12. Aplikace klient – server

Obsah

- Základní pojmy a koncepty
- Socket a jeho využití
- Adresování serveru
- Navázání komunikace a možná úskalí
- Komunikace požadavek – odpověď (request – response)
- Vhodné protokoly a jazyky (HTTP, XML, JSON apod.)
- Obsluha socketu vlastním vláknem a výjimky
- Vhodné užití a příklady

Základní pojmy a koncepty

Klient-server architektura

Klient-server je síťová architektura, která odděluje klienty (žadatele o služby) od serverů (poskytovatelů služeb).

Server

- Počítačový program nebo zařízení, které poskytuje služby jiným počítačům (klientům)
- Pasivně čeká na požadavky od klientů
- Zpracovává požadavky a odesílá odpovědi
- Může obsluhovat mnoho klientů současně
- Obvykle běží nepřetržitě
- Příklady: webový server, databázový server, e-mailový server

Klient

- Počítačový program nebo zařízení, které využívá služby poskytované serverem
- Aktivně iniciuje komunikaci se serverem
- Odesílá požadavky na server a zpracovává odpovědi
- Příklady: webový prohlížeč, e-mailový klient, databázový klient

Rozdíl mezi architekturou klient-server a P2P

Klient-server

- Jasně definované role: server poskytuje služby, klient je využívá
- Centralizovaná struktura (server je centrálním bodem)
- Snazší správa a zabezpečení
- Závislost na dostupnosti serveru (single point of failure)
- Možnost škálování zvýšením výkonu serveru

Peer-to-peer (P2P)

- Každý účastník může být zároveň klient i server
- Decentralizovaná struktura
- Vyšší odolnost proti výpadku (není zde SPOF)
- S rostoucím počtem účastníků roste robustnost sítě
- Obtížnější správa a zabezpečení
- Příklady: BitTorrent, některé VoIP služby, Blockchain technologie

Socket a jeho využití

Co je socket?

Socket je koncový bod komunikace mezi dvěma programy běžícími v síti. Je to softwarová abstrakce, která představuje kombinaci IP adresy a portu, a poskytuje programátorům jednoduché rozhraní pro síťovou komunikaci.

Typy socketů

1. **TCP Socket** (Stream Socket)
   - Spolehlivý, spojovaný přenos
   - Garantuje doručení dat ve správném pořadí
   - Používá TCP protokol
2. **UDP Socket** (Datagram Socket)
   - Nespolehlivý, nespojovaný přenos
   - Negarantuje doručení ani pořadí dat
   - Rychlejší než TCP
   - Používá UDP protokol
3. **Raw Socket**
   - Přímý přístup k síťové vrstvě
   - Používá se pro specializované účely (např. ICMP)

Základní operace se sockety

Na straně serveru:

1. Vytvoření socketu
2. Svázání (bind) socketu s IP adresou a portem
3. Naslouchání (listen) příchozím spojením
4. Přijetí (accept) spojení od klienta
5. Přijímání a odesílání dat
6. Uzavření spojení

Na straně klienta:

1. Vytvoření socketu
2. Připojení (connect) k serveru
3. Odesílání a přijímání dat
4. Uzavření spojení

Příklad implementace v Javě

TCP Server

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

public class SimpleServer {
    public static void main(String[] args) {
        try {
            // Vytvoření server socketu a naslouchání na portu 8080
            ServerSocket serverSocket = new ServerSocket(8080);
            System.out.println("Server běží a čeká na klienty na portu 8080...");

            // Nekonečná smyčka - server běží stále
            while (true) {
                // Přijmutí spojení od klienta
                Socket clientSocket = serverSocket.accept();
                System.out.println("Klient připojen: " + clientSocket.getInetAddress());
```

```
                // Nastavení vstupního a výstupního streamu
                BufferedReader in = new BufferedReader(
                    new InputStreamReader(clientSocket.getInputStream()));
                PrintWriter out = new PrintWriter(
                    clientSocket.getOutputStream(), true);

                // Čtení zprávy od klienta
                String message = in.readLine();
                System.out.println("Přijata zpráva: " + message);

                // Odeslání odpovědi klientovi
                out.println("Server přijal zprávu: " + message);

