**Hálózat nyújtotta lehetőségek:** A hálózati kapcsolat lehet összeköttetésen alapuló (TCP) vagy összeköttetés mentes(UDP).

**Összeköttetésen alapuló (TCP):**Egy összeköttetésen alapuló kliens-szerver kapcsolat szerver lépései:

* A szerver lefoglal egy TCP portot, melyen keresztül fogadja a kliens kéréseit.
* A szerver elkezd várakozni a kérésekre.
* Ha a kliens közli a szerverrel a kapcsolat felvételi szándékát, akkor a szerver felveszi a klienssel a kapcsolatot, és kiszolgálja azt.

A TCP felépített kapcsolat kétirányú, küldhet a kliens és a szerver is.

Valós életbeli példa a TCP és UDP összehasonlítására:

TCP a telefonhálózaton történő kapcsolatfelvétel, míg az UDP a postai levelezéshez hasonlítható. Miután a telefon kicseng, és a partner felvette a telefont, kialakul közöttük egy vonal, amin tarthatják a kapcsolatot anélkül, hogy minden egyes szó vagy mondat kimondása után újra kellene tárcsázni azt a számot, ahova a legutolsó mondatot küldték. A postai levelezés pedig UDP-re azért jó példa, mert minden levélre rá kell írni a címzettet.

**Összeköttetés mentes kapcsolat:** A kapcsolat minden résztvevőjének létre kell hoznia egy UDP kommunikációs portot, ahol fogadhatja a többiektől érkezett csomagokat, illetve erről a portról küldhet nekik csomagokat. A kommunikáció feltétele, hogy a felek ismerjék a másik címét. Ez a megvalósítás járhat problémákkal, hisz ugyanis nem garantált, hogy az ilyen formában küldött csomagok megérkeznek a helyükre, sőt az sem garantált, ha megérkezik, akkor csak egyszer érkezik meg(UDP protokoll jellemzője, hogy csomagokat veszthet és duplázhat). Az UDP protokollal küldendő csomagoknak van egy méret limitje, azaz bele kell férniük egy IP csomagba.

Az UDP felhasználói datagram protokoll. Lehetővé teszi a felhasználóknak, hogy összeköttetés felépítése és lebontása nélkül üzenetet küldhessenek. Nem garantálja sem az üzenetek kézbesítését, sem azok sorrendtartását.(A TCP ezeket biztosítja).

**Kommunikációs folyamat további kérdései:**

A kommunikáció előkészítésében nagyon fontos, hogy a kliens tudja, hogy a szerver mely számítógépen fut (erőforráscím), és azon belül milyen azonosítójú TCP vagy UDP porton keresztül érhető el.

**Hálózatkezelési osztályok:** A Java.net csomag elemeivel fogunk megismerkedni.

**Összeköttetés alapú kommunikáció javaban:** Az összeköttetés alapú szerver kommunikációs végpontokat implementáló osztály a **ServerSocket.** TCP kommunikációs port absztrakciója.

Két konstruktora van: Az egyik csak egy egész számot vár, ami a létrehozandó szerver TCP portjának száma. A másiknál a második paraméter az, hogy a szerverrel való kommunikációra legfejlebb hány kliens várakozhat (a szerver végpontjához tartozó várakozási sorban).

A szerverrel kapcsolatot felépíteni akaró kliensek egy várakozási sorba kerülnek, érkezési sorrendjükben. A szerver ebből a sorból mindig a legelsővel hozhat létre kommunikációs kapcsolatot a **ServerSocket** accept metódusával (ez egy **Socket** objektumot ad vissza, ez azonosítja a kialakítot kliens-szerver kapcsolatot, adatátviteli csatornát). A **Socket** osztály rendelkezik getInputStream és getOutputStream metódusokkal, ezekkel az adatátvitel I/O csatornákhoz férhetünk hozzá. A gyakorlatban a DataInputStream és DataOutputStream objektumokkal dolgozunk.

