sostanzialmente lo stesso per qualunque applicazione.

A livello di terminologia, il thread principale del server (quello che esegue le `accept()`) è spesso chiamato **master**, mentre i thread creati per gestire i singoli client sono detti **slave**.

5 Esempio: echo con client multipli

Come primo esempio di server in grado di gestire più client, si riprende il servizio di "echo" visto in precedenza.

5.1 Server

Il main del server implementa il thread master, seguendo esattamente lo schema appena descritto:

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class MultiEchoServer {
    public static final int PORT = 8080;
```

```
    public static void main(String[] args) throws IOException {
        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
            System.out.println("Server Started");
            while (true) {
                Socket socket = serverSocket.accept();
                try {
                    new OneEchoServer(socket);
                } catch (Exception e) {
                    // If thread creation fails, close the socket.
                    // Otherwise, normally, the socket will be closed by
                    // the thread.
                    socket.close();
                    throw e;
                }
            }
        }
    }
}
```

Il codice del thread slave, che gestisce la comunicazione con il client, non presenta novità rispetto ai server visti in precedenza (a parte il fatto di essere eseguito, appunto, in un thread separato):

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

public class OneEchoServer extends Thread {
    private final Socket socket;

    public OneEchoServer(Socket socket) {
        this.socket = socket;
        start();
    }

    public void run() {
        try (
            BufferedReader in = new BufferedReader(
                new InputStreamReader(socket.getInputStream())
            );
            PrintWriter out = new PrintWriter(
                new BufferedWriter(
                    new OutputStreamWriter(socket.getOutputStream())
                ),
                true // Auto-flush
```

```
            )
        ) {
            echo(in, out);
            System.out.println("Closing...");
        } catch (IOException e) {
            System.err.println("IO Exception");
        } finally {
            try {
                socket.close();
            } catch (IOException e) {
                System.err.println("Socket not closed");
            }
        }
    }

    private void echo(BufferedReader in, PrintWriter out)
            throws IOException {
        while (true) {
            String str = in.readLine();
            if (str.equals("END")) break;
            System.out.println("Echoing: " + str);
            out.println(str);
        }
    }
}
```

5.2 Client

Per verificare che il server sia veramente in grado di gestire più client, si creano molti client che si collegano allo stesso server.

Ciascun client è un thread con un proprio `Socket`, e rimane connesso per un tempo limitato. Nel main, verrà usata una costante `MAX_THREADS` per fissare il numero massimo di thread contemporaneamente in esecuzione, cioè il carico del server: cambiandone il valore, si può vedere si può vedere quando il server comincia ad avere problemi di prestazioni.

```
import java.io.*;
import java.net.InetAddress;
import java.net.Socket;
import java.util.concurrent.ThreadLocalRandom;

public class EchoClientThread extends Thread {
    private final InetAddress serverAddr;
```

```
private final int serverPort;
private final int id;
private static int nextId = 0;
private static int threadCount = 0;

public static int threadCount() {
    return threadCount;
}

public EchoClientThread(InetAddress serverAddr, int serverPort) {
    this.serverAddr = serverAddr;
    this.serverPort = serverPort;

    synchronized (EchoClientThread.class) {
        id = nextId;
        nextId++;
        threadCount++;
    }

    start();
}

public void run() {
    System.out.println("Making client " + id);

    try (
        Socket socket = new Socket(serverAddr, serverPort);
        BufferedReader in = new BufferedReader(
            new InputStreamReader(socket.getInputStream())
        );
        PrintWriter out = new PrintWriter(
            new BufferedWriter(
                new OutputStreamWriter(socket.getOutputStream())
            ),
            true // Auto-flush
        )
    ) {
        for (int i = 0; i < 25; i++) {
            out.println("Client " + id + ": " + i);
            Thread.sleep(
                ThreadLocalRandom.current().nextInt(200, 800)
            );
            System.out.println(in.readLine());
        }
```

```
            out.println("END");
        } catch (IOException e) {
            System.err.println("IO Exception");
        } catch (InterruptedException e) {
            System.err.println("Interrupted");
        } finally {
            synchronized (EchoClientThread.class) {
                threadCount--;
            }
        }
    }
}
```

Questa classe ha alcuni attributi statici:

- `nextId` è usato per fornire un ID univoco a ciascun client (in modo da permettere di distinguerli nell'output del programma);
- `threadCount` indica quanti client sono al momento attivi, permettendo così di decidere se crearne altri o aspettare, in modo da non superare il limite `MAX_THREADS`.

A parte ciò, il codice di `EchoClientThread` è sostanzialmente uguale ai client visti finora.

