**实验八 Socket支持下的点对点通信的实现**

**【实验目的】**

1．理解socket的基本概念和原理。

2．掌握socket的建立、监听、连接、发送数据和接收数据。

3．本实验综合了高级语言程序设计、点对点通信协议、运输层相关原理及Socket编程思想、方法等知识。

**【实验学时】**

2学时

**【实验性质】**

综合性实验

**【实验环境】**

硬件：两台互联的计算机

软件：Java语言

**【实验报告】**

1．简述TCP/IP体系中的运输层有哪两种协议，并说明每种协议的特点及区别。（15分）

答：

**Protocol，传输控制协议）和UDP（User Datagram Protocol，用户数据报协议）**。这两种协议在功能、特点和应用场景上存在显著的区别。

**1. TCP（Transmission Control Protocol）**

**特点：**

* **面向连接：** TCP在数据传输之前需要建立连接，传输完成后需要释放连接。这种连接是全双工的，即双方可以同时发送和接收数据。
* **可靠传输：** TCP通过确认机制、重传机制、流量控制和拥塞控制等手段，确保数据的可靠传输。如果数据包丢失或损坏，TCP会自动重传。
* **有序传输：** TCP保证数据包按照发送顺序到达接收端。
* **流量控制：** TCP使用滑动窗口机制来控制发送方的发送速率，以防止接收方缓冲区溢出。
* **拥塞控制：** TCP通过慢启动、拥塞避免、快速重传和快速恢复等算法来避免网络拥塞。

**应用场景：**

* 适用于对数据传输可靠性要求高的应用，如文件传输（FTP）、电子邮件（SMTP）、网页浏览（HTTP）等。

**2. UDP（User Datagram Protocol）**

**特点：**

* **无连接：** UDP在数据传输之前不需要建立连接，每个数据包（称为数据报）都是独立的，传输完成后不需要释放连接。
* **不可靠传输：** UDP不提供数据包的确认、重传、流量控制和拥塞控制等机制，数据包可能会丢失、重复或乱序到达。
* **无序传输：** UDP不保证数据包的顺序，接收端可能接收到乱序的数据包。
* **低开销：** 由于UDP不提供复杂的控制机制，其协议头开销较小，传输效率较高。

**应用场景：**

* 适用于对实时性要求高、对数据可靠性要求相对较低的应用，如实时音视频传输（VoIP、视频会议）、在线游戏、DNS查询等。

**区别总结：**

* **连接性：** TCP是面向连接的，UDP是无连接的。
* **可靠性：** TCP提供可靠的数据传输，UDP不保证数据的可靠性。
* **有序性：** TCP保证数据包的有序传输，UDP不保证数据包的顺序。
* **开销：** TCP的协议头开销较大，UDP的协议头开销较小。
* **应用场景：** TCP适用于对数据可靠性要求高的应用，UDP适用于对实时性要求高、对数据可靠性要求相对较低的应用。

2．简述socket的基本概念和原理。（15分）

答：

\*\*Socket（套接字）\*\*是网络通信中的一种抽象概念，它提供了一种编程接口，使得应用程序可以通过网络进行数据的发送和接收。Socket是应用层与传输层之间的接口，它封装了底层的网络协议（如TCP/IP）的细节，为开发者提供了一种简单的方式来实现网络通信。

**基本概念**

1. **Socket的定义：**
   * Socket是一种通信的端点，它包含了网络地址（IP地址）和端口号（Port Number），用于标识网络中的一个进程。
   * Socket可以看作是应用程序与网络之间的桥梁，应用程序通过Socket发送和接收数据。
2. **Socket的类型：**
   * **流式Socket（Stream Socket）：** 基于TCP协议，提供可靠的、面向连接的、有序的数据传输服务。适用于需要可靠数据传输的应用，如文件传输、网页浏览等。
   * **数据报Socket（Datagram Socket）：** 基于UDP协议，提供无连接的、不可靠的、无序的数据传输服务。适用于对实时性要求高、对数据可靠性要求相对较低的应用，如实时音视频传输、在线游戏等。
3. **Socket的组成：**
   * **IP地址：** 标识网络中的主机。
   * **端口号：** 标识主机上的进程。
   * **协议类型：** 标识使用的传输层协议（如TCP或UDP）。

