Kurs rozszerzony języka Python

Wykład 11.

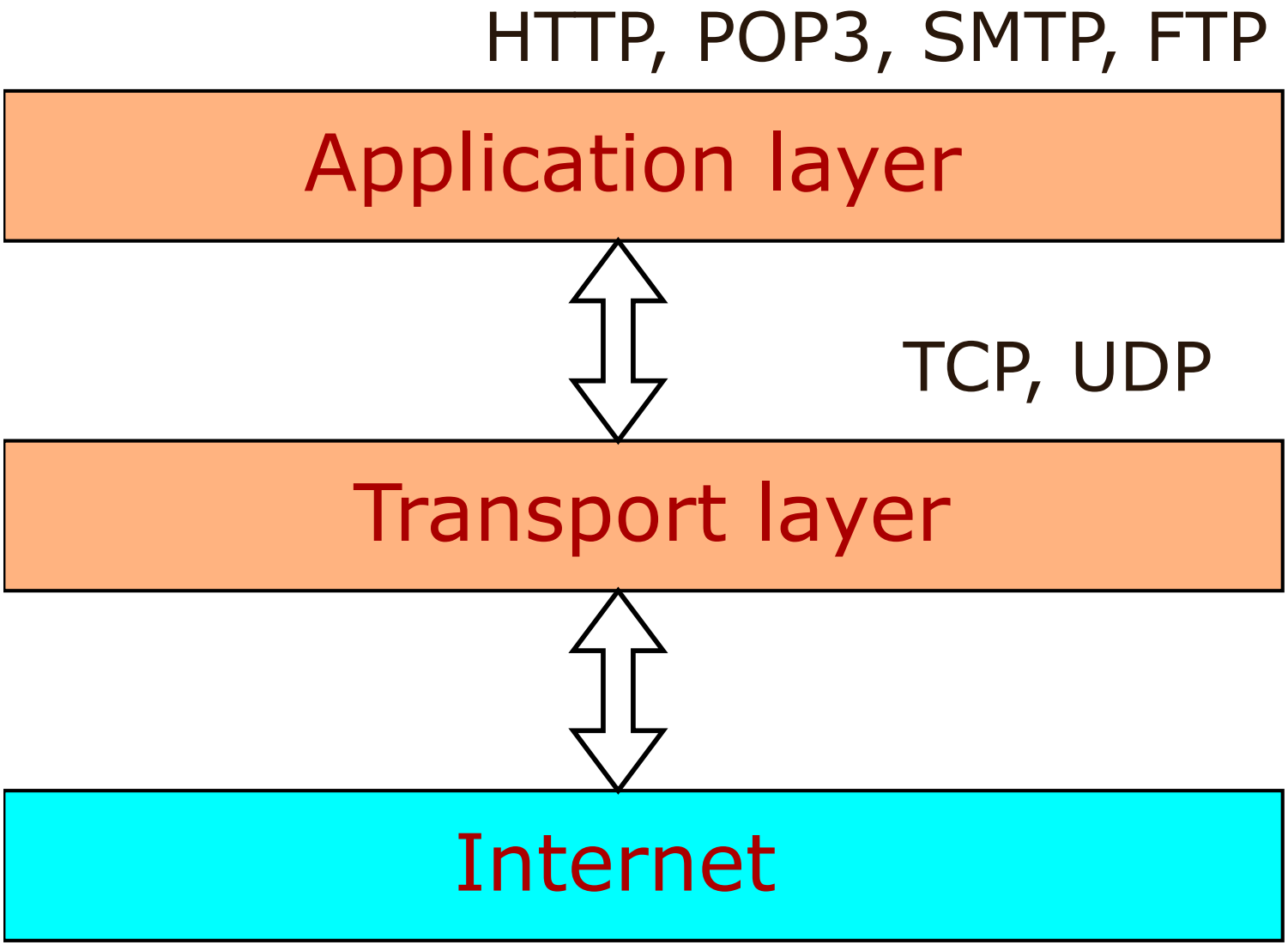
Marcin Młotkowski

17 grudnia 2024

Plan wykładu

Plan wykładu

Schemat sieci: przypomnienie



Protokół UDP

Cechy protokołu

- Protokół jest bardzo prosty
- Brak kontroli dostarczonych komunikatów
- Stosowany tylko w lokalnych (niezawodnych) sieciach
- Komunikacja za pomocą gniazd

Przykład

Zadanie

Przesłać z komputera (nadawca/klient) do komputera (odbiorca/serwer) komunikat "Hello Python".

