Modul 5. TCP Socket Programming

Tugas Pendahuluan :

1. Apa yang dimaksud dengan TCP Socket Programming dan mengapa itu penting dalam pengembangan aplikasi jaringan?
2. Bagaimana peran socket dalam TCP Socket Programming? Jelaskan perbedaan antara server socket dan client socket.
3. Apa yang membedakan protokol TCP (Transmission Control Protocol) dari protokol transport lainnya, dan mengapa TCP sering digunakan dalam Socket Programming?
4. Jelaskan secara singkat bagaimana proses pertukaran data terjadi antara server dan klien dalam TCP Socket Programming.
5. Apa keuntungan utama dari menggunakan TCP Socket Programming dalam pengembangan aplikasi jaringan dibandingkan dengan metode komunikasi lainnya?

Materi :

TCP adalah salah satu protokol transport yang beroperasi di lapisan transport dalam model referensi OSI. Protokol ini menyediakan koneksi yang handal, terjamin, dan terorientasi koneksi untuk mentransfer data antara dua titik akhir (endpoints) dalam jaringan. Socket programming adalah pendekatan dalam pengembangan perangkat lunak yang memungkinkan aplikasi untuk berkomunikasi melalui jaringan menggunakan soket (socket). Soket merupakan antarmuka pemrograman aplikasi (API) yang memungkinkan aplikasi di satu mesin berkomunikasi dengan aplikasi di mesin lain.

Dalam konteks TCP Socket Programming, socket berfungsi sebagai endpoint atau titik akhir dalam koneksi jaringan. Socket mencakup informasi seperti alamat IP dan nomor port yang digunakan untuk mengidentifikasi aplikasi atau layanan tertentu pada suatu mesin. Setiap koneksi TCP melibatkan setidaknya dua socket: satu di titik awal (contohnya, klien) dan satu di titik akhir (contohnya, server). Socket pada klien dan server berinteraksi untuk membentuk koneksi dan bertukar data. Berikut ilustrasi Sederhana:

- Bayangkan TCP sebagai "jembatan" yang dapat membawa data antara dua ujung jaringan.
- Socket pada klien dan server adalah "gerbang" atau "titik akses" ke jembatan tersebut.
- Ketika klien dan server berbicara satu sama lain, mereka menggunakan socket untuk membuka dan menutup pintu di jembatan TCP.

```
// Membuat socket klien
Socket clientSocket = new Socket("server-ip", 8080);

// Membuat socket server
ServerSocket serverSocket = new ServerSocket(8080);
Socket connectionSocket = serverSocket.accept();
```

Dalam contoh di atas, `clientSocket` adalah socket pada sisi klien, dan `connectionSocket` adalah socket pada sisi server. Socket ini adalah pintu masuk dan keluar untuk data yang dikirim antara klien dan server.

Pentingnya Socket dalam TCP Socket Programming:

- Socket programming memberikan fleksibilitas dalam komunikasi jaringan dan memungkinkan pengembang untuk membuat aplikasi yang dapat berkomunikasi di seluruh jaringan.
- Socket menyediakan abstraksi yang memungkinkan aplikasi untuk berfokus pada pertukaran data tanpa perlu terlalu terikat pada detail implementasi protokol jaringan di tingkat yang lebih rendah.

Meskipun istilah "TCP Socket programming" sering digunakan bersamaan dengan "model stream," sebenarnya ada perbedaan antara keduanya. Mari kita lihat perbedaan utama antara TCP dan model stream:

1. TCP (Transmission Control Protocol):

- Protokol Transport: TCP adalah protokol transport yang menyediakan koneksi yang handal dan terjamin antara dua titik akhir dalam jaringan.
- Koneksi Terorientasi: TCP merupakan protokol terorientasi koneksi, yang berarti ada proses pembentukan koneksi sebelum data dapat ditransfer, dan koneksi tersebut dijaga selama pertukaran data.
- Reliable dan Ordered: TCP menjamin pengiriman data yang handal (tidak ada kehilangan data) dan memastikan urutan yang benar dari data yang dikirim.