                // Uzavření spojení s klientem
                clientSocket.close();
            }
        } catch (IOException e) {
            System.out.println("Chyba serveru: " + e.getMessage());
        }
    }
}
```

TCP Klient

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

public class SimpleClient {
    public static void main(String[] args) {
        try {
            // Připojení k serveru na localhost:8080
            Socket socket = new Socket("localhost", 8080);
            System.out.println("Připojeno k serveru");

            // Nastavení vstupního a výstupního streamu
            BufferedReader in = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(
                socket.getOutputStream(), true);

            // Odeslání zprávy serveru
            out.println("Ahoj, servere!");

            // Přijetí odpovědi od serveru
            String response = in.readLine();
            System.out.println("Odpověď serveru: " + response);

            // Uzavření socketu
            socket.close();
        } catch (IOException e) {
            System.out.println("Chyba klienta: " + e.getMessage());
        }
    }
}
```

Adresování serveru

IP adresa

- Jedinečný identifikátor zařízení v síti
- IPv4: 32-bitové adresy (např. 192.168.1.1)
- IPv6: 128-bitové adresy (např. 2001:0db8:85a3:0000:0000:8a2e:0370:7334)
- Statická (pevně přidělená) nebo dynamická (přidělená dočasně, např. pomocí DHCP)

Port

- 16-bitové číslo (0-65535), které identifikuje konkrétní aplikaci/službu na daném zařízení
- Umožňuje, aby na jednom zařízení běželo více služeb
- Známé porty (0-1023): přiřazeny standardním službám (HTTP: 80, HTTPS: 443, FTP: 21)
- Registrované porty (1024-49151): registrovány pro konkrétní služby
- Dynamické porty (49152-65535): volně použitelné

Doménové jméno a DNS

- Doménové jméno: snadno zapamatovatelný název, který se mapuje na IP adresu
- DNS (Domain Name System): systém pro překlad doménových jmen na IP adresy
- Hierarchická struktura: subdomény.doména.TLD (např. www.example.com)
- Umožňuje adresovat server, i když se změní jeho IP adresa

URL (Uniform Resource Locator)

- Kompletní adresa zdroje na internetu
- Skládá se z:
  - Protokol (http, https, ftp...)
  - Hostname (doménové jméno nebo IP adresa)
  - Port (volitelný)
  - Cesta k zdroji
  - Parametry (volitelné)
  - Fragment (volitelný)
- Příklad: https://www.example.com:443/path/to/resource?param=value#section

Navázání komunikace a možná úskalí

TCP Handshake (three-way handshake)

Proces navázání TCP spojení probíhá ve třech krocích:

1. **SYN**: Klient posílá paket s příznakem SYN (synchronize) a počátečním sekvenčním číslem
2. **SYN-ACK**: Server odpovídá paketem s příznaky SYN a ACK (acknowledge) a vlastním sekvenčním číslem
3. **ACK**: Klient potvrzuje spojení paketem s příznakem ACK

Po úspěšném dokončení třícestného handshaku je TCP spojení ustanoveno a obě strany mohou začít přenášet data.

Možná úskalí a problémy při navázání komunikace

Nedostupnost serveru

- Server není spuštěn nebo je nedostupný
- Řešení: implementace opakovaných pokusů o připojení s exponenciálním čekáním

Chybná adresa nebo port

- Nesprávná IP adresa nebo port
- Řešení: ověření správnosti adresy, použití DNS pro získání aktuální IP adresy

Blokování firewallem

- Firewall na straně klienta nebo serveru může blokovat spojení
- Řešení: konfigurace firewallu, použití alternativních portů

Přetížení serveru

- Server odmítá nová spojení z důvodu přetížení
- Řešení: load balancing, zvýšení kapacity serveru, implementace front požadavků

Timeout

- Vypršení časového limitu při pokusu o spojení
- Řešení: nastavení vhodných timeoutů, opakované pokusy

Chyby při autentizaci

- Neplatné přihlašovací údaje, expirovaný nebo neplatný certifikát
- Řešení: ověření přihlašovacích údajů, aktualizace certifikátů

Problémy s NAT a proxy

- Network Address Translation (NAT) může způsobit problémy při mapování portů
- Řešení: implementace technik pro průchod NAT (NAT traversal), použití STUN/TURN serverů

Příklad ošetření chyb při navazování spojení v Javě

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

public class RobustClient {
    private static final int MAX_ATTEMPTS = 5;
    private static final int INITIAL_TIMEOUT = 1000; // 1 sekunda

    public static void main(String[] args) {
        int attempts = 0;
        int timeout = INITIAL_TIMEOUT;

        while (attempts < MAX_ATTEMPTS) {
            try {
                // Nastavení timeoutu pro spojení
                Socket socket = new Socket();
                socket.connect(new InetSocketAddress("example.com", 8080), timeout);