Nadawca

Implementacja

```
import socket

port = 8081
host = "localhost"

s = socket.socket(socket.AF_INET,
                  socket.SOCK_DGRAM)
s.sendto("Hello, Python!!!", (host, port))
```

Odbiorca

Implementacja

```
import socket
port = 8081

s = socket.socket(socket.AF_INET,
                  socket.SOCK_DGRAM)
s.bind(("", port)
print("Nasłuch na porcie", port)
while True:
    data, addr = s.recvfrom(1024)
    print("Nadawca", addr, "dane", data)
```

Zastosowania

Video streaming

Odczyt z kamery

```
import cv2 # pip install opencv-python

kamera = cv2.VideoCapure(0)

status, frame = kamera.read()
```

`status`: czy udało się poprawnie odczytać
`frame`: numpy.ndarray (u mnie 640 x 480 x 3)

Spakowanie i odczyt danych

Spakowanie danych

```
import pickle

data = pickle.dump(frame)
```

Na początku powinna być długość danych jako 4-bajtowa liczba

```
import pack

header = struct.pack("Q", len(data))

message = header + data
```

Nadawca

```
import cv2
import pickle, struct

kamera = cv2.VideoCapture(0)
client_socket = socket.socket(socket.AF_INET,
    socket.SOCK_STREAM)
client_socket.connect(('127.0.0.1', 8080))

while True:
    status, frame = kamera.read()
    data = pickle.dumps(frame)
    message = struct.pack("Q", len(data)) + data
    client_socket.sendall(message)
```

Odbiorca

```
server_socket = socket.socket(socket.AF_INET,
    socket.SOCK_STREAM)
server_socket.bind(('0.0.0.0', 8080))
server_socket.listen()
connection, addr = server_socket.accept()

payload_size = struct.calcsize("Q")

data = b""

while True:
    ...
```

Odczyt danych

Uwagi:

- będziemy odczytywać strumień danych w porcjach po 4KB;
- w jednej porcji danych może być koniec jednej klatki i początek następnej;
- nie znamy faktycznego rozmiaru klatki (rozmiar może się zmieniać);
- wiemy tylko, że rozmiar nagłówka to `struct.calcsize("Q")`

Odbiorca

```
while len(data) < payload_size:
    packet = connection.recv(4 * 1024)  # 4K buffer si
    if not packet:
        break
    data += packet

packed_msg_size = data[:payload_size]
data = data[payload_size:]
msg_size = struct.unpack("Q", packed_msg_size)[0]

while len(data) < msg_size:
    data += connection.recv(4 * 1024)

frame_data = data[:msg_size]
data = data[msg_size:]
frame = pickle.loads(frame_data)

cv2.imshow('Receiver', frame)
```

"Prawdziwe" serwery

```
socketserver.TCPServer
socketserver.UDPServer
```

Serwery wielowątkowe

```
# Pomocnicze klasy w module socketserver
socketserver.ForkingMixIn
socketserver.ThreadingMixIn
```

Prawdziwy serwer wielowątkowy:

```
class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    ...
```

Plan wykładu

Proste zadanie

Zdalne monitorowanie pracy wielu komputerów za pomocą przeglądarki:

- Na każdym komputerze jest uruchomiony serwer http;
- żądanie jakiejś strony powoduje wykonanie odpowiedniej akcji, np.:

  http://host/uptime

  powoduje wykonanie polecenia `uptime` i zwrócenie outputu polecenia do przeglądarki;
- domyślnie jest wysyłana lista dostępnych funkcji.

Szczegóły protokołu http, żądanie

Klient

GET / HTTP/1.1
Host: www.ii.uni.wroc.pl
User-Agent: Mozilla/5.0

Szczegóły protokołu http, odpowiedź

Serwer

HTTP/1.1 200 OK
Date: Mon, 15 Dec 2023 11:14:01 GMT
Server: Apache/2.0.54 (Debian GNU/Linux)
Content-Length: 37402

<dane>

Obsługa HTTP

Serwer webowy: obiekt klasy `http.server.HTTPServer`

- Obsługuje protokół http;
- **nie** obsługuje żądań.

Obsługa HTTP

Serwer webowy: obiekt klasy `http.server.HTTPServer`

- Obsługuje protokół http;
- **nie** obsługuje żądań.

Obsługa żądań: klasa `http.server.SimpleHTTPRequestHandler`

- klasa bazowa do rozbudowy własnej funkcjonalności;
- metody obsługujące żądania (GET, POST, HEADER,...);
- metody konstrukcji odpowiedzi.