2. Model Stream:

- Aliran Data yang Kontinu: Model stream merujuk pada pendekatan di mana data dikirim sebagai aliran byte yang kontinu antara dua titik akhir. Data dianggap sebagai aliran tanpa batas, tanpa pemisahan paket yang jelas.
- Tidak Terbatas pada TCP: Meskipun TCP sering digunakan untuk mengimplementasikan model stream, model stream sendiri tidak terbatas pada TCP. Protokol lain seperti UDP (User Datagram Protocol) juga dapat digunakan untuk mentransfer data dalam model stream.

Perbedaan Utama:

- TCP adalah protokol transport yang menyediakan layanan terjamin dan teratur untuk mentransfer data.
- Model Stream adalah Pendekatan di mana data dianggap sebagai aliran kontinu tanpa batas, tidak terbatas pada TCP saja.

Dengan kata lain, TCP adalah salah satu implementasi dari model stream dalam konteks socket programming. Model stream sendiri dapat diimplementasikan menggunakan protokol lainnya selain TCP, seperti UDP, tetapi karakteristik kehandalan dan urutan yang terkait dengan TCP mungkin tidak ada dalam implementasi tersebut.

TCP sebagai Protokol Transport:

Transmission Control Protocol (TCP) adalah salah satu protokol transport yang beroperasi di lapisan transport dalam model referensi OSI. TCP memiliki karakteristik khas yang membuatnya sangat sesuai untuk aplikasi yang memerlukan pengiriman data yang handal dan teratur. Dengan fokus pada kehandalan, pengaturan ulang, dan koneksi terorientasi, TCP memastikan bahwa data dapat dikirim dengan aman antara dua titik akhir dalam jaringan. Karakteristik-karakteristik ini membedakan TCP dari protokol transport lainnya, seperti User Datagram Protocol (UDP).

1. Kehandalan TCP:

TCP dirancang untuk memberikan tingkat kehandalan tinggi dalam pengiriman data. Protokol ini menggunakan mekanisme konfirmasi penerimaan (acknowledgment) untuk memastikan bahwa setiap paket data yang dikirimkan telah diterima oleh penerima. Jika penerima tidak mengonfirmasi penerimaan data, TCP akan mengirim ulang paket tersebut.

2. Pengaturan Ulang (Retransmission):

Pengaturan ulang (retransmission) adalah salah satu aspek kunci dari kehandalan TCP. Jika suatu paket data hilang atau rusak selama perjalanan melalui jaringan, TCP akan menginisiasi pengaturan ulang dengan mengirim ulang paket tersebut. Hal ini memastikan bahwa data yang dikirimkan akan sampai dengan utuh dan tidak terjadi kehilangan informasi.

3. Koneksi Terorientasi:

TCP merupakan protokol terorientasi koneksi, yang berarti proses pembentukan koneksi harus dilakukan sebelum data dapat ditransfer. Koneksi ini bersifat handshaking, di mana klien dan server saling berkomunikasi untuk menetapkan parameter koneksi sebelum pertukaran data dimulai. Koneksi terorientasi ini memberikan jaminan bahwa data akan diterima oleh penerima dengan benar dan dalam urutan yang benar.

4. Pengiriman Data yang Handal dan Urutan yang Benar:

TCP menggunakan mekanisme pengaturan ulang dan konfirmasi penerimaan untuk memastikan bahwa data yang dikirimkan tiba dengan aman dan dalam urutan yang benar di titik akhir penerima. Setiap paket data memiliki nomor urut yang unik, dan penerima mengonfirmasi penerimaan paket tersebut. Jika paket hilang atau rusak, TCP akan mengirim ulang paket tersebut untuk memastikan kehandalan dan keutuhan data.

Dengan demikian, TCP sebagai protokol transport memastikan bahwa aplikasi yang mengandalkan kehandalan dan urutan dalam pengiriman data dapat beroperasi secara efektif dan dapat diandalkan di atas jaringan yang mungkin tidak selalu stabil.

Model Stream dalam TCP:

Konsep model stream dalam TCP menggambarkan pendekatan di mana data dianggap sebagai aliran byte yang kontinu antara dua titik akhir (endpoints) komunikasi. Dalam model ini, data tidak dibagi menjadi paket terpisah seperti yang terjadi pada model paket, melainkan dianggap sebagai aliran yang tidak terbatas.

