Návrh systémů IoT

3. Architektury a protokoly komunikačních rozhraní

Stanislav Vítek
Katedra radioelektroniky
České vysoké učení technické v Praze

Obsah přednášky

1. Formáty pro výměnu dat
2. HTTP protokol
3. Modely komunikace
4. Architektury komunikačních rozhraní

Formáty výměny dat

1. XML
2. JSON
3. BJSON
4. Protocol Buffers

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<book>
    <title>Python Basics</title>
    <page_count>635</page_count>
    <pub_date>2021-03-16</pub_date>
    <authors>
        <author>
            <name>David Amos</name>
        </author>
        <author><name>Joanna Jablonski</name></author>
        <author><name>Dan Bader</name></author>
        <author><name>Fletcher Heisler</name></author>
    </authors>
    <isbn13>978-1775093329</isbn13>
    <genre>Education</genre>
</book>
```

JSON (JavaScript Object Notation)

- Založen na podmnožině standardu programovacího jazyka JavaScript ECMA-262 3rd Edition.
- JSON je způsob ukládání a komunikace dat se specifickými pravidly (jako např. XML, YAML atd.)
- Soubory JSON mají příponu .json
- Používá dvojice klíč-hodnota
- Navržen tak, aby byl čitelný pro lidi i stroje a nezávislý na jazyce

http://json.org

https://json-schema.org/

https://jsonformatter.curiousconcept.com/

JSON příklad

```
{
   "members":[
      {
         "name":"Ringo Star",
         "alive":true,
         "birth":1940
      },
      {
         "name":"John Lennon",
         "alive":false,
         "birth":1940
      },
      {
         "name":"Paul McCartney",
         "alive":true,
         "birth":1942
      },
      {
         "name":"George Harrison",
         "alive":false,
         "birth":1943
      }
   ]
}
```

JSON - datové typy

- Řetězec
  - "Hello world!"
  - sekvence znaků v kódování Unicode
  - znak je reprezentován jako jednoznakový řetězec
- Číslo
  - 10 1.5 1.3e20
  - čísla typu integer, float, včetně čísel v exponenciálním tvaru
- Logický datový typ (boolean)
  - true false
- Null
  - hodnota může být null

JSON - datové typy

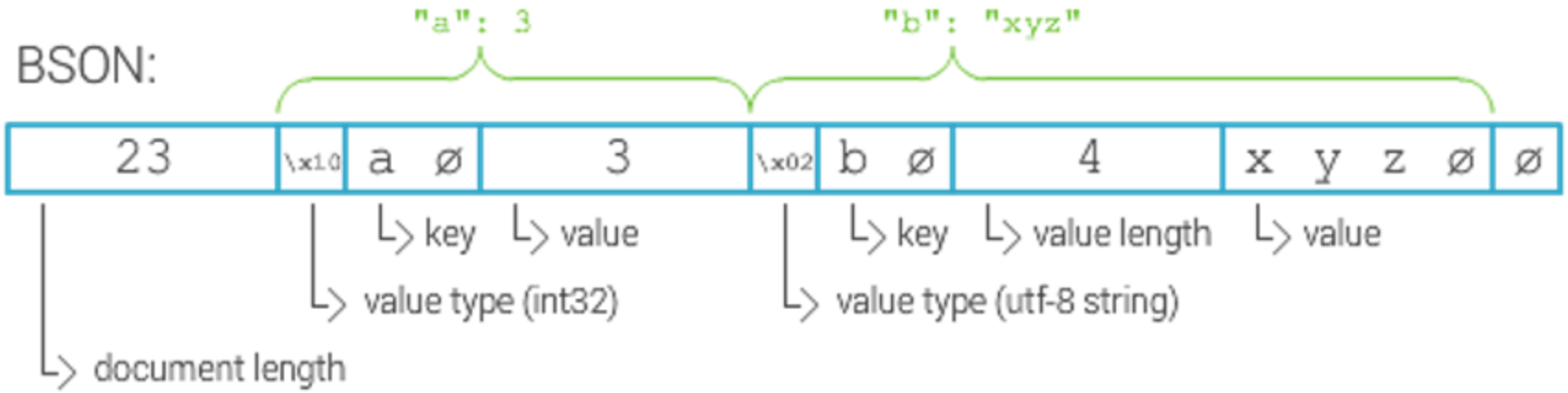
- Pole
  - [1, 2 3] ["Hello", "World"]
  - položky oddělené čárkou
  - položky nemusí být stejného datového typu
- Objekt
  - {"name": "John", "age": 43}
  - páry klíč-hodnota oddělené čárkou

BSON - Binary JSON

JSON:

```
{
  "a": 3,
  "b": "xyz"
}
```

BSON:



Protocol Buffers

- Jazykově neutrální formát dat pro binární serializaci dat
- Lze popsat název služeb, procedur a zpráv.
- Protocol Buffers se obvykle zkracují na Protobuf, soubory mají příponou .proto.
- Dokumentace zde.

```
syntax = "proto3";

message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
  repeated Product products = 4;
}
```

```
message Person {
    string name = 1;
    int32 id = 2;
    string email = 3;

    enum PhoneType {
        MOBILE = 0;
        HOME = 1;
        WORK = 2;
    }

    message PhoneNumber {
        string number = 1;
        PhoneType type = 2 [default = HOME];
    }

    repeated PhoneNumber phone = 4;
}
```

Jak Protobuf funguje?

- Vygenerování tříd potřebných pro čtení / zápis zpráv.
- Ke generování se používá kompilátor protoc.