Il main crea un client ogni (ad esempio) 100 millisecondi, fino a un massimo di (ad esempio) 40 thread concorrenti. Una volta raggiunto il limite `MAX_THREADS`, prima di creare altri client si aspetta che terminino alcuni di quelli attualmente in esecuzione.

```
import java.io.IOException;
import java.net.InetAddress;

public class MultiEchoClient {
    private static final int MAX_THREADS = 40;

    public static void main(String[] args)
            throws IOException, InterruptedException {
        InetAddress serverAddr = InetAddress.getByName(null);
        while (true) {
            if (EchoClientThread.threadCount() < MAX_THREADS) {
                new EchoClientThread(serverAddr, MultiEchoServer.PORT);
            }
            Thread.sleep(100);
        }
    }
}
```

Oltre al valore di `MAX_THREADS`, anche la durata dello sleep presente nel ciclo del main influenza il numero di thread concorrenti. Infatti, siccome ciascun thread ha una durata limitata, con uno sleep lungo si potrebbe non raggiungere mai il limite imposto da `MAX_THREADS`. Invece, più è breve lo sleep, più è probabile raggiungere il limite; come caso estremo, se lo sleep venisse del tutto rimosso sarebbe di fatto garantito che il numero di client in esecuzione sia sempre pari a `MAX_THREADS`.

5.3 Output

Un esempio di output del client (a sinistra) e del server (a destra) è il seguente:

```
Making client 0
Making client 1
Making client 2
Making client 3
Client 0: 0
Making client 4
Client 2: 0
Making client 5
Client 1: 0
Making client 6
Client 0: 1
Client 4: 0
Making client 7
Making client 8
Client 2: 1
Making client 9
Client 1: 1
Client 4: 1
Making client 10
Client 5: 0
Client 6: 0
Client 3: 0
Making client 11
Client 7: 0
Client 1: 2
Making client 12
...
```

```
Server Started
Echoing: Client 0: 0
Echoing: Client 1: 0
Echoing: Client 2: 0
Echoing: Client 3: 0
Echoing: Client 0: 1
Echoing: Client 4: 0
Echoing: Client 2: 1
Echoing: Client 5: 0
Echoing: Client 1: 1
Echoing: Client 6: 0
Echoing: Client 0: 2
Echoing: Client 4: 1
Echoing: Client 7: 0
Echoing: Client 8: 0
Echoing: Client 2: 2
Echoing: Client 9: 0
Echoing: Client 1: 2
Echoing: Client 4: 2
Echoing: Client 10: 0
Echoing: Client 5: 1
Echoing: Client 6: 1
Echoing: Client 3: 1
Echoing: Client 11: 0
Echoing: Client 7: 1
Echoing: Client 1: 3
...
```

6 Esempio: domande e risposte

Quest'ultimo esempio mostra un server che pone delle domande al client; quando il client invia una risposta, il server verifica se essa sia giusta, e in caso contrario invia al client la risposta corretta.

Il server legge le domande e le risposte corrette da un file `QnA.txt`, nel quale la prima riga è una domanda, la seconda è la risposta corrispondente, e così via. Un esempio di contenuto di questo file è il seguente:

```
What caused the craters on the moon?
meteorites
How far away is the moon (in miles)?
239000
How far away is the sun (in millions of miles)?
93
Is the Earth a perfect sphere?
no
What is the internal temperature of the Earth (in degrees F)?
9000
```

Il server verrà prima presentato nella versione che gestisce un solo client alla volta, e poi modificato in modo da poter gestire più client contemporaneamente.

6.1 Client

Il client è un programma sequenziale (con un unico thread), che riceve messaggi (domande) dal server, li mostra all'utente, e legge da quest'ultimo le risposte da rimandare al server.

L'unica particolarità del codice è che l'indirizzo del server viene letto da riga di comando.

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

public class QAClient {
    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            System.out.println("Usage: java QAClient <server-address>");
            System.exit(1);
        }
        String serverAddr = args[0];
```

```
        try (
            Socket socket = new Socket(serverAddr, QAServer.PORT);
            BufferedReader in = new BufferedReader(
                new InputStreamReader(socket.getInputStream())
            );
            PrintWriter out = new PrintWriter(
                new BufferedWriter(
                    new OutputStreamWriter(socket.getOutputStream())
                ),
                true // Auto-flush
            );
            BufferedReader userIn = new BufferedReader(
                new InputStreamReader(System.in)
            )
        ) {
            String question;
            while ((question = in.readLine()) != null) {
                System.out.println("Server: " + question);
                if (question.equals("END")) break;
                String answer = userIn.readLine();
                out.println(answer);
            }
        }
    }
}
```

6.2 Server che gestisce un client alla volta

Il server, anche nella versione che gestisce solo un client alla volta, è decisamente più complesso rispetto al client.

La gestione dei socket è la solita: la complessità deriva invece dalla logica dell'applicazione. A tal proposito, per facilitare la gestione delle coppie domanda-risposta, si è scelto di rappresentarle mediante un'apposita classe `QA`:

```
public class QA {
    private final String question;
    private final String answer;

    public QA(String question, String answer) {
        this.question = question;
        this.answer = answer;
    }
```