1. Data sebagai Aliran Byte yang Kontinu:

Dalam TCP, data dikirim dan diterima sebagai aliran byte yang terus menerus. Artinya, data tidak dibagi menjadi blok-blok yang terpisah secara eksplisit, dan tidak ada batasan paket yang jelas seperti pada protokol transport lain seperti UDP. Aliran byte ini membuat TCP sangat sesuai untuk aplikasi yang memerlukan transfer data yang konsisten dan terus menerus, seperti transfer file, streaming video, atau protokol aplikasi lainnya.

2. Perbedaan dengan Model Paket (seperti UDP):

Perbedaan kunci antara model stream dalam TCP dan model paket dalam protokol transport seperti UDP adalah cara data diorganisir dan dikirim:

1) Model Stream TCP:

- Data dianggap sebagai aliran byte yang terus menerus.
- Tidak ada batasan paket yang jelas; data dipisahkan dalam bentuk byte tanpa paket yang dapat dibedakan.
- TCP memastikan bahwa data dikirim dengan urutan yang benar dan kehandalan tinggi.

2) Model Paket UDP
   - Data dibagi menjadi paket terpisah dengan batasan ukuran tertentu.
   - Setiap paket berisi informasi header yang menyertakan metadata, seperti alamat pengirim, alamat penerima, dan nomor urut paket.
   - UDP menyediakan layanan tanpa koneksi dan tidak menjamin pengiriman data atau urutan yang benar.

Ilustrasi:

- Model Stream TCP: Bayangkan sebagai aliran air yang terus mengalir tanpa pembatasan, di mana setiap tetesan air mewakili byte data.
- Model Paket UDP: Bayangkan sebagai kumpulan balon air yang terpisah, di mana setiap balon memiliki nomor urut dan informasi pengirim/penerima.

Pentingnya Model Stream dalam TCP:

- Model stream memungkinkan aplikasi untuk bekerja dengan data dalam format yang kontinu dan konsisten.
- Kesesuaian TCP dengan model stream membuatnya ideal untuk aplikasi yang membutuhkan transfer data yang handal dan terus menerus.

Aliran Data dalam TCP:

Dalam model stream TCP, data diatur sebagai aliran byte tanpa pemisahan paket yang jelas. Ini berarti data yang dikirimkan antara pengirim dan penerima dianggap sebagai aliran kontinu tanpa batasan paket yang terpisah. Dalam konteks ini, mari kita jelaskan lebih lanjut cara aliran data diatur dan bagaimana proses pembentukan koneksi serta pertukaran data diimplementasikan dalam model stream TCP.

Cara Data diatur sebagai Aliran Byte:

1. Kirim dan Terima Tanpa Pembatasan:
- Pengirim dapat mengirim data sepanjang waktu tanpa harus memikirkan pembagian data menjadi paket.
- Penerima menerima data sebagai aliran kontinu tanpa harus menangani pembatasan paket yang terpisah.
2. Kontinuitas Byte:
- Data dianggap sebagai serangkaian byte yang terus menerus tanpa pemisahan paket yang jelas.
- Tidak ada "tanda batas" yang memisahkan satu bagian data dengan bagian data lainnya.

Proses Pembentukan Koneksi dan Pertukaran Data dalam Model Stream TCP:

1. Pembentukan Koneksi:
- Proses dimulai dengan pengiriman permintaan koneksi (three-way handshake) dari klien ke server.

- Server merespons permintaan dan mengonfirmasi koneksi.
- Setelah koneksi terbentuk, data dapat mulai ditransfer.

2. Pertukaran Data:

- Data ditransfer dalam bentuk aliran byte antara klien dan server.
- Data yang dikirim oleh pengirim dipisahkan oleh TCP menjadi segmen-segmen yang dikirimkan melalui jaringan.
- Penerima menerima dan menyusun kembali segmen-segmen tersebut menjadi aliran data yang kontinu.

3. Konfirmasi Penerimaan:

- Setiap segmen yang diterima oleh penerima dikonfirmasi dengan mengirimkan acknowledgment (ACK) ke pengirim.
- Jika pengirim tidak menerima konfirmasi dalam batas waktu tertentu, segmen akan dianggap hilang, dan pengirim akan mengirim ulang segmen tersebut.