                System.out.println("Připojeno k serveru");

                // Komunikace se serverem...

                socket.close();
                return; // Úspěšné připojení, ukončíme metodu
            } catch (SocketTimeoutException e) {
                System.out.println("Timeout při připojování k serveru");
            } catch (ConnectException e) {
                System.out.println("Server odmítl spojení");
            } catch (UnknownHostException e) {
                System.out.println("Neznámý hostitel");
                break; // Chybná adresa, nemá smysl opakovat
            } catch (IOException e) {
                System.out.println("Chyba I/O: " + e.getMessage());
```

```
            }

            attempts++;
            timeout *= 2; // Exponenciální čekání
            System.out.println("Pokus " + attempts + " z " + MAX_ATTEMPTS +
                               ", další pokus za " + (timeout/1000) + " sekund");

            try {
                Thread.sleep(timeout);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                break;
            }
        }

        System.out.println("Nepodařilo se připojit k serveru po " + MAX_ATTEMPTS + " pokusech");
    }
}
```

Komunikace požadavek – odpověď (request – response)

Princip komunikace request-response

1. Klient vytvoří a odešle požadavek (request) na server
2. Server přijme požadavek, zpracuje ho a připraví odpověď
3. Server odešle odpověď (response) klientovi
4. Klient přijme odpověď a zpracuje ji

Synchronní vs. asynchronní komunikace

Synchronní komunikace

- Klient po odeslání požadavku čeká na odpověď od serveru
- Jednodušší implementace, ale blokuje další činnost klienta
- Vhodná pro jednoduché operace nebo když potřebujeme okamžitou odpověď

Asynchronní komunikace

- Klient po odeslání požadavku pokračuje v činnosti a nezůstává blokován
- Odpověď serveru je zpracována později (např. callback, událost)
- Složitější implementace, ale umožňuje lepší využití zdrojů
- Vhodná pro dlouho trvající operace nebo uživatelská rozhraní

Stavová vs. bezstavová komunikace

Stavová komunikace

- Server si pamatuje stav komunikace s klientem mezi jednotlivými požadavky
- Výhoda: jednodušší správa kontextu, menší režie při opakovaných požadavcích
- Nevýhoda: náročnější na zdroje serveru, problémy při škálování

Bezstavová komunikace

- Server neuchovává stav mezi požadavky, každý požadavek obsahuje všechny potřebné informace
- Výhoda: jednodušší škálování, odolnost proti výpadkům
- Nevýhoda: větší objem přenášených dat, složitější správa kontextu na straně klienta
- Příklad: HTTP (základní protokol je bezstavový, stavovost je implementována pomocí cookies, sessionů)

Příklad komunikace request-response v Javě

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

public class RequestResponseServer {
    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(8080);
            System.out.println("Server běží na portu 8080");

            while (true) {
                Socket clientSocket = serverSocket.accept();
                System.out.println("Klient připojen: " + clientSocket.getInetAddress());

                // Zpracování v novém vlákně
                new Thread(() -> handleClient(clientSocket)).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static void handleClient(Socket clientSocket) {
        try {
            // Nastavení streamu pro čtení požadavku
            BufferedReader reader = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));

            // Nastavení streamu pro zápis odpovědi
            PrintWriter writer = new PrintWriter(clientSocket.getOutputStream(), true);

            // Čtení požadavku
            String request = reader.readLine();
            System.out.println("Přijat požadavek: " + request);

            // Zpracování požadavku (v reálné aplikaci by bylo složitější)
            String response;
            if (request.startsWith("GET")) {
                response = "200 OK: Data pro GET požadavek";
            } else if (request.startsWith("POST")) {
                response = "200 OK: Data byla přijata";
            } else {
                response = "400 Bad Request: Neznámý typ požadavku";
            }

            // Odeslání odpovědi
            writer.println(response);
            System.out.println("Odeslána odpověď: " + response);

            // Zavření spojení
            clientSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Vhodné protokoly a jazyky (HTTP, XML, JSON apod.)

Protokoly pro komunikaci klient-server

HTTP (Hypertext Transfer Protocol)

- Nejpoužívanější protokol pro webové aplikace
- Bezstavový, text-based protokol
- Základní metody: GET, POST, PUT, DELETE, HEAD, OPTIONS, PATCH
- Odpovědi obsahují stavové kódy (200 OK, 404 Not Found, 500 Internal Server Error, atd.)
- HTTP/1.1: jedno spojení pro více požadavků (keep-alive)
- HTTP/2: multiplexing, server push, komprese hlaviček
- HTTP/3: postaveno na QUIC místo TCP, lepší podpora mobilních zařízení

HTTPS (HTTP Secure)

- HTTP s šifrováním pomocí TLS/SSL
- Chrání proti odposlouchávání a man-in-the-middle útokům
- Výchozí port 443 (místo 80 pro HTTP)

WebSocket

- Plně duplexní komunikační protokol přes TCP
- Umožňuje dlouhodobé spojení mezi klientem a serverem
- Server může posílat data klientovi bez nutnosti nového požadavku
- Vhodné pro aplikace vyžadující aktualizace v reálném čase (chat, hry, sledování cenných papírů)

REST (Representational State Transfer)

- Architektonický styl pro návrh webových služeb
- Využívá HTTP metody pro CRUD operace (Create, Read, Update, Delete)
- Bezstavový, klientské požadavky musí obsahovat všechny potřebné informace
- Snadno škálovatelný, jednoduchá implementace

SOAP (Simple Object Access Protocol)

- Protokol pro výměnu strukturovaných informací v distribuovaných systémech
- Založen na XML
- Komplexnější než REST, ale nabízí více funkcí (transakce, spolehlivé doručení)
- Méně závislý na HTTP než REST

gRPC

- Moderní RPC (Remote Procedure Call) framework od Google
- Využívá HTTP/2 a Protocol Buffers
- Vysoký výkon, silně typované rozhraní
- Podporuje streaming a autentizaci

Formáty pro výměnu dat

XML (eXtensible Markup Language)

- Značkovací jazyk pro strukturované dokumenty
- Čitelný pro člověka i stroj
- Samopopisný, hierarchická struktura
- Široká podpora parsování v různých jazycích
- Relativně objemný (mnoho značek)

Příklad:

```xml
<osoba>
  <jmeno>Jan</jmeno>
  <prijmeni>Novák</prijmeni>
  <vek>30</vek>
  <adresa>
    <ulice>Hlavní 123</ulice>
    <mesto>Praha</mesto>
  </adresa>
</osoba>
```

JSON (JavaScript Object Notation)

- Odlehčený formát pro výměnu dat
- Snadno čitelný i zapisovatelný, menší overhead než XML
- Nativní podpora v JavaScriptu, široká podpora v ostatních jazycích
- Datové typy: string, number, object, array, boolean, null

Příklad:

```json
{
  "jmeno": "Jan",
  "prijmeni": "Novák",
  "vek": 30,
  "adresa": {
    "ulice": "Hlavní 123",
    "mesto": "Praha"
  }
}
```

Protocol Buffers

- Binární formát pro serializaci strukturovaných dat od Google
- Kompaktnější a rychlejší než XML nebo JSON
- Vyžaduje předem definovaný schema
- Silně typované
- Vhodné pro komunikaci mezi interními systémy

MessagePack

- Binární serializační formát podobný JSON, ale kompaktnější
- Rychlejší serializace/deserializace než JSON
- Podporuje stejné datové typy jako JSON

YAML (YAML Ain't Markup Language)

- Lidsky čitelný formát pro serializaci dat
- Méně syntaktického šumu než JSON nebo XML
- Podporuje komentáře, reference, komplexní datové typy

Výběr vhodného protokolu a formátu

Při výběru protokolu a formátu dat je třeba zvážit:

1. **Kompatibilita** - podpora v různých prostředích a programovacích jazycích
2. **Výkon** - rychlost serializace/deserializace, objem přenášených dat
3. **Čitelnost** - potřeba manuálního debugování nebo editace
4. **Flexibilita** - snadnost změn v datové struktuře
5. **Bezpečnost** - ochrana přenášených dat
6. **Škálovatelnost** - chování při velkém počtu klientů nebo velkém objemu dat

Obsluha socketu vlastním vláknem a výjimky

Multithreaded server

Aby server mohl obsluhovat více klientů současně, je třeba každé spojení zpracovávat v samostatném vlákně.

Implementace multithreaded serveru v Javě

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

public class MultithreadedServer {
    private static final int PORT = 8080;
    private static final int THREAD_POOL_SIZE = 50;

    public static void main(String[] args) {
        // Vytvoření thread poolu pro obsluhu klientů
        ExecutorService threadPool = Executors.newFixedThreadPool(THREAD_POOL_SIZE);

        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
            System.out.println("Server běží na portu " + PORT);

            while (true) {
                try {
                    // Čekání na připojení klienta
                    Socket clientSocket = serverSocket.accept();
                    System.out.println("Nové spojení: " + clientSocket.getInetAddress());

                    // Předání obsluhy klienta do thread poolu
                    threadPool.execute(new ClientHandler(clientSocket));
                } catch (IOException e) {
                    System.err.println("Chyba při přijímání spojení: " + e.getMessage());
                }
            }
        } catch (IOException e) {
            System.err.println("Chyba při vytváření server socketu: " + e.getMessage());
            threadPool.shutdown();
        }
    }

    // Třída pro obsluhu jednoho klienta
    static class ClientHandler implements Runnable {
        private final Socket clientSocket;

        public ClientHandler(Socket socket) {
            this.clientSocket = socket;
        }

        @Override
        public void run() {
            try (
                BufferedReader in = new BufferedReader(
                    new InputStreamReader(clientSocket.getInputStream()));
                PrintWriter out = new PrintWriter(
                    clientSocket.getOutputStream(), true)
            ) {
                String inputLine;
```

```
