**package** utilities;

**import** java.io.ByteArrayInputStream;

**import** java.io.ByteArrayOutputStream;

**import** java.io.ObjectInputStream;

**import** java.io.ObjectOutputStream;

**import** org.apache.commons.lang3.exception.ExceptionUtils;

**import** org.apache.log4j.Logger;

**public** **class** Encoder {

**static** Logger *logger* = Logger.*getLogger*(Encoder.**class**);

**public** **static** **byte**[] encode(Message \_data) {

**try** {

ByteArrayOutputStream bStream = **new** ByteArrayOutputStream();

ObjectOutputStream objOutput = **new** ObjectOutputStream(bStream);

objOutput.writeObject(\_data);

objOutput.flush();

**byte**[] data = bStream.toByteArray();

objOutput.close();

bStream.close();

**return** data;

} **catch** (Exception e) {

e.printStackTrace();

*logger*.error(ExceptionUtils.*getStackTrace*(e));

**return** **null**;

}

}

**public** **static** Message decode(**byte**[] \_bytes) {

**try** {

**if** (\_bytes == **null** || \_bytes.length == 0)

**return** **null**;

ByteArrayInputStream bInputS = **new** ByteArrayInputStream(\_bytes);

ObjectInputStream oIs = **new** ObjectInputStream(bInputS);

TranslationMessage msg = (TranslationMessage) oIs.readObject();

oIs.close();

bInputS.close();

**return** msg;

} **catch** (Exception e) {

e.printStackTrace();

*logger*.error(ExceptionUtils.*getStackTrace*(e));

**return** **null**;

}

}

}

//////////////////////////////////////////////////////////////////

**package** utilities;

**import** java.io.Serializable;

**public** **class** Message **implements** Serializable {

}

//////////////////////////////////////////////////////////////////

**package** utilities;

**import** java.util.UUID;

**public** **class** TranslationMessage **extends** Message {

**private** **static** **final** **long** *serialVersionUID* = 1L;

**private** UUID requestId = UUID.*randomUUID*();

**private** UUID responseId;

**private** String data1 = "";

**private** String data2 = "";

**private** String response = "";

**public** TranslationMessage(String \_response, TranslationMessage request) {

**super**();

**this**.response = \_response;

**this**.responseId = request.getRequestId();

}

**public** TranslationMessage(String \_response) {

**super**();

**this**.response = \_response;

}

**public** TranslationMessage(String \_data1, String \_data2) {

**super**();

**this**.data1 = \_data1;

**this**.data2 = \_data2;

}

**public** String getData1() {

**return** data1;

}

**public** **void** setData1(String data) {

**this**.data1 = data;

}

@Override

**public** String toString() {

**return** data1;

}

**public** String getData2() {

**return** data2;

}

**public** **void** setData2(String data2) {

**this**.data2 = data2;

}

**public** String getResponse() {

**return** response;

}

**public** **void** setResponse(String response) {

**this**.response = response;

}

**public** UUID getRequestId() {

**return** requestId;

}

**public** **void** setRquestId(UUID requestId) {

**this**.requestId = requestId;

}

**public** UUID getResponseId() {

**return** responseId;

}

**public** **void** setResponseId(UUID responseId) {

**this**.responseId = responseId;

}

}

//////////////////////////////////////////////////////////////////

**package** interactive;

**import** java.io.IOException;

**import** java.net.InetSocketAddress;

**import** java.nio.ByteBuffer;

**import** java.nio.channels.SelectionKey;

**import** java.nio.channels.Selector;

**import** java.nio.channels.ServerSocketChannel;

**import** java.nio.channels.SocketChannel;

**import** java.nio.channels.spi.SelectorProvider;

**import** java.util.Iterator;

**import** junit.framework.Assert;

**import** org.apache.commons.lang3.exception.ExceptionUtils;

**import** org.apache.log4j.Logger;

**import** org.junit.Test;

**import** utilities.Encoder;

**import** utilities.TranslationMessage;

**public** **class** Server **extends** Thread {

Logger \_logger = Logger.*getLogger*(Server.**class**);

SelectionKey selkey = **null**;

Selector sckt\_manager = **null**;

ByteBuffer buffer = ByteBuffer.*allocateDirect*(2048);

