Síťové služby 2

V tomto semináři si představíme základní nástroje pro tvorbu a přístup k síťovým službám. Jmenovitě se zaměříme na služby postavené na protokolu TCP/IP. Tato kombinace protokolů patří mezi nejrozšířenější a současně se na jednotlivá spojení můžeme dívat jako na proud bytů od klienta k serveru (či opačně) a můžeme tak využít prostředky, které již známe například z práce se soubory.

1 Rychlé řešení pro protokol HTTP

Náš popis prostředků pro práci se síťovými službami začneme u protokolu HTTP(S), který se využívá nejen pro sdílení dokumentů, k čemuž byl původně navržen, ale má celou řadu dalších uplatnění, např. u webových služeb postavených na architektuře REST[1].

1.1 Objekt URL

Třída `URL` slouží primárně k uložení adresy ve formě URL[2]. V této třídě můžeme najít metodu `URL.openStream()`, která se postará vytvoření spojení a vrátí data z dané URL v podobě objektu třídy `InputStream`.[3] Ukažme si to na následujícím příkladu, který bude číst data řádek po řádku z webu katedry a vypisovat je na standardní výstup.

```
String line;
URL source = new URL("https://www.inf.upol.cz/");
try (InputStream is = source.openStream();
     BufferedReader rd = new BufferedReader(new InputStreamReader(is))) {
     while ((line = rd.readLine()) != null) {
         System.out.println(line);
     }
}
```

Nejdříve jsme si vytvořili pomocnou proměnnou `line`, do které budeme načítat jednotlivé řádky, a dále jsme si vytvořili objekt třídy `URL`, který reprezentuje adresu, jejíž obsah chceme přečíst. Metoda `URL.openStream()` vrací data ve formě `InputStream`, což je obecný proud bytů. Jelikož chceme s dokumentem pracovat jako s textem, převedeme tento proud bytů pomocí třídy `InputStreamReader` na proud

[1] `https://en.wikipedia.org/wiki/Representational_state_transfer`
[2] `https://en.wikipedia.org/wiki/URL`
[3] Podporované jsou protokoly HTTP, HTTPS, FILE, JAR.

znaků (třídu `Reader`), který ještě obalíme do třídy `BufferedReader`, která umožňuje načítat data po řádcích.[4] Následuje smyčka, která načítá jednotlivé řádky, dokud je co číst.

1.2 Http Client 2.0

Třída `URL` umožňuje pouze načítat data z dané URL. Pokud máme ambice s protokolem HTTP dělat něco složitějšího, například posílat požadavky, využívat všech možností verze protokolu HTTP 2.0 měli bychom sáhnout po HTTP klientovi, který je k dispozici od verze Java 11. Princip práce s tímto klientem si ukážeme na následujícím příkladu.

```
1 URL url = new URL("https://www.inf.upol.cz/mapa-stranek");
2 HttpClient httpClient = HttpClient.newBuilder().version(Version.HTTP_1_1).build();
3 HttpRequest request   = HttpRequest.newBuilder().uri(url.toURI()).GET().build();
4 HttpResponse<String> response = httpClient.send(request, BodyHandlers.ofString());
5 System.out.println("Status: " + response.statusCode());
6 System.out.println("Content: " + response.body());
```

Všimněme si, že se nejdříve vytvoří klient (řádek č. 2), následně požadavek (řádek č. 3), a ten je pak předán klientu (řádek č. 4). Pro vytvoření klienta i požadavku se používá navrhový vzor Builder[5], který podle potřeby umožňuje specifikovat různé vlastnosti ať už klienta (protokol, nastavení proxy, autentizace), tak i požadavku (typ, hlavičky, timeout). V našem příkladu jsme u klienta nastavili preferovaný protokol HTTP/1.1 a u požadavku jsme nastavili URI[6] a to, že má být odeslán jako požadavek typu GET.

Součástí odeslání požadavku (řádek č. 4) je i parametr, který udává, jak naložit s tělem odpovědi. V našem případě je tělo odpovědi převedeno na řetezec, ale je možné nechat odpověď uložit do souboru či převést na obecný proud bytů (`InputStream`).

Narozdíl od objektu typu `URL` můžeme přečíst z odpovědi i další stavové informace (např. chybový kód odpovědi[7]). HTTP Client podporuje celou řadu dalších vlastností, jako je například asynchronní komunikace, to však jde nad rámec tohoto semináře a je ponecháno na prostudování p.t. čtenářům.