                while ((inputLine = in.readLine()) != null) {
                    // Zpracování požadavku
                    System.out.println("Přijato od " + clientSocket.getInetAddress() + ": " + inputL

                    // Odeslání odpovědi
                    out.println("Echo: " + inputLine);

                    // Ukončení spojení při přijetí "bye"
                    if ("bye".equalsIgnoreCase(inputLine)) {
                        break;
                    }
                }
            } catch (IOException e) {
                System.err.println("Chyba při komunikaci s klientem: " + e.getMessage());
            } finally {
                try {
                    clientSocket.close();
                } catch (IOException e) {
                    System.err.println("Chyba při zavírání spojení: " + e.getMessage());
                }
                System.out.println("Spojení ukončeno: " + clientSocket.getInetAddress());
            }
        }
    }
}
```

Ošetření výjimek při síťové komunikaci

Při síťové komunikaci může dojít k mnoha typům výjimek, které je třeba správně ošetřit:

Běžné výjimky v síťové komunikaci (Java)

1. **IOException** - obecná výjimka pro I/O operace
2. **SocketException** - chyby při operacích se sockety
3. **ConnectException** - chyba při připojování k serveru
4. **SocketTimeoutException** - timeout při operacích se sockety
5. **UnknownHostException** - nelze přeložit hostname na IP adresu
6. **BindException** - chyba při bindování socketu (např. port již používán)

Strategie ošetření výjimek

1. **Logování** - zaznamenání chyby pro pozdější analýzu
2. **Opakované pokusy** - při přechodných chybách
3. **Graceful degradation** - poskytnutí omezené funkčnosti při problémech
4. **Upozornění uživatele** - srozumitelné chybové hlášky
5. **Ukončení spojení** - bezpečné uzavření zdrojů

Příklad robustního klienta s ošetřením výjimek

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

public class RobustNetworkClient {
    private static final int MAX_RETRIES = 3;
    private static final int CONNECT_TIMEOUT = 5000; // 5 sekund
    private static final int READ_TIMEOUT = 10000; // 10 sekund

    public static void main(String[] args) {
```

```
        for (int retry = 0; retry <= MAX_RETRIES; retry++) {
            try {
                if (retry > 0) {
                    System.out.println("Pokus o připojení č. " + retry);
                    // Exponenciální čekání před opakováním
                    Thread.sleep(1000 * (long)Math.pow(2, retry - 1));
                }

                // Vytvoření socketu s timeouty
                Socket socket = new Socket();
                socket.connect(new InetSocketAddress("example.com", 80), CONNECT_TIMEOUT);
                socket.setSoTimeout(READ_TIMEOUT);

                try (
                    BufferedReader in = new BufferedReader(
                        new InputStreamReader(socket.getInputStream()));
                    PrintWriter out = new PrintWriter(
                        socket.getOutputStream(), true)
                ) {
                    // Komunikace se serverem
                    System.out.println("Připojeno k serveru");
                    out.println("GET / HTTP/1.1");
                    out.println("Host: example.com");
                    out.println("Connection: close");
                    out.println();

                    // Čtení odpovědi
                    String line;
                    while ((line = in.readLine()) != null) {
                        System.out.println(line);
                    }
                }

                // Úspěšná komunikace, konec cyklu
                break;

