Distributed Systems

Chapter 4: Designing of a Client/Server-Software

Change history

Date	Changes	Name
x.x.2022	Translation of slides 1 to 22	Khader Karkar
12.05.2024	Revision of slides 1 to 22	Bora Derici
20.06.2024	Translation of slides 28 to 103	Bora Derici

Translation assisted by ChatGPT-4

4.1 The Client/Server Model

The design of distributed applications today is often based on the so-called **client/server model**. In this interaction model, the units communicating with each other are divided into **clients** and **servers**. A server is a service provider that provides a specific functionality. The client is a service consumer who requests the execution of a specific service from a server. Because the client and server run independently of one another - often on different computer systems - communication between the two must be synchronized. Otherwise, data exchange may never occur between them.

Rendezvous problem

The problem addressed is also known as the rendezvous problem. If you start the client and server on different machines in rapid succession, the following processes can occur.

1. The client starts first. It tries to contact the corresponding server. If this has not yet been started, the client sees it as finished. The server is also started immediately after the client. If the server cannot find its communication partner (this has already been terminated), it also sees that it is terminated. No interaction can take place.
2. The server starts first. It tries (just like in the first case) to find his communication partner. Unfortunately, the client is not yet active, which is why the server is shut down. The client started immediately after the server cannot find its communication partner either and is therefore terminated.

The rendezvous problem arises because both sides are **actively** trying to find their communication partner. To solve the problem, only one of the two sides is allowed to actively establish communication. The other side must wait **passively** for a request to arrive. The definition of client and server introduced must be expanded to include this aspect.

Server: A server is a **service provider** that provides a specific functionality for a client in the form of a service and **passively** waits for a client to make a request to it.

Client: A client is a **service user** who **actively** requests a service from a server and then waits for the server to provide the requested service.

Servers are often more **complex** than clients. In addition to communicating with the client and providing the specific service, they also have to carry out other tasks:

- **Authentication:** The server must be able to uniquely identify the client requesting a service.
- **Authorization:** Once the identity of a client has been uniquely determined, it must be verified whether the Client is allowed to request the desired service.
- **Data protection:** Clients should not be allowed to freely access personal data.
- **Data security:** All information offered by a server should be protected against manipulation and destruction.
- **Protection of the operating system:** The server must prevent a client from accessing any resources of the operating system and potentially abusing them.

There are a number of advantages in favor of the wide distribution and use of the client/server model. The main advantages of the client/server model are:

- The client/server model can be used in **heterogeneous environments**. It therefore enables communication despite different computer and operating systems.
- The terms client and server have prevailed in theory and practice. These are **fixed terms** that enable clear communication among IT professionals.
- Because of the clear separation between service provider and service user, **the interaction between the two sides is clearly defined**.
- The client/server model can be used not only for software development but also for **hardware design**. For example, when developing systems for controlling production plants, one hardware component can act as a client (e.g. sensor that measures temperature) and another as a server (e.g. components for processing the measurement data)

The disadvantages of the client/server model are:

- Although the client/server model supports heterogeneous environments, it does not hide them from the programmer (**no transparency**). The programmer must consider heterogeneity when implementing distributed applications.
- In practice, there are often applications that do not allow for **a clear separation of client and server.** For example, a server can simultaneously act as a client for another server.
- The client/server model is **relatively old** and inadequate for many modern forms of communication. New communication models have to be developed here, such as remote method calls or message-oriented communication.

The messages exchanged between client and server are also generally referred to as **requests** and **responses**. After the server has started, it passively waits for a request to arrive. As soon as a request arrives, it provides the requested service and sends a response back to the client. The process described is shown in Figure 4.1 (according to [Tanenbaum 2003] p. 62).

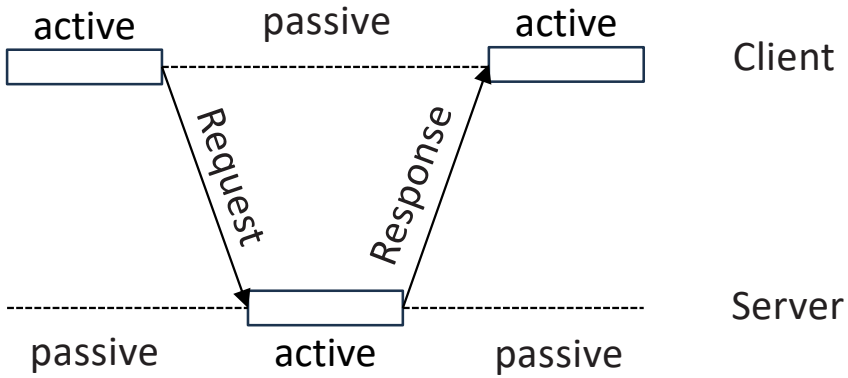


Figure 4.1: Structure of the client/server communication

4.1.1 Client Design

According to [Comer 2001], three different aspects must be distinguished when designing clients:

- parameterizable vs. non-parameterizable clients
- iterative vs. parallel clients
- connectionless vs. connection-oriented clients

Parameterizable vs. non-parameterizable clients

First of all, the question arises as to whether the client should be freely configurable or not. A freely configurable client makes it possible to select the server to be addressed via parameters - for example command line parameters - and to set other communication parameters. The main advantage of such a Client lies in the ability to **establish a connection to various Servers**. For instance, if a Server is unreachable due to network problems, the request can simply be sent to another Server. A client that cannot be freely parameterized always communicates with the same server. The user can only start the client but cannot provide any command-line parameters. Such clients offer a **high degree of security** because they only allow communication with a server classified as trustworthy. In practice, freely parameterizable Clients are often found because they necessitate connections to multiple Servers, selectable by the user.

If you implement a freely parameterizable client, the programmer must specify where the client gets its parameters from. There are several possibilities here:

- The client evaluates the transferred **command line parameters**.
- The user can pass the configuration parameters to the client via a GUI.
- The parameters are stored in a local **configuration file** that is read by the client.
- Relevant parameters are made available to the client via a **directory service** (e.g. Lightweight Directory Access Protocol (LDAP) / X.500).

Iterative vs. parallel clients

The second important design aspect is the question of an iterative or parallel implementation of clients. An iterative client has only one thread of control, i.e: it can only do one specific task at a time. A parallel client, on the other hand, has several execution fields and can therefore process several tasks (quasi) in parallel. A parallel Client is advantageous when the following tasks need to be fulfilled:

- The client should be able to interact with several servers at the same time. For example, a client can send a request to multiple servers at the same time. The response from the server that responds first is used, or the responses provided can be averaged for greater accuracy.
- During the client's communication with the server, the user wants to make further entries. The client must therefore be able to process the communication and the inputs (quasi) in parallel.

connectionless vs. connection-oriented clients

The third and final design aspect when implementing a client is the question of the transport protocol used. In today's distributed systems, the connection-oriented TCP (Transmission Control Protocol) or the connectionless UDP (User Datagram Protocol) are predominantly used as the transport protocol. Accordingly, a distinction is made between connectionless and connection-oriented clients. Whether a client works connectionless or connection-oriented depends very much on the server. The client must always use the same transport protocol as the server.

4.1.2 Design of Servers

When designing Servers, there are similar issues to be discussed as when designing clients. The most important design aspect is whether the server should work iteratively or in parallel.

Iterative server:

An iterative server processes incoming requests in the order in which they arrive in a strictly **sequential** manner, according to the **first-come-first-served (FCFS)** principle, which is also known from operating systems. With an iterative server, requests that require a relatively short period of time to be processed may have to wait for requests that take a long time to process. From a user's point of view, such a way of working is unsatisfactory. Intuitively (and rightly so), every user expects a "small" request to be processed quickly.