2 Obecný síťový klient

V předchozí kapitole jsme si ukázali klienty, kteří zpřístupňují komunikaci pomocí protokolu HTTP. Tito klienti nás odstiňují od implementačních detailů komunikace. Co kdybychom si chtěli napsat vlastního klienta, nebo klienta pro jinou službu, například SMTP? Proto si nyní ukážeme tvorbu jednoduchého HTTP klienta, přičemž použitý princip lze přenést i na další komunikační protokoly.

Připomeňme, jak vypadá takový jednoduchý HTTP požadavek.

```
GET /cesta HTTTP/1.0\r\n
Host: www.example.com\r\n
```

[4] Pokud bychom chtěli přenášet binární data, např. obrázky, tuto konverzi neprovádíme.
[5] `https://en.wikipedia.org/wiki/Builder_pattern`
[6] `https://en.wikipedia.org/wiki/Uniform_Resource_Identifier`
[7] To je opět důležité například pro služby postavené na architektuře REST.

`\r\n`

Na začátku máme typ požadavku (GET, POST atp.), následuje cesta a verze protokolu. Součástí požadavku jsou hlavičky ve tvaru `Hlavička: hodnota`. V našem příkladu je uvedeno jen jméno serveru, ke kterému se připojujeme. Konec požadavku je vyznačen prázdným řádkem (`\r\n` v příkladu označuje ukončení řádků pomocí řídících znaků CR a LF, jak je vyžaduje standard jazyka HTTP). Po odeslání požadavku HTTP server vrátí stavovou informaci, seznam hlaviček oddělených opět prázdným řádkem a samotný obsah stránky.

Že komunikace probíhá tímto způsobem, si můžeme vyzkoušet například pomocí programu telnet, který je sice určený pro komunikaci archaickým protokolem Telnet[8], ale můžeme jej použít i pro práci s jednoduchými textovými protokoly, jako je třeba HTTP nebo SMTP.

Nejdříve si spustíme telnet, uvedeme adresu serveru a port, a dostaneme úvodní informace od programu telnet.

```
> telnet www.inf.upol.cz 80
Trying 158.194.92.6...
Connected to www.inf.upol.cz.
Escape character is '^]'.
```

Nyní můžeme zadat požadavek, použijeme tu nejjednodušší varinatu.

```
GET / HTTP/1.0
Host: www.inf.upol.cz
```

Poté, co požadavek odešleme vložením prázdného řádku, získáme odpověď.

```
HTTP/1.1 301 Moved Permanently
Date: Wed, 24 Feb 2021 09:40:00 GMT
Server: Apache/2.4.38 (Debian)
Location: https://www.inf.upol.cz/
Content-Length: 232
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="https://www.inf.upol.cz/">here</a>.</p>
</body></html>
Connection closed by foreign host.
```

[8] https://en.wikipedia.org/wiki/Telnet

Když víme, jak vypadá komunikace mezi klientem a HTTP serverem, můžeme naprogramovat klienta v jazyce Java.

K vytvoření síťového spojení použijeme objekt třídy `Socket`, kterému v konstruktoru předáme adresu a port, kam se chceme připojit. Z tohoto objektu můžeme získat dva proudy dat (i) `OutputStream`, kterým posíláme data na server a (ii) `InputStream`, kterým čteme data ze serveru. Jelikož se jedná o textově orientovaný protokol, je dobré tyto proudy bytů převést na proudy znaků, tj. na objekty tříd `Reader` a `Writer`. V následujícím příkladu používáme ještě obalení do tříd `BufferedReader` a `BufferedWriter`, které jednotlivé operace ukládají do paměti (bufferu) a optimalizují tak přenos dat.

```
try (Socket socket = new Socket("www.inf.upol.cz", 80);
     InputStream inpStream = socket.getInputStream();
     OutputStream outStream = socket.getOutputStream();
     BufferedReader rd = new BufferedReader(new InputStreamReader(inpStream));
     BufferedWriter wr = new BufferedWriter(new OutputStreamWriter(outStream))) {

     /* komunikace se serverem */
}
```

Nyní máme vytvořené objekty pro komunkaci se serverem a můžeme začít předávat data. K odeslání použijeme objekt typu `BufferedWriter`, kterému předáme celý požadavek.

```
wr.write("GET / HTTP/1.0\r\n");
wr.write("Host: www.inf.upol.cz\r\n");
wr.write("\r\n");
wr.flush();
```

Důležité je zavolat metodu `BufferedWriter.flush()`, která zajistí, že je obsah bufferu vyprázdněn a odeslán na server.