**基本原理**

1. **创建Socket：**
   * 应用程序通过系统调用（如socket()函数）创建一个Socket，操作系统会为该Socket分配一个文件描述符。
   * 创建Socket时需要指定协议类型（TCP或UDP）、地址族（如IPv4或IPv6）等参数。
2. **绑定Socket：**
   * 应用程序通过bind()函数将Socket绑定到一个特定的IP地址和端口号上。
   * 绑定后，Socket就可以通过该地址和端口号进行通信。
3. **监听Socket：**
   * 对于流式Socket，应用程序可以通过listen()函数将Socket设置为监听状态，等待客户端的连接请求。
4. **连接Socket：**
   * 客户端通过connect()函数向服务器发起连接请求。
   * 服务器通过accept()函数接受客户端的连接请求，并创建一个新的Socket用于与客户端通信。
5. **数据传输：**
   * 应用程序通过send()和recv()函数（或write()和read()函数）进行数据的发送和接收。
   * 对于流式Socket，数据传输是可靠的、有序的；对于数据报Socket，数据传输是不可靠的、无序的。
6. **关闭Socket：**
   * 通信结束后，应用程序通过close()函数关闭Socket，释放系统资源。

**总结**

Socket是网络通信中的一个重要概念，它为应用程序提供了一种简单的方式来实现网络通信。通过Socket，应用程序可以创建、绑定、监听、连接、发送和接收数据，从而实现网络通信功能。Socket的原理涉及操作系统提供的系统调用，通过这些调用，应用程序可以与底层的网络协议进行交互，实现数据的传输。

Socket的基本概念和原理是网络编程的基础，理解Socket的工作机制对于开发网络应用程序至关重要。

3．请简要说明程序编写的步骤，分别写出服务器端和客户端应用程序的代码，并分析程序设计思想(程序必须有相应的说明及注释，本页不够可附页)。（60分）

**程序编写的步骤**

1. **需求分析：**
   * **服务器端：创建一个图形化服务器界面，监听特定端口，处理多个客户端连接，支持客户端之间的消息广播。**
   * **客户端：创建一个图形化客户端界面，连接服务器，发送和接收消息，支持多客户端实例。**
2. **设计：**
   * **网络通信设计：**
     + **服务器端：使用ServerSocket监听指定端口，为每个连接的客户端创建独立的ClientHandler线程，支持多客户端并发连接，实现消息广播机制。**
     + **客户端：使用Socket连接服务器，创建独立的接收消息线程(RecvThread)，支持手动输入服务器IP、端口和用户名，实现消息的双向传输。**
   * **图形界面设计：**
     + **服务器端：使用Swing组件构建服务器界面，显示本机IP和监听端口，提供“监听”和“关闭”功能按钮，实时展示服务器和客户端连接日志。**
     + **客户端：使用Swing组件构建客户端界面，提供连接、关闭和创建新客户端功能，实时显示聊天消息，支持用户输入消息。**
3. **编码：**
   * **服务器端：**
     + **创建Server类，继承JFrame，负责界面和服务器整体逻辑。**
     + **创建ClientHandler内部类，处理每个客户端的连接和消息传输。**
     + **实现startServer()方法，启动服务器监听。**
     + **实现disconnect()方法，关闭服务器。**
     + **实现broadcastMessage()方法，广播消息给其他客户端。**
   * **客户端：**
     + **创建Client类，继承JFrame，负责客户端主逻辑。**
     + **创建RecvThread内部类，处理服务器消息接收。**
     + **实现connect()方法，连接服务器。**
     + **实现disconnect()方法，断开连接。**
     + **实现actionPerformed()方法，处理界面交互事件。**
4. **测试：**
   * **编写测试用例，验证服务器端和客户端的基本功能，包括连接、发送消息、接收消息、断开连接等。**
   * **测试多客户端并发连接和消息广播功能。**
5. **调试：**
   * **修复程序中的错误，确保服务器端和客户端能够正常运行。**
   * **处理网络连接相关异常，确保程序的健壮性。**
6. **优化：**
   * **优化代码，提高程序的性能和可读性。**
   * **添加重连机制、完善错误处理、实现消息持久化、增强安全性等。**