Serwer

```
from http.server import HTTPServer,
         SimpleHTTPRequestHandler
import os

class MyHttpHandler(SimpleHTTPRequestHandler):
```

Implementacja klasy MyHttpHandler

```
def do_GET(self):
    self.send_response(200)
    self.send_header('Content-type', 'text/html')
    self.end_headers()

    self.wfile.write(b'<html><head><title>test</title></hea
    self.wfile.write(b'<body>')
    if self.path == '/uptime': self.uptime()
    else: self.menu()

    self.wfile.write(b'</body></html>')
```

Implementacja uptime

```
def uptime(self):
    res = bytes(os.popen("uptime").read(), 'utf-8')
    self.wfile.write(b'<h1>Rezultat</h1>')
    self.wfile.write(b'<tt>' + res + b'</tt>')
```

Implementacja serwera

```
def menu(self):
    self.wfile.write(b'<h1>Serwer</h1>')
    self.wfile.write(b'<ul>')
    self.wfile.write(b'<li><a href="uptime">uptime</a></li>
    self.wfile.write(b'</ul>')
```

Serwer http

Uruchomienie całego serwera

```
address = ('', 8000)
httpd = HTTPServer(address, MyHttpHandler)
httpd.serve_forever()
```

Co to jest

Framework do obsługi różnych protokołów sieciowych. Oparty jest na paradygmacie *sterowania zdarzeniami*.

Obsługa żądań

```
from twisted.internet import reactor
from twisted.web import http

class MyRequestHandler(http.Request):

  def process(self):
    self.setHeader('Content-type', 'text/html')

    self.write(b'<html><head><title>test</title></head>')
    self.write(b'<body>')
    self.write(self.path)
    print(self)
    if self.path == b'/uptime': self.uptime()
    else: self.menu()

    self.write(b'</body></html>')
    self.finish()

  def uptime(self): pass
```

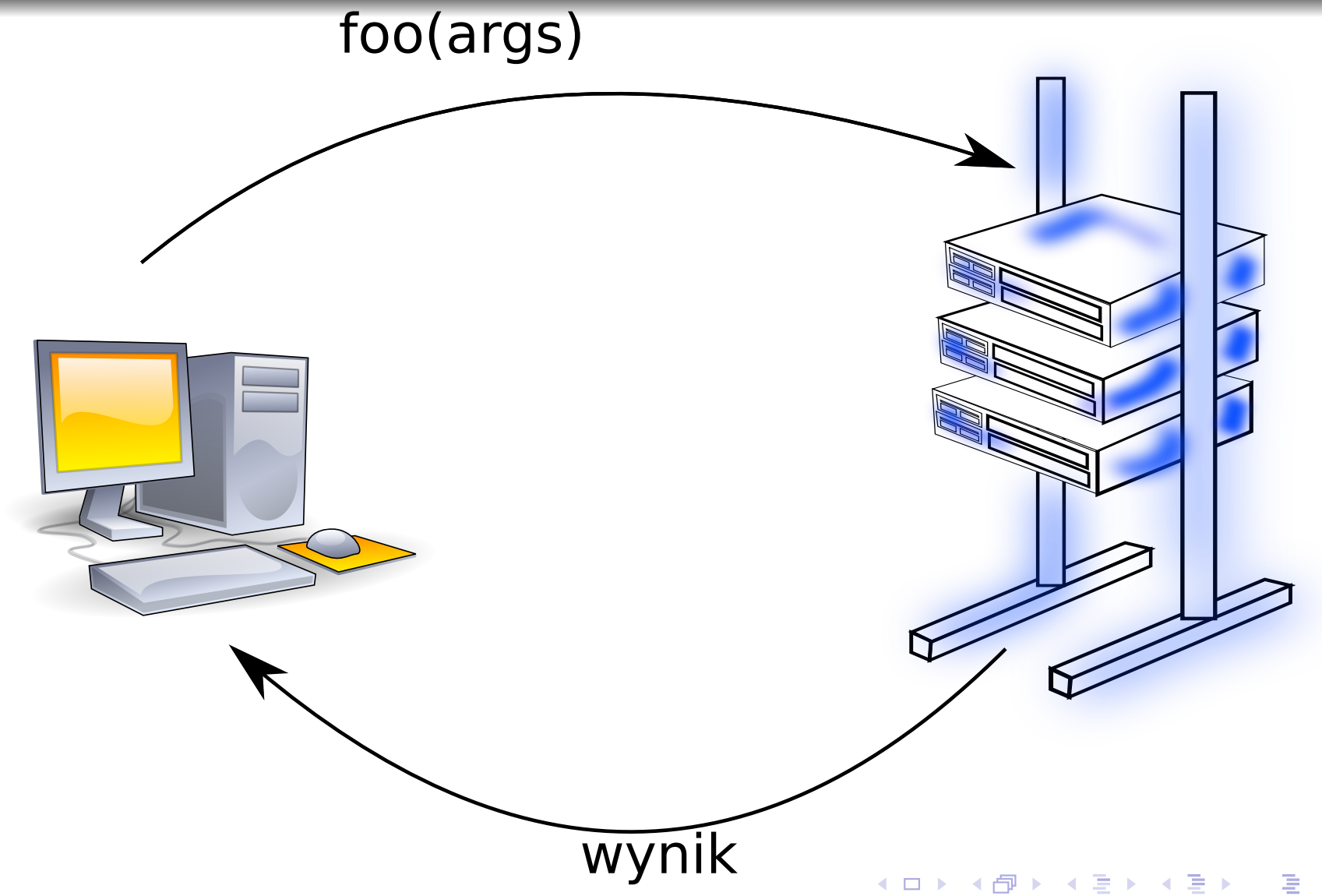
Uruchomienie serwera

```
class MyHTTP(http.HTTPChannel):
    requestFactory = MyRequestHandler


class HTTPServerFactory(http.HTTPFactory):
    def buildProtocol(self, addr):
        return MyHTTP()

reactor.listenTCP(8001, HTTPServerFactory())
reactor.run()
```