            } catch (UnknownHostException e) {
                System.err.println("Neznámý hostitel: " + e.getMessage());
                // Nemá smysl opakovat při neznámém hostiteli
                break;
            } catch (SocketTimeoutException e) {
                System.err.println("Timeout: " + e.getMessage());
                // Pokračuje v cyklu pro další pokus
            } catch (ConnectException e) {
                System.err.println("Chyba připojení: " + e.getMessage());
                // Pokračuje v cyklu pro další pokus
            } catch (IOException e) {
                System.err.println("I/O chyba: " + e.getMessage());
                // Pokračuje v cyklu pro další pokus
            } catch (InterruptedException e) {
                System.err.println("Přerušeno: " + e.getMessage());
                Thread.currentThread().interrupt();
                break;
            }
        }
    }
}
```

Non-blocking I/O (NIO)

Pro vysokovýkonné servery je vhodné použít non-blocking I/O, které umožňuje obsluhovat mnoho spojení s menším počtem vláken.

Základy Java NIO

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.*;
import java.util.*;

public class NIOServer {
    public static void main(String[] args) throws IOException {
        // Vytvoření a konfigurace selektoru
        Selector selector = Selector.open();

        // Vytvoření a konfigurace server socketu
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        serverSocketChannel.socket().bind(new InetSocketAddress(8080));
        serverSocketChannel.configureBlocking(false);

        // Registrace serveru se selektorem pro přijímání spojení
        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);

        System.out.println("NIO server běží na portu 8080");

        while (true) {
            // Čekání na události
            selector.select();

            // Zpracování všech připravených klíčů
            Set<SelectionKey> selectedKeys = selector.selectedKeys();
            Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

            while (keyIterator.hasNext()) {
                SelectionKey key = keyIterator.next();

                if (key.isAcceptable()) {
                    // Nové spojení
                    handleAccept(key, selector);
                } else if (key.isReadable()) {
                    // Data připravena ke čtení
                    handleRead(key);
                } else if (key.isWritable()) {
                    // Socket připraven k zápisu
                    handleWrite(key);
                }

                keyIterator.remove();
            }
        }
    }

    private static void handleAccept(SelectionKey key, Selector selector) throws IOException {
        ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();
        SocketChannel clientChannel = serverChannel.accept();
        clientChannel.configureBlocking(false);
```

```
            // Registrace klienta pro čtení
            clientChannel.register(selector, SelectionKey.OP_READ);
            System.out.println("Přijato nové spojení od: " + clientChannel.getRemoteAddress());
        }

        private static void handleRead(SelectionKey key) throws IOException {
            SocketChannel clientChannel = (SocketChannel) key.channel();
            ByteBuffer buffer = ByteBuffer.allocate(1024);

            try {
                int bytesRead = clientChannel.read(buffer);

                if (bytesRead == -1) {
                    // Konec spojení
                    clientChannel.close();
                    key.cancel();
                    System.out.println("Spojení ukončeno: " + clientChannel.getRemoteAddress());
                    return;
                }

                buffer.flip();
                byte[] data = new byte[buffer.limit()];
                buffer.get(data);

                String message = new String(data).trim();
                System.out.println("Přijato: " + message);

                // Příprava odpovědi
                String response = "Echo: " + message;
                ByteBuffer responseBuffer = ByteBuffer.wrap(response.getBytes());

                // Odeslání odpovědi
                clientChannel.write(responseBuffer);

            } catch (IOException e) {
                System.err.println("Chyba při čtení: " + e.getMessage());
                clientChannel.close();
                key.cancel();
            }
        }

        private static void handleWrite(SelectionKey key) throws IOException {
            // Implementace dle potřeby - zde zjednodušeno, protože zápis provádíme přímo v handleRead
        }
    }
```

Vhodné užití a příklady

Příklady aplikací využívajících architekturu klient-server

1. Webové aplikace

- **Server**: Apache, Nginx, Tomcat
- **Klient**: webový prohlížeč
- **Protokol**: HTTP/HTTPS
- **Formát dat**: HTML, CSS, JavaScript, JSON, XML

2. E-mailové služby

- **Server**: SMTP, POP3, IMAP servery
- **Klient**: e-mailový klient (Outlook, Thunderbird)
- **Protokoly**: SMTP, POP3, IMAP
- **Formát dat**: MIME

3. Databázové systémy

- **Server**: MySQL, PostgreSQL, Oracle, SQL Server
- **Klient**: aplikace nebo databázový klient
- **Protokol**: proprietární protokoly specifické pro každou databázi
- **Formát dat**: SQL, binární formáty

