ZPR projekt: klon gry Agar.io

Kajetan Śpionek
Wojciech Przybysz

1. Struktura klas serwera

W projekcie wykorzystano przykład *CHAT_SERVER* ze strony internetowej biblioteki boost, ilustrujący wykorzystanie biblioteki boost::asio do stworzenia serwera http. Jest on udostępniony pod licencją *Boost Software License*, która umożliwia bezpłatne użycie, modyfikacje i rozprzestrzenianie kodu. Całość kodu z przykładu umieszczono w przestrzeni nazw *http*. Na tę przestrzeń nazw składają się struktury:

Header - definiująca nagłówek http
Reply - definiująca strukturę odpowiedzi i jej przetwarzanie
Request - definująca zapytanie http

oraz klasy:

RequestHandler
RequestParser

zajmujące się poprawnym przetwarzaniem zapytania.

Klasy: Dataframe i DataframeParser - zajmują sie przetwarzaniem ramek zgodnie z protokołem WebSocket (RFC 6455).

Klasy: GameBoard, Element, Ball, FoodItem - implementują logikę gry.

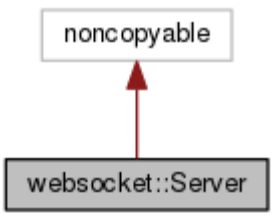
Klasy: Player, Session, Server - implementują nawiązywanie połączenia i przetwarzanie komunikatów.

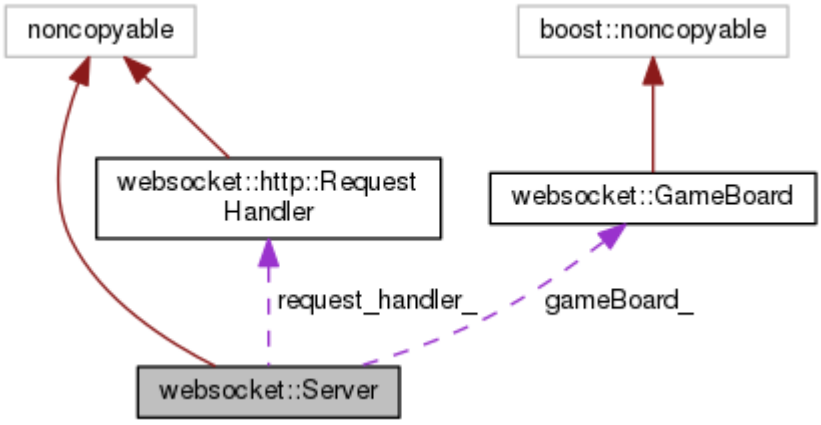
Serwer wykorzystuje asynchroniczną komunikację boost::asio. W pliku server.cpp są zaimplementowane metody tworzące serwer I nasłuchiwanie na porcie 7777 (konieczne jest podanie portu jako argumentu przy uruchamianiu programu). Klasa Session posiada odwołania do odpowiednich parserów, stanów połączenia oraz do metod klasy GameBoard, gdzie zaimplementowana jest logika gry.

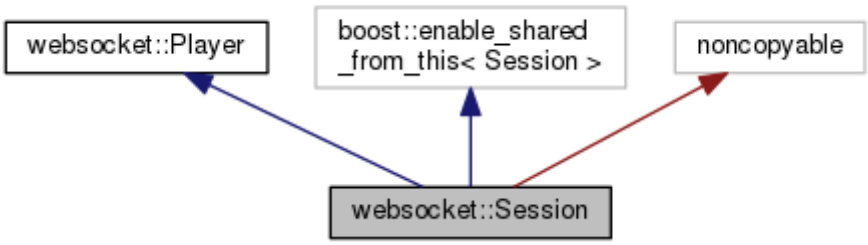
DIAGRAM KLAS:

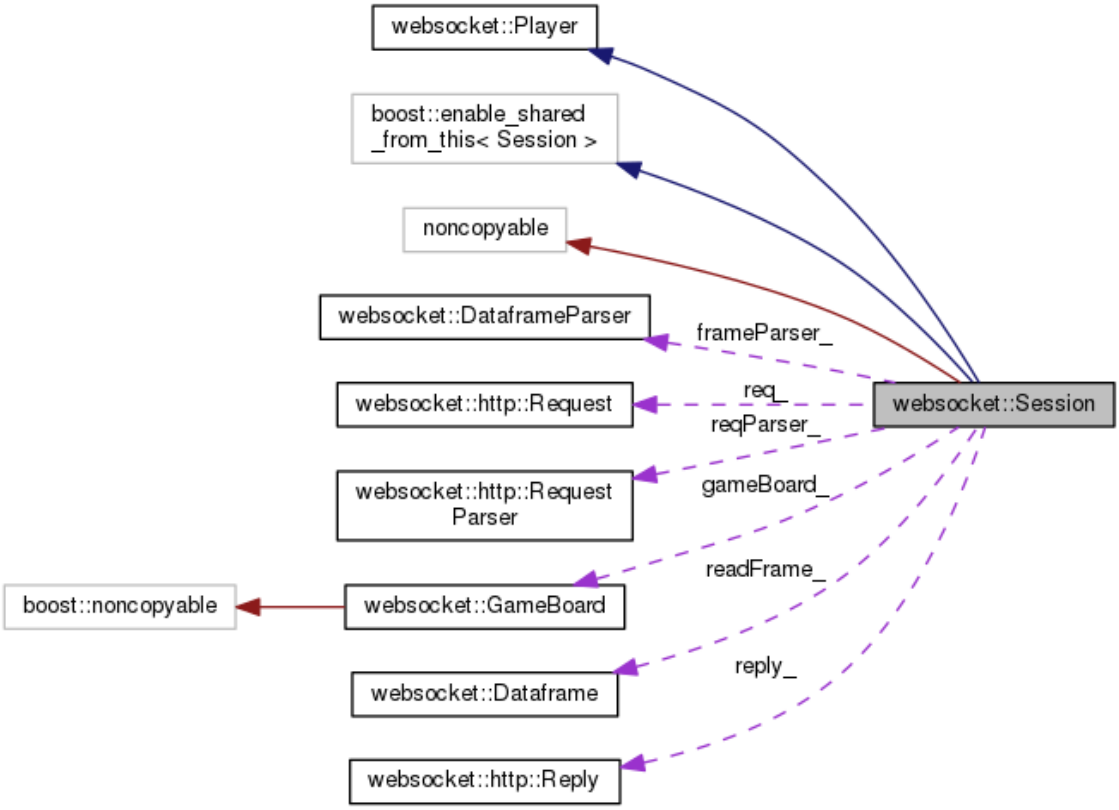
legenda:
- niebieska strzałka - dziedziczenie publiczne
- czerwona strzałka - dziedziczenie prywatne
- fioletowe strzałki – reprezentują inne klasy wykorzystywane przez klasę

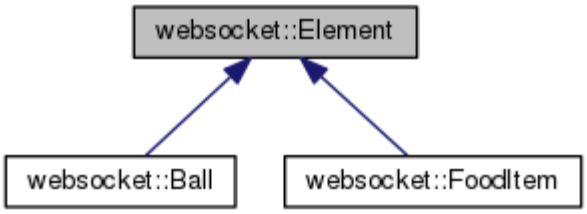
Klasa Server

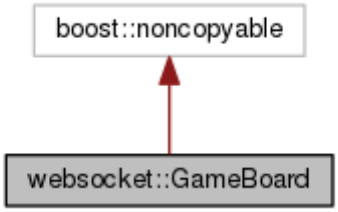



noncopyable
boost::noncopyable
websocket::http::Request Handler
websocket::GameBoard
request_handler_
gameBoard_
websocket::Server


websocket::Player
boost::enable_shared _from_this< Session >
noncopyable
websocket::Session


websocket::Player
boost::enable_shared _from_this< Session >
noncopyable
websocket::DataframeParser
frameParser_
websocket::http::Request
req_
reqParser_
websocket::Session
websocket::http::Request Parser
gameBoard_
readFrame_
boost::noncopyable
websocket::GameBoard
reply_
websocket::Dataframe
websocket::http::Reply