Po odeslání požadavku již můžeme čekat na odpověd a číst ji z objektu typu `BufferedReader` stejně jako jsme četli data poskytnutá objektem `URL`.

```
String line;
while ((line = rd.readLine()) != null)
    System.out.println(line);
```

3 Jednoduchý síťový server

V předchozích dvou kapitolách jsme si ukázali, jak vytvořit síťového klienta pro protokol HTTP případně pro obecný textově orientovaný protokol. Nyní si ukážeme, jak vytvořit vlastní síťovou službu. Vytvoříme jednoduchý server, který bude přijímat řádky textu a bude vracet ty samé řádky, jen v nich převede znaky z malých písmen na velké.[9].

[9]Takto pojatá síťová služba není příliš užitečná, ale umožňuje nám ukázat princip síťové komunikace a převod na cokoliv užitečnějšího by neměl být nijak náročný.

Jak vytvořit server si ukážeme v několika krocích.

Nejdříve musíme vytvořit socket, na kterém náš server bude naslouchat jednotlivým klientům. K tomu sloužím třída `ServerSocket`, které v konstruktoru předáme port, na kterém má naslouchat.

```
ServerSocket srvSocket = new ServerSocket(4242);
```

Zavoláním metody `Serversocket.acccept()` začneme naslouchat na daném portu. Jinými slovy, tato metoda zablokuje běh aktuálního vlákna, dokud se nepřipojí nějaký klient. V takovém případě metoda vrátí objekt typu `Socket`, kterým můžeme komunikovat s daným klientem, jak ukazuje následující příklad, kde pro větší pohodlí, komunikaci balíme do tříd `BufferedReader` a `BufferedWriter`.

```
try (Socket clientSocket = srvSocket.accept();
     BufferedReader rd = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
     BufferedWriter wr = new BufferedWriter(new OutputStreamWriter(clientSocket.getOutputStream())

     processRequests(rd, wr);
}
```

Komunikace s klientem probíhá právě pomocí těchto objektů. Z objektu typu `Reader` čteme data od klienta, objektem typu `Writer` data odesíláme, jak ukazuje následující příklad.

```
private static void processRequests(BufferedReader rd, BufferedWriter wr) throws IOException {
    while (true) {
        String line = rd.readLine();
        if (line == null) break;

        if (line.equalsIgnoreCase("quit")) break;
        String resp = line.toUpperCase();
        wr.write(resp);
        wr.write('\n');
        wr.flush();
    }
}
```

Ve smyčcce načítáme jednotlivé řádky, a pokud klient ukončí spojení nebo pošle řetezec `quit`, smyčku ukončíme, jinak odešleme zpět řetezec, kde jsou písmena převedena z malých na velké.

Že naše služba funguje, si můžeme vyzkoušet buď pomocí programu telnet nebo si můžeme vytvořit vlastního klienta, jako v předchozí kapitole.

Nedá se však říct, že takto vytvořená služba funguje dobře, jelikož po ukončení prvního spojení služba přestane běžet. Je potřeba začít opět naslouchat na daném portu. Proto je vhodné dát volaní metody `ServerSocket.accept()` do smyčky.

```
while (!stopServer) {
    try (Socket clientSocket = srvSocket.accept();
```

```
        /* inicializace */) {
        /* zpracovani pozadavku */
    }
}
```

Jak bude ukončení služby (nastavení proměnné `stopServer`) vypadat, záleží na konkrétní aplikaci. V přiložené ukázkové aplikaci jsme pro maximální zjednodušení k ukončení serveru použili speciální požadavek od klienta, což není úplně dobré, protože každý může danou službu vypnout. V praxi by se to řešilo buď jiným komunikačním kanálem určeným k ovládání serveru nebo odchycením požadavku na ukončení virtuálního stroje.

Použitím smyčky, ve které čekáme na připojení, jsme vyřešili jeden problém, avšak naše služba stále nefunguje dobře. Vyzkoušejme se připojit současně se dvěma klienty. Komunikace bude probíhat jen s prvním připojeným klientem, druhý bude čekat v pořadí. Plyne to z povahy daného kódu.

Abychom mohli komunikovat s více klienty současně, je potřeba komunikaci s každým klientem řešit v samostatném vláknu. Jedná se o přímočarou úpravu, kdy `Socket`, který získáme, předáme vláknu, které si již řídí komunikaci samo.

```
Socket clientSocket = srvSocket.accept();
Thread serverThread = new ServerThread(clientSocket);
serverThread.start();
```

V přiloženém kódu najdete úplné řešení, to se příliš neliší od toho, co jsme viděli v předchozích příkladech, avšak je potřeba brát v potaz, že máme-li jakékoliv proměnné nebo data přístupná z více vláken je potřeba přístup k nim synchronizovat pomocí metod označených jako `synchronized`, viz přístup k atributu `stopServer`.