```
protoc -I=$SRC_DIR --python_out=$DST_DIR $SRC_DIR/person.proto
```

```
import person_pb2
person = person_pb2.Person()
person.id = 1234
person.name = "John Doe"
person.email = "jdoe@example.com"
phone = person.phones.add()
phone.number = "555-4321"
phone.type = person_pb2.Person.HOME
```

Protobuf vs XML

- Protocol buffer:
  - je jednodušší
  - 3 - 10x menší
  - 20 - 100x rychlejší

```
<person>
    <name>John Doe</name>
    <email>jdoe@example.com</email>
</person>
```

```
person {
    name: "John Doe"
    email: "jdoe@example.com"
}
```

HTTP (Hypertext Transfer Protocol)

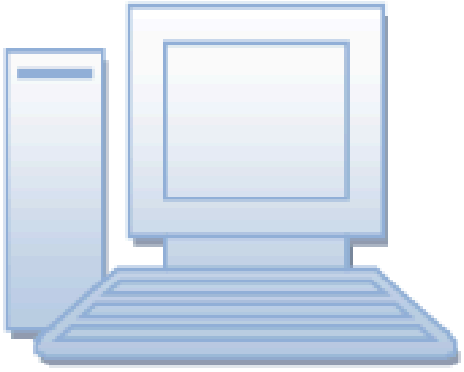
- Protokol pro přenos objektů libovolného typu (stránky, obrázky, …) mezi webovým serverem a prohlížečem
- Používá se i pro odesílání formulářových dat
- Jednoduchý aplikační protokol vystavený nad protokolem TCP, nepodporuje UDP
- Bezstavový protokol modelu požadavek/odpověď – přináší problémy pro webové aplikace
- Několik verzí – HTTP 0.9, HTTP 1.0, HTTP 1.1, HTTP/2, HTTP/3

Základní model protokolu

1. Navázání spojení
2. Zaslání požadavku klientem
3. Zaslání odpovědi serverem
4. Uzavření spojení

- Pro stránky s mnoha vloženými objekty (obrázky apod.) je tento způsob pomalý, a proto novější verze HTTP umožňují během jednoho spojení vyřídit několik požadavků/odpovědí

(1) User issues URL from a browser
http://host:port/path/file



(5) Browser formats the response and displays

Client (Browser)

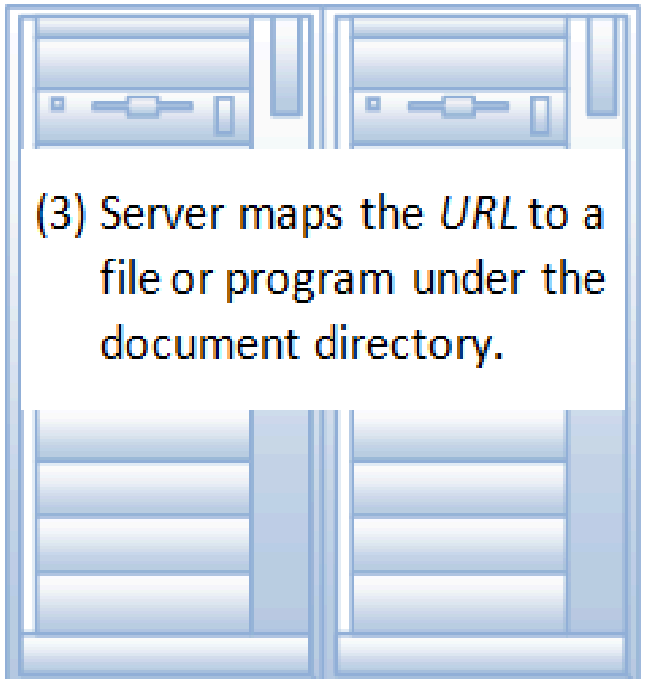
(2) Browser sends a request message

```
GET URL HTTP/1.1
Host: host:port
.................
.................
```

(4) Server returns a response message