2. Struktura połączenia

Połączenie nawiązywane jest przez klienta, który wysyła ramkę *newPlayerName:* ze swoim nickiem. Serwer sprawdza dostępność i odsyła ramkę zwrotną *newPlayerValidNick:* z informacją *OK* bądź *TAKEN*, co pozwala klientowi na ewentualną zmianę nicku. Następnie klient wysyła ramkę gotowości do gry *newPlayerStatus:* . Serwer na informację o gotowości do gry klienta przesyła:

-*mapSize:* aktualne wymiary mapy,
-*gameState:* aktualny stan gry, czyli położenie kulek,
-*newBall:* przypisuje kulkę graczowi i informuje innych graczy,
-*newFoodItem:* dodaje n nowych statycznych kulek do gry.

Następnie rozpoczyna się właściwa gra. Klient wysyła do serwera unormowane do jedynki położenia myszki. Serwer przetwarza zmianę położenia, ustala prędkość kulki według jej aktualnego promienia. Jeśli w obrębie nowego położenia kulki znajduje się inna kulka o mniejszym promieniu, zostaje ona ‘zjedzona’ - usunięta z gry. Wtedy promień kulki, która zjadła inną kulke się powiększa oraz zostają zapisane jej statystyki.

3. Działanie klienta

Początkowo klient wyświetla stronę startową, na której użytkownik może wpisać nazwę, która będzie go reprezentowała w rozgrywce. Dodatkowo wyświetlają się też błędy połączenia z serwerem, błędnego wyboru nazwy użytkownika (dozwolone są tylko znaki alfanumeryczne). Po naciśnięciu start, rozpoczynana jest wymiana ramek z informacjami z serwerem.

Na podstawie otrzymanych z serwera informacji klient wyświetla na canvasie aktualny stan gry. W trakcie trwania rozgrywki klient zajmuje się wyświetlaniem aktualnego odświeżaniem stanu gry, oraz przesyłaniem położenia myszki użytkownika. W momencie przegranej, wyświetlony zostaje komunikat, z końcowymi statystykami gracza.

Architektura klienta wykorzystuje wbudowane w HTML5 Websockets oraz Canvas. Dodatkowo w bocie został zastosowany HTML5 Web worker.

4. Zrealizowane funkcjonalności

W projekcie udało się zrealizować wszystkie założone w specyfikacji funkcjonalności. W trakcie realizacji, zdecydowaliśmy odejść nieco od założonej w szkielecie struktury, po to by lepiej dopasować serwer do przykładu ze strony boost::asio. W rezlizacji projektu mogliśmy użyć poznane na wykładzie udogodnienia języka oraz bibliotek, między innymi: kontenery stl, sprytne wskaźniki, std::bind, boost::asio, boost::lexical_cast, cppunit. Ponad to mieliśmy okazję poznać udogodnienia HTML5 : Canvas, Websockets, WebWorker.

Udało się uzyskać zadawalający efekt oraz możliwosci gry. Warto byłoby popracować nad pokryciem testami oraz obsługą wyjątków. W kolejnej iteracji projektu możnaby stworzyć własny kontener przechowujący kulki. Obecne użycie std::map nie jest rozwiązaniem optymalnym ze względu na czas dostępu, ale jest rozwiązaniem skutecznym.

5. Planowany czas, a rzeczywista czasochłonność projektu.

Zadania	**Czas planowany**	**Czas rzeczywisty**
Implementacja Serwera	35	57
▪ Stworzenie interfejsu dla WebSocket	10	30
▪ Podstawowe połączenie z pojedynczym klientem	3	2
▪ Nawiązanie komunikacji z wieloma klientami	7	2
▪ Implementacja logiki gry	5	10
▪ Przesyłanie docelowych danych do klientów	5	5
▪ Testowanie i korekcja błędów	5	8
Implementacja Klienta	15	43
▪ Stworzenie podstawowego interfejsu przesyłania danych	4	12
▪ Stworzenie interfejsu graficznego	3	14
▪ Stworzenie docelowego schematu komunikacyjnego	5	9
▪ Testowanie	3	8
Sumaryczny czas:	50	100

6. Podsumowanie długości kodu

Language	files	blank	comment	code
Javascript	155	309	321	2646
C++	13	371	56	1594
C/C++ Header	14	188	167	471