4. Pemastian Urutan yang Benar:

- TCP memastikan bahwa data yang dikirimkan dari satu titik akhir ke titik akhir lainnya sampai dengan benar dan dalam urutan yang benar.
- Setiap byte dalam aliran data memiliki nomor urut, dan penerima memastikan urutan byte yang benar saat menerima data.

Pentingnya Aliran Data dalam TCP:

- Aliran data tanpa pembatasan paket mempermudah pengembangan aplikasi yang memerlukan transfer data kontinu, seperti transfer file atau streaming media.
- Proses pembentukan koneksi dan pertukaran data dalam model stream TCP memberikan kehandalan dan urutan yang kritis dalam lingkungan jaringan yang mungkin tidak selalu stabil.

Peer to Peer di TCP

Konsep Peer-to-Peer (P2P) dalam konteks TCP (Transmission Control Protocol) melibatkan komunikasi langsung antara dua atau lebih titik akhir (peers) tanpa keterlibatan server pusat. Pada umumnya, koneksi P2P dapat diimplementasikan menggunakan model stream TCP atau model datagram UDP, tergantung pada kebutuhan aplikasi. Berikut adalah beberapa konsep utama terkait dengan P2P dalam konteks TCP:

1. Koneksi Langsung antara Peers: Dalam model P2P menggunakan TCP, setiap peer dapat bertindak sebagai pengirim dan penerima secara bersamaan. Setiap peer memiliki kemampuan untuk membuka koneksi ke peer lainnya dan mengirim serta menerima data.
2. Inisiasi Koneksi oleh Peer: Sebuah peer dapat menginisiasi pembentukan koneksi dengan peer lainnya tanpa memerlukan server pusat. Ini memungkinkan terjadinya komunikasi langsung antara peers tanpa keterlibatan intermediari.
3. Model Stream atau Datagram : P2P dapat diimplementasikan baik dengan menggunakan model stream TCP maupun model datagram UDP, tergantung pada karakteristik aplikasi. Model stream digunakan untuk pertukaran data yang memerlukan kehandalan dan urutan, sementara model datagram dapat digunakan untuk pertukaran data yang lebih bersifat terputus.
4. Penanganan Koneksi dan Pertukaran Data: Setiap peer bertanggung jawab untuk menangani pembentukan koneksi dan pertukaran data dengan peer lainnya. Ini dapat melibatkan proses handshake untuk pembentukan koneksi dan pengaturan ulang data jika terjadi kehilangan atau kesalahan.

5. Koordinasi Peer: Dalam sistem P2P, terdapat kebutuhan untuk koordinasi antara peers. Koordinasi ini dapat mencakup pertukaran informasi seperti alamat IP, status ketersediaan, atau informasi lainnya yang diperlukan untuk berkomunikasi.
6. Skema Topologi P2P: P2P dapat diimplementasikan dengan berbagai skema topologi, seperti fully decentralized (setiap peer setara), hybrid (kombinasi peer setara dan peer yang berfungsi sebagai bootstrap), atau fully centralized (dengan server pusat untuk membantu koneksi awal).
7. Keamanan dan Manajemen Peer: Dalam implementasi P2P, diperlukan keamanan dan manajemen peer yang efektif. Ini termasuk autentikasi peer, manajemen koneksi, dan manajemen sumber daya jaringan.

P2P menggunakan TCP memungkinkan pembentukan jaringan terdesentralisasi yang dapat berkomunikasi langsung satu sama lain. Hal ini sering digunakan dalam aplikasi file sharing, video streaming, dan aplikasi kolaboratif lainnya. Penerapan P2P dengan TCP membutuhkan desain dan manajemen koneksi yang cermat untuk memastikan kehandalan dan efisiensi komunikasi.