Parallel server:

A parallel server processes multiple requests **quasi-parallel**, either using multiple processors (parallel) or a **round-robin** approach (single processor). Implementing a parallel server requires the operating system to support parallelism. Common methods include using multiple processes or multiple threads. Other techniques, such as asynchronous I/O, can also achieve parallelism.

Response Time and Processing Time:

The decision to implement either an iterative or parallel server primarily depends on the client's response time and the server's processing time.

- Processing Time: The time a server needs to handle a request, influenced by the complexity of the request.
- Response Time: The total time from the client's perspective, including processing time, transmission time over the network, and waiting time in server buffers.

Response time is always greater than processing time. An iterative server handles requests sequentially, causing other requests to wait in a queue, which increases response time. If the response time becomes unacceptable, switching from an iterative to a parallel server is advisable.

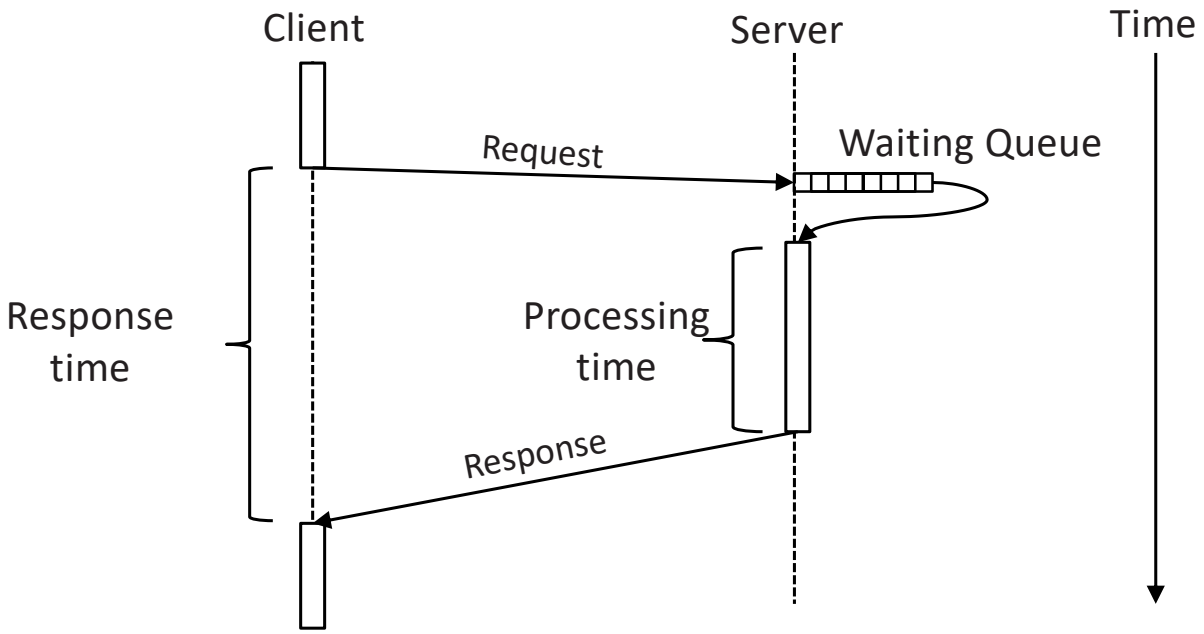


Figure 4.2: Response and processing timeline

Iterative Server and Parallel Server

- **Iterative Server Design**: Requests are processed sequentially. If three requests arrive simultaneously, each subsequent request must wait for the previous ones to complete. For example, if Request 1 (R1) takes t time units, Request 2 (R2) takes 2t time units, and Request 3 (R3) takes 3t time units, the total response time for R3 is 6t time units. The average response time across the three requests is 3.33t time units.

- **Parallel Server Design**: Requests are processed concurrently. Each request is handled as soon as it arrives, so the response times are just the processing times of the individual requests: t for R1, 2t for R2, and 3t for R3. The average response time here is reduced to 2t time units.

The parallel server design significantly improves both individual and average response times. However, the improved response time is offset by the server's increased overhead due to the management and creation of parallel processes.

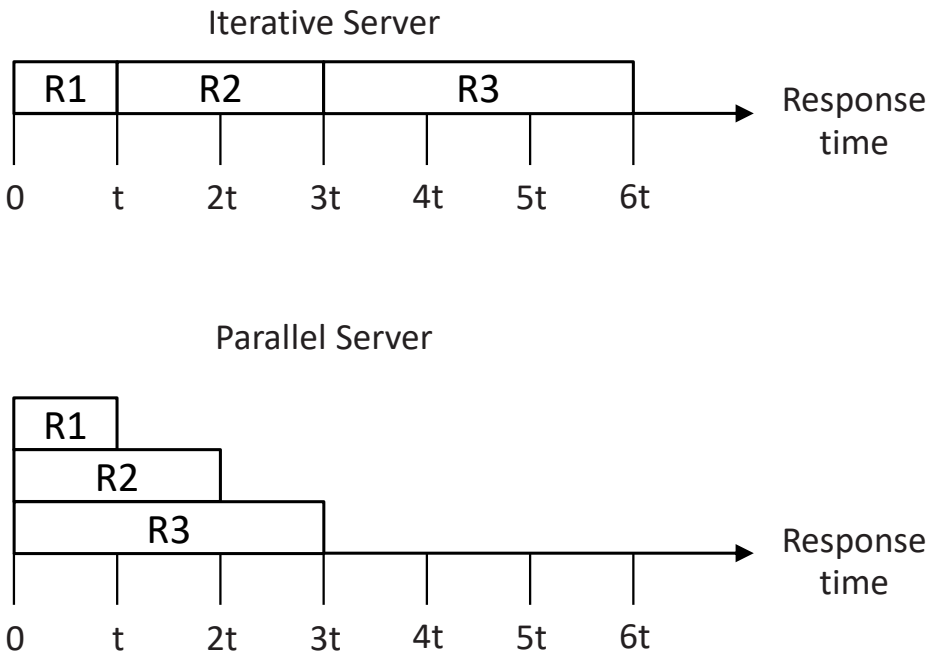


Figure 4.3: Response time with iterative and parallel server

- Unfortunately, the described time advantage only exists on a multiprocessor system. On a uniprocessor system, parallelism is implemented through a time-sharing method. If the length of a time slice is t time units, the sequence in Figure 4.4 can occur.

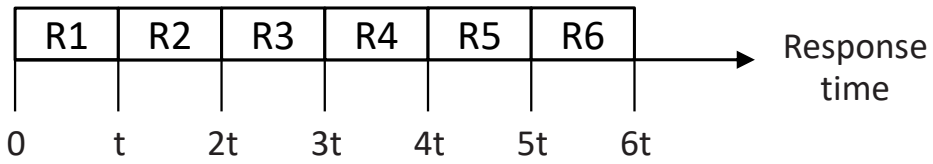


Figure 4.4: Parallel server over time slices

In a multiprocessor system, parallel processing can significantly reduce response times. However, on a uniprocessor system using time-sharing, response times increase with each additional request—R1's response time is t, R2's is 4t, and R3's is 6t, averaging 3.67t. This increase in response time with more requests can potentially lead to server overload, highlighting the limitations of time-sharing on uniprocessor systems.

Master/Slave principle:

Many parallel servers are implemented using the Master/Slave principle (Figure 4.5). The process involves:

1. The Master waits in an endless loop for requests at a specific port.
2. Upon receiving a request, the Master creates a Slave to handle the request and then goes back to waiting for new requests.
3. The Slave processes the received request and communicates with the client using its own port.