4. Aplikace pro chat a komunikaci

- **Server**: messaging server
- **Klient**: chat aplikace
- **Protokol**: XMPP, proprietární protokoly, WebSocket
- **Formát dat**: JSON, XML, binární formáty

5. Distribuované výpočetní systémy

- **Server**: koordinační server
- **Klient**: výpočetní uzly
- **Protokoly**: různé (HTTP, TCP/UDP, gRPC)
- **Formát dat**: binární formáty, JSON

Jednoduchý HTTP server v Javě

```
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpHandler;
import com.sun.net.httpserver.HttpServer;

import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;

public class SimpleHTTPServer {
    public static void main(String[] args) throws IOException {
        // Vytvoření HTTP serveru na portu 8000
        HttpServer server = HttpServer.create(new InetSocketAddress(8000), 0);

        // Přidání handleru pro cestu "/hello"
        server.createContext("/hello", new HttpHandler() {
            @Override
            public void handle(HttpExchange exchange) throws IOException {
                String response = "Hello, World!";
                exchange.sendResponseHeaders(200, response.length());
                try (OutputStream os = exchange.getResponseBody()) {
                    os.write(response.getBytes());
                }
            }
        });

        // Přidání handleru pro cestu "/info"
        server.createContext("/info", new HttpHandler() {
            @Override
            public void handle(HttpExchange exchange) throws IOException {
                // Získání informací o požadavku
                String requestMethod = exchange.getRequestMethod();
```

```
                String remoteAddress = exchange.getRemoteAddress().toString();

                String response = "Request Method: " + requestMethod + "\n" +
                                  "Remote Address: " + remoteAddress + "\n" +
                                  "URI: " + exchange.getRequestURI();

                exchange.sendResponseHeaders(200, response.length());
                try (OutputStream os = exchange.getResponseBody()) {
                    os.write(response.getBytes());
                }
            }
        });

        // Nastavení executor pro zpracování požadavků
        server.setExecutor(null); // Použije výchozí executor

        // Spuštění serveru
        server.start();
        System.out.println("Server běží na portu 8000");
    }
}
```

REST API server s použitím Javy a Spring Boot

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.*;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.AtomicLong;

@SpringBootApplication
public class RestApiServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(RestApiServerApplication.class, args);
    }
}

// Model pro uživatele
class User {
    private long id;
    private String name;
    private String email;

    public User(long id, String name, String email) {
        this.id = id;
        this.name = name;
        this.email = email;
    }

    // Gettery a settery
    public long getId() { return id; }
    public void setId(long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getEmail() { return email; }
```

```
    public void setEmail(String email) { this.email = email; }
}

// REST controller
@RestController
@RequestMapping("/api/users")
class UserController {
    private final List<User> users = new ArrayList<>();
    private final AtomicLong counter = new AtomicLong();

    // Konstruktor s inicializací vzorových dat
    public UserController() {
        users.add(new User(counter.incrementAndGet(), "Jan Novák", "jan@example.com"));
        users.add(new User(counter.incrementAndGet(), "Eva Svobodová", "eva@example.com"));
    }

    // Získání všech uživatelů
    @GetMapping
    public List<User> getAllUsers() {
        return users;
    }

    // Získání uživatele podle ID
    @GetMapping("/{id}")
    public User getUserById(@PathVariable Long id) {
        return users.stream()
                .filter(user -> user.getId() == id)
                .findFirst()
                .orElseThrow(() -> new RuntimeException("Uživatel s ID " + id + " nenalezen"));
    }

    // Vytvoření nového uživatele
    @PostMapping
    public User createUser(@RequestBody User user) {
        user.setId(counter.incrementAndGet());
        users.add(user);
        return user;
    }

    // Aktualizace uživatele
    @PutMapping("/{id}")
    public User updateUser(@PathVariable Long id, @RequestBody User updatedUser) {
        User user = getUserById(id);
        user.setName(updatedUser.getName());
        user.setEmail(updatedUser.getEmail());
        return user;
    }

    // Smazání uživatele
    @DeleteMapping("/{id}")
    public void deleteUser(@PathVariable Long id) {
        users.removeIf(user -> user.getId() == id);
    }
}
```

WebSocket server pro real-time chat

```
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import javax.websocket.OnClose;
import javax.websocket.OnError;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.PathParam;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/chat/{username}")
public class ChatEndpoint {

    private static final Map<String, Session> sessions = new HashMap<>();