Untuk mengimplementasikan Peer-to-Peer (P2P) di Java, Anda dapat menggunakan soket (socket) untuk menangani komunikasi langsung antara peer. Berikut adalah langkah-langkah umum yang dapat Anda ikuti:

1. Buat Kode untuk Peer: Setiap peer perlu memiliki kode Java yang dapat membuat soket untuk menerima koneksi dan mengirim dan menerima data.

```
1.  import java.io.*;
2.  import java.net.*;
3.
4.  public class Peer {
5.      public static void main(String[] args) {
6.          try {
7.              // Buat server socket untuk menerima koneksi
8.              ServerSocket serverSocket = new ServerSocket(12345);
9.
10.             System.out.println("Peer Menunggu Koneksi...");
11.
12.             // Terima koneksi dari peer lain
13.             Socket clientSocket = serverSocket.accept();
14.
15.             // Buat input stream untuk menerima data
16.                                     BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
17.
18.             // Baca data dari peer lain
19.             String receivedData = in.readLine();
20.             System.out.println("Data diterima: " + receivedData);
21.
22.             // Buat output stream untuk mengirim data
23.             PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
24.
25.             // Kirim data ke peer lain
26.             out.println("Hello from Peer!");
27.
28.             // Tutup soket dan stream
29.             in.close();
30.             out.close();
31.             clientSocket.close();
32.             serverSocket.close();
```

```
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    ```

```

2. Buat Kode untuk Peer Lain : Buat kode yang serupa untuk peer lain sehingga mereka dapat saling berkomunikasi. Pastikan untuk mengganti alamat IP dan port dengan informasi yang sesuai.
3. Uji Peer-to-Peer : Jalankan kode peer di dua terminal atau dua program Java yang berbeda untuk melihat apakah mereka dapat saling berkomunikasi.
4. Penanganan Kesalahan : Tambahkan penanganan kesalahan dan manajemen eksepsi untuk memastikan aplikasi dapat menangani situasi yang tidak diinginkan.

Perlu diingat bahwa implementasi P2P sederhana di atas hanya menunjukkan dasar-dasar komunikasi antara dua peer. Untuk skenario P2P yang lebih kompleks, Anda mungkin perlu menambahkan manajemen koneksi, enkripsi, dan fitur-fitur keamanan lainnya.

Selain itu, untuk mendukung banyak peer, Anda dapat mengimplementasikan pendekatan terdesentralisasi di mana setiap peer bertanggung jawab untuk menemukan dan berkomunikasi dengan peer lainnya. Ini dapat melibatkan penggunaan teknologi seperti UPnP (Universal Plug and Play) untuk mengatasi masalah penanganan NAT (Network Address Translation).

Tentu, berikut adalah contoh sederhana untuk aplikasi Peer-to-Peer di Java yang dapat menangani lebih dari dua peer. Aplikasi ini akan memungkinkan setiap peer untuk mengirim pesan ke semua peer lainnya dalam jaringan. Kami akan menggunakan thread untuk menangani koneksi dari setiap peer.

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.List;

public class MultiPeerServer {
    private static final int PORT = 12345;
    private static List<PeerHandler> peers = new ArrayList<>();

    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(PORT);
            System.out.println("Server is running on port " + PORT);

            while (true) {
                Socket clientSocket = serverSocket.accept();
                System.out.println("New peer connected: " + clientSocket.getInetAddress());

                // Membuat handler baru untuk peer dan menambahkannya ke daftar
                PeerHandler peerHandler = new PeerHandler(clientSocket);
                peers.add(peerHandler);
```
```

```
26.
27.                 // Memulai thread untuk menangani koneksi dengan peer
28.                 new Thread(peerHandler).start();
29.             }
30.         } catch (IOException e) {
31.             e.printStackTrace();
32.         }
33.     }
34.
35.     // Broadcast pesan ke semua peer
36.     public static void broadcastMessage(String message, PeerHandler sender) {
37.         for (PeerHandler peer : peers) {
38.             if (peer != sender) {
39.                 peer.sendMessage(message);
40.             }
41.         }
42.     }
43. }
44.
45. class PeerHandler implements Runnable {
46.     private Socket peerSocket;
47.     private BufferedReader reader;
48.     private PrintWriter writer;
49.
50.     public PeerHandler(Socket peerSocket) {
51.         try {
52.             this.peerSocket = peerSocket;
53.                                                 this.reader    =    new    BufferedReader(new
InputStreamReader(peerSocket.getInputStream()));
54.             this.writer = new PrintWriter(peerSocket.getOutputStream(), true);
55.         } catch (IOException e) {
56.             e.printStackTrace();
57.         }
58.     }
59.
60.     @Override
61.     public void run() {
62.         try {
63.             String username = reader.readLine();
64.             System.out.println("New peer connected: " + username);
65.
66.             // Broadcast pesan bahwa peer baru telah bergabung
67.             MultiPeerServer.broadcastMessage(username + " has joined the chat.", this);
68.
69.             String peerMessage;
70.             while ((peerMessage = reader.readLine()) != null) {
71.                 // Broadcast pesan ke semua peer
72.                 MultiPeerServer.broadcastMessage(username + ": " + peerMessage, this);
73.             }
74.
75.         } catch (IOException e) {
76.             e.printStackTrace();
77.         } finally {
78.             // Handle jika peer terputus
79.             try {
80.                 reader.close();
81.                 writer.close();
82.                 peerSocket.close();
83.
84.                 // Hapus handler dari daftar
```

```
                MultiPeerServer.peers.remove(this);

                // Broadcast pesan bahwa peer telah keluar
                MultiPeerServer.broadcastMessage("Peer has left the chat.", this);

            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    // Mengirim pesan ke peer
    public void sendMessage(String message) {
        writer.println(message);
    }
}
```