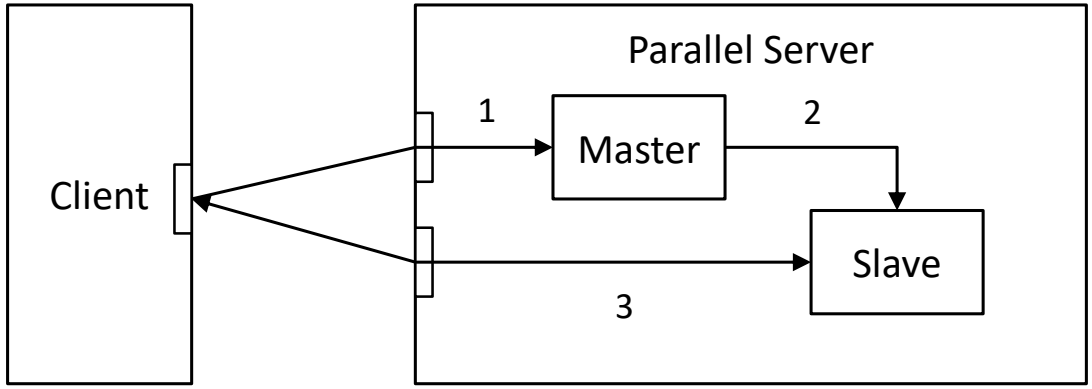


Figure 4.5:Master/Slave principle

Connectionless vs Connection-Oriented Protocols:

- **Connectionless Protocols (UDP)**: Servers using these are efficient but require developers to manually ensure data reliability and order.
- **Connection-Oriented Protocols (TCP)**: These servers provide built-in reliability through the TCP/IP stack, ensuring data integrity and order, but are less efficient due to the overhead of maintaining connections.

Developers need to choose between **efficiency** and **reliability**.

Stateless and Stateful servers:

- **Stateless Servers**: These servers do not retain any information about previous interactions with clients. Each request is processed independently, which **simplifies implementation** and **enhances robustness** against failures because there’s no need to restore any previous state after a server crash.

- **Stateful Servers**: Unlike stateless servers, stateful servers track the history of client interactions, affecting how they respond to new requests. This setup is similar to a finite state machine. In case of failures, it's crucial to restore the last known state to prevent issues in client-server communication.

1) **Connectionless vs Connection-oriented**
2) **Iterative vs Parallel**
3) **Stateless vs Stateful**

Combining these attributes in different ways yields $2^3 = 8$ possible server types, providing a systematic framework for understanding server architecture. This highlights the range of options available when configuring servers to meet specific operational needs or architectural preferences.

Multiprotocol-Server

- **Supports Multiple Protocols**: Multi-protocol servers handle multiple transport protocols simultaneously, such as UDP and TCP.
- **Code Efficiency**: They allow for code reuse across different protocols, reducing the need to duplicate similar functionalities in separate programs.
- **Resource Conservation**: By running multiple protocols through a single server process, these servers save on system resources like process table entries and main memory.

Multiservice-Server

Offers multiple services simultaneously for a specific transport protocol, such as providing both TIME and DAYTIME services.

Resource Efficiency: Shares the advantages of a Multi-protocol Server by requiring only one server process to deliver multiple services, thereby conserving system resources.

Super-Server

Provides several services at once across different transport protocols, improving service availability and expanding protocol support.

- **Resource Efficiency**: Only one process is needed to offer all services, significantly conserving system resources.
- **Dynamic Configuration**: Can be dynamically configured by a system administrator, allowing new services to be added or existing ones to be terminated without needing a server restart. An example is the inetd Super-Server used in many Linux systems, which is configurable via a file.

4.2 Implementation of the client/server model over sockets

The Socket API, essential for implementing the Client/Server model, was introduced in 1982 by the University of California, Berkeley, as part of BSD UNIX 4.1c. It facilitates easy access to protocol stack functions necessary for sending and receiving data, aligning with UNIX's principle that "everything is a file". In network communications, a socket serves as an endpoint using the TCP/IP or UDP/IP stack, comprising an IP address and a port number. Effective data exchange between two systems requires two connected sockets, which together form a **communication channel or pipe.**

4.2.1 Overview of Socket Primitives

The original Socket API, implemented in C, includes essential functions designed to establish, manage, and terminate network communications between clients and servers. These functions are detailed to enhance understanding of their specific roles and how they facilitate effective network interaction.

socket() function initializes a new communication endpoint suitable for TCP or UDP protocols. This newly created socket is generic and can be adapted for use by either a client or a server, with its specific role defined by subsequent function calls.

bind() function associates a socket with a specific local address, typically comprising an IP address and a port number.

listen() function is integral to connection-oriented communication, as it configures a socket to operate in a **passive mode**, making it ready for server use. It also establishes the length of the connection queue, which holds incoming connection requests when the server is otherwise occupied. This is particularly crucial for iterative servers, which cannot handle new connections while engaged in communication with a client.

accept() function is used by servers in connection-oriented communication to wait for incoming connection requests. It operates by **blocking** server operations until a client initiates a connection. Once a connection request is made, accept() provides a new socket dedicated to handling the communication with that client.

connect() function is used by clients to initiate a connection to a remote server. This connection is established through a three-way handshake, ensuring a reliable link before data transfer begins.

read()/write() : [data read and write]

the read() function is used to receive data from a socket, while the write() function is responsible for sending data through a socket. These functions are fundamental for data exchange between connected devices.

sendto()/recvfrom() : [sending and reciving datagram]

In connectionless communication, specifically with UDP, the functions sendto() and recvfrom() serve similar roles to read() and write() in connection-oriented contexts. sendto() is used to send UDP datagrams, while recvfrom() is used to receive the next UDP datagram from the protocol stack, facilitating data exchange without a persistent connection.

close()

After the communication between client and server has ended, the reserved sockets can be released using close().

The Socket API requires correct function call sequencing and differentiates between connectionless and connection-oriented communications. Figure 4.6 demonstrates this sequence for connectionless communication, with dashed arrows for data exchanges and solid arrows marking the function call order.

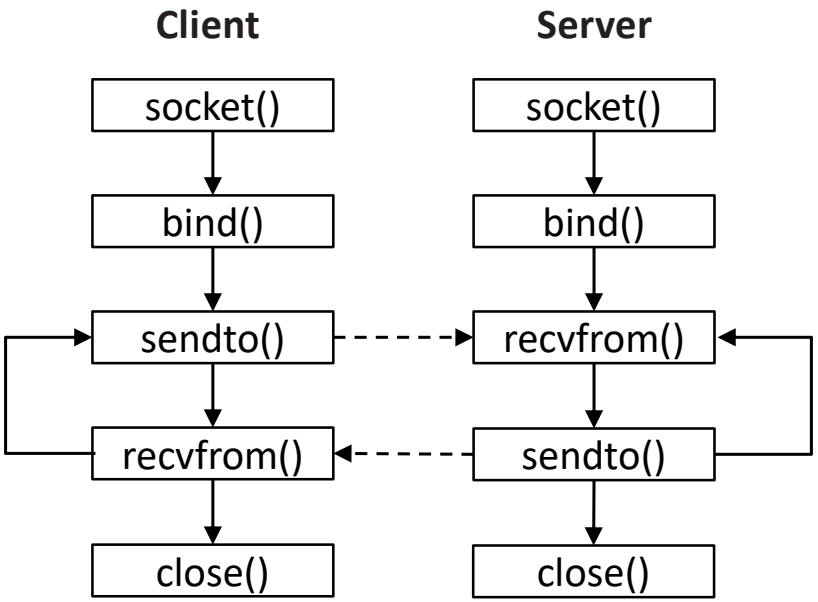
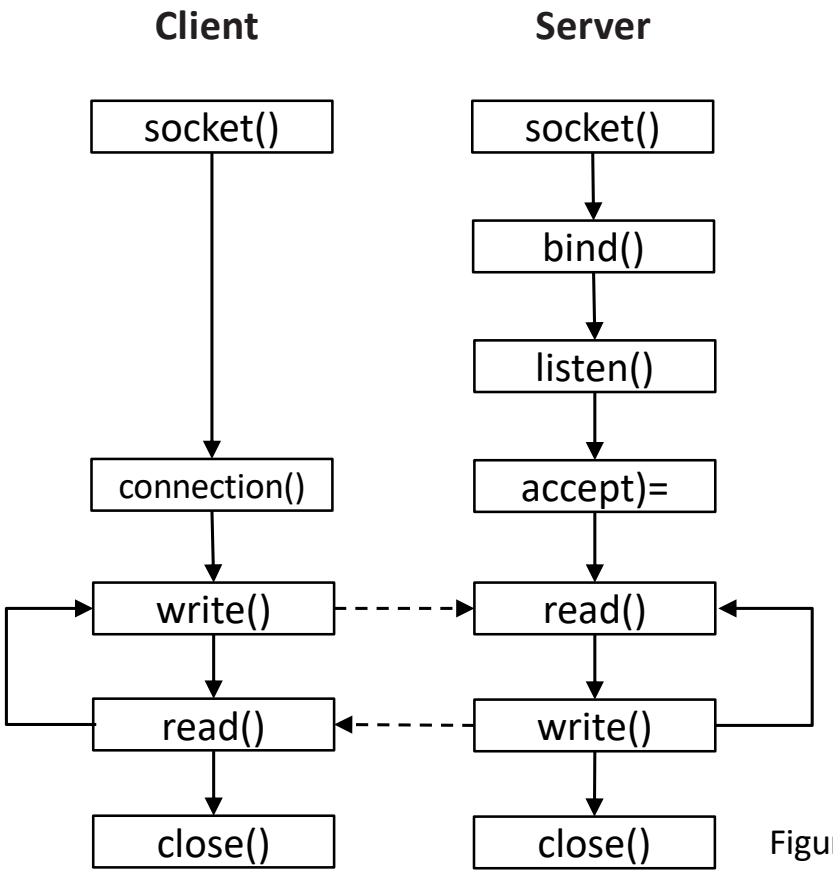