```
HTTP/1.1 200 OK
.................
.................
.................
```

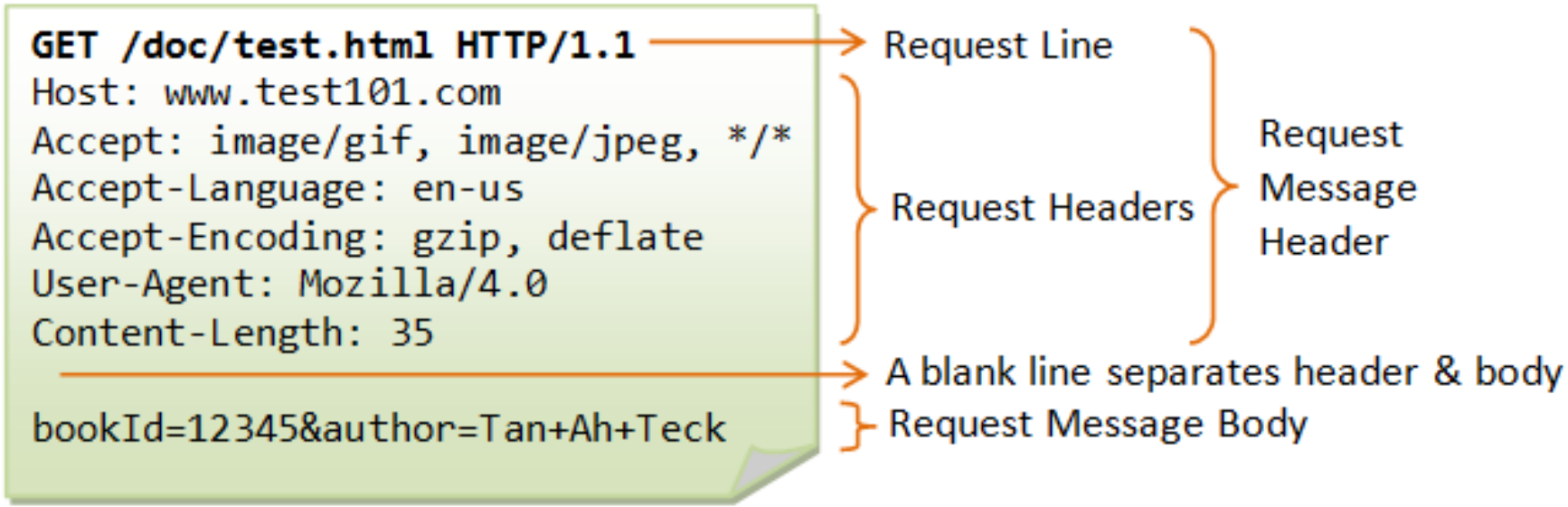
HTTP (Over TCP/IP)



Server (@ host:port)

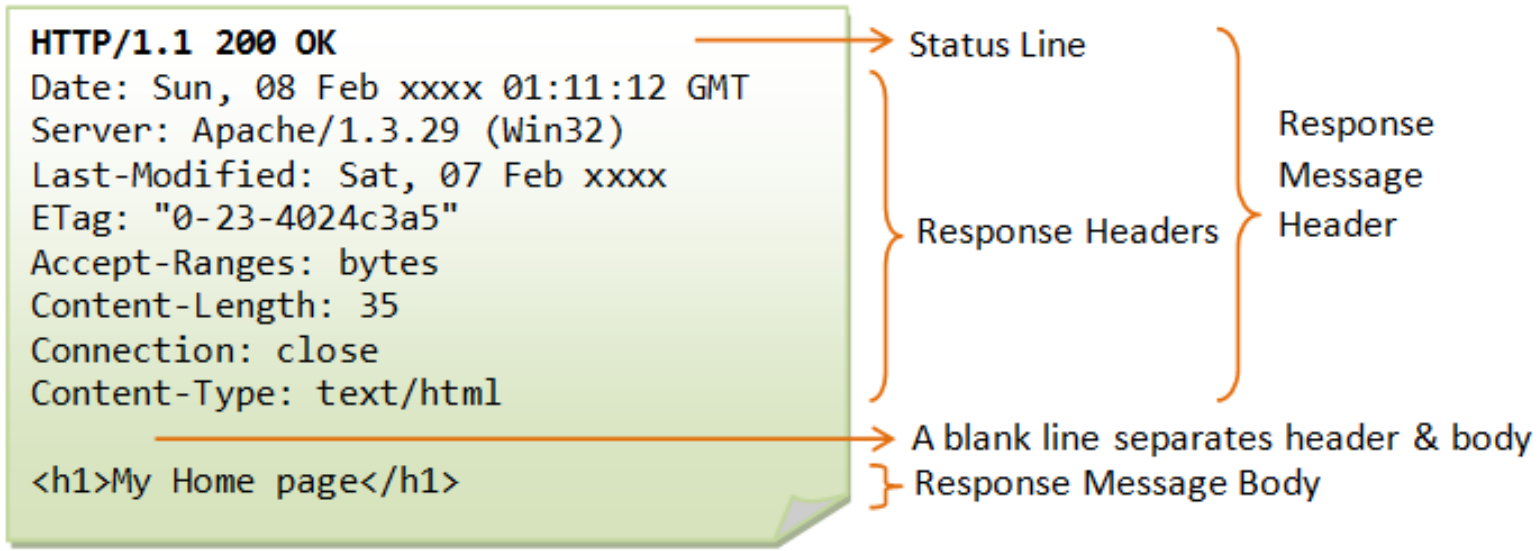
HTTP požadavek

- Hlavička
  - Request line - identifikuje HTTP metodu, URI a verzi protokolu
  - Request headers
- Tělo požadavku



HTTP odpověď

- Hlavička
  - Status line - identifikuje verzi protokolu a kód odpovědi
  - Response headers
- Tělo



HTTP metody

- **GET** Klient může použít požadavek GET k získání webového zdroje ze serveru.
- **HEAD** Klient může použít požadavek HEAD k získání hlavičky, kterou by získal požadavkem GET. Protože hlavička obsahuje datum poslední změny dat, lze ji použít ke kontrole proti místní kopii mezipaměti.
- **POST** Slouží k odeslání dat na webový server.
- **PUT** Požádá server o uložení dat.
- **DELETE** Požádá server o odstranění dat.
- **TRACE** Požádá server, aby vrátil diagnostickou stopu provedených akcí.
- **OPTIONS** Požádejte server, aby vrátil seznam metod požadavků, které podporuje.
- **CONNECT** Slouží k tomu, aby proxy server navázal spojení s jiným hostitelem a jednoduše odpověděl na obsah, aniž by se jej pokusil analyzovat nebo uložit do mezipaměti. Často se používá k navázání spojení SSL prostřednictvím proxy.

Kódy HTTP odpovědí

- 1xx - téměř nepoužívaný
- 2xx - úspěch
  - 200 OK - requests succeeded, usually contains data
  - 201 Created - returns a Location header for new resource
  - 202 Accepted - server received request and started processing
  - 204 No Content - request succeeded, nothing to return
- 3xx - přesměrování
  - 304 Not Modifed - resource not modifed, cached version can be used

- 4xx - client error
  - 400 Bad Request - malformed syntax
  - 401 Unauthorized - authentication required
  - 403 Forbidden - server has understood, but refuses request
  - 404 Not Found - resource not found
  - 405 Method Not Allowed - specified method is not supported
  - 409 Conflict - resource conflict with client data
  - 415 Unsupported Media Type - server does not support media type
- 5xx - server error
  - 500 Internal Server Error - server encountered error and failed to process request

Zpracování chybových kódů ve Flasku

