PSI Zespół 57 – Serwer CoAP

Autorzy: Michał Machnikowski, Maksymilian Nowak, Bruno Sienkiewicz

Treść zadania

Celem projektu jest implementacja serwera CoAP w języku Python, który będzie obsługiwał przynajmniej żądania `GET`, `POST`, `PUT`, oraz `DELETE`. Serwer powinien posiadać odpowiednio skonstruowaną architekturę – tak, aby łatwo było dodać nową funkcję obsługującą żądania przychodzące na dany URL.

Założenia

Funkcjonalne

- Serwer będzie obsługiwał żądania zgodnie z protokołem CoAP
- Serwer będzie obsługiwał żądania `GET`, `POST`, `PUT`, oraz `DELETE`
- Serwer będzie zwracał odpowiednie kody odpowiedzi
- Serwer będzie generował logi ze wszystkimi zdarzeniami, które wystąpiły podczas jego działania
- Serwer będzie organizował dane w hierachię zasobów, gdzie każdy zasób będzie miał swój unikalny URL
- Serwer będzie pozwalał na obsługę wielu klientów jednocześnie

Niefunkcjonalne

- Serwer ma architekturę pozwalającą na łatwą rozbudowę aplikacji o nowe żądania
- Serwer pozwala na jednoczesną obsługę wielu klientów bez utraty wydajności
- Serwer ma poprawnie działać na systemie Linux

Przypadki użycia

Serwer będzie służył do obsługi żądań do sieci urządzeń IoT. Przykładowo, zakładając, że serwer będzie obsługiwał czujniki temperatury, to możliwe przypadki użycia to:

- `GET`: Odczytywanie bieżącej temperatury na czujniku
- `POST`: Zarejestrowanie nowego czujnika
- `PUT`: Aktualizacja temperatury czujnika
- `DELETE`: Usunięcie czujnika

Analiza sytuacji błędnych

W przypadku wystąpienia sytuacji błędnej, serwer zwróci odpowiedni dla tej sytuacji kod, zgodnie ze specyfikacją RFC 7252:

- Kod `4.xx` dla błędów po stronie klienta
- Kod `5.xx` dla błędów po stronie serwera

Każdy błąd będzie zapisany w logach do dalszej analizy; rekord będzie zawierał podstawowe informacje, takie jak data i godzina zdarzenia, adres IP klienta, adres URL żądania oraz kod uzyskany w odpowiedzi.

Środowisko sprzętowo-programowe

- System operacyjny:
  - Ubuntu 24.04
- Konteneryzacja:
  - Docker
  - `docker compose`
- Język programowania:
  - Python 3.11
- Zarządzanie pakietami:
  - `poetry`
- Linter:
  - `ruff`
- Formatter:
  - `ruff`
- Analiza statyczna:
  - `mypy`
- Testowanie:
  - `pytest`

Architektura rozwiązania

Do symulacji architektury rozwiązania zastosujemy Dockera.

Poszczególne kontenery będą reprezentować elementy architektury, które będą komunikować się za pomocą sieci docekrowej.

API

W projekcie planowane są trzy główne bloki funkcjonalne:

- Uruchomienie serwera
- Nasłuchiwanie przychodzących żądań
- Obsługa żądań

Sposób testowania

- Testy jednostkowe
- Testy integracyjne
  - poprawność komunikacji klient-serwer
  - stabilność systemu
- Testy manualne, z opisanym ich przebiegiem w dokumentacji końcowej

Podział prac w zespole

- Przygotowanie środowiska – Bruno Sienkiewicz
- Szkielet aplikacji – Bruno Sienkiewicz, Maksymilian Nowak, Michał Machnikowski
- Metoda `GET` (+ testy) – Michał Machnikowski
- Metoda `POST` (+ testy) – Maksymilian Nowak