Figure 4.6: Connectionless communication over sockets

- Both client and server create a socket and bind it to a local address.
- The server waits for incoming UDP diagrams using the recvfrom() function, which blocks it until data arrives.
- The client sends a datagram using sendto() , unblocks the server when the datagram is received.
- Upon receiving the datagram, the server processes it and sends a response back to the client using sendto().
- The client waits server's response.
- The process can repat multiple times to exchange all necessary data.
- When the communication is complete, both server and client call close() to release their sockets.



The client and server start by creating sockets. The server binds its socket, listens, and waits in a blocked state on accept() waiting for a client connection. Once connected through a connect() and a three-way handshake, data exchange begins. The client sends data using write(), which the server receives and processes via read(). The server responds similarly, and the client processes the received response. This data exchange cycle continues until all information is transferred. Both parties then close their sockets to end the communication.

Figure 4.7: connection-oriented communication over sockets

4.2.2 Socket API in JAVA

The Java Socket API is implemented within the **java.net package**, which contains 27 classes, 6 interfaces, and 11 exceptions. Although the package appears extensive, only a subset of its functionality is typically required for developing client/server applications. The critical functions for these applications are

- Address- and Name resolution
  1. InetAddress
  2. Inet4Address
  3. Inet6Address

- UDP (User Datagram Protocol) - communication
  1. DatagramPacket
  2. DatagramSocket

- TCP (Transmission Control Protocol) – communication
  1. ServerSocket
  2. Socket

In Java, besides network communication classes, classes for data input and output are crucial. Input and output are handled through data streams, which are a key component in handling data efficiently in Java applications.

- DataInputStream
- DataOutputStream
- BufferedReader
- PrintWriter

InetAddress, Inet4Address and Inet6Address

The InetAddress class in Java serves as a general representation for IP addresses. It has two subclasses Inet4Address for IPv4 addresses and Inet6Address for IPv6 addresses. These classes enable the mapping of hostnames to IP addresses and vice versa. Instead of being directly instantiated, instances of these classes are obtained through a static get() method. The hierarchy and its functions are shown in figure 4.8

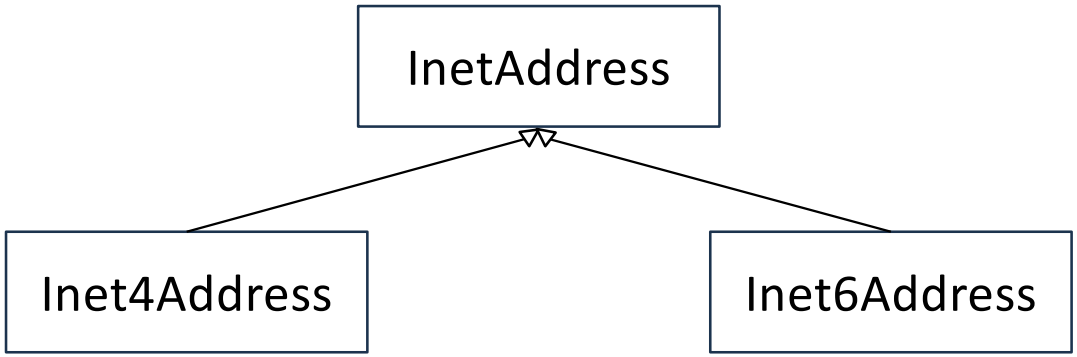


Figure 4.8: Hierarchy of Address class

The InetAddress class in Java includes two static methods for converting hostnames or URLs into IP addresses: getByName(String host) and getAllByName(String host). getByName() method returns the primary IP address for a given hostname, whereas getAllByName() provides an array of InetAddress objects representing all IP addresses of the host. If a hostname cannot be resolved, an UnknownHostException is thrown.

Program 4.1 demonstrates this functionality by showing the resolution of the URL www.fh-fulda.de.

```
InetAddress ipAddr = null;
try{
   //Name resolution
   ipAddr = InetAddress.getByName("www.fh-fulda.de");
}catch(UnknownHostException e){
   e.printStackTrace();
   System.exit(-1);
}
```

Programm 4.1: Resolution of a Hostname

Implementation task. Modify program 4.1 so that all IP addresses of a specific computer are determined. Check whether the addresses are IPv4 or IPv6 addresses.

DatagramPacket and DatagramSocket

Java.net package includes the DatagramSocket and DatagramPacket classes for connectionless communication over networks. The DatagramSocket class is used for UDP socket operations, and DatagramPacket is used for handling UDP packets. These classes are essential for implementing UDP-based communication in Java applications. To create a DatagramSocket, various constructors can be used. The three most important constructors are:

- DatagramSocket()
- DatagramSocket(int port)
- DatagramSocket(int port, InetAddress laddr)

The DatagramSocket class in Java provides various constructors to suit different networking needs. The default constructor creates a UDP socket and binds it to any available port, ideal for clients who let the operating system handle port selection. For defining a specific port, the DatagramSocket(int port) constructor is used, which is particularly useful for servers that require a fixed port. Additionally, the DatagramSocket(int port, InetAddress laddr) constructor allows setting both the port and a specific IP address, offering precise control over the socket's communication endpoints for clients or servers.

Sending and receiving UDP packets:

The DatagramSocket class in Java includes two key methods for UDP communication: send() and receive(). The send() method is used to dispatch a UDP datagram to the specified address within the datagram itself, while the receive() method blocks the operation until the next UDP datagram arrives, ensuring synchronous communication. These methods enable efficient handling of UDP data transfers via sockets.

UDP-Socket release:

The close() method is used to release a UDP socket after it is no longer needed. If any threads are actively waiting for a datagram on this socket at the time close() is invoked, they will be interrupted and a SocketException will be thrown.