```python
from flask import abort, render_template, request

# a username needs to be supplied in the query args
# a successful request would be like /profile?username=jack
@app.route("/profile")
def user_profile():
    username = request.arg.get("username")
    # if a username isn't supplied in the request, return a 400 bad request
    if username is None:
        abort(400)

    user = get_user(username=username)
    # if a user can't be found by their username, return 404 not found
    if user is None:
        abort(404)

    return render_template("profile.html", user=user)
```

- Zpracování 404 - **Page Not Found**

```
from flask import render_template

@app.errorhandler(404)
def page_not_found(e):
    # note that we set the 404 status explicitly
    return render_template('404.html'), 404
```

- Využití Application Factories

```
from flask import Flask, render_template

def page_not_found(e):
  return render_template('404.html'), 404

def create_app(config_filename):
    app = Flask(__name__)
    app.register_error_handler(404, page_not_found)
    return app
```

Předávání formulářových dat

1. Metoda GET
2. Metoda POST
3. Jakou metodu vybrat
4. Ukázka zpracování ve Flasku

Metoda GET

- Standardní metoda

```
<form method="GET">
  <input type="text" name="nazev1"/>
  <input type="text" name="nazev2"/>
  <input type="submit" name="Odeslat">
</form>
```

- před odesláním prohlížeč všechna data z formuláře zakóduje do jednoho dlouhého řetězce

```
název1=hodnota1&název1=hodnota2&...
```

- Hodnoty polí jsou upraveny tak, aby je šlo zapsat jako součást URL
  - mezera → +
  - speciální znaky, znaky s diakritikou apod. → %xx, kde xx je reprezentuje jednotlivé bajty z textu reprezentovaného v kódování UTF-8
  - zakódovaná data přidána za URL požadavku (za znak ?)
- webový server typicky předá skriptu data v proměnné prostředí QUERY_STRING
- Většina jazyků pro psaní webových aplikací však data zpřístupní pohodlnějším způsobem

Metoda POST

- Data se kódují podobně jako při použití metody GET
- Data se přenášejí v těle požadavku HTTP
- Webový server data předává skriptu na jeho standardní vstup
- Ve většině jazyků lze data číst pohodlně bez nutnosti parsovat standardní vstup

Výběr metody

GET

- odeslání formuláře lze simulovat pomocí zadání URL adresy
- vhodné pro operace, které nemění stav backendu
- lze uložit do záložek, poslat emailem, …

POST

- pro větší objemy dat (nevejdou se do URL)
- nutné pro operace měnící stav backendu

Další možnosti

- standardně se formuláře odesílají jako typ **application/x-www-form-urlencoded**
- při nahrávání souborů lze používá typ **multipart/form-data**
- metodu lze vybrat i ručně

```
<form action="..." method="post" enctype="multipart/form-data">
<!-- -->
</form>
```

Zpracování HTTP požadavků ve Flasku

K načtení dat zaslaných s požadavkem můžete použít následující atributy:

- **request.data** → Přístup k příchozím datům požadavku jako řetězci
- **request.args** → Přístup k analyzovaným parametrům URL.
- **request.form** → Přístup k parametrům formuláře.
- **request.values** → Kombinuje args a form
- **request.json** → Vrací rozparsovaná data JSON, pokud je mimetype application/json
- **request.files** → Vrací objekt MultiDict, který obsahuje všechny nahrané soubory. Každý klíč je název souboru a hodnota je objekt FileStorage.
- **request.authorization** → Vrací objekt třídy Authorization. Představuje hlavičku Authorization odeslanou klientem.

Příklad zpracování formuláře

- Formulář pro přihlášení do aplikace

```
<html>
   <body>
      <form action = "http://localhost:5000/login" method = "post">
         <p>Enter Name:</p>
         <p><input type = "text" name = "nm" /></p>
         <p><input type = "submit" value = "submit" /></p>
      </form>
   </body>
</html>
```

Zpracování formuláře

```
@app.route('/success/<name>')
def success(name):
   return 'welcome %s' % name