**public** Server() {

}

**public** **void** run() {

**try** {

coreServer();

} **catch** (Exception e) {

\_logger.error(ExceptionUtils.*getStackTrace*(e));

}

}

**public** **static** **void** main(String args[]) {

Server \_server = **new** Server();

\_server.start();

}

**private** **void** coreServer() {

**try** {

ServerSocketChannel ssc = ServerSocketChannel.*open*();

**try** {

// Establishing New Channel

ssc.socket().bind(**new** InetSocketAddress(8897));

sckt\_manager = SelectorProvider.*provider*().openSelector();

ssc.configureBlocking(**false**);

SocketChannel client = **null**;

ssc.register(sckt\_manager, SelectionKey.*OP\_ACCEPT*);

\_logger.debug("Channel Establishd");

TranslationMessage msg = **null**;

**while** (**true**) {

sckt\_manager.select();

**for** (Iterator<SelectionKey> i = sckt\_manager.selectedKeys()

.iterator(); i.hasNext();) {

SelectionKey key = i.next();

i.remove();

**if** (key.isConnectable()) {

((SocketChannel) key.channel()).finishConnect();

}

// Accepting a new Client

**if** (key.isAcceptable()) {

client = ssc.accept();

client.configureBlocking(**false**);

client.socket().setTcpNoDelay(**true**);

client.register(sckt\_manager, SelectionKey.*OP\_READ*);

\_logger.debug("A new client established");

}

// reading and writing data

**if** (key.isReadable()) {

buffer = ByteBuffer.*allocateDirect*(2048);

buffer.clear();

**while** (client.read(buffer) <= 0)

;

{

buffer.flip();

// New Input Message

msg = (TranslationMessage) convertBufferToMessage(buffer);

\_logger.debug("Received " + msg.getData1());

**if** (msg.getData1().equals("quit")) {

\_logger.debug("Now disconnecting the client");

client.close();

**return**;

}

}

**int** num = 1 + (**int**) (Math.*random*() \* ((1 - 4) + 1));

Thread.*sleep*(num \* 200);

**int** result = LevenshteinDistance(msg.getData1(),

msg.getData2());

msg = **new** TranslationMessage(

"Levenshtein Distance between string : "

+ msg.getData1() + " and string "

+ msg.getData2() + " is " + result, msg);

buffer.clear();

buffer = ByteBuffer.*wrap*(Encoder.*encode*(msg));

client.write(buffer);

\_logger.debug("Sending " + msg.getResponse());

// if (msg.getData1().equals("quit")

// || msg.getData2().equals("quit")) {

// client.close();

// return;

// }

}

}

}

} **catch** (IOException e) {

\_logger.error(ExceptionUtils.*getStackTrace*(e));

} **finally** {

**try** {

**if** (ssc != **null**) {

ssc.close();

}

} **catch** (IOException e) {

\_logger.error(ExceptionUtils.*getStackTrace*(e));

}

}

} **catch** (Exception e) {

e.printStackTrace();

\_logger.error(ExceptionUtils.*getStackTrace*(e));

}

}

**private** TranslationMessage convertBufferToMessage(ByteBuffer buffer) {

TranslationMessage message = **null**;

**byte**[] bytes = **new** **byte**[buffer.remaining()];

buffer.get(bytes);

message = (TranslationMessage) Encoder.*decode*(bytes);

buffer.clear();

buffer = ByteBuffer.*wrap*(Encoder.*encode*(message));

**return** message;

}

**int** LevenshteinDistance(CharSequence str1, CharSequence str2) {

**int**[][] distance = **new** **int**[str1.length() + 1][str2.length() + 1];

**for** (**int** i = 0; i <= str1.length(); i++)

distance[i][0] = i;

**for** (**int** j = 1; j <= str2.length(); j++)

distance[0][j] = j;

**for** (**int** i = 1; i <= str1.length(); i++)

**for** (**int** j = 1; j <= str2.length(); j++)

distance[i][j] = *minimum*(

distance[i - 1][j] + 1,

distance[i][j - 1] + 1,

distance[i - 1][j - 1]

+ ((str1.charAt(i - 1) == str2.charAt(j - 1)) ? 0

: 1));