A java.nio új I/O csomagban bevezették a getChannel metódust, amellyel I/O csatornára hivatkozó objektumot kapunk, ami az új mechanizmusokat támogatja. Ennek fontos eleme a várakozásmentes I/O alapú hálózati kapcsolatok készítése. Ezáltal a szervereknek nem kell önálló programszálat létrehozni több kliens kiszolgálására.

Blokkoló I/O műveletek: A bemenetfolyamok olvasó metódusai blokkoló műveletek, ez azt jelenti, hogy a read metódus felfüggeszti a műveletet meghívó szálat, amíg kellő beolvasható adat nem lesz, vagy meg nem szűnik a csatorna.

A kliens-szerver koncepciónál lehet probléma. A kliens kéréseket küld, a szerver pedig ezeket olvassa, és az olvasó művelet blokkol.

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

String keres = in.*readUTF*(in);

System.***out***.println(keres+"asd");

out.writeUTF(keres + " asd");

out.flush();

}

Látható, hogy fut egy végtelen ciklus, ami az adatfolyamról olvas, majd kiírja az olvasott adatot egy asd szöveggel kiegészítve. Viszont ez a read művelet addig blokkolja a szálat, amíg nem tud beolvasni valamit(a cliens nem küld el szöveget, amíg nem küld a kliens).Mihelyst jön valami a klienstől tovább megy a sysora, majd megint a ciklus elejéről indulunk, és blokkoljuk a szálat várakozással.

Ezért csinálják azt szerveroldalon, hogy minden kliens kapcsolathoz új szálat hoznak létre, hogy csak velük foglalkoznak, és akkor a szál várakozhat, hisz csak egy klienst szolgál ki.

A következőben megvalósítok egy chat alkalmazást (kliens és szerver oldalon is java), amely blokkoló I/O műveletekkel dolgozik, ezért minden kliensnek külön szálat indít:

Server:

**public** **class** Server {

**public** **static** List<EgyKliensKiszolgalasa> *list* = **new** ArrayList<EgyKliensKiszolgalasa>();

/\*Ebbe a listába tárolom a létező kliens kiszolgálásokat\*/

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

ServerSocket kapcsolodasihely = **new** ServerSocket(9999);

/\*A szerver ezen a porton figyel.\*/

/\*\*/

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

/\*Egy folyamatosan futó ciklus, ami figyeli a várakozási listát,amiben a klienssek

kérési sorrenben helyezkednek el.\*/

Socket kapcsolat = **null**;

EgyKliensKiszolgalasa egy = **null**;

**try**{

System.***out***.println("szerver waiting....");

/\*A várakozási sorból, amiben a kliensek vannak(kérvényküldési sorrendben) a következő

klienssel egy Socket objektumot ad vissza, ami a kliennsel való kapcsolatot

reprezentálja.(adatátviteli csatorna)\*/

kapcsolat = kapcsolodasihely.accept();

/\*\*/

egy = **new** EgyKliensKiszolgalasa(kapcsolat);

*list*.add(egy);

/\*elindítom a szálat, ami az aktuális kliens kiszolgálását végzi.\*/

egy.start();

System.***out***.println("szerver done...");

/\*Ha létrehozta a szálat a kliensenek, akkor újra kezdi előlről a figyelést,azaz nézi, hogy

van-e kliens a várakozási sorban\*/

}**catch**(IOException e){

/\*Ha hiba keletkezik\*/

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

/\*Ha a kapcsolat létrejött, de rossz, akkor lezárjuk, és a szálakat

tartalmazó kezelő listából is kitöröljük.\*/

**try**{

kapcsolat.close();

*list*.remove(egy);

}**catch**(IOException ioe){

}

}

}

}

}

}

A szerver folyamatosan fut, és figyeli, hogy érkezik-e hozzá kérés, ha igen, akkor új szálat indít neki, ami kiszolgálja.