@app.route('/login',methods = ['POST', 'GET'])
def login():
   if request.method == 'POST':
      user = request.form['nm']
      return redirect(url_for('success',name = user))
   else:
      user = request.args.get('nm')
      return redirect(url_for('success',name = user))
```

HTTP hlavičky

- **Date**
  - datum a čas požadavku/odpovědi
- **Content-Type**
  - druh zasílaných dat
- **Host**
  - doménová adresa serveru
  - umožňuje správnou funkci více virtuálních serverů na jedné společné adrese
- **Location**
  - přesměrování na jinou stránku

Ovládání mezipaměti, proxy a načítání stránek

- **Cache-Control**
  - řízení proxy serverů a vyrovnávacích pamětí
- **Expires**
  - datum, kdy vyprší platnost stránky
- **If-Modified-Since**, **If-Unmodified-Since**, **If-None-Match**
  - podmíněné načtení stránky
- **Last-Modified**
  - datum poslední modifikace souboru

Identifikační údaje

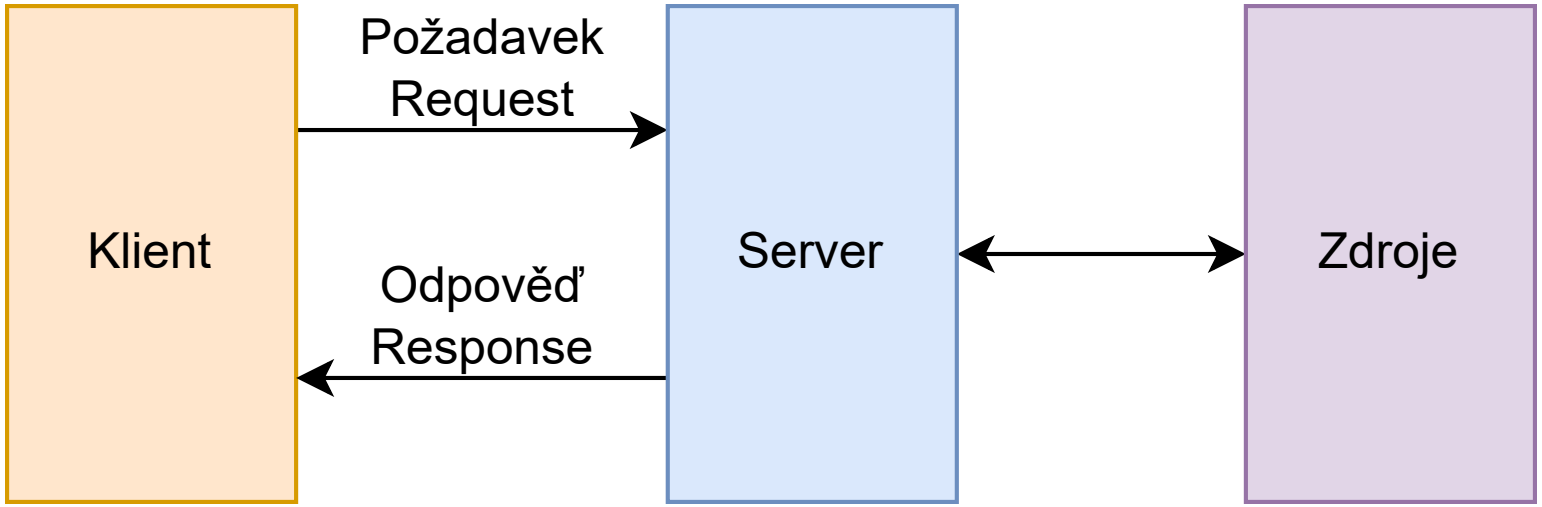
- **User-Agent**
  - identifikace klienta
- **Server**
  - identifikace serveru
- **Referer**
  - adresa stránky, kde bylo získáno URL právě kladeného požadavku (lze použít pro analýzu typu „odkud přišli“)

Modely komunikace

1. Request-Response
2. Publish-Subscribe
3. Push-Pull
4. Exclusive-Pair

Request-Response

- Aplikace (klient) posílá požadavky službě (server)
- Server obdrží požadavek, rozhodne se jak odpoví, načte data, načte zdroj reprezentace (šablona odpovědi), sestaví odpověď a odešle ji klientovi

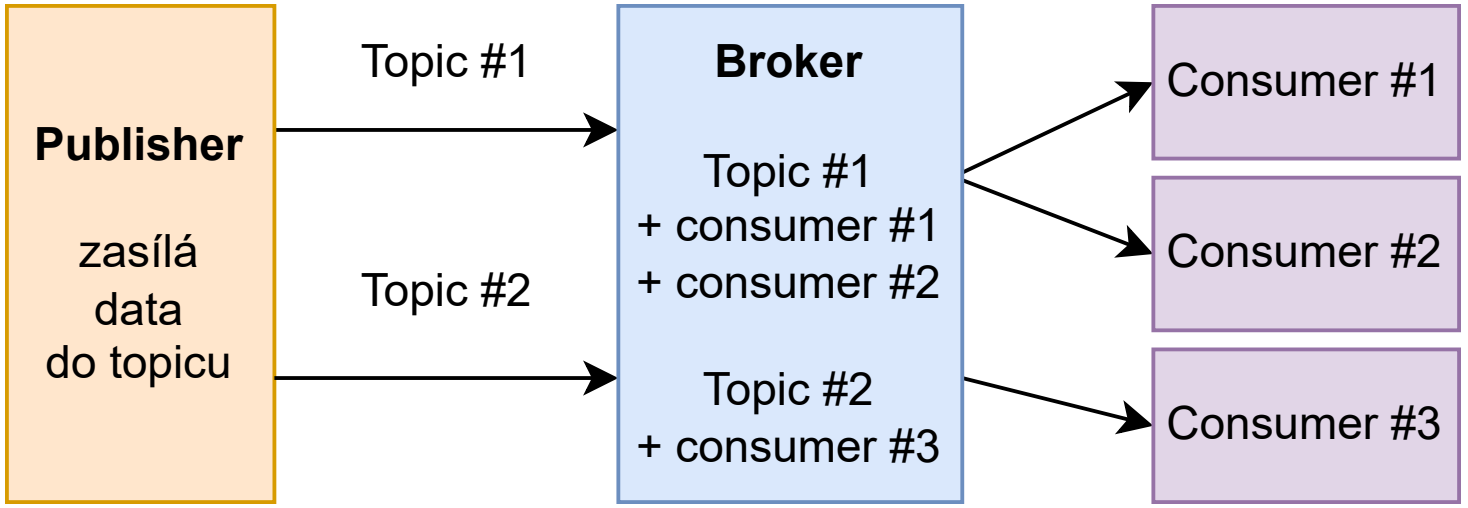


Vlastnosti modelu Request-Response

- Request-Response je bezstavový model.
  - Každá dvojice požadavek-odpověď je nezávislá na ostatních.
- Příkladem je protokol HTTP
  - HTTP funguje jako protokol dotaz-odpověď mezi klientem a serverem.
  - Klientem může být webový prohlížeč a serverem aplikace v počítači, která podporuje webové stránky.
  - Klient (prohlížeč) odešle serveru požadavek HTTP a server vrátí klientovi odpověď.
- Další příklady: CoAP (Constrained Application Protocol)

Publish-Subscribe

- Zahrnuje tři účastníky:
  - vydavatel (publisher) - posílá data v rámci téma (topic)
  - spotřebitel (consumer) - přihlašuje (subscribe) k odběru téma, o zdroji nic neví
  - zprostředkovatele (mediator, broker) - posílá spotřebiteli data v rámci téma
- Role vydavatel/spotřebitel jsou čistě logické



Vlastnosti modelu Publish-Subscribe

1. Interakce many-to-many

- Stejná informace může být doručena ve stejný okamžik různým spotřebitelům.
- Každý spotřebitel dostává informace od různých producentů.

2. Oddělení v prostoru

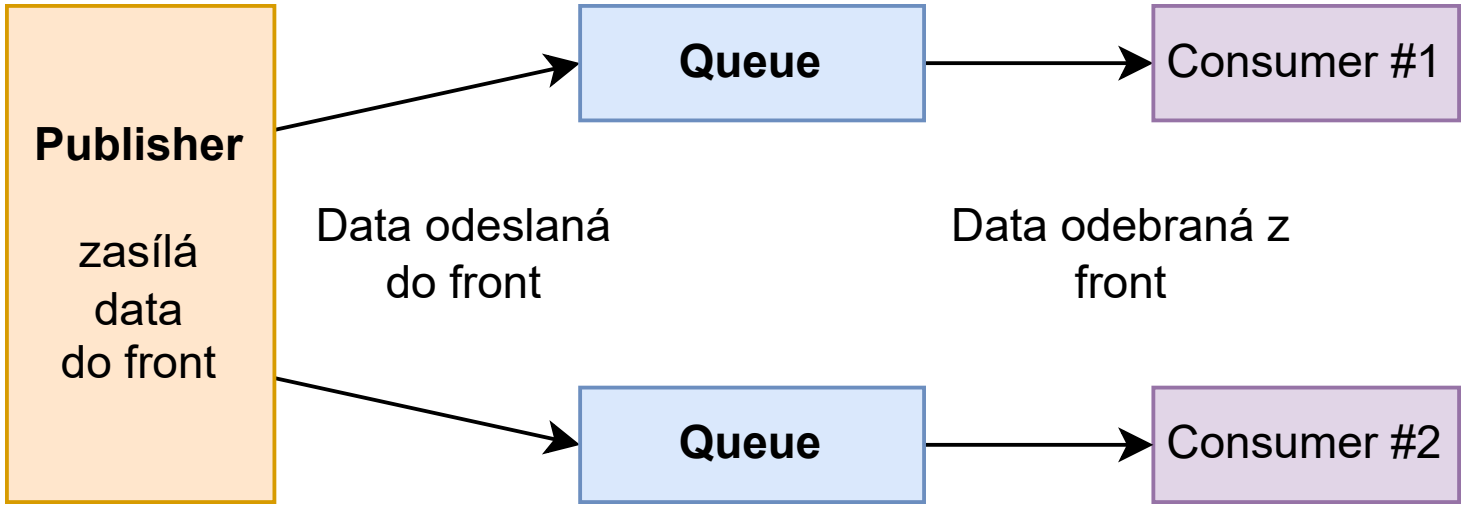
- Interagující strany se nemusí navzájem znát.
- Adresování zpráv je na základě jejich obsahu.

3. Oddělení v čase

- Interagující strany se nemusejí aktivně účastnit komunikace ve stejnou dobu.

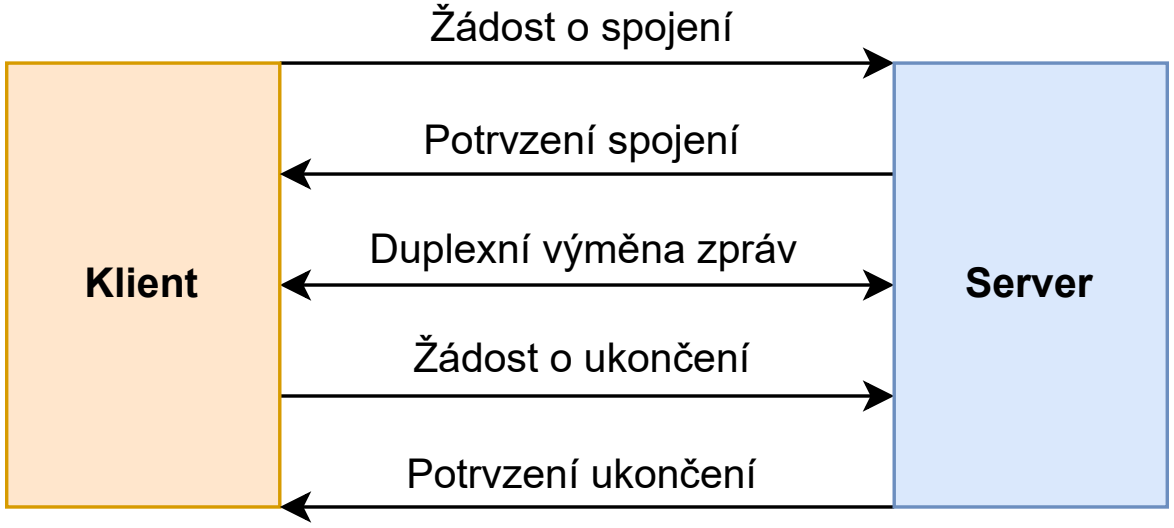
Push-Pull

- Producenti dat odesílaji data do front a konzumenti je z front odebírají