- Metoda `PUT` (+ testy) – Michał Machnikowski
- Metoda `DELETE` (+ testy) – Bruno Sienkiewicz
- Dokumentacja – Michał Machnikowski, Maksymilian Nowak, Bruno Sienkiewicz

Przewidywane funkcje do zademonstrowania w ramach odbioru częściowego

- Szkielet aplikacji
- Deklaracja każdej z metod (mocki)
- W pełni działająca metoda `GET`

Plan pracy z podziałem na tygodnie

- 23.12 – 29.12: Przygotowanie środowiska
- 30.12 – 05.01: Stworzenie szkieletu aplikacji
- 06.01 – 12.01: Implementacja metody `GET`
- 13.01 – 19.01: Implementacja pozostałych metod, przygotowanie testów
- 20.01 – 24.01: Weryfikacja poprawności rozwiązania, przygotowanie dokumentacji końcowej

Opis protokołu

Protokół CoAP jest protokołem binarnym bazującym na protokole UDP, zaprojektowanym do użytku na urządzeniach o ograniczonych zasobach sprzętowych, w szczególności dla urządzeń IoT. Wzorowany jest na HTTP i używa podobnej struktury zasobów jak model REST. Żądania oznaczone są kodami `0.*`, po których zwracana jest odpowiedź z jednym z 3 typów kodów.

Przetwarzanie wiadomości

Struktura wiadomości:

- Nagłówek, 4 bajty
  - Wersja
  - Typ wiadomości
    - żądanie, potwierdzane lub niepotwierdzane
    - odpowiedź, z danymi lub dane dosłane później
  - Długość tokena
  - Kod wiadomości
  - Identyfikator wiadomości
- Token, od 0 do 8 bajtów
- Opcje (opcjonalne)
- Payload (opcjonalne)
  - Zaczyna się za znacznikiem `0xFF`

Po przetworzeniu zapytania zwracana jest odpowiedź lub błąd, pomyślne odpowiedzi oznaczone są kodem `2.*`, błędy po stronie klienta `4.*` i błędy po stronie serwera `5.*`.

Instrukcja uruchomienia

1. Sklonuj repozytorium

```
git clone https://gitlab-stud.elka.pw.edu.pl/mmachnik/psi-projekt.git
```

2. Przejdź do katalogu z projektem

```
cd psi-projekt
```

Za pomocą dockera

3. Zbuduj obraz

```
docker build -t coap_server .
```

4. Uruchom kontener

```
docker run -p 5683:5683 coap_server
```

Za pomocą pythona

3. Zainstaluj wszystkie potrzebne zależności:

```
make install
```

4. Uruchom serwer:

```
make run
```

Opis najważniejszych struktur i funkcji

`server.py`

Jest to plik zawierający definicję głównej klasy serwera, `CoAPServer`. Klasa ta przyjmuje jako parametry wejściowe strukturę `routes`, zawierającą informacje o dostępnych zasobach, `host` z adresem IP serwera oraz `port` z numerem portu, na którym serwer ma nasłuchiwać.

Klasa posiada dwie metody – `start()` oraz `shutdown()`. Pierwsza z nich odpowiada za uruchomienie serwera, natomiast druga za jego zatrzymanie.

`request_handler.py`

Plik ten zawiera definicję klasy `RequestHandler`. W konstruktorze klasy przekazywana jest struktura `routes`, analogicznie do klasy `CoAPServer`. Klasa ta posiada metodę `handle_request()`, przyjmujący jako parametr wejściowy ciąg bajtów reprezentujący żądanie CoAP. Metoda ta zwraca odpowiedź serwera w postaci ciągu bajtów.

Przetworzenie żądania wykorzystuje funkcje `parse_message()` oraz `encode_message()`, które zostaną opisane poniżej.

`utils/parser.py`

`parse_message()`

Funkcja ta przyjmuje jako parametr wejściowy ciąg bajtów reprezentujący żądanie CoAP. Funkcja ta zwraca strukturę typu `CoapMessage`, zawierającą informacje o żądaniu.