服务端

import java.awt.\*;  
import java.awt.event.\*;  
import java.io.\*;  
import java.net.\*;  
import java.util.ArrayList;  
import java.util.concurrent.CopyOnWriteArrayList;  
  
import javax.swing.\*;  
  
public class Server extends JFrame implements *ActionListener* {  
 ServerSocket ss = null;  
 private boolean serverRunning = false;  
 JLabel ipLabel = new JLabel();  
 JLabel portLabel = new JLabel("端口");  
 JTextField portTextField = new JTextField("8000");  
 JButton listenButton = new JButton("监听");  
 JButton closeButton = new JButton("关闭");  
 JPanel panel = new JPanel();  
 JTextArea contentTextArea = new JTextArea();  
  
 public Server() {  
 setTitle("服务器端");  
 setLayout(new BorderLayout());  
 panel.setLayout(new FlowLayout());  
 try {  
 ipLabel.setText("本机IP信息：" + InetAddress.*getLocalHost*().toString());  
 } catch (HeadlessException e) {  
 e.printStackTrace();  
 } catch (UnknownHostException e) {  
 e.printStackTrace();  
 }  
 panel.add(ipLabel);  
 panel.add(portLabel);  
 panel.add(portTextField);  
 panel.add(listenButton);  
 panel.add(closeButton);  
 add(panel, BorderLayout.*NORTH*);  
 add(new JScrollPane(contentTextArea), BorderLayout.*CENTER*);  
 listenButton.addActionListener(this);  
 closeButton.addActionListener(this);  
  
 addWindowListener(new WindowAdapter() {  
 @Override  
 public void windowClosing(WindowEvent e) {  
 disconnect();  
 }  
 });  
 setSize(500, 300);  
 // 将窗体位于屏幕的中央  
 setLocationRelativeTo(null);  
 setVisible(true);  
 }  
  
 public void startServer() {  
 try {  
 int port = Integer.*parseInt*(portTextField.getText());  
 ss = new ServerSocket(port);  
 System.*out*.println("服务器启动，等待客户端连接...");  
 contentTextArea.append("服务器启动，等待客户端连接...\n");  
 serverRunning = true;  
  
 while (serverRunning) {  
 Socket s = ss.accept();  
 new ClientHandler(s).start();  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 contentTextArea.append("启动服务器失败: " + e.getMessage() + "\n");  
 } finally {  
 try {  
 if (ss != null)  
 ss.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 public void disconnect() {  
 serverRunning = false;  
 try {  
 if (ss != null)  
 ss.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 System.*exit*(0);  
 }  
  
 public void actionPerformed(ActionEvent e) {  
 if (e.getSource() == listenButton) {  
 listenButton.setEnabled(false);  
 startServer();  
 } else if (e.getSource() == closeButton) {  
 disconnect();  
 }  
 }  
  
 private class ClientHandler extends Thread {  
 private Socket socket;  
 private DataInputStream is;  
 private DataOutputStream os;  
 private String clientName;  
  
 public ClientHandler(Socket socket) {  
 this.socket = socket;  
 }  
  
 @Override  
 public void run() {  
 try {  
 is = new DataInputStream(socket.getInputStream());  
 os = new DataOutputStream(socket.getOutputStream());  
 clientName = is.readUTF(); // 读取客户端名字  
 System.*out*.println(clientName + " 连接成功");  
 contentTextArea.append(clientName + " 连接成功\n");  
  