**return** distance[str1.length()][str2.length()];

}

**private** **static** **int** minimum(**int** a, **int** b, **int** c) {

**return** Math.*min*(Math.*min*(a, b), c);

}

@Test

**public** **void** testStringDifference() {

String s = "kitten";

String l = "sittin";

**int** distance = LevenshteinDistance(s, l);

Assert.*assertEquals*(2, distance);

s = "kitten";

l = "";

distance = LevenshteinDistance(s, l);

Assert.*assertEquals*(6, distance);

s = "";

l = "";

distance = LevenshteinDistance(s, l);

Assert.*assertEquals*(0, distance);

s = "kitten";

l = "sittin";

distance = LevenshteinDistance(l, s);

Assert.*assertEquals*(2, distance);

s = "kittn";

l = "sittn";

distance = LevenshteinDistance(l, s);

Assert.*assertEquals*(1, distance);

s = "kitten";

l = "sittn";

distance = LevenshteinDistance(l, s);

Assert.*assertEquals*(2, distance);

}

}

//////////////////////////////////////////////////////////////////

**package** interactive;

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.net.InetSocketAddress;

**import** java.nio.ByteBuffer;

**import** java.nio.channels.SelectionKey;

**import** java.nio.channels.Selector;

**import** java.nio.channels.SocketChannel;

**import** org.apache.commons.lang3.exception.ExceptionUtils;

**import** org.apache.log4j.Logger;

**import** utilities.Encoder;

**import** utilities.TranslationMessage;

**public** **class** Client **extends** Thread {

Logger \_logger = Logger.*getLogger*(Client.**class**);

SelectionKey selkey = **null**;

Selector sckt\_manager = **null**;

ByteBuffer buffer = ByteBuffer.*allocateDirect*(2048);

ByteBuffer readBuf = ByteBuffer.*allocateDirect*(2048);

BufferedReader bufReader = **null**;

**public** Client() {

}

**public** **void** coreClient() {

String \_data1 = **null**;

String \_data2 = **null**;

SocketChannel sc = **null**;

**try** {

// Connecting to Server

sc = SocketChannel.*open*();

sc.configureBlocking(**false**);

sc.connect(**new** InetSocketAddress(8897));

**while** (!sc.finishConnect())

; // wait until the connection gets established

\_logger.debug("Connection is accepted by server");

**while** (**true**) {

**if** (sc.isConnected()) {

**try** {

**if** (sc != **null**) {

System.*out*

.println("===============Levenshtein Translator======================");

System.*out*.print("Enter String 1 : ");

bufReader = **new** BufferedReader(

**new** InputStreamReader(System.*in*));

\_data1 = bufReader.readLine();

System.*out*.print("Enter String 2 : ");

bufReader = **new** BufferedReader(

**new** InputStreamReader(System.*in*));

\_data2 = bufReader.readLine();

TranslationMessage msg = **null**;

**if** (\_data1 != **null** && \_data2 != **null**) {

msg = **new** TranslationMessage(\_data1, \_data2);

buffer = ByteBuffer.*wrap*(Encoder.*encode*(msg));

sc.write(buffer);

\_logger.debug("Sending strings '"

+ msg.getData1() + "' and '"

+ msg.getData2() + "'");

**if** (msg.getData1().equals("quit")

|| msg.getData2().equals("quit")) {

sc.close();

**return**;

}

}

buffer.clear();

readBuf.clear();

**while** (sc.read(readBuf) <= 0)

;

readBuf.flip();

msg = (TranslationMessage) convertBufferToMessage(readBuf);

System.*out*.println("Received " + msg.getResponse());

\_logger.debug("Received " + msg.getResponse());

}

} **catch** (Exception e) {

e.printStackTrace();

\_logger.error(ExceptionUtils.*getStackTrace*(e));

}

}

}

} **catch** (IOException e) {

e.printStackTrace();

\_logger.error(ExceptionUtils.*getStackTrace*(e));

} **finally** {

**try** {

**if** (sc.isConnected()) {

sc.close();

}

**if** (bufReader != **null**) {

bufReader.close();

}

} **catch** (IOException e) {

\_logger.error(ExceptionUtils.*getStackTrace*(e));

}

}

}

**public** **void** run() {

**try** {

coreClient();