Untuk menjalankan contoh aplikasi Peer-to-Peer di Java yang menangani lebih dari dua peer, Anda perlu mengikuti beberapa langkah:

1. Salin kode untuk `MultiPeerServer` dan `PeerHandler` ke file Java yang berbeda. Misalnya, simpan `MultiPeerServer` di file `MultiPeerServer.java` dan `PeerHandler` di file `PeerHandler.java`.
2. Kompilasi Kode
   - Buka terminal atau command prompt.
   - Navigasi ke direktori tempat Anda menyimpan file-file Java tersebut.
   - Kompilasi kode dengan menjalankan perintah berikut untuk masing-masing file:

```
javac MultiPeerServer.java
javac PeerHandler.java
```

3. Jalankan Server di terminal atau command prompt yang sama, jalankan server dengan perintah:

```
java MultiPeerServer
```

4. Jalankan Peer:
   - Buka terminal atau command prompt baru untuk setiap peer yang ingin Anda jalankan.
   - Jalankan peer dengan perintah berikut (gantilah `YourUsername` dengan nama pengguna yang diinginkan):

```
java PeerHandler YourUsername
```

5. Mulai Berkomunikasi, Setelah server dan setidaknya dua peer berjalan, Anda akan melihat output bahwa mereka terhubung. ulailah mengirim pesan dari salah satu peer, dan pesan tersebut akan disiarkan ke semua peer lainnya.
6. Eksperimen dengan Lebih Banyak Peer
   Jalankan lebih banyak instance `PeerHandler` dengan nama pengguna yang berbeda untuk melibatkan lebih banyak peer dalam percakapan.

Pastikan untuk menjalankan langkah-langkah ini pada lingkungan pengembangan Java yang sudah diinstal di sistem Anda. Selain itu, pastikan untuk memahami bahwa pada lingkungan produksi,

aplikasi P2P yang lebih canggih mungkin memerlukan manajemen koneksi yang lebih rumit dan keamanan yang lebih baik.

TCP flow control and congestion control

TCP Flow Control dan Congestion Control adalah dua mekanisme yang dibangun ke dalam protokol TCP (Transmission Control Protocol) untuk mengoptimalkan dan mengelola aliran data melalui jaringan. Keduanya berfokus pada cara TCP menyesuaikan diri dengan kondisi jaringan yang berubah dan memastikan efisiensi pengiriman data. Berikut adalah penjelasan singkat tentang keduanya:

1. TCP Flow control adalah mekanisme yang mengatur laju pengiriman data antara pengirim dan penerima untuk menghindari kelebihan beban penerima. Cara Kerjanya adalah Penerima memberi tahu pengirim berapa banyak data yang dapat diterimanya melalui TCP Window Size. Pengirim memonitor ukuran jendela ini dan mengatur laju pengiriman agar sesuai dengan kapasitas penerima.
   Tujua TCP Flow control adalah mencegah pengirim mengirim data terlalu cepat untuk penerima yang mungkin tidak dapat mengaturnya.
2. TCP Congestion Control adalah mekanisme untuk mengelola dan menghindari kemacetan dalam jaringan, yang terjadi ketika ada terlalu banyak data yang bergerak melalui jaringan dan menyebabkan penurunan kinerja. Cara Kerjanya pengirim memantau sinyal keadaan jaringan dan mengatur laju pengiriman sesuai dengan kondisi jaringan. Dengan adanya tanda-tanda kemacetan (misalnya, paket hilang), pengirim mengurangi laju pengiriman.
   Tujuan TCP Congestion Control adalah menghindari kelebihan beban jaringan dan memastikan setiap paket sampai ke tujuannya dengan efisien dan mencegah terjadinya kemacetan yang dapat mengakibatkan penurunan kinerja secara keseluruhan.