`encode_message()`

Funkcja ta przyjmuje jako parametr wejściowy strukturę `CoapMessage`, zawierającą informacje o odpowiedzi serwera. Funkcja ta zwraca ciąg bajtów reprezentujący odpowiedź serwera.

`utils/construct_response.py`

Plik ten zawiera funkcję `construct_response()`, wykorzystywaną przez konkretne zasoby. Przyjmuje ona żądanie w postaci struktury `CoapMessage`, kod odpowiedzi oraz treść odpowiedzi. Funkcja ta zwraca strukturę `CoapMessage` reprezentującą odpowiedź serwera.

Opis interfejsu użytkownika

Serwer CoAP nie posiada interfejsu użytkownika. Po uruchomieniu serwera widoczne są jedynie logi zdarzeń, które mają miejsce podczas jego działania.

Flagi konfiguracyjne

Nasza implementacja serwera CoAP posiada flagi konfiguracyjne, podawane w postaci argumentów wiersza poleceń. Są to:

- `--host` – adres IP serwera
- `--port` – numer portu, na którym serwer ma nasłuchiwać
- `--verbose` – poziom szczegółowości logów
  - `-v` – tylko ostrzeżenia
  - `-vv` – ostrzeżenia oraz logi informacyjne
  - `-vvv` – ostrzeżenia, logi informacyjne oraz debugowe

Postać logów

Logi mają następujący format:

```
<timestamp> [<log_level>] <message>
```

Gdzie:

- `timestamp` – data i godzina zdarzenia
- `log_level` – poziom logów (DEBUG, INFO, WARNING, ERROR, CRITICAL)
- `message` – treść logu

Oto przykładowy zapis logów:

```
2025-01-24 08:50:01,812 [DEBUG] Received data from ('127.0.0.1', 48990)
2025-01-24 08:50:01,812 [DEBUG] Handling request: CoapMessage(header_version=1,
header_type=0, header_token_length=4, header_code=<CoapCode.GET: '0.01'>,
header_mid=1337, token=b'1234', options={<CoapOption.URI_PATH: 11>: b'/sensors'},
payload=b'')
2025-01-24 08:50:01,815 [INFO] Received GET request for URI: /sensors
2025-01-24 08:50:01,815 [DEBUG] Returning all sensor data
2025-01-24 08:50:01,815 [INFO] Request to /sensors handled successfully
2025-01-24 08:50:01,815 [DEBUG] Sent response to ('127.0.0.1', 48990)
```

Wykorzystane narzędzia

- `Docker` – konteneryzacja aplikacji
- `poetry` – zarządzanie zależnościami
- `ruff` – linter oraz formatter
- `mypy` – analiza statyczna
- `pytest` – testy jednostkowe oraz integracyjne
- `typer` – tworzenie interfejsu CLI

Narzędzie CLI

W celu ułatwienia wysyłania żądań do serwera CoAP, przygotowaliśmy narzędzie CLI, przypominające narzędzie `curl`. Narzędzie to pozwala na wysyłanie żądań `GET`, `POST`, `PUT`, oraz `DELETE` do serwera CoAP.

Uruchomienie

Aby uruchomić narzędzie CLI, należy wykonać poniższą komendę:

```
python cli.py <uri> [options]
```

Gdzie:

- `uri` – adres URL serwera CoAP
- `options` – dodatkowe opcje, zależne od żądania:
  - `--method` – metoda żądania (`GET`, `POST`, `PUT`, `DELETE`)
  - `--data` – dane do przesłania w żądaniu (dla `POST` i `PUT`)
  - `--verbose` – poziom szczegółowości logów
    - `-v` – tylko ostrzeżenia
    - `-vv` – ostrzeżenia oraz logi informacyjne

- `-vvv` – ostrzeżenia, logi informacyjne oraz debugowe

Testy

Testy jednostkowe

Nasz projekt zawiera testy jednostkowe tylko dla funkcji parsujących wiadomości CoAP. Testy te sprawdzają poprawność działania funkcji `parse_message()` oraz `encode_message()`.