Constructors of DatagramPacket:

The DatagramPacket class in Java represents a UDP packet, combining a data buffer with a specified destination address in one abstraction. To create a UDP packet, the following constructors are available:

- DatagramPacket(byte[] buf, int length)
- DatagramPacket(byte[] buf, int length, InetAddress address, int port)

The DatagramPacket class offers two constructors for creating UDP packets. The first constructor reserves only a data buffer, requiring subsequent setting of the destination IP address and port using the setAddress() and setPort() methods. The second constructor allows immediate specification of both the destination address and port during packet instantiation, enabling the packet to be sent without further configuration.

Read and write the content of a datagram:

The DatagramPacket class provides getData() and setData() methods to read and write the content of UDP packets, which fundamentally consist of byte arrays. To send data such as strings via UDP, the data must first be converted into a byte array using the getBytes() method before it can be transmitted.

Socket and ServerSocket

The Socket and ServerSocket classes in Java are fundamental for creating connection-oriented client/server applications. The Socket class acts as a communication endpoint, enabling the sending and receiving of data. The most important constructors of the Socket class are:

- Socket()
- Socket(String host, int port)

The Socket class provides a default constructor that creates a socket without connecting it. To establish a connection, you must use the connect() method, supplying it with a SocketAddress that includes the remote host's IP address and port number. Alternatively, a simpler method involves using a second constructor that directly accepts the hostname and port number for immediate connection setup. Errors in socket creation or connection attempts are handled by throwing an IOException if a general error occurs, or an UnknownHostException if the remote host's address cannot be resolved.

Input and output via sockets:

Once the socket has been successfully instantiated and connected to the remote socket, data streams for reading and writing data can be obtained using getInputStream() and getOutputStream().

Close TCP-socket:

After completing communication via a socket, you can close and release it using the close() method. If any threads are still blocked waiting for data when the socket is closed, they will throw a SocketException. This ensures that all resources associated with the socket are properly cleaned up.

Constructors of ServerSocket:

The ServerSocket class in Java is used by servers to listen for and accept incoming connections. It features specific constructors essential for setting up these listening sockets. The two most important constructors of ServerSocket are:

- ServerSocket(int port)
- ServerSocket(int port, int backlog)

The ServerSocket(int port) constructor creates a passive socket that binds to a specified port and sets up a connection queue capable of holding up to 50 incoming connection requests.

The ServerSocket(int port, int backlog) constructor offers flexibility to specify a different length for this connection queue, allowing for customization based on server load and requirements.

Waiting for connections:

After creating a ServerSocket, accept() method is used to block and wait for connection requests. You can specify a maximum waiting time with the setSoTimeout(int timeout) method. If no connection request is received within this timeframe, a SocketTimeoutException is thrown. Successful connection attempts return a Socket object through accept(), which can then be used for data exchange.

To shut down a ServerSocket, close() method should be used. This will prompt any threads that are blocked on the accept() method to throw a SocketException, thereby stopping their activities.

4.3 Case Studies

The practical application of client/server theories and the Java Socket API will be explored through detailed examples involving three different services. Each service is thoroughly described, from its general concept and functionality to design considerations and specific implementation.

4.3.1 DAYTIME

The DAYTIME service, designated to port 13 for both UDP and TCP, provides the current system time as a human-readable string.

In UDP mode:

- The server waits for a datagram
- Ignores the content of incoming datagrams
- Responds with the system time.

In TCP mode:

- The server waits for a connection request
- Sends the time after establishing a connection and then closes the connection.

The time format is recommended by RFC 867 to be in the order of weekday, month, day of the month, year, and time-zone (e.g., "Sun. Aug 29, 2004 16:41:32-CEST").

Design Aspects

- The DAYTIME service requires minimal computational effort, resulting in short processing times. Therefore, an iterative implementation of the server is sufficient.
- DAYTIME communication consist only of a request and a response, the server can operate stateless.
- Both UDP and TCP are suitable as transport protocols according to RFC 867.

An example of implementing a connectionless DAYTIME client and server will be on the following slide.

```
import java.net.*;
import java.io.*;
import java.text.*;
import java.util.*;
public class DaytimeServer_v1 {
   // 1KB buffer size for datagrams
   private static final int BUF_SIZE = 1024;
   private int port; // local port
   private DatagramSocket udpSocket;
   private DatagramPacket requestPacket; // Request
   private DatagramPacket responsePacket; // Response
   private SimpleDateFormat formatter; // Time format
   private String currentTime; // Current server time
```

```
// Constructor
public DaytimeServer_v1(int port) {
   this.port = port;
   formatter = new SimpleDateFormat("E, MMM d, yyyy HH:mm:ss-z");
}

// In an infinite loop: wait for incoming datagram, determine time, send time back to client.
public void startServer() {
   try {

      // Open UDP socket for communication
      udpSocket = new DatagramSocket(port);
      System.out.println("Waiting for UDP packet on port " + port + " ... ");
      requestPacket = new DatagramPacket(new byte[BUF_SIZE], BUF_SIZE);
      responsePacket = new DatagramPacket(new byte[BUF_SIZE], BUF_SIZE);
```

```
        while (true) {
            udpSocket.receive(requestPacket);
            // Get current system time and store in response
            currentTime = formatter.format(new Date());
            responsePacket.setData(currentTime.getBytes());
            // Set destination address and port of client
            responsePacket.setAddress(requestPacket.getAddress());
            responsePacket.setPort(requestPacket.getPort());
            // Send time to client
            udpSocket.send(responsePacket);
            // Status message
            System.out.println(" [" + currentTime + " sent to " + requestPacket.getAddress().getHostName() + "]");
        }
    } catch (IOException e) {
        e.printStackTrace();
        System.exit(-1);
    }
}
```

```
// Main
   public static void main(String[] args) {
      switch (args.length) {
         case 1:
            try {
               (new DaytimeServer_v1(Integer.parseInt(args[0]))).startServer();
            } catch (NumberFormatException e) {
               e.printStackTrace();
               System.exit(-1);
            }
            break;
         default:
            System.err.println("\nSyntax: java DaytimeServer_v1 <Port>\n");
            System.exit(-1);
            break;
      }}}
```

Program 4.2: Connectionless and Iterative DAYTIME Server

- The DAYTIME server expects the local port number as a parameter.
- The main() method creates a new server object if necessary and calls startServer().
- A new UDP socket is created for the designated port with the statement ”udpSocket = new DatagramSocket(port);”
- The DAYTIME service is implemented in the infinite loop.

```
while (true) {
   requestPacket = new DatagramPacket(new byte[BUF_SIZE], BUF_SIZE);
   responsePacket = new DatagramPacket(new byte[BUF_SIZE], BUF_SIZE);
   udpSocket.receive(requestPacket);
   // Get current system time and store in response
   currentTime = formatter.format(new Date());
   responsePacket.setData(currentTime.getBytes());
   // Set destination address and port of client
   responsePacket.setAddress(requestPacket.getAddress());
   responsePacket.setPort(requestPacket.getPort());
   // Send time to client
   udpSocket.send(responsePacket);
   // Status message
   System.out.println(" [" + currentTime + " sent to " + requestPacket.getAddress().getHostName() + "]");}
```