Perbedaan Utama:

- Flow Control: Fokus pada hubungan antara pengirim dan penerima untuk memastikan bahwa pengirim tidak mengirim data terlalu cepat untuk penerima.
- Congestion Control: Fokus pada keseimbangan beban di seluruh jaringan, memastikan tidak terjadi kemacetan yang dapat mengakibatkan penurunan kinerja secara keseluruhan.

Kedua mekanisme ini bekerja bersama untuk memastikan efisiensi pengiriman data melalui jaringan TCP, dengan tujuan utama untuk mencegah kemacetan dan mengoptimalkan penggunaan sumber daya jaringan.

Implementasi TCP Flow Control dan Congestion Control di Java biasanya tidak memerlukan tindakan langsung dari pengembang Java karena kedua mekanisme ini telah diimplementasikan di lapisan TCP sendiri. Mekanisme ini terjadi di bawah tingkat aplikasi dan dikelola oleh stack protokol jaringan dan sistem operasi.

Namun, sebagai pengembang aplikasi Java, Anda dapat memanfaatkan beberapa properti dan metode dari kelas `Socket` dan `ServerSocket` untuk memantau dan mengelola aliran dan kemacetan. Berikut adalah beberapa contoh:

1. TCP Flow Control, Menggunakan, menggunakan metode `getReceiveBufferSize()` untuk mendapatkan ukuran buffer penerima:

```
    Socket socket = new Socket("host", port);
    int receiveBufferSize = socket.getReceiveBufferSize();
    ```
```

2. TCP Congestion Control, memantau informasi koneksi, seperti paket yang hilang atau timeout:

```
    Socket socket = new Socket("host", port);
    socket.setSoTimeout(1000); // Timeout dalam milidetik

    // Selanjutnya, Anda dapat menangani SocketTimeoutException untuk mendeteksi paket yang hilang.
```

Menggunakan metode `setTcpNoDelay()` untuk mengatur pengiriman segera tanpa menunggu buffer:

```
    Socket socket = new Socket("host", port);
    socket.setTcpNoDelay(true);
    ```
```

Penting untuk dicatat bahwa kebanyakan implementasi flow control dan congestion control terjadi di lapisan TCP, dan pengembang aplikasi jarang perlu campur tangan secara langsung. Alat-alat ini dirancang untuk beroperasi secara otomatis dan tersembunyi dari pengguna aplikasi untuk memberikan kinerja jaringan yang baik secara default.

TCP Flow Control dan Congestion Control adalah mekanisme internal pada lapisan TCP, dan implementasinya tidak memerlukan tindakan langsung dari pengembang aplikasi. Namun, saya dapat memberikan contoh sederhana penggunaan kelas `Socket` dalam Java, yang secara tidak langsung memanfaatkan mekanisme tersebut:

Contoh untuk TCP Flow Control (Menggunakan `Socket`):

```
1. import java.io.*;
2. import java.net.*;
3.
4. public class TCPSender {
5.     public static void main(String[] args) {
6.         try {
7.             Socket socket = new Socket("localhost", 12345);
8.
9.             // Mendapatkan output stream untuk mengirim data
10.            OutputStream outputStream = socket.getOutputStream();
11.            OutputStreamWriter outputStreamWriter = new OutputStreamWriter(outputStream);
12.            BufferedWriter bufferedWriter = new BufferedWriter(outputStreamWriter);
13.
14.            // Mengirim data
15.            bufferedWriter.write("Hello, Server!");
16.            bufferedWriter.newLine();
17.            bufferedWriter.flush();
18.
19.            // Menutup koneksi
20.            bufferedWriter.close();
21.            socket.close();
22.        } catch (IOException e) {
23.            e.printStackTrace();
24.        }
25.    }
26. }
```

```
27. ```
28.
```