} **catch** (Exception e) {

e.printStackTrace();

\_logger.error(e);

}

}

**public** **static** **void** main(String args[]) {

Client \_client = **new** Client();

\_client.start();

}

**private** TranslationMessage convertBufferToMessage(ByteBuffer buffer) {

TranslationMessage message = **null**;

**byte**[] bytes = **new** **byte**[buffer.remaining()];

buffer.get(bytes);

message = (TranslationMessage) Encoder.*decode*(bytes);

buffer.clear();

buffer = ByteBuffer.*wrap*(Encoder.*encode*(message));

**return** message;

}

}

//////////////////////////////////////////////////////////////////

**package** aspects;

**import** org.apache.log4j.Logger;

**public** **class** Stats {

Logger \_logger = Logger.*getLogger*(Stats.**class**);

**private** **int** completedConv = 0;

**private** **double** avgTurnAroundTime = 0;

**private** **double** totalTurnAroundTime = 0;

**private** **double** maxTurnAroundTime = Integer.*MIN\_VALUE*;

**private** **double** minTurnAroundTime = Integer.*MAX\_VALUE*;

**private** **double** currentTurnAroundTime = 0;

**private** **int** receives = 0;

**public** **int** getReceives() {

**return** receives;

}

**public** **int** getCompletedConv() {

**return** completedConv;

}

**public** **double** getAvgTurnAroundTime() {

**return** avgTurnAroundTime;

}

**public** **double** getTotalTurnAroundTime() {

**return** totalTurnAroundTime;

}

**public** Stats(**int** completedConv, **long** avgAroundTime) {

**super**();

**this**.completedConv = completedConv;

**this**.avgTurnAroundTime = avgAroundTime;

}

**public** **void** addAvgTurnAroundTime(**double** \_turnAroundTime) {

// \_logger.debug("Stats : time "+\_turnAroundTime);

**this**.currentTurnAroundTime = \_turnAroundTime;

**this**.totalTurnAroundTime += \_turnAroundTime;

// \_logger.debug("Stats : total turnaround time "+totalTurnAroundTime);

**if** (\_turnAroundTime < minTurnAroundTime)

minTurnAroundTime = \_turnAroundTime;

// \_logger.debug("Stats : MIN turnaround time "+minTurnAroundTime);

**if** (\_turnAroundTime > maxTurnAroundTime)

maxTurnAroundTime = \_turnAroundTime;

// \_logger.debug("Stats : MAX turnaround time "+maxTurnAroundTime);

avgTurnAroundTime = ((completedConv \* avgTurnAroundTime) + \_turnAroundTime)

/ (++completedConv);

// \_logger.debug("Stats : avgTurnAroundTime "+avgTurnAroundTime);

}

**public** **double** getMaxTurnAroundTime() {

**return** maxTurnAroundTime;

}

**public** **void** setMaxTurnAroundTime(**double** maxTurnAroundTime) {

**this**.maxTurnAroundTime = maxTurnAroundTime;

}

**public** **double** getMinTurnAroundTime() {

**return** minTurnAroundTime;

}

**public** **void** setMinTurnAroundTime(**double** minTurnAroundTime) {

**this**.minTurnAroundTime = minTurnAroundTime;

}

**public** **double** getCurrentTurnAroundTime() {

**return** currentTurnAroundTime;

}

**public** **void** setCurrentTurnAroundTime(**double** currentTurnAroundTime) {

**this**.currentTurnAroundTime = currentTurnAroundTime;

}

}

//////////////////////////////////////////////////////////////////

**package** aspects;

**import** org.apache.log4j.Logger;

**import** org.junit.Test;

**public** **class** PerformanceMeasure {

Logger \_logger = Logger.*getLogger*(PerformanceMeasure.**class**);

**private** **int** windowSize = 5;

**private** Stats[] statsList = **new** Stats[windowSize];

**private** **int** currentStatIndex = 0;

Stats currentStat = **new** Stats(0, 0);

**public** **void** updateRollingStatsWindow(**double** newTurnAroundTime) {

// \_logger.debug("Current log time is "+ newTurnAroundTime);

**if** (currentStatIndex >= windowSize) {

// Reset the Rolling Window

currentStatIndex = 0;

currentStat = **null**;