Serwery aplikacyjne



Wykorzystywane protokoły

- General InterORB Protocol
- Remote Java Invocation
- RPC
- .NET Remoting
- XML RPC
- ...

Protokół XML-RPC

Postać komunikatu:

```
<?xml version="1.0"?>
<methodCall>
  <methodName>ack</methodName>
  <params>
    <param>
    <value><i4>2</i4></value>
    <value><i4>2</i4></value>
    </param>
  </params>
</methodCall>
```

Protokół XML-RPC

Sposób przesyłania komunikatu: żądanie HTTP

Zadanie

Serwer obliczający zdalnie *n*-tą liczbę Fibonacciego

Serwer

```
from xmlrpc.server import SimpleXMLRPCServer
```

Implementacja funkcjonalności

```
def fib(n):
    if n < 2: return 1
    return fib(n - 1) + fib(n - 2)
```

Serwer

```
from xmlrpc.server import SimpleXMLRPCServer
```

Implementacja funkcjonalności

```
def fib(n):
    if n < 2: return 1
    return fib(n - 1) + fib(n - 2)
```

Implementacja serwera

```
from SimpleXMLRPCServer import *

server = SimpleXMLRPCServer(('localhost', 8002))
server.register_function(fib)
server.register_function(lambda x, y: x + y, 'add')
server.serve_forever()
```

Klient

Implementacja

```
import xmlrpc.client
server = xmlrpc.client.ServerProxy("http://localhost:8002")
print(server.fib(10))
print(server.add(2,3))
```

Serwer wielowątkowy

Wielodziedziczenie

```
from socketserver import ThreadingMixIn
from xmlrpc.server import  SimpleXMLRPCServer
class ThreadedXMLRPCServer(
  ThreadingMixIn,
  SimpleXMLRPCServer):
      pass

server = ThreadedXMLRPCServer(('', 8003))

server.register_function(ack, "ack")

print(f"Serwer XML RPC nasluchuje na porcie 8003")
server.serve_forever()
```

Plan wykładu

Flask: mikroframework do tworzenia aplikacji webowych

Flask: mikroframework do tworzenia aplikacji webowych

Pinterest, LinkedIn

Przykład

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def main():
    return "<p>Kurs programowania w Pythonie</p>"

@app.route("/wyklad12")
def wyklad():
    return "<p>Usługi sieciowe</p>"
```

Uruchomienie

```
$ flask run
```

Architektura REST

REST

Representational state transfer

Web service API

RESTful

Representational state transfer

Jak to wygląda w praktyce

Operacja	żądanie http
Create	PUT
Read	GET
Update	POST
Delete	DELETE

Odwołania do elementów

Pobranie elementu `osoba` o identyfikatorze `ide`: żądanie GET

`http://localhost/osoba/15`

Jako wynik jest zwracany json zawierający osobę bądź komunikat błędu (też w formacie json).

Nowy element

Utworzenie elementu `osoba`: żądanie PUT

`http://localhost/osoba/`

Jako wynik jest zwracany json zawierający nową osobę bądź komunikat błędu (też w formacie json).

Implementacja protokołu we Flasku

Wysłanie żądanego obiektu

```
import json
from flask import Flask, request, jsonify

@app.route('/osoba/<int:ide>', methods=['GET'])
def get_osoba(ide):
    print(ide)
    return jsonify({'imie': 'Maksymilian',
                    'nazwisko': 'Debeściak'})
```

Klient, żądanie pobrania nowej osoby

```
res = requests.get("http://localhost:5000/osoba/123")
print(res.json())
```

Utworzenie nowego elementu

```
@app.route('/osoba/', methods=['PUT'])
def new_osoba():
    print("nowa osoba")
    return jsonify({'msg': 'nowa osoba'})
```

Klient, żądanie utworzenia nowej osoby

```
res = requests.put("http://localhost:5000/osoba")
print(res.json())
```

Plan wykładu