Contoh untuk TCP Congestion Control (Menggunakan `Socket`):

```
 1. import java.io.*;
 2. import java.net.*;
 3.
 4. public class TCPReceiver {
 5.     public static void main(String[] args) {
 6.         try {
 7.             ServerSocket serverSocket = new ServerSocket(12345);
 8.             System.out.println("Server is listening on port 12345...");
 9.
10.             Socket clientSocket = serverSocket.accept();
11.             System.out.println("Client connected: " + clientSocket.getInetAddress());
12.
13.             // Menerima data
14.             InputStream inputStream = clientSocket.getInputStream();
15.             InputStreamReader inputStreamReader = new InputStreamReader(inputStream);
16.             BufferedReader bufferedReader = new BufferedReader(inputStreamReader);
17.
18.             String receivedData = bufferedReader.readLine();
19.             System.out.println("Received data: " + receivedData);
20.
21.             // Menutup koneksi
22.             bufferedReader.close();
23.             clientSocket.close();
24.             serverSocket.close();
25.         } catch (IOException e) {
26.             e.printStackTrace();
27.         }
28.     }
29. }
30. ```
31.
```

Penting untuk dicatat bahwa contoh ini hanya menunjukkan penggunaan dasar `Socket` untuk mengirim dan menerima data. TCP Flow Control dan Congestion Control sebenarnya diatur oleh stack protokol TCP dan sistem operasi, dan Anda tidak perlu mengelolanya secara eksplisit dalam aplikasi level tinggi seperti di atas.

Latihan :

1. Perbaiki Program TCP Socket Programming

Tujuan Latihan:

Mengidentifikasi dan memperbaiki kesalahan syntax dan logika dalam implementasi sederhana TCP Socket Programming.

Instruksi:

Berikan solusi atau perbaikan untuk setiap program yang memiliki kesalahan syntax atau logika. Jelaskan alasan dari setiap perubahan yang Anda buat.

Program 1: Server Side (TCPReceiver.java)

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

public class TCPReceiver {
    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(12345);
            System.out.println("Server is listening on port 12345...");

            Socket clientSocket = serverSocket.accept();
            System.out.println("Client connected: " + clientSocket.getInetAddress());

            InputStream inputStream = clientSocket.getInputStream();
            InputStreamReader inputStreamReader = new InputStreamReader(inputStream);
            BufferedReader bufferedReader = new BufferedReader(inputStreamReader);

            String receivedData = bufferedReader.readLine();
            System.out.println("Received data: " + receivedData);

            bufferedReader.close();
            clientSocket.close();
            serverSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Program 2: Client Side (TCPSender.java)

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

public class TCPSender {
    public static void main(String[] args) {
        try {
            Socket socket = new Socket("localhost", 12345);

            OutputStream outputStream = socket.getOutputStream();
            OutputStreamWriter outputStreamWriter = new OutputStreamWriter(outputStream);
            BufferedWriter bufferedWriter = new BufferedWriter(outputStreamWriter);

            bufferedWriter.write("Hello, Server!");
            bufferedWriter.newLine();
            bufferedWriter.flush();

            bufferedWriter.close();
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Instruksi Tambahan:

1. Identifikasi kesalahan syntax atau logika di setiap program.
2. Koreksi setiap kesalahan dengan memberikan solusi yang benar.
3. Jelaskan alasan di balik setiap perubahan yang Anda buat.
4. Pastikan program dapat berjalan tanpa kesalahan setelah perbaikan.

Tugas Rumah :

Mengimplementasikan aplikasi sederhana menggunakan TCP untuk mentransfer file antara dua peer (komputer) dalam jaringan.

Petunjuk:

1. Implementasikan dua program terpisah: satu untuk peer pengirim (sender) dan satu untuk peer penerima (receiver).
2. Program sender harus dapat memilih file yang akan dikirim.
3. Program receiver harus dapat menerima file yang dikirim dan menyimpannya di lokasi yang ditentukan.
4. Pastikan untuk menangani situasi yang mungkin terjadi, seperti file tidak ditemukan atau kesalahan selama transfer.
5. Berikan feedback yang jelas kepada pengguna, misalnya, tampilkan kemajuan transfer.