 while (true) {  
 String message = is.readUTF();  
 System.*out*.println(clientName + ": " + message);  
 contentTextArea.append(clientName + ": " + message + "\n");  
 broadcastMessage(message, clientName);  
 }  
 } catch (EOFException e) {  
 System.*out*.println(clientName + " 断开连接");  
 contentTextArea.append(clientName + " 断开连接\n");  
 } catch (IOException e) {  
 e.printStackTrace();  
 } finally {  
 try {  
 if (is != null)  
 is.close();  
 if (os != null)  
 os.close();  
 if (socket != null)  
 socket.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 private void broadcastMessage(String message, String sender) throws IOException {  
 for (ClientHandler client : getAllClients()) {  
 if (!client.equals(this)) {  
 client.os.writeUTF(sender + ": " + message);  
 client.os.flush();  
 }  
 }  
 }  
  
 private synchronized List<ClientHandler> getAllClients() {  
 return new ArrayList<>(Server.this.clientHandlers);  
 }  
 }  
  
 private final List clientHandlers = new CopyOnWriteArrayList<>();  
  
 private synchronized void addClient(ClientHandler clientHandler) {  
 clientHandlers.add(String.*valueOf*(clientHandler));  
 }  
  
 private synchronized void removeClient(ClientHandler clientHandler) {  
 clientHandlers.remove(clientHandler.getPriority());  
 }  
  
 public static void main(String[] args) {  
 new Server();  
 }  
}  
客户端

import java.awt.\*;  
import java.awt.event.\*;  
import java.io.\*;  
import java.net.\*;  
  
import javax.swing.\*;  
  
public class Client extends JFrame implements *ActionListener* {  
 private static int *clientCount* = 1; // 静态变量用于跟踪客户端编号  
  
 Socket s = null;  
 DataOutputStream os = null;  
 DataInputStream is = null;  
 private boolean connected = false;  
 JLabel nameLabel = new JLabel("名字");  
 JTextField nameTextField = new JTextField("Client" + *clientCount*++, 10);  
 JLabel ipLabel = new JLabel("IP");  
 JTextField ipTextField = new JTextField("127.0.0.1", 10);  
 JLabel portLabel = new JLabel("端口");  
 JTextField portTextField = new JTextField("8000", 5);  
 JButton connectButton = new JButton("连接");  
 JButton closeButton = new JButton("关闭");  
 JButton newClientButton = new JButton("新建客户端");  
 JPanel panel = new JPanel();  
 JTextField inputTextField = new JTextField();  
 JTextArea contentTextArea = new JTextArea();  
 Thread recvThread = new Thread(new RecvThread());  
  
 public Client() {  
 setTitle("客户端");  
 setLocation(400, 300);  
 setSize(1000, 500);  
 setLayout(new BorderLayout());  
 panel.setLayout(new FlowLayout());  
 panel.add(nameLabel);  
 panel.add(nameTextField);  
 panel.add(ipLabel);  
 panel.add(ipTextField);  
 panel.add(portLabel);  
 panel.add(portTextField);  
 panel.add(connectButton);  
 panel.add(closeButton);  
 panel.add(newClientButton);  
 add(panel, BorderLayout.*NORTH*);  
 add(new JScrollPane(contentTextArea), BorderLayout.*CENTER*);  
 add(inputTextField, BorderLayout.*SOUTH*);  
 connectButton.addActionListener(this);  
 closeButton.addActionListener(this);  
 inputTextField.addActionListener(this);  
 newClientButton.addActionListener(this);  
  
 addWindowListener(new WindowAdapter() {  
 @Override  
 public void windowClosing(WindowEvent e) {  
 disconnect();  
 }  
 });  
 // 将窗体位于屏幕的中央  
 setLocationRelativeTo(null);  
 setVisible(true);  
 }  
  
 public void connect() {  
 try {  
 String host, port, name;  
 host = ipTextField.getText();  
 port = portTextField.getText();  
 name = nameTextField.getText().trim();  
 s = new Socket(host, Integer.*parseInt*(port));  
 os = new DataOutputStream(s.getOutputStream());  
 is = new DataInputStream(s.getInputStream());  
 os.writeUTF(name); // 发送名字到服务器  
 os.flush();  
 connected = true;  
 contentTextArea.append("已连接到服务器作为 " + name + "\n");  
 } catch (UnknownHostException e) {  
 e.printStackTrace();  
 contentTextArea.append("无法连接到服务器: " + e.getMessage() + "\n");  
 } catch (IOException e) {  
 e.printStackTrace();  
 contentTextArea.append("连接失败: " + e.getMessage() + "\n");  
 }  
 }  
  
 public void disconnect() {  
 try {  
 if (os != null)  
 os.close();  
 if (is != null)  
 is.close();  
 if (s != null)  
 s.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 System.*exit*(0);  
 }  
  