    @OnOpen
    public void onOpen(Session session, @PathParam("username") String username) {
        sessions.put(username, session);
        broadcastMessage("Server", username + " se připojil k chatu!");
    }

    @OnMessage
    public void onMessage(String message, @PathParam("username") String username) {
        broadcastMessage(username, message);
    }

    @OnClose
    public void onClose(Session session, @PathParam("username") String username) {
        sessions.remove(username);
        broadcastMessage("Server", username + " opustil chat!");
    }

    @OnError
    public void onError(Throwable error) {
        error.printStackTrace();
    }

    private void broadcastMessage(String sender, String message) {
        String formattedMessage = sender + ": " + message;
        sessions.forEach((username, session) -> {
            try {
                if (session.isOpen()) {
                    session.getBasicRemote().sendText(formattedMessage);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        });
    }
}
```

Jednoduchý UDP klient a server

UDP Server

```java
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class UDPServer {
    public static void main(String[] args) {
        final int PORT = 9876;

        try (DatagramSocket socket = new DatagramSocket(PORT)) {
            System.out.println("UDP server běží na portu " + PORT);

            byte[] receiveBuffer = new byte[1024];

            while (true) {
                // Příprava paketu pro příjem dat
                DatagramPacket receivePacket = new DatagramPacket(receiveBuffer, receiveBuffer.lengt

                // Čekání na příchozí paket
                socket.receive(receivePacket);

                // Zpracování přijatých dat
                String message = new String(receivePacket.getData(), 0, receivePacket.getLength());
                System.out.println("Přijato od " + receivePacket.getAddress() + ": " + message);

                // Získání adresy a portu odesílatele
                InetAddress clientAddress = receivePacket.getAddress();
                int clientPort = receivePacket.getPort();

                // Příprava odpovědi
                String response = "Echo: " + message;
                byte[] sendBuffer = response.getBytes();

                // Odeslání odpovědi
                DatagramPacket sendPacket = new DatagramPacket(
                    sendBuffer, sendBuffer.length, clientAddress, clientPort);
                socket.send(sendPacket);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

UDP Klient

```java
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.util.Scanner;

public class UDPClient {
    public static void main(String[] args) {
        final String SERVER_HOSTNAME = "localhost";
        final int SERVER_PORT = 9876;

        try (DatagramSocket socket = new DatagramSocket();
```

```
             Scanner scanner = new Scanner(System.in)) {

            InetAddress serverAddress = InetAddress.getByName(SERVER_HOSTNAME);

            System.out.println("UDP klient připraven. Zadejte zprávu (nebo 'exit' pro ukončení):");

            while (true) {
                // Čtení vstupu od uživatele
                String message = scanner.nextLine();

                if ("exit".equalsIgnoreCase(message)) {
                    break;
                }

                // Příprava paketu s daty
                byte[] sendBuffer = message.getBytes();
                DatagramPacket sendPacket = new DatagramPacket(
                    sendBuffer, sendBuffer.length, serverAddress, SERVER_PORT);

                // Odeslání paketu
                socket.send(sendPacket);

                // Příprava paketu pro příjem odpovědi
                byte[] receiveBuffer = new byte[1024];
                DatagramPacket receivePacket = new DatagramPacket(receiveBuffer, receiveBuffer.lengt

                // Čekání na odpověď
                socket.receive(receivePacket);

                // Zpracování odpovědi
                String response = new String(receivePacket.getData(), 0, receivePacket.getLength());
                System.out.println("Odpověď serveru: " + response);
            }

            System.out.println("Klient ukončen.");

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

Shrnutí

Architektura klient-server je základním stavebním kamenem mnoha síťových aplikací. Klíčové aspekty této architektury zahrnují:

1. **Socket** - koncový bod komunikace, který umožňuje programům komunikovat přes síť
2. **Adresování serveru** - způsob, jak klient najde a připojí se k serveru (IP adresa, port, doménové jméno)
3. **Navázání komunikace** - proces ustanovení spojení mezi klientem a serverem, včetně řešení různých problémů
4. **Komunikace request-response** - základní vzor výměny zpráv mezi klientem a serverem
5. **Protokoly a formáty dat** - standardy pro komunikaci a reprezentaci dat (HTTP, JSON, XML)
6. **Multithreading a NIO** - techniky pro efektivní obsluhu více klientů současně
7. **Ošetření výjimek** - správné zacházení s chybami, které mohou během síťové komunikace nastat