}

**if** (currentStatIndex > 0)

currentStat = statsList[currentStatIndex - 1];

**else** **if** (currentStat == **null** && currentStatIndex == 0)

currentStat = statsList[windowSize - 1];

currentStat.addAvgTurnAroundTime(newTurnAroundTime);

statsList[currentStatIndex] = currentStat;

currentStatIndex++;

}

**public** Stats getCurrentStats() {

**return** statsList[currentStatIndex - 1];

}

**public** String printCurrentStats() {

Stats curStats = getCurrentStats();

String str = "================== Stats for Performance Measure ========================\n";

str += " Total Receives " + curStats.getCompletedConv();

str += "\navg Turnaround Time for conv. "

+ curStats.getAvgTurnAroundTime()

+ " Overall total time for conv "

+ curStats.getTotalTurnAroundTime();

str += "\ncurrent Turnaround Time : "

+ curStats.getCurrentTurnAroundTime()

+ " avg Minimum Turnaround Time for conv. "

+ curStats.getMinTurnAroundTime()

+ " Maximum Turnaround time for conv "

+ curStats.getMaxTurnAroundTime();

**return** str;

}

@Test

**public** **void** testUpdateRollingWindowStats() {

**long**[] turnAroundTimes = **new** **long**[] { 2, 1, 3, 6, 1, 3, 6, 1, 1, 2, 4,

3 };

**for** (**int** i = 0; i < 12; i++) {

updateRollingStatsWindow(turnAroundTimes[i]);

}

System.*out*.println(printCurrentStats());

}

}

//////////////////////////////////////////////////////////////////

**package** aspects;

**import** java.nio.ByteBuffer;

**import** java.nio.channels.SocketChannel;

**import** java.util.Date;

**import** java.util.HashMap;

**import** java.util.UUID;

**import** interactive.Client;

**import** utilities.TranslationMessage;

**import** utilities.Encoder;

**public** **aspect** ConversationTime {

**private** PerformanceMeasure pm = **new** PerformanceMeasure();

HashMap<UUID, Long> conversationMap = **new** HashMap<UUID, Long>();

**private** **pointcut** ChannelWrite(SocketChannel \_channel, ByteBuffer \_buffer) :

**call**(\* SocketChannel+.write(ByteBuffer)) && **target**(\_channel) && **args**(\_buffer);

**int** **around**(SocketChannel \_channel, ByteBuffer \_buffer) : ChannelWrite(\_channel, \_buffer) {

ByteBuffer tempBuf = \_buffer.duplicate();

Object obj = **thisJoinPoint**.getThis();

**if** (obj **instanceof** Client) {

TranslationMessage msg = (TranslationMessage) convertBufferToMessage(tempBuf);

conversationMap.put(msg.getRequestId(), **new** Date().getTime());

}

**return** **proceed**(\_channel, \_buffer);

}

**private** **pointcut** ChannelRead(SocketChannel \_channel, ByteBuffer \_buffer) :

**call**(\* SocketChannel+.read(ByteBuffer)) && **target**(\_channel) && **args**(\_buffer);

**int** **around**(SocketChannel \_channel, ByteBuffer \_buffer) : ChannelRead(\_channel, \_buffer) {

ByteBuffer tempBuf = \_buffer.duplicate();

**int** readBytes = **proceed**(\_channel, \_buffer);

**if** (readBytes > 0) {

Object obj = **thisJoinPoint**.getThis();

**if** (obj **instanceof** Client) {

TranslationMessage msg = (TranslationMessage) convertBufferToMessage(tempBuf);

**long** convStartTime = conversationMap.get(msg.getResponseId());

**long** convEndTime = **new** Date().getTime();

pm.updateRollingStatsWindow((**double**) (convEndTime - convStartTime));

System.out.println(pm.printCurrentStats());

}

}

**return** readBytes;

}

**private** TranslationMessage convertBufferToMessage(ByteBuffer buffer) {

TranslationMessage message = **null**;

**byte**[] bytes = **new** **byte**[buffer.remaining()];

buffer.get(bytes);

message = (TranslationMessage) Encoder.decode(bytes);

buffer.clear();

buffer = ByteBuffer.wrap(Encoder.encode(message));

**return** message;

}

}