- Two UDP datagrams are instantiated to store the incoming request and the outgoing response, in this example each capable of holding 1 kilobyte (BUF_SIZE bytes).
- The server uses the receive() method to block and wait for an incoming datagram.
- After receiving a datagram, the server determines the current system time by creating a new Date object. Then the system time is converted into human readable string using SimpleDateFormat method.
- The formatted time string is stored in the response datagram and the datagram is configured with the requester's IP address and port number.
- The server response back to the client using the send() method.

```
import java.io.*;
import java.net.*;
public class DaytimeClient_v1 {
   // 1KB buffer size for datagrams
   private static final int BUF_SIZE = 1024;
   private int port; // Server's port number
   private String host; // Server's hostname
   private DatagramSocket udpSocket; // UDP socket for communication
   private InetAddress ipAddress; // Server's IP address
   private DatagramPacket request; // Datagram for sending request
   private DatagramPacket response; // Datagram for receiving response
   private String serverTime; // Server's system time received

   // Constructor
   public DaytimeClient_v1(String host, int port) {
      this.host = host;
      this.port = port;
   }
```

```
// Method to start the client
public void startClient() {
   try {
      udpSocket = new DatagramSocket();
      // Determine server's IP address
      ipAddress = InetAddress.getByName(host);
      // Create and send an empty datagram to the server
      request = new DatagramPacket(new byte[BUF_SIZE], BUF_SIZE, ipAddress, port);
      udpSocket.send(request);
      // Create datagram for response and wait for the reply
      response = new DatagramPacket(new byte[BUF_SIZE], BUF_SIZE);
      udpSocket.receive(response);
      // Convert response to String
      serverTime = new String(response.getData());
      System.out.println("Received server time: " + serverTime);
      // Release resources
      udpSocket.close();
   } catch (IOException e) { e.printStackTrace(); System.exit(-1); }}
```

```
// Main
   public static void main(String[] args) {
      switch (args.length) {
         case 2:
            try {
               (new DaytimeClient_v1(args[0], Integer.parseInt(args[1]))).startClient();
            } catch (NumberFormatException e) {
               e.printStackTrace();
               System.exit(-1);
            }
            break;
         default:
            System.err.println("\nSyntax: java DaytimeClient_v1 <Host> <Port>\n");
            System.exit(-1);
            break;
      }
   }}
```

Program 4.3: Connectionless and Iterative DAYTIME Client

- The DAYTIME client needs two parameters: the server’s host name and port number.
- main() method checks for these parameters and creates a new DAYTIME client if they are correctly provided.
- Unbound UDP socket is created by the client for communication with the server and determines the IP address of the provided hostname using the static method InetAddress.getByName().
- The client creates and configures a request datagram with the server's IP address and port and sends the datagram to the server using send() method.
- Before waiting for the server's response, the client creates a new datagram to store the received response.
- The client uses the receive() method to wait for the response from the server. This operation blocks until the response datagram from the server is received.
- Once the response datagram is received, its contents are converted into a string. This string is then displayed on the console.
- After completing its task, the client closes the UDP socket and terminates its execution.

4.3.2 ECHO

The ECHO service is defined in RFC 862 [IETF/RFC] and is used for debugging and performance measurement. We can distinguish between the connectionless and connection-oriented ECHO services based on their handling of network traffic.

Connection-Oriented ECHO Server:

- Waits on port 7 for incoming connections.
- Sends back all received data unchanged to the client until the client closes the connection.

Connectionless ECHO Server:

- Also listens on port 7, but for incoming datagrams.
- Immediately returns the content of any received datagram in a new datagram.

Design Aspects

- The ECHO server only returns the received request, it does not need to store any client information.
- To support multiple clients simultaneously, a parallel implementation of the server is used.
- The parallel structure prevents delays caused by clients sending large data loads from impacting those sending less data.
- A connection-oriented, parallel, and stateless design is chosen for the ECHO server

On the next slide we do an example.

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

public class EchoServer_v1 {
   private int port; // Local port
   private ServerSocket serverSocket; // Passive socket
   private Socket socket; // Socket for connection

   // Constructor
   public EchoServer_v1(int port) {
      this.port = port;
   }
```

```
// Method to start the server
public void startServer() {
   try {
      // Create a passive socket and bind to local port
      serverSocket = new ServerSocket(port);
      while (true) {
         System.out.println("Waiting for connection on port " + port + "...");
         socket = serverSocket.accept();
         System.out.println("Connection request from " + socket.getInetAddress().getHostName() + " - starting
EchoThread");
         // Start a new thread to handle the connection
         (new Thread(new EchoServer_v1.EchoThread(socket))).start();
      }
   } catch (IOException e) {
      e.printStackTrace();
      System.exit(-1);
   }
}
```

```
// Inner class to perform ECHO service
private class EchoThread implements Runnable {
   private Socket socket; // Socket for connection
   private InputStream in; // Reading from socket
   private OutputStream out; // Writing to socket


   // Constructor
   public EchoThread(Socket socket) {
      this.socket = socket;
   }
```

```
// Run method to read and send data byte-by-byte

public void run() {
   try {
      in = socket.getInputStream();
      out = socket.getOutputStream();
      int data;
      while ((data = in.read()) != -1) {
         out.write(data);
         out.flush();
      }
      in.close();
      out.close();
      socket.close();
   } catch (IOException e) {
      e.printStackTrace();
   }
}}
```

```
// Main
   public static void main(String[] args) {
      switch(args.length){
         case 1:
            try{
               (new EchoServer_v1(Integer.parseInt(args[0]))).startServer();
            }catch (NumberFormatException e){
               e.printStackTrace();
               System.exit(-1);
            }
            break;
         default:
            System.err.println("\nSyntax: java EchoServer_v1 <Port>\n");
            System.exit(-1);
            break;
      }}}
```

Program 4.4: Connection-oriented, parallel ECHO server

- The main() method serves as the program's entry point, checking for exactly one command-line argument. It terminates the program if the number of arguments is incorrect.
- Upon receiving the correct port number, the EchoServer_v1 class is initialized and the server is started using startServer() method.
- First, a ServerSocket object is created and bound to the provide port. In an infinite loop, the server waits for new connection requests using accept().
- For each incoming connection, a new thread is started. This thread runs an instance of the EchoThread inner class, which handles the echo functionality.
- Data transmissions are handled by InputStream and OutputStream from the connected socket, echoing received data back to the client.

The actual ECHO service is realized through the following loop:

```
int data;
while((data = in.read()) != -1){
   out.write(data);
   out.flush();
}
```

Program 4.5: Core of the ECHO Server

- The read() method reads bytes from the input stream one at a time, returning each as an integer value. It returns -1 if the input stream is closed or no more data is available.
- Bytes read from the input stream are written to the output stream using the write() method. To ensure the data is sent over the network, the flush() method is called after writing.
- If the client closes the connection, read() returns -1, causing the termination of the while-loop. Subsequently, the EchoThread closes the input and output streams and the socket using close() method.
- After closing all streams and the socket, the EchoThread completes all operations and the thread ends.

The functionality of the ECHO client is demonstrated in Program 4.6.

```
import java.net.*;
import java.io.*;
public class EchoClient_v1 {
   private String host; // Hostname of the ECHO server
   private int port; // Port number of the ECHO server

   // Constructor
   public EchoClient_v1(String host, int port) {
      this.host = host;
      this.port = port;
   }
```

```
// Method to start the client
public void startClient() {
   try {
      System.out.print("Connecting to " + host + ":" + port + "...");
      Socket socket = new Socket(host, port);
      System.out.println("Connection established");
      OutputStream out = socket.getOutputStream();
      InputStream in = socket.getInputStream();
      int data;
      // Read data from keyboard, send to ECHO server, receive response, and print it
      while ((data = System.in.read()) != -1) {
         out.write((byte) data);
         out.flush();
         System.out.print((char) in.read());  }
      in.close(); // Close all streams and socket
      out.close();
      socket.close();
   } catch (IOException e) {
      e.printStackTrace();
      System.exit(-1);
   }
}
```

```
// Main
   public static void main(String[] args) {
      switch (args.length){
         case 2:
            try {
               (new EchoClient_v1(args[0], Integer.parseInt(args[1]))).startClient();
            } catch (NumberFormatException e) {
               e.printStackTrace();
               System.exit(-1);
            }

      break;
      default:
         System.err.println("\nSyntax: java EchoClient_v1 <Host> <Port>\n");
         System.exit(-1);
      }}}
```

Program 4.6: Connection-oriented, parallel ECHO Client

- The main() method verifies the command line arguments for the host name and port number of the ECHO server.
- If parameters are valid, an EchoClient_v1 object is created then its startClient() method is called to establish a connection with the ECHO server.
- Once connected, getInputStream() and getOutputStream() are used to handle data transmission.
- Inputs from the keyboard are read and sent to the remote ECHO server, the server's responses are displayed on the console.

The described process is implemented by the following loop:

```
int data;
while((data = System.in.read()) != -1){
   out.write((byte) data);
   out.flush ();
   System.out.print((char) in.read());
}
```

Program 4.7: Core of the ECHO Client

The input stream System.in, which is always open in Java, can be used to read something from the keyboard. System.in allows reading the next byte of data stored in the keyboard buffer using the read() method. As long as data is available, it is sent byte by byte to the ECHO server and the server's response is read byte by byte.

ECHO client and ECHO server has two major disadvantages (see tasks at the end of the chapter):

1. In the ECHO client and server, data is always read and written byte by byte. This leads to many unnecessary input/output operations, especially since the input and output streams also offer methods to read multiple bytes at once.
2. The ECHO client does not provide users with a way to terminate the program in a defined manner. All inputs are interpreted as data. A special input to terminate the client would be desirable.
3. The ECHO server allows access by an unlimited number of ECHO clients. This makes a denial-of-service attack possible.

4.3.3 FILE

- The FILE service is not described in any RFC, was specially developed for this example to showcase a simple yet effective client/server model.
- Designed to be connection-oriented, parallel, and stateful.
- Both FILE client and server offer basic functionality, focusing on demonstrating core principles rather than extensive features.
- The service enables file operations over a network, where the FILE client can read and write to a file.
- New entries added by the client are appended to the end of the file, regardless of the client's current read position.
- Reading and writing operations are performed independently, which is also illustrated in a specific diagram (Figure 4.9).

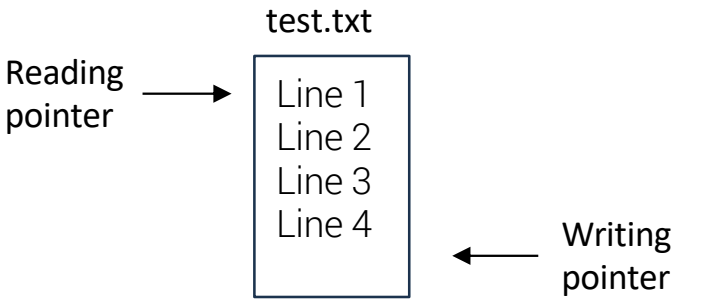


Figure 4.9: Reading and writing a file

If the client wants to read, it always starts at line 1 and successively reads all the lines. If it wants to write new data, these are always appended at the end of the file.

- Only strings are exchanged between the client and server to enable simple communication.
- The client sends a command to the server and the server evaluates the command, performs the requested operation, and returns the result.
- The server recognizes five different commands, which are detailed in Table 4.1.

Command	Meaning
OPEN filename	Opens the file with the name filename.
READ	Reads the next line from the previously opened file
WRITE line	Writes a new line to the end of the file.
CLOSE	Closes the previously opened file.
SHUTDOWN	The client ends communication with the server.

Table 4.1: Server commands

Status of the server

The server can be in **two** different states:

1. (FILE_NOT_OPEN) indicates no active file, either no file has been opened by the client yet or any previously opened file has been closed.
2. The server currently has a file open for reading and writing (FILE_OPEN).

The server's response to a read or write command depends on these two states. If no file is open, read and write operations will result in an error message. Otherwise, reading is performed from the open file.

FileServer

The FILE server's functionality is encapsulated in the FileServer class, which includes an inner class named FileThread. The main tasks of FileServer are to block and wait for new connection requests, and upon receiving one, it creates a FileThread to manage the connection. The FileThread is responsible for processing commands and executing corresponding file operations. The structure and flow of these operations are initiated in the main() method, detailed in Program 4.8.

```
// Entry point
public static void main(String[] args) {
   switch (args.length) {
      case 2:
         try {
            (new FileServer(Integer.parseInt(args[0]), args[1])).startServer();
         } catch (NumberFormatException e) {
            System.err.println(e.toString());
            System.exit(-1);
         }
         break;
         default:
         System.err.println("\nSyntax: java FileServer <Port> <Path>\n");
         System.exit(-1);
         break;
   }
}
```

Program 4.8: main() method of FileServer

The main() method of the FileServer application requires two user-provided arguments: a port number and a path to a local **working directory** where files to be accessed by the client and server are stored. If the user inputs an incorrect number of arguments, the program will display an error message and terminate. If the arguments are valid, the program creates a new FileServer object and calls its startServer() method, which is detailed in Program 4.9.

```
public void startServer() {
   ServerSocket serverSocket = null; // passive socket
   Socket socket = null; // Socket for connection
   try {
      serverSocket = new ServerSocket(port); // Open server socket on specified port
      while (true) {
         System.out.println("Waiting for connection on port " + port + "...");
         socket = serverSocket.accept(); // Accept incoming connections
         System.out.println("Connection request from " + socket.getInetAddress().getHostName() + " - starting FileThread");
         // Start a new thread for handling file operations
         (new Thread(new FileServer.FileThread(socket, path))).start();
      }} catch (IOException e) {
      System.err.println(e.toString());
      System.exit(-1); }}
```

Program 4.9: startServer() method of FileServer

- A passive socket is established on a specified port.
- The server enters an infinite loop where it uses accept() to block and wait for incoming connections.
- Once accept() secures a connection and returns a new socket, a FileThread is initiated to handle this connection.
- The FileThread is given the new socket and the path to the working directory.
- The details of the FileThread's operations are defined in its run() method, as shown in Program 4.10.

```
public void run() {
   boolean running = true; // Flag to end the thread
   try {
      reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));
      writer = new PrintWriter(socket.getOutputStream(), true);
      // Read and evaluate commands
      while (running) {
         String command = reader.readLine();
         // Response to commands
         if (command.startsWith("OPEN")) {
            open(command);
            continue;
         }
         if (command.equals("CLOSE")) {
            close();
            continue;
         }
```

The program continues on the following slide

```
        if (command.equals("READ")) {
            read();
            continue;
        }
        if (command.startsWith("WRITE")) {
            write(command);
            continue;
        }
        if (command.equals("SHUTDOWN")) {
            shutdown();
            running = false;
            continue;
        }
        // Response to an invalid command
        writer.println("Invalid command: " + command);
    }
} catch (IOException e) {
    System.err.println(e.toString());}}
```

Program 4.10: run() method of FileThread

BufferedReader and PrintWriter

- getInputStream() and getOutputStream() are used to fetch the input and output streams from a socket.
- For easier string handling, these streams are wrapped in a BufferedReader and a PrintWriter.
- The BufferedReader uses an InputStreamReader connected to the socket’s input stream, allowing line-by-line reading via readLine().
- The PrintWriter connects to the socket's output stream and can optionally enable auto-flush, which automatically triggers flush() on each println() call to ensure data is sent immediately.
- Using a PrintWriter with auto-flush is advised to prevent errors and reduce manual flush calls.
- Programs 4.11 and 4.12 demonstrate similar outcomes using configurations with and without auto-flush.

