|  |
| --- |
|  |
| МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ |
| Федеральное государственное бюджетное образовательное учреждение высшего образования  **"МИРЭА - Российский технологический университет"**  **РТУ МИРЭА** |
| Институт информационных технологий (ИТ) |
| Кафедра инструментального и прикладного программного обеспечения |

|  |  |
| --- | --- |
|  | |
| **ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ №3** | |
| **по дисциплине** | |
| **«Архитектура клиент-серверных приложений»**  Тема: «Программирования Java сокетов» | |
| Выполнил студент группы **ИКБО-20-21** | Сидоров С.Д. |
| Принял преподаватель кафедры ИиППО | Волков М.Ю. |

|  |  |  |
| --- | --- | --- |
| Практические работы выполнены | « » 2023г. |  |
| «Зачтено» | « » 2023г. |  |

Москва 2023

**Теоретическое введение**

Клиент-серверная архитектура - это самая известная структура приложений в Интернете. В этой архитектуре клиенты (например, персональные компьютеры, устройства IoT и т. Д.) вначале запрашивают ресурсы с сервера, а затем сервер отправляет соответствующие ответы на запросы клиентов. Чтобы это произошло, как на клиентах, так и на серверах должен быть реализован какой-то механизм, поддерживающий эту сетевую транзакцию. Этот механизм называется коммуникацией через сокеты.

Стоит отметить, что существует два типа сокетов для TCP и UDP. Поскольку большинство сетевых приложений используют TCP, **в тексте работы речь пойдёт только о TCP-сокетах и их реализации.** Основное различие между ними заключается в том, что UDP не имеет соединения, то есть между клиентом и сервером нет сеанса, в то время как TCP ориентирован на соединение, то есть сначала должно быть установлено эксклюзивное соединение между клиентом и сервером для связи.

Сокет работает по типичной модели запрос / ответ, где в java-программе, называемой клиентом, вызывается другая программа, называемая сервером, работающая на другой JVM. Клиент должен отправить запрос, и сервер отвечает ответом. В этой модели обмен всегда инициируется клиентом; сервер не может отправлять какие-либо данные без предварительного запроса клиента. Стоит отметить, что данная модель хорошо работала во всемирной паутине, когда клиенты время от времени запрашивали документы, которые нечасто менялись, но ограничения этого подхода становятся все более актуальными, поскольку контент меняется быстро и пользователи ожидают более интерактивного взаимодействия в сети. Например, WebSocket устраняет эти ограничения, предоставляя полнодуплексный канал связи между клиентом и сервером. Протокол WebSocket (JSR 356) будет рассмотрен в следующей практической работе. Но на практике, сокеты активно используются при реализации клент-серверного взаимодействия.

Сокет - это одна из конечных точек двустороннего канала связи между двумя программами, работающими в сети. Сокет привязан к номеру порта, чтобы уровень TCP мог идентифицировать приложение, данные предназначенные для отправки.

Каждый сервер - это программа, которая работает в определенной системе и прослушивает определенный порт. Сокеты привязаны к номерам портов, и когда мы запускаем любой сервер, он просто слушает сокет и ждет клиентских запросов. Например, сервер tomcat, работающий на порту 8080, ожидает клиентских запросов и, получив любой клиентский запрос, отвечает на них.

Следовательно, для запуска любого сокета необходим запускаемый порт. Очевидно, что он размещен на машине, идентифицируемой именем хоста и уникальной адресованной IP.

В Java классы для поддержки программирования сокетов упакованы в пакет java.net. В соответствии с Java, java.net пакет поставляется с двумя классами Socket и ServerSocket для функций клиента и сервера соответственно.

**Постановка задачи**

Необходимо создать клиент-серверное приложение на языке JAVA с использованием socket, для широковещательного общения пользователей. Приложение может быть как консольным, так и оснащённым полноценным GUI. Клиентское приложение считывает данные из стандартного ввода и отсылает сообщение серверу (с помощью TCP/IP). Сервер, в свою очередь, накапливает сообщения и раз в 5 секунд осуществляет массовую рассылку всем клиентам. Если сообщений за указанный период не поступило, то рассылка не производится. Клиент, получивший сообщение, отображает на экране текст данного сообщения. Структуру и поведение данного клиент-серверного приложения, в том числе, **например**, в части регистрации конкретного клиента и формата широковещательного сообщения, студент определяет самостоятельно.

**Программный код**

Листинг 1 – Код класса Server.java

import java.io.IOException;  
import java.io.PrintWriter;  
import java.net.ServerSocket;  
import java.net.Socket;  
import java.time.LocalDateTime;  
import java.util.Queue;  
import java.util.Scanner;  
import java.util.concurrent.Callable;  
import java.util.concurrent.ConcurrentLinkedQueue;  
import java.util.concurrent.Executors;  
import java.util.concurrent.ScheduledExecutorService;  
import java.util.concurrent.TimeUnit;  
import java.util.concurrent.atomic.AtomicInteger;  
  
public class Server {   
 private static final ScheduledExecutorService *scheduledExecutorService* = Executors.*newScheduledThreadPool*(20);   
 private static final Queue<Callable<Integer>> *mailingTasks* = new ConcurrentLinkedQueue<>();   
 private static final AtomicInteger *numberOfMessagesInBuffer* = new AtomicInteger(0);   
 private static final StringBuffer *globalMessageStringBuffer* = new StringBuffer();  
  