- Producenti nemusí vědět o konzumentech
- Fronty fungují jako vyrovnávací paměť, řeší nesoulad rychlostí vkládání a odebírání



Exclusive-Pair

- Obousměrný, plně duplexní, používá trvalé spojení mezi klientem a serverem.
- Spojení zůstává otevřené, dokud klient odešle požadavek na ukončení.
- Klient a server mohou po navázání spojení navzájem posílat zprávy .



Architektury komunikačních rozhraní

1. API
2. REST
3. GraphQL
4. SOAP
5. RPC & gRPC
6. WebSocket
7. WebHook

API (Application Programming Interface)

- API označuje rozhraní pro programování aplikací
- Jsou to mosty, které propojují různé systémy a umožňují efektivní výměnu dat, volání funkcí a celkovou integraci.
- Styly architektury API definují vzory a struktury, které určují, jak jsou rozhraní API navrhována a organizována.
- Tyto styly poskytují rámec, který zajišťuje konzistenci, škálovatelnost a udržovatelnost projektů vývoje rozhraní API.

REST (Representational State Transfer)

- Zaměřen na zdroje systému (Resources) jejich adresaci a přenos.
- Zdroje jsou vytvářeny (**Create**), čteny (**Read**), aktualizovány (**Update**) a mazány (**Delete**) - **CRUD** - pomocí bezestavového protokolu HTTP
- Klient přistupuje ke zdrojům prostřednitvím koncového bodu URI
- REST využívá HTTP metody
  - **GET** - získání existujího zdroje
  - **POST** - vytvoření nového zdroje
  - **PUT** - aktualizace existujícího zdroje
  - **PATCH** - částečná aktualizace existujícího zdroje
  - **DELETE** - smazání zdroje

Vlastnosti REST

- **Bezestavový** - server neudržuje žádný stav mezi požadavky od klienta.
- **Klient-server architektura** - klient a server musí být od sebe odděleny, aby se mohly vyvíjet nezávisle.
- **Možnost ukládání do mezipaměti** - data získaná ze serveru by měla být kešovatelná buď klientem, nebo serverem.
- **Jednotné rozhraní** - server poskytuje jednotné rozhraní pro přístup ke zdrojům bez definování jejich reprezentace.
- **Vrstvený systém** - klient může přistupovat ke zdrojům na serveru nepřímo prostřednictvím dalších vrstev, jako je proxy nebo load balancer.
- **Kód na vyžádání** (volitelný) - server může klientovi předat kód, který může spustit, například JavaScript pro jednostránkovou aplikaci.

- **Výhody**
  - Je jednoduchý a snadno se používá a chápe.
  - Řídí se pravidly webu a používá stávající standardy a protokoly.
  - Je rychlý a zvládne mnoho požadavků, protože podporuje ukládání do mezipaměti a bezstavovost.
  - Je flexibilní a může používat různé formáty a typy médií.
- **Nevýhody**
  - Nemá jasnou strukturu ani schéma, což může způsobit jeho nepřehlednost a nekonzistentnost.
  - Nepodporuje složité dotazy nebo operace, kvůli čemuž může potřebovat mnoho požadavků a získat příliš mnoho nebo příliš málo dat.
  - Nepracuje dobře s chybami nebo výjimkami, protože používá stavové kódy HTTP, které nejsou vždy jasné nebo správné.