```
PrintWriter autoFlusher = new PrintWriter(outputStream. true);
autoFlusher.println("Hello World!");
```

Program 4.11: PrintWriter with Auto-flush

```
PrintWriter noAutoFlusher = new PrintWriter(outputStream);
noAutoFlusher.println("Hello World!");
noAutoFlusher.flush();
```

Program 4.12: PrintWriter without Auto-flush

Read and responds to commands

- Once input and output streams are established, FileThread enters a continuous while loop that continues until the running flag is set to false.
- The loop reads commands from the socket using readLine() and verifies their validity (e.g., OPEN, READ, WRITE, CLOSE, SHUTDOWN).
- If a correct command is received, the corresponding method is called.
- Incorrect commands trigger an error message to be sent back to the client and the loop starts again from the beginning.

Opening a file

- Opening a file is implemented through the open() method, which is shown in Program 4.13.
- It first verifies that the OPEN command is complete, containing both the keyword OPEN followed by a filename.
- If no file is currently open, a new RandomAccessFile object is created, allowing random read and write access.
- After the file has been opened, an attempt is made to obtain an exclusive lock on the file. This involves calling the getChannel() method on the RandomAccessFile object and then the tryLock() method on the returned channel.
- The tryLock() method attempts to acquire a lock on the file. If this is not possible, tryLock() returns immediately, meaning tryLock() operates in non-blocking.
- If the lock is acquired, a success message is sent, otherwise, an error message is sent.
- If another file is already open, it is closed before attempting to open a new file and secure a lock.

```
private void open(String command) {
    // Early return if status check or command format is not addressed first
    int index = command.indexOf(' ');
    if (index == -1 || index == (command.length() - 1)) {
        writer.println("Incorrect command! Syntax: OPEN <filename>");
        return;
    }
    if (status == FileThread.FILE_NOT_OPEN) {
        try {
            file = new RandomAccessFile(path + command.substring(index + 1), "rw");
            // Attempt to obtain a lock on the file, if possible
            if (file.getChannel().tryLock() == null) {
                writer.println("File is already in use");
                return;
            }
            // Update current file name and status
            currentFilename = path + command.substring(index + 1);
            writer.println(currentFilename + " has been opened");
            status = FileThread.FILE_OPEN;
        }
```

The program continues on the following slide

```
catch (IOException e) {
            System.err.println(e.toString());
        }
        status = FileThread.FILE_OPEN;
    }else {
        try {
            file.close();
            file = new RandomAccessFile(path + command.substring(index + 1), "rw");
            // Attempt to get a lock on the file, if possible
            if (file.getChannel().tryLock() == null) {
                writer.println("File is already in use");
                return;
            }
            lastFilename = currentFilename;
            // Construct file name
            currentFilename = path + command.substring(index + 1);
            writer.println(lastFilename + " was closed, " + currentFilename + " has been opened");
        } catch (IOException e) {
            System.err.println(e.toString());
        }
    }
}
```

Program 4.13: open() method

Reading a File

- It first verifies if a file is currently open, ensuring the FileThread is not in the FILE_NOT_OPEN state.
- If no file is open, it sends an error message to the client via the socket.
- Otherwise, it reads the next line from the file and sends it to the client.
- If an attempt to read fails because the file's end has been reached, it sends an error message back to the client.
- The read() method reads from the opened file see Program 4.14

```
private void read() {
   if (status == FileThread.FILE_NOT_OPEN) {
      writer.println("No file opened");
   } else {
      String line = null;
      try {
         line = file.readLine();
         if (line != null) {
            writer.println(line);
         } else {
            writer.println("End of file reached");
         }
      } catch (IOException e) {
         System.err.println(e.toString());
      }
   }
}
```

Program 4.14: read() method

Write to a file (Program 4.15)

- It first checks if the file is open, ensuring that the FileThread is not in the FILE_NOT_OPEN state.
- The WRITE command, which includes a keyword "WRITE" followed by the data to be written, is verified for correctness similarly to the OPEN command.
- If no file is open, the method sends an error message back to the client.
- If the command is correct and a file is open, the method writes the data as a new line at the end of the file. The following steps are necessary.

1. file.getFilePointer() is used to remember the current position in the file.
2. The method call file.seek(file.length()) moves the file pointer to the end of the file.
3. file.writeBytes() method is called to enter the information as a new line in the file.
4. file.seek() is used to return to the previous file position.

```
private void write(String command) {
   if (status == FileThread.FILE_NOT_OPEN) {
      writer.println("No file opened");
   } else {
      int index = command.indexOf(' ');
      if (index == -1 || index == (command.length() - 1)) {
         writer.println("Incorrect command! Syntax: WRITE <Line>");
         return;
      }
      String line = command.substring(index + 1);
      try {
         // Remember current position in file
         long filePointer = file.getFilePointer();
         // Jump to the end of the file
         file.seek(file.length());
         // Write line at the end of the file
         file.writeBytes(line + System.getProperty("line.separator"));
         // Return to the previous position
         file.seek(filePointer);
         writer.println(line + " written in " + currentFilename);
      } catch (IOException e) {
         System.err.println(e.toString());}}}
```

Program 4.15: write() method

The close() method closes an already opened file and resets the FileThread to the FILE_NOT_OPEN state. It is shown in Program 4.16.

```
private void close() {
   if (status == FileThread.FILE_NOT_OPEN) {
      writer.println("No file opened");
   } else {
      try {
         file.close();
         writer.println(currentFilename + " has been closed");
      } catch (IOException e) {
         System.err.println(e.toString());
         System.exit(-1);
      }
      status = FileThread.FILE_NOT_OPEN;
   }
}
```

Program 4.16: close() method

- SHUTDOWN command ends communication with the FileServer.
- The SHUTDOWN command triggers the shutdown() method. In this method, any open file as well as the input and output streams of the socket, and the socket itself, are closed.

```
private void shutdown(){
  if(status == FileThread.FILE_OPEN){
    try{
      file.close();
    }catch (IOException e){
      System.err.println(e.toString());
    }} // end if
  try{
    reader.close(); // Close input/output streams
    writer.println("Communication is ending");
    writer.close();
    // Close socket
    socket.close();
  }catch (IOException e){
    System.err.println(e.toString());}}
```

Program 4.17: shutdown() method

FileClient

- The main() method of the FileClient class begins by checking the provided command-line arguments, requiring the hostname and the port number where the FILE service is accessible.
- A new object of the FileClient class is created, startClient() method establishes a new connection to the FileServer, securing input and output streams from this connection.
- In a loop, the client reads commands from the keyboard and sends them to the server.
- If the server returns a null response, all input and output streams as well as the socket are closed, then terminates.
- The complete implementation of FileClient is detailed in Program 4.18.

```
import java.io.*;
import java.net.*;
public class FileClient {
   private String host; // Hostname of the server
   private int port; // Port number of the FILE service
   // Constructor
   public FileClient(String host, int port) {
      this.host = host;
      this.port = port;
   }
   // Start client to handle commands and server communication
   public void startClient() {
      boolean running = true; // Flag to end client
      Socket socket;
      BufferedReader reader; // Read line by line from socket
      PrintWriter writer; // Write line by line to socket
      BufferedReader stdin; // Read line by line from keyboard
```

```
try {
   System.out.print("Connecting to " + host + ":" + port + "...");
   socket = new Socket(host, port);
   System.out.println("Connected");
   reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));
   writer = new PrintWriter(new OutputStreamWriter(socket.getOutputStream()), true);
   stdin = new BufferedReader(new InputStreamReader(System.in));
   String command = null;
   String response = null;
   while (running) {
      // Read command
      System.out.print("Command: ");
      command = stdin.readLine();
      // Send command to Server
      writer.println(command);
      // Wait for response from server
      response = reader.readLine();
```

```
if (response != null) {
            System.out.println(response);
        } else {
            try {
                writer.close();
                reader.close();
                socket.close();
                stdin.close();
                running = false;
            } catch (IOException e) {
                System.err.println(e.toString());
                System.exit(-1);
            }
        }
    }
  } catch (IOException e) {
    System.err.println(e.toString());
    System.exit(-1);
  }}
```

```
// Entry point
   public static void main(String[] args) {
      switch (args.length){
         case 2:
            try{
               (new FileClient(args[0].Integer.parseInt(args[1]))).startClient();
            }catch (NumberFormatException e) {
               System.err.println(e.toString());
               System.exit(-1);
            }
            break;
         default:
            System.err.println("\nUsage: java FileClient <Host> <Port>\n");
            break;
      }
   }
}
```