 // 使用内部类创建接收数据线程  
 private class RecvThread implements *Runnable* {  
 public void run() {  
 try {  
 while (connected) {  
 String str = is.readUTF();  
 contentTextArea.append(str + "\n");  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 public void actionPerformed(ActionEvent e) {  
 if (e.getSource() == connectButton) {  
 connectButton.setEnabled(false);  
 connect();  
 recvThread.start();  
 } else if (e.getSource() == closeButton) {  
 disconnect();  
 } else if (e.getSource() == inputTextField) {  
 String str = inputTextField.getText().trim();  
 inputTextField.setText("");  
 try {  
 os.writeUTF(str);  
 os.flush();  
 contentTextArea.append("你: " + str + "\n");  
 } catch (IOException e1) {  
 e1.printStackTrace();  
 }  
 } else if (e.getSource() == newClientButton) {  
 new Client(); // 创建新的客户端实例  
 }  
 }  
  
 public static void main(String[] args) {  
 new Client();  
 }

**1. 总体设计目标**

**服务器端**：

* 创建一个图形化服务器界面。
* 监听特定端口，处理多个客户端连接。
* 支持客户端之间的消息广播。

**客户端**：

* 创建一个图形化客户端界面。
* 连接服务器，发送和接收消息。
* 支持多客户端实例。

**2. 关键设计思路**

**网络通信设计**：

* **服务器端**：
  + 使用ServerSocket监听指定端口。
  + 为每个连接的客户端创建独立的ClientHandler线程。
  + 支持多客户端并发连接。
  + 实现消息广播机制。
* **客户端**：
  + 使用Socket连接服务器。
  + 创建独立的接收消息线程(RecvThread)。
  + 支持手动输入服务器IP、端口和用户名。
  + 实现消息的双向传输。

**图形界面设计**：

* **服务器端**：
  + 使用Swing组件构建服务器界面。
  + 显示本机IP和监听端口。
  + 提供“监听”和“关闭”功能按钮。
  + 实时展示服务器和客户端连接日志。
* **客户端**：
  + 使用Swing组件构建客户端界面。
  + 提供连接、关闭和创建新客户端功能。
  + 实时显示聊天消息。
  + 支持用户输入消息。

**3. 代码结构解析**

**主要类和方法**：

* **服务器端**：
  + Server类：服务器主类，继承JFrame，负责界面和服务器整体逻辑。
  + ClientHandler内部类：处理每个客户端的连接和消息传输。
  + startServer()：启动服务器监听。
  + disconnect()：关闭服务器。
  + broadcastMessage()：广播消息给其他客户端。
* **客户端**：
  + Client类：客户端主类，继承JFrame，负责客户端主逻辑。
  + RecvThread内部类：处理服务器消息接收。
  + connect()：连接服务器。
  + disconnect()：断开连接。
  + actionPerformed()：处理界面交互事件。

**4. 具体实现特点**

**网络通信**：

* **服务器端**：
  + 使用DataInputStream和DataOutputStream进行数据传输。
  + 通过Socket进行点对点通信。
  + 每个客户端使用独立线程处理。
* **客户端**：
  + 使用DataInputStream和DataOutputStream进行数据传输。
  + 通过Socket连接服务器。
  + 独立线程接收服务器消息。