Bezestavový síťový protokol

- Klient pošle serveru požadavek a server pošle zpět odpověď podle aktuálního stavu
- Server není povinen uchovávat informace o relaci nebo stavu každého komunikačního partnera pro více požadavků.
- Po havárii není třeba obnovovat žádný stav, selhávající server lze restartovat.
- Příklady: HTTP, UDP (User Datagram Protocol), DNS (Domain Name System).

Stavový síťový protokol

- Pokud server neodpoví na požadavek, klient pošle požadavek znovu.
- Servery musí uchovávat informace o stavu a další podrobnosti o relaci, takže je možné v případě selhání komunikace přenos obnovit
- Příklady: FTP (File Transfer Protocol), Telnet.

GraphQL

- Technologie, která vznikla ve společnosti Facebook, ale nyní je open-source.
- Základním mechanismem pro provádění dotazů a mutací je HTTP metoda POST.
- Jak název napovídá, GraphQL je určena k reprezentaci dat v grafu.
    - Grafovou databázi si můžete představit jako rozsáhlou množinu objektů (uzlů), které spolu souvisejí různým způsobem (hrany).
- Graf je definován podle jazyka schémat, který je specifický pro jazyk GraphQL.
- Příklad jednoduchého schématu jazyka GraphQL si můžete prohlédnout zde.

Ke GraphQL se vrátíme v 5. přednášce o databázích.

SOAP (Simple Object Access Protocol)

- Protokol pro výměnu informací zakódovaných v jazyce XML mezi klientem a procedurou nebo službou, která se nachází na internetu.
- Specifikace byla zveřejněna v roce 1999 a jako otevřený standard ji publikuje W3C.
- Kromě HTTP lze používat i další přenosové protokoly, například FTP a SMTP. (Obvyklým vzorem je použití protokolu HTTP pro synchronní výměnu dat a protokolu SMTP nebo FTP pro asynchronní interakce).
- Pro podporu konzistence při strukturování dat používá SOAP standardní schéma XML (XSL) pro kódování XML. Kromě toho mohou vývojáři vytvářet vlastní schémata XML a přidávat do zpráv SOAP vlastní prvky XML.

WSDL (Web Service Description Language)

- SOAP se obvykle používá s jazykem WSDL.
- Význam použití WSDL spočívá v tom, že vývojáři a stroje mohou kontrolovat webovou službu, která podporuje SOAP, a zjistit tak specifika výměny informací po síti.
- Kromě toho WSDL popisuje, jak strukturovat zprávy s požadavky a odpověďmi SOAP, které daná služba podporuje.
- Zjišťování pomocí WSDL zjednodušuje programování webových služeb využívajících protokol SOAP.

Stuktura SOAP zprávy

- Hierarchická struktura, jejímž kořenovým prvkem je **<soap:Envelope>**.
- Může mít tři podřízené elementy **<soap:Header>**, **<soap:Body>** a **<soap:Fault>**.
- Element **<soap:Body>** je povinný.
- Prvky **<soap:Header>** a **<soap:Fault>** jsou nepovinné.
- Pokud je použit nepovinný prvek **<soap:Header>**, musí být prvním dceřiným prvkem v rámci nadřazeného prvku **<soap:Envelope>**, a pokud je použit nepovinný prvek **<soap:Fault>**, musí být dcem prvku **<soap:Body>**.

Příklad SOAP zprávy