Testy integracyjne

Testy integracyjne sprawdzają poprawność komunikacji klient-serwer. Symulują one wysłanie żądania do serwera, a następnie sprawdzają, czy serwer poprawnie odpowiedział na dane żądanie i czy zasoby zostały poprawnie zaktualizowane.

Dodatkowo testy integracyjne sprawdzają stabilność systemu, czyli czy serwer poprawnie obsługuje wielu klientów jednocześnie.

Testy manualne

Testy manualne polegają na ręcznym wysłaniu żądań do serwera CoAP za pomocą narzędzia CLI oraz sprawdzeniu, czy serwer poprawnie odpowiedział na dane żądanie.

Oto przykładowy test manualny:

1. Uruchomienie serwera:

```
make run
```

2. Wysłanie żądania GET do zasobu `/sensors`:

```
poetry run python cli.py coap://localhost:5683/sensors
```

lub

```
source .venv/bin/activate
python cli.py coap://localhost:5683/sensors
```

3. Otrzymane rezultaty:

Klient:

```
(psi) (coap-server-py3.11) 09:06:22 maks@RYZEN:~/psi-projekt$ python cli.py
coap://localhost:5683/sensors
```

```
Response Code: 2.05 Content
Data: {"1": {"name": "Sensor 1", "temperature": 21}, "2": {"name": "Sensor 2",
"temperature": 25}}
(psi) (coap-server-py3.11) 09:48:58 maks@RYZEN:~/psi-projekt$
```

Serwer:

```
2025-01-24 09:48:51,001 [INFO] Starting CoAP Server on 127.0.0.1:5683
2025-01-24 09:48:51,001 [INFO] CoAP Server started on 127.0.0.1:5683
2025-01-24 09:48:57,998 [DEBUG] Received data from ('127.0.0.1', 40865)
2025-01-24 09:48:57,999 [DEBUG] Handling request: CoapMessage(header_version=1,
header_type=0, header_token_length=4, header_code=<CoapCode.GET: '0.01'>,
header_mid=1337, token=b'1234', options={<CoapOption.URI_PATH: 11>: b'/sensors'},
payload=b'')
2025-01-24 09:48:57,999 [INFO] Received GET request for URI: /sensors
2025-01-24 09:48:57,999 [DEBUG] Returning all sensor data
2025-01-24 09:48:57,999 [INFO] Request to /sensors handled successfully
2025-01-24 09:48:57,999 [DEBUG] Sent response to ('127.0.0.1', 40865)
```

Żądanie zostało obsłużone poprawnie i serwer zwrócił kod odpowiedzi `2.05 Content` oraz listę dostępnych czujników.

Dodatkowo, w celu weryfikacji poprawności działania naszego narzędzia CLI, uruchomiliśmy ten sam test, korzystając z zewnętrznej biblioteki klienckiej `aiocoap` (`cli_aiocoap.py`). W odpowiedzi otrzymaliśmy ten sam rezultat, co potwierdza poprawne zaimplementowanie protokołu CoAP.

Pokrycie

Pokrycie testami kodu naszego serwera wynosi 82%. W pliku `htmlcov/index.html` (wygenerowanym przez `make test`) można w czytelny sposób zobaczyć, ile wynosi pokrycie poszczególnych plików oraz które dokładnie linie nie są pokryte.

Podsumowanie

- Nasz serwer CoAP działa poprawnie i obsługuje żądania `GET`, `POST`, `PUT`, oraz `DELETE`.
- Posiada on modularną architekturę, pozwalającą na łatwą rozbudowę żądań.
- Serwer również poprawnie obsługuję wielu klientów jednocześnie.
- Spełnione zostały wszystkie założenia funkcjonalne oraz niefunkcjonalne.