 public static void main(String[] args) throws IOException {  
 try (ServerSocket serverSocket = new ServerSocket(50001)) {  
 while (true) {  
 Socket clientSocket = serverSocket.accept();  
 Callable<Integer> mailingTask = Server.*registerClientMailingTask*(clientSocket);  
 *scheduledExecutorService*.execute(new ClientSession(clientSocket, mailingTask, new Scanner(clientSocket.getInputStream()).nextLine()));  
 }  
 }  
 }  
  
 public static Callable<Integer> registerClientMailingTask(Socket socket) {  
 Callable<Integer> mailingTask = () -> {  
 try {  
 PrintWriter out = new PrintWriter(socket.getOutputStream(), true);  
 out.println(*globalMessageStringBuffer*);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 return 0;  
 };  
 if (Server.*mailingTasks*.add(mailingTask)) {  
 return mailingTask;  
 } else {  
 throw new RuntimeException("Failed to create mailing task for the client: " + socket);  
 }  
 }

Продолжение листинга 1

public static void makeMailing() {  
  
 synchronized (*globalMessageStringBuffer*) {  
 *scheduledExecutorService*.schedule(() -> {  
 try {   
 *scheduledExecutorService*.invokeAll(*mailingTasks*);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {   
 *globalMessageStringBuffer*.setLength(0);  
 Server.*numberOfMessagesInBuffer*.getAndSet(0);  
 }  
 }, 5, TimeUnit.*SECONDS*);  
 }  
 }  
  
 private static class ClientSession implements Runnable {  
 private final Socket socket;  
 private final Callable<Integer> mailingTask;  
 private final String nickname;  
  
 public ClientSession(Socket clientSocket, Callable<Integer> clientMailingTask, String clientNickname) {  
 this.socket = clientSocket;  
 this.mailingTask = clientMailingTask;  
 this.nickname = clientNickname;  
 }  
  
 @Override  
 public void run() {  
 try {  
 Scanner in = new Scanner(socket.getInputStream());  
 while (in.hasNextLine()) {  
 //using just one append for sync purpose  
 *globalMessageStringBuffer*.append(LocalDateTime.*now*() + " " + nickname + ": " + in.nextLine() + "\n");  
 System.*out*.println("Buffer updated: \n" + *globalMessageStringBuffer*);  
 if (Server.*numberOfMessagesInBuffer*.incrementAndGet() == 1) {  
 Server.*makeMailing*();  
 }  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 } finally {  
 try {  
 socket.close();  
 *mailingTasks*.remove(mailingTask);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
}

Листинг 2 – Код класса Client.java

import java.io.PrintWriter;  
import java.net.Socket;  
import java.util.Scanner;  
  
public class Client {  
  
 public static void main(String[] args) throws Exception {  
  
 try (Socket socket = new Socket("localhost", 50001)) {  
 Scanner scanner = new Scanner(System.*in*);  
  
 Scanner in = new Scanner(socket.getInputStream());  
 PrintWriter out = new PrintWriter(socket.getOutputStream(), true);  
  
 System.*out*.println("Input your nickname");  
 out.println(scanner.nextLine());  
  
 while (scanner.hasNextLine()) {  
 out.println(scanner.nextLine());  
 while (in.hasNextLine()) {  
 System.*out*.println(in.nextLine());  
 }  
 }  
 }  
 }  
}

**Вывод программы**

Чтобы начать работу приложения необходимо запустить сервер и начать отсылать с клиентов к нему запросы. Для этого было одновременно открыто 4 консоли, в одной из них был запущен сервер командой «java Server.java», а в трёх остальных – по клиенту. Отправим с клиентов по одному сообщению на сервер (рисунки 1-3).

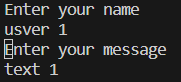


Рисунок 1 – Сообщение, отправленное с клиента №1

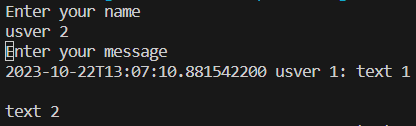


Рисунок 2 – Сообщение, отправленное с клиента №2

На стороне клиента при попадании первого сообщения в буфер начинается обратный отсчёт до рассылки. За это время сервер продолжает регистрировать новые сообщения и добавлять их в буфер. Так, на рисунке 3 можно наблюдать процесс регистрации поступающих сообщений.

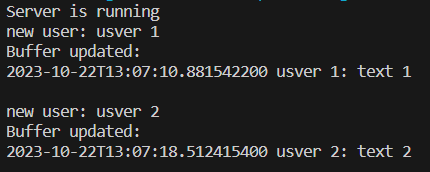
****

Рисунок 3 – Регистрация входящих сообщений сервером

По истечении 5-ти секунд, сервер рассылает всем сокетам по установленным соединениям содержимое буфера и очищает его (рисунки 4-5).

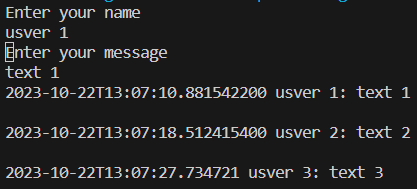
****

Рисунок 4 – Сообщение, полученное клиентом №1

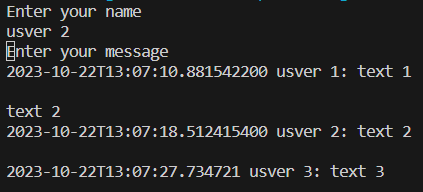
****

Рисунок 5 – Сообщение, полученное клиентом №2

**Вывод**

В результате выполнения данной практической работы было проведено ознакомление с одной из базовых технологии реализации конечной точки для передачи и приема данных по сети - сокетом.