EgyKliensKiszolgálása:

**public** **class** EgyKliensKiszolgalasa **extends** Thread{

/\*Ez a kiszolgáló szál a klienseknek.Ezt listába tárolom.\*/

/\*Ismernie kell a kapcsolatot amin keresztül kommunikálhat.\*/

**private** Socket socket = **null**;

**private** DataInputStream in = **null**;

**private** DataOutputStream out = **null**;

**public** EgyKliensKiszolgalasa(Socket socket){

**this**.socket = socket;

**try** {

in = **new** DataInputStream(socket.getInputStream());

out = **new** DataOutputStream(socket.getOutputStream());

} **catch** (IOException e) {

e.printStackTrace();

}

}

@Override

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

**try**{

/\*Ezt a kódot futtatja a szál.A blokoló olvasás probál kiolvasni az adatfolyamról, ha küld

rá valamit a kliens, akkor az összes végrehajtó szálnak továbbítja, és azok elküldik a

kliensüknek, hogy jelenítsék meg az üzenetet, amit ez a kliens küldött\*/

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

String keres = in.*readUTF*(in);

**for**(**int** i=0;i<Server.*list*.size();i++){

Server.*list*.get(i).out.writeUTF(keres);

Server.*list*.get(i).out.flush();

}

}

}**catch**(EOFException eof){

}**catch**(Throwable t){

}**finally**{

**try**{

socket.close();

Server.*list*.remove(**this**);

}**catch**(Throwable t){

}

}

}

}

Kliens:

**public** **class** Client **extends** JFrame{

/\*Grafikus panel van hozzá.\*/

**private** JPanel contentPane;

**private** JTextField textField;

**private** **static** DataInputStream *in*;

**private** **static** DataOutputStream *out*;

**private** **static** String *text*;

**private** **static** JTextArea *textArea*;

/\*\*

\* Launch the application.

\* **@throws** IOException

\* **@throws** UnknownHostException

\*/

**public** **static** **void** main(String[] args) **throws** UnknownHostException, IOException {

Socket socket = **new** Socket("localhost",9999);

*in* = **new** DataInputStream(socket.getInputStream());

*out* = **new** DataOutputStream(socket.getOutputStream());

EventQueue.*invokeLater*(**new** Runnable() {

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

**try** {

Client frame = **new** Client();

frame.setVisible(**true**);

} **catch** (Exception e) {

e.printStackTrace();

}

}

});

*text* = "";

/\*A kliens folyamatosan figyeli, hogy amíg nem kap exit szöveget, addig olvasná az adatfolyamot,

hogy kap-e valami szöveget, ha kap a szervertől akkor megjeleníti.\*/

**while**(!*text*.equals("exit")){

*text* = *in*.readUTF();

*textArea*.append(System.*lineSeparator*());

*textArea*.append(*text*);

}

}

/\*\*

\* Create the frame.

\*/

**public** Client() {

setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

setBounds(100, 100, 450, 300);

contentPane = **new** JPanel();

contentPane.setBorder(**new** EmptyBorder(5, 5, 5, 5));

setContentPane(contentPane);

contentPane.setLayout(**null**);

textField = **new** JTextField();

textField.setBounds(10, 11, 201, 20);

contentPane.add(textField);

textField.setColumns(10);

JButton btnSend = **new** JButton("Send");

btnSend.setBounds(254, 10, 89, 23);

btnSend.addActionListener(**new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent arg0) {

/\*Ha a gombra kattintok, akkor az adatfolyamra a socketen keresztül

továbbítom a textfieldbe lévő szöveget.\*/

**if**(textField.getText().length() > 0){

**try** {

*out*.writeUTF(textField.getText());

textField.setText("");

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

});

contentPane.add(btnSend);

*textArea* = **new** JTextArea();

*textArea*.setBounds(20, 42, 289, 208);

contentPane.add(*textArea*);

}

}

Ennek a módszernek a működése:A Server folyamatosan figyeli a kliensek csatlakozását, ha csatlakoznak, akkor egy új szálat indít, ami kiszolgálja őket.

A szálak azt csinálják, hogy várják az üzenetet a klienstől, majd a megkapott üzenetet elküldik az összes többi kliensnek.

**Nem blokkoló I/O művelet:**