Mateusz Ratajczak 136790
Dominik Rolewski 136792

Prosty serwer HTTP zgodny z RFC 2616 co najmniej w zakresie żądań GET, HEAD, PUT, DELETE

Opis protokołu:

Request HTTP:
<Nazwa Methody HTTP> <URL> HTTP/1.1\r\n
Content-Length: <długość>\r\n
\r\n
<request body>

Response HTTP:
<HTTP/1.1 <Response-code> <Request-name>\n
Content-type: <text/html>/<application/json>\n
\n
<response body>

Request czytany od użytkownika jest bajt po bajcie przy użyciu funkcji read. W przypadku metody POST, PUT aż do momentu wykrycia, zakończenia request body czyli znaku „}", natomiast w pozostałych do wykrycia \r\n\r\n. Tak przeczytany request przesyłany jest do procedury build_request gdzie jest wywoływana odpowiednia funkcja HTTP wywołana przez klienta lub zwraca błąd w przypadku uzycia niezaimplementwanej metody. Serwer korzysta z 4 mutexów:
- mutex → do synchronizacji na zmiennych warunkowych
- mutex_writer → do synchronizacji pisarzy
- var_reader_mutex → do blokowania aktualizacji zmiennej zliczającej czytelników
- var_writer_mutex → do blokowania aktualizacji zmiennej zliczającej pisarzy
Oraz 2 zmiennych warunkowych, usypianych przy uzyciu metody wait i wybudzany przez signal.
- cond_read → jeżeli pisarz zmienia źródło danych to wątek wykonywany przez czytelnika jest zawieszany na powyższej zmiennej. Wybudzony jest on na koniec wykonania procesu przez pisarza.

- cond → jeżeli jakikolwiek czytelnik wykorzystuje dane, czyli readers > 0 to wątek pisarza jest zawieszany. Signal na zmienną cond wysyłany jest przez wątek czytelnika, gdy readers == 0;

Nasz serwer umożliwia wielowątkowy odczyt z plików bazy, dzięki czemu pliki nie są monopolizowane przez jednego użytkownika, a serwer jest szybszy, niż w przypadku blokowania plików przez klienta.

Inspiracją do zaimplementowanego modelu wzajemnego wykluczania był dla nas problem czytelników i pisarzy. Nasze rozwiazanie tego faworyzuje czytelników, wynika to z faktu, że pisarz może dokonać zmian jedynie w momencie, gdy żaden inny czytelnik nie jest obsługiwany przez serwer. Przyjeliśmy to rozwiązanie jako bardziej korzystne, gdyż do tego typu serwerów jak nasz z większą częstotliwością wysyłane są zapytania GET i HEAD niż te administrujące dane czyli PUT, DELETE lub POST. Dzięki temu, mamy pewność że w pierwszej kolejności serwer obsłuży mniej zasobożerne żadanie, czyli zwyczajne odczytanie i wysłanie danych. Request wysyłany jest również znak po znaku.

Opis implementacji:

Naszym zadaniem było wykonanie prostego serwera HTTP zdolnego wykonywać żądania GET, HEAD, PUT, DELETE.

W naszym rozwiązaniu każda z powyższych metod została zaimplementowana w osobnej funkcji. W części main zawarliśmy podstawowe elementy tworzenia serwera TCP. Pozostałe funkcje dotyczą samej obsługi żadań wysyłanych przez klienta oraz tworzenia nowych wątków aplikacji.

Mateusz Ratajczak 136790
Dominik Rolewski 136792

GET: (wywoływane bez ciała) - reader

Pozwala wylistować całą listę książek, bądź konkretny element z listy. W naszym podejściu, przy kompletowaniu odpowiedzi serwer tworzy pliki odpowiedzi indywidualne dla każdego z uzytkowników, do których zapisuje response, czyli kod response'a i jego treść w postaci strony HTML lub rzeczywiste dane przeczcytane z pliku JSON. Funkcja wykona się gdy żaden writer nie dokonuje modyfikacji danych.

HEAD: (wywoływane bez ciała) - reader

Po wpisaniu adresu niezależnie czy odwołując się do konkretnej książki czy do całego zbioru, zwraca nagłówek informujący o powodzeniu bądź błędzie wykonania. Funkcja wykona się gdy żaden writer nie dokonuje modyfikacji danych.

PUT: (wywoływana z ciałem) - writer

Pozwala dodawać nowe książki do zbioru, a także modyfikować już istniejące. Jeśli podamy w body zapytania rekord o nowym id zostanie on dodany do bazy, jeśli podamy rekord o numerze id istniejącym w bazie, taki rekord zostanie zmodyfikowany. Metoda ta zakłada chwilową blokadę na modfikowanym pliku aby zapewnić pozostałym klientom spójność odczytywanych danych. Wykonywane jest to przy uzyciu niezbędnych zmiennych warunkowych, które w odpowiednim momencie usypiają wątek lub go wybudzają.

DELETE (wywoływane bez Ciała) - writer

Usuwa daną pozycję z bazy danych oraz zwraca zaktualizowaną listę wszystkich książek bez usuniętej pozycji. Na chwilę plik jest blokowany aż do momentu zakończenia usuwania podanego rekordu.

Dodatkowo zaimplementowaliśmy obsługę **POST** (wywoływana z ciałem) - writer

Pozwala na dodanie nowego rekordu do bazy danych. Niezależnie od wejścia ta metoda zawsze zwróci ten sam wynik, na takich samych danych.

Opis sposobu kompilacji i uruchomienia projektu:

W celu kompilacji należy otworzyć folder z projektem. Następnie, będąc w katalogu głównym projektu, należy wykonać w terminalu komendę `cd /src`

Kolejnym krokiem jest kompilacja projektu poprzez wykonanie polecenia:

```
gcc –pthread server.c –Wall –o server
```

Następnie można uruchomić server przy pomocy polecenia `./server`

Uruchomienie klienta

Klientem naszej aplikacji jest aplikacja Postman bądź przeglądarka internetowa. W aplikacji klienta wystarczy podać adres ip serwera, dla lokalnego serwera jest to adres 127.0.0.1 bądź inny w przypadku komunikacji poprzez sieć oraz wskazać numer portu. Nasza aplikacja działa na porcie nr. 8080.

Nasza aplikacja wykonuje polecenia na zbiorze rekordów, zawierających opis książek, które znajdują się w pliku .json.

W pliku README.md na gitlabie umieściliśmy sposoby wykonania zapytań przy użyciu aplikacji Postman oraz lokalnego adresu IP.