**并发处理**：

* **服务器端**：
  + 使用CopyOnWriteArrayList管理客户端列表。
  + 使用synchronized关键字保证线程安全。

**异常处理**：

* **服务器端**：
  + 捕获并处理网络连接相关异常。
  + 优雅地处理客户端断开连接情况。
* **客户端**：
  + 捕获网络连接相关异常。
  + 在界面上显示错误信息。

**5. 程序不足和改进建议**

**服务器端**：

1. **缺少用户身份验证机制**：可以添加用户名和密码验证，确保连接的客户端是合法用户。
2. **没有实现私聊功能**：可以添加私聊功能，允许客户端之间进行一对一的私密聊天。
3. **错误处理和日志记录可以更加完善**：可以添加更详细的日志记录和错误处理机制，提高程序的健壮性。
4. **客户端列表管理方法存在小瑕疵**：addClient和removeClient方法可以进一步优化，确保线程安全。

**客户端**：

1. **缺少重连机制**：可以添加重连机制，当连接断开后自动尝试重新连接服务器。
2. **没有实现完善的错误处理**：可以添加更详细的错误处理机制，提高程序的健壮性。
3. **接收消息线程没有优雅的停止方式**：可以改进接收消息线程的停止方式，确保线程能够优雅地退出。
4. **未实现消息持久化**：可以添加消息持久化功能，将聊天记录保存到本地文件或数据库中。
5. **安全性较低，没有加密通信**：可以添加加密通信机制，确保消息传输的安全性。

4. 实验总结。(10分)

本次实验的主要目的是通过编写一个基于Java Swing的网络聊天应用程序，深入理解TCP/IP网络通信的基本原理，掌握Socket编程的基本方法，并熟悉多线程编程和图形用户界面（GUI）设计。实验内容包括服务器端和客户端的设计与实现。服务器端创建了一个图形化界面，使用ServerSocket监听指定端口，处理多个客户端连接，并为每个连接的客户端创建独立的ClientHandler线程，实现多客户端并发连接和消息广播机制。客户端也创建了一个图形化界面，使用Socket连接服务器，支持手动输入服务器IP、端口和用户名，并创建独立的接收消息线程(RecvThread)，实现消息的双向传输。

在实验过程中，我们首先进行了需求分析，明确了服务器端和客户端的功能需求。然后，设计了程序的结构，包括类、方法、变量等。接着，根据设计编写代码，实现了服务器端和客户端的功能。编写完成后，我们编写了测试用例，验证程序的正确性和稳定性，并进行了调试，修复程序中的错误，确保服务器端和客户端能够正常运行。最后，我们对代码进行了优化，提高了程序的性能和可读性。

通过本次实验，我们成功实现了一个简单的网络聊天应用程序。服务器端能够监听指定端口，处理多个客户端连接，并将客户端发送的消息广播给所有其他客户端。客户端能够连接服务器，发送和接收消息，并支持多客户端实例。实验结果表明，程序设计思想是正确的，代码结构是合理的，程序的可靠性和稳定性得到了验证。

在实验过程中，我们深入理解了TCP/IP网络通信的基本原理，掌握了ServerSocket和Socket的使用方法，实现了数据的可靠传输。通过为每个客户端创建独立的线程，我们实现了多客户端并发连接，熟悉了多线程编程的基本方法，掌握了线程的创建、启动和停止。此外，我们使用Java Swing组件构建了服务器端和客户端的图形界面，熟悉了Swing组件的使用方法，掌握了布局管理器和事件处理机制。在异常处理方面，我们在服务器端和客户端中捕获并处理了网络连接相关异常，确保程序的健壮性，并通过在界面上显示错误信息，提高了程序的可维护性。

尽管实验取得了成功，但仍有一些不足之处需要进一步改进和优化。例如，服务器端缺少用户身份验证机制，可以添加用户名和密码验证；客户端缺少重连机制，可以添加自动重连功能；接收消息线程没有优雅的停止方式，可以改进线程的停止方式；未实现消息持久化和加密通信，可以进一步优化程序的安全性和可靠性。未来，我们将继续优化和扩展程序功能，提高程序的安全性和用户体验。

实验成绩：

批改时间：

评阅教师：