Sprawozdanie z projektu

Konrad Szychowiak 144564
Julia Auguścik 145172

Temat: System wymiany komunikatów publish/subscribe

Opis protokołu komunikacyjnego

Wiadomości przesyłane przez tcp mają następującą postać:

- Każda wiadomość zaczyna się od symbolu typu komunikatu i kończy znakiem `;`

```
<wiadomość> := <symbol-typu> <lista-argumentów> ;
```

- Po symbolu znajdują się argumenty, zależne of typu:

```
<lista-argumentów> := \t string
                    | \t string <lista-argumentów>
```

  argumenty są ciągami znaków oddzielonymi od siebie znakiem tabluacji `\t`

Typy komunikatów

symbol	rozwinięcie	argumenty	wysyła
`C`	***C**reate conversation*	`name: string` – tytuł konwersacji, `uuid: string` – identyfikator nadany przez autora	klient
`D`	***D**elete conversation*	`id: int` – identyfikator konwersacji	klient
`S`	***S**ubscribe*	`id: int` – identyfikator konwersacji	klient
`U`	***U**nsubscribe*	`id: int` – identyfikator konwersacji	klient
`P`	***P**ost message*	`id: int` – identyfikator konwersacji, `content: string` – treść wiadomości	klient
`N`	***N**ew message* (created)	`id: int` – identyfikator konwersacji, `content: string` – treść wiadomości	serwer
`L`	***L**ist of conversations*	**ciąg następujących po sobie trójek**: `id: int` – identyfikator konwersacji, `name: string` – tytuł konwersacji, `uuid: string` (j/w)	serwer

- `id` oznacza identyfikator taki, jaki trzyma i używa serwer
- `int` oznacza, że przesłany ciąg znaków interpretowany będzie jako liczba

---

- po stronie klienta interpretacja przychodzących wiadomości ma miejsce w `TcpMiddleware.onData()` w pliku `client/src/tcp/TcpMiddleware.ts` a wiadomości tworzone są przez `Pack(.ts)`
- po stronie klienta wiadomosci odbiera `ThreadBehavior` w `server/src/main.cpp` a wysyłają klasy oparte na klasie `Listener` na podstawie wiadomości stworzonych przez `Visitor`-ów.

Struktura projektu - opis implementacji

Serwer

> serwer oparty na C++20

1. Wątek główny (funkcja `main`) uruchamia serwer i rozpoczyna nasłuchiwanie.
   - serwer tworzony jest przez klasę `Server` opartą na klasie `Socket`
   - serwer odbiera połącznie używając funkcji `accept()` i przekazuje sterowanie do funkcji `connectionHandlerFactory`
2. `connectionHandlerFactory` zapisuje do struktury `thread_data_t` deskryptor gniazda połączenia a następnie uruchamia funkcję `ThreadBehavior` w nowym wątku

3. `ThreadBehavior` odpowiada za obsługę operacji wykonywanych przez pojedynczego klienta:
    - rejestruje `ConversationsListener`, który zostanie powiadomiony przy każdej zmainie w liście konwersacji (dodanie/usunięcie)
    - nasłuchuje w pętli na komunikaty przychodzące na socket używając funckji `read()` a następnie wykonuje odpowiednie akcje:
        - *Subscribe*: wyszukuje żądaną konwersację i tworzy dla niej `MessagesListener` – klasę która zostaje powiadomiona przy publikacji wiadomości
        - *Unsubscribe*: usuwa `MessagesListener` dla podanej konwersacji
        - *Post message*: dodaje wiadomość do wskazanej konwersacji i powiadamia o tym wszystkich `MessagesListener`-ów.
        - *Create conversation*: jeżli klient utworzy konwersację, `ThreadBehavior` dodaje ją do globalnego stanu przechowującego konwersacje i powiadamia wszystkich podłączonych klientów
        - *Delete conversation*: usuwa wskazaną konwersację i powiadamia o tym
4. Po zakończeniu połączenia `ThreadBehavior` usuwa wszystkie stworzone przez siebie konwersacje i `Listener`-y oraz zamyka gniazdo połączenia
    - stworzone konwersacje są pamiętane w `createdConversations: vector<Conversation *>`
    - zarejestrowane `Listener`-y są pamiętane w `createdListeners: map<int, MessagesListener *>` oraz w `conversationsListener: ConversationsListener*`

Klient

> klient oparty na Electron (NodeJS + Chromium) oraz Typescript

1. `src/main.ts` uruchomiony zostaje w głównym procesie Electrona.
2. `src/renderer/renderer.js` zostaje uruchomiony w osobnym procesami odpowiedzialnym za wyrenderowanie strony w html stanowiącej interfejs graficzny
3. Komunikacja między tymi dwoma procesami odbywa się za pomocą `Event`-ów (jest to mechanizm wbudowany w Electrona).
4. Gdy użytkownik poda dane połączenia (w procesie renderera) generowany jest odpowiedni event i proces główny nawiązuje połączenie poprzez `net.Socket`.
5. Za każdym razem gdy użytkownik chce wykonać jakąś akcję generowany jest w rendererze event, na podstawie którego wysyłany jest komunikat poprzez `net.Socket.write()`
    1. Komunikaty tworzone są przez klasę `Pack`
6. Za każdym razem gdy serwer wysyła jakieś dane generowany jest event `'data'` który następnie zostaje obsłużony przez klasę `TcpMiddleware`
    1. `TcpMiddleware` używa dekoratora `@reEmmit('event')` który powoduje przesłanie do renderera eventu o nazwie `'event'` i danymi takimi jakie zwraca metoda – ułatwia to przekazanie rendererowi odpowiednio przetworzone dane
    2. Stan konwersacji jest przechowywany w klasie `Store` (src/main/Store.ts), tam zapisywana jest lista konwersacji przekazana przez serwer i stamtąd jest odczytywana

Sposób kompilacji i uruchomienia programu

Serwer

> Serwer wymaga `cmake` oraz kompilatora C++20, np. gnu `g++`

```
# server/
cmake .
cmake --build .
# wygeneruje plik wykonywalny o nazwie `server`
./server
```

> Serwer można także skompilować i uruchomić jako kontener dokerowy, używając pliku `Dockerfile` dostępnego w katalogu `server/`.

```
# server/
docker build -t tcp-server .
```

Klient

> Klient bazuje na frameworku Electron, który jest wymagany do uruchomienia projektu. Dodatkowo wymagany jest nodeJS oraz npm.

```
# client/
npm install # instaluje wymagane zależności
npm start # uruchomi projekt na podstawie skryptu zawartego w package.json
          # oraz stworzy katalog client/build/ zawierający skompilowany javascript
```

> Projekt może zostać skompilowany do plików wykonywalnych używając np. electron forge lub electron builder. Skompilowane pliki wykonywalne znajdują się w repozytorium GitHub.