```
    public String getQuestion() { return question; }
    public String getAnswer() { return answer; }
}
```

Il server è implementato da un'istanza della classe `QAServer`, che viene creata nel metodo main della stessa classe, dopo aver caricato le domande e risposte dal file `QnA.txt`:

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

public class QAServer {
    public static final int PORT = 1234;

    private final List<QA> questionsAndAnswers;
    private final Random rand = new Random();

    public QAServer(List<QA> questionsAndAnswers) {
        this.questionsAndAnswers = questionsAndAnswers;
    }

    public void serve() throws IOException {
        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
            System.out.println("Server up and running...");
            while (true) {
                try (
                    Socket socket = serverSocket.accept();
                    BufferedReader in = new BufferedReader(
                        new InputStreamReader(socket.getInputStream())
                    );
                    PrintWriter out = new PrintWriter(
                        new BufferedWriter(
                            new OutputStreamWriter(
                                socket.getOutputStream()
                            )
                        ),
                        true // Auto-flush
                    )
                ) {
                    serveClient(in, out);
                }
            }
        }
```

```
}

private void serveClient(BufferedReader in, PrintWriter out)
        throws IOException {
    String another;
    do {
        QA qa = newQuestion();
        out.println(qa.getQuestion());

        String answer = in.readLine();
        String correct = qa.getAnswer();
        if (correct.equalsIgnoreCase(answer)) {
            out.println("That's correct! Want another? (y/n)");
        } else {
            out.println(
                "Wrong, the correct answer is " + correct
                + ". Want another? (y/n)"
            );
        }

        another = in.readLine();
    } while ("y".equalsIgnoreCase(another));

    out.println("END");
}

private QA newQuestion() {
    int index = rand.nextInt(questionsAndAnswers.size());
    return questionsAndAnswers.get(index);
}

public static void main(String[] args) throws IOException {
    File qaFile = new File("QnA.txt");
    List<QA> questionsAndAnswers = new ArrayList<>();
    try (
        BufferedReader br =
            new BufferedReader(new FileReader(qaFile))
    ) {
        while (true) {
            String question = br.readLine();
            String answer = br.readLine();
            if (question == null || answer == null) break;
            questionsAndAnswers.add(new QA(question, answer));
        }
```

```
        }

        new QAServer(questionsAndAnswers).serve();
    }
}
```

6.3 Server multi-client

Come nell'esempio precedente, per consentire la gestione concorrente di più client bisogna modificare il server in modo che avvii un nuovo thread per ogni connessione:

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

public class QAServer {
    public static final int PORT = 1234;
    private final List<QA> questionsAndAnswers;

    public QAServer(List<QA> questionsAndAnswers) {
        this.questionsAndAnswers = questionsAndAnswers;
    }

    public void serve() throws IOException {
        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
            System.out.println("Server up and running...");
            while (true) {
                Socket socket = serverSocket.accept();
                try {
                    new QAServerThread(socket, questionsAndAnswers)
                        .start();
                } catch (Exception e) {
                    socket.close();
                    throw e;
                }
            }
        }
    }

    public static void main(String[] args) throws IOException {
        File qaFile = new File("QnA.txt");
        List<QA> questionsAndAnswers = new ArrayList<>();
```

```
        try (
            BufferedReader br =
                new BufferedReader(new FileReader(qaFile))
        ) {
            while (true) {
                String question = br.readLine();
                String answer = br.readLine();
                if (question == null || answer == null) break;
                questionsAndAnswers.add(new QA(question, answer));
            }
        }

        new QAServer(questionsAndAnswers).serve();
    }
}
```

L'implementazione del thread slave, che gestisce il singolo client, è la seguente:

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

public class QAServerThread extends Thread {
    private final List<QA> questionsAndAnswers;
    private final Socket socket;
    private final Random rand = new Random();

    public QAServerThread(Socket socket, List<QA> questionsAndAnswers) {
        this.socket = socket;
        this.questionsAndAnswers = questionsAndAnswers;
    }

    public void run() {
        try {
            try (
                BufferedReader in = new BufferedReader(
                    new InputStreamReader(socket.getInputStream())
                );
                PrintWriter out = new PrintWriter(
                    new BufferedWriter(
                        new OutputStreamWriter(socket.getOutputStream())
                    ),
                    true // Auto-flush
                )
```

```
            ) {
                serveClient(in, out);
            } finally {
                socket.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void serveClient(BufferedReader in, PrintWriter out)
        throws IOException {
        String another;
        do {
            QA qa = newQuestion();
            out.println(qa.getQuestion());

            String answer = in.readLine();
            String correct = qa.getAnswer();
            if (correct.equalsIgnoreCase(answer)) {
                out.println("That's correct! Want another? (y/n)");
            } else {
                out.println(
                    "Wrong, the correct answer is " + correct
                    + ". Want another? (y/n)"
                );
            }

            another = in.readLine();
        } while ("y".equalsIgnoreCase(another));

        out.println("END");
    }

    private QA newQuestion() {
        int index = rand.nextInt(questionsAndAnswers.size());
        return questionsAndAnswers.get(index);
    }
}
```

Non è invece necessario apportare alcuna modifica al client.

Osservazione: Il codice applicativo del server è stato suddiviso in due parti: quella specifica per il singolo client è stata trasferita nel thread slave, mentre quella di validità generale

(in particolare, il caricamento delle domande e risposte) è rimasta in `QAServer`.

In generale, quando si passa da un server in grado di gestire un unico client alla volta a un server multi-client, non si tratta di scrivere del codice in più (a parte le poche righe necessarie alla gestione dei thread), ma solo di ristrutturare il codice esistente.