```
POST /BobsTickers HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 275
SOAPAction: "http://cooltickers.org/soap"

<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
  xmlns:m="http://www.exampletickers.org">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetStockPriceRequest>
      <m:StockName>IBM</m:StockName>
    </m:GetStockPriceRequest>
  </soap:Body>
</soap:Envelope>
```

SOAP v Pythonu - knihovna requests

```
import requests
url = "http://webservices.oorsprong.org/websamples.countryinfo/CountryInfoService.wso"

payload = """
<?xml version=\"1.0\" encoding=\"utf-8\"?>
<soap:Envelope xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\">
  <soap:Body>
    <CountryIntPhoneCode xmlns=\"http://www.oorsprong.org/websamples.countryinfo\">
      <sCountryISOCode>IN</sCountryISOCode>
    </CountryIntPhoneCode>
  </soap:Body>
</soap:Envelope>"""
# headers
headers = {
  'Content-Type': 'text/xml; charset=utf-8'
}
# POST request
response = requests.request("POST", url, headers=headers, data=payload)
# prints the response
print(response.text)
print(response)
```

RPC - Remote Procedure Call

- RPC umožňuje lepší definici sémantiky
- Napodobuje volání lokálních procedur přes síťové rozhraní
- Je velmi častou volbou v případě architektury mikroslužeb
- Volba protokolu je libovolná (protocol agnostic), lze přizpůsobit aplikaci
  - Při použití HTTP se používají pouze metody GET a POST
- Významnou implementací je gRPC od Google
- Rozmazává hranici mezi serverem a klientem - server může být klientem a klient serverem (protože prostě jen volají procedury)

gRPC

- Technologie vyvinutá společností Google a později zpřístupněná jako open-source.
- Stejně jako GraphQL je to specifikace, která je implementována v různých jazycích.
- Na rozdíl od REST a GraphQL, které používají textové formáty dat, gRPC používá binární formát Protocol Buffers.
- Je navržen pro vysoce výkonnou komunikaci s nízkou latencí.
- Podporuje obousměrné streamování, takže je ideální pro aplikace v reálném čase.
- Dobře se hodí pro architekturu mikroslužeb.

Nevýhody gRPC

- Složitost: Nastavení a konfigurace gRPC může být v porovnání s REST složitější, zejména pro vývojáře, kteří nemají zkušenosti s vyrovnávacími paměťmi protokolů a binární serializací.
- Podpora jazyků: Přestože gRPC podporuje více programovacích jazyků, nemusí být tak univerzálně podporován jako REST, což omezuje výběr jazyků pro klienty.
- Velikost požadavků a odpovědí: Binární serializace používaná protokolem gRPC může vést ke kompaktnějším zprávám, ale nemusí být nejlepší volbou pro aplikace, které potřebují vyměňovat rozsáhlé textové dokumenty.
- Zvýšená režie: Další funkce poskytované protokolem gRPC, jako je obousměrné streamování a vyrovnávání zátěže, mohou přinést dodatečnou režii, která může v některých scénářích ovlivnit výkon.

Websocket

- Protokol, který umožňuje plně duplexní obousměrné komunikační kanály prostřednictvím jediného připojení TCP.
- Je ideální pro aplikace v reálném čase, jako je chat, online hry a platformy pro finanční obchodování. Mezi klíčové vlastnosti protokolu WebSocket patří:
    - Nízká latence: WebSocket API poskytuje nízkou latenci, takže je ideální pro interaktivní aplikace.
    - Obousměrná komunikace: Komunikaci může zahájit klient i server.
    - Jediné připojení: Udržuje jediné dlouhodobé spojení.
- WebSocket je klíčový pro aplikace, které vyžadují okamžité aktualizace a interaktivitu v reálném čase.

Nevýhody

- Udržování stavu v připojeních WebSocket může být složité, zejména ve scénářích s vysokým počtem připojených klientů.
- Připojení WebSocket spotřebovávají prostředky serveru a velký počet otevřených připojení může zatížit výkon serveru.
- Podpora WebSocket nemusí být univerzální ve všech prohlížečích a platformách, což vyžaduje záložní mechanismy pro zajištění kompatibility.
- WebSocket přináší určitou režii protokolu, která nemusí být pro určité případy použití tak efektivní jako gRPC, zejména pokud se jedná o vysokofrekvenční komunikaci nebo velmi velké užitečné zatížení.

Webhook

- Umožňuje aplikacím odesílat data v reálném čase jiným aplikacím, když dojde k určitým událostem.
- Fungují na modelu publish-subscribe a upozorňují odběratele na příslušné události.
- Mezi klíčové vlastnosti patří:
  - **Řízení událostí:** Webhooks jsou zásadní pro architektury řízené událostmi, které zajišťují aktualizace v reálném čase.
  - **Asynchronní:** Umožňují asynchronní komunikaci mezi aplikacemi.
  - **Přizpůsobitelná integrace:** Webhooks lze přizpůsobit konkrétním případům použití a typům událostí.

Jak Webook pracuje

- Odesílatel zjistí událost nebo změnu, o které chce příjemce informovat.
- Odesílatel vytvoří požadavek HTTP POST obsahující příslušné informace o události.
- Odesílatel odešle tento POST požadavek na předem definovanou adresu URL, kterou mu poskytne příjemce. Tato adresa URL se často označuje jako koncový bod webhooku.
- Příjemce naslouchá příchozím požadavkům HTTP POST na koncovém bodě webhook.
- Když příjemce přijme požadavek POST, zpracuje data v užitečném zatížení požadavku a provede potřebné akce na základě informací o události.