Rebelia Programowalnych Zaskrońców

Artur Dobrogowski Dominik Januszewicz Paweł Wąsowski

21 stycznia 2015

Spis treści

Rozdział 1

Opis działania

1.1 Użyte narzędzia

Do kompilacji projektu użyliśmy narzędzia *cmake*, do organizacji pracy repozytorium *git*, do testów i ich automatyzacji *boost::test* i *cmake - make test*. Do generacji dokumentacji narzędzia *doxygen*, a do utrzymania stałej konwencji kodu - *astyle*. Posługiwaliśmy się też prostymi skryptami *bash* automatyzujące pewne czynności. W celu kompilacji pod windowsem używaliśmy narzędzia *cygwine*.

1.2 Temat

Tematem naszego projektu jest generyczny serwer zaprojektowany pod gry multiplayer.

1.2.1 Zaplanowana funkcjonalność

Przewidzieliśmy w naszym projekcie cztery podstawowe funkcjonalności:

- Funkcja czatu umożliwiająca komunikację graczy połączonych z serwera. Naturalnie z funkcją czatu powinna być połączona funkcja wyświetlania listy graczy podłączonych do serwera, żeby było wiadomo z kim się rozmawia.
- Funkcja zakładania i dołączania do pokoi w których gracze zbierają się przed rozpoczęciem gry.
- Funkcja dystrybucji ustawień wśród graczy danej rozgrywki zapisanych w języku *LUA* przed rozpoczęciem rozgrywki.
- Oraz naturalnie, pewna gra (której logika nie była celem naszego projektu, więc zostawiliśmy na nią miejsce udając, że się rozgrywa)

Aby dostarczyć powyższą funkcjonalność, musieliśmy też dostarczyć wiele innych mechanizmów, takich jak obsługa połączenia przez internet, serializacja/deserializacja danych przesyłanych, “rejestracja” klientów, testy itp.

1.2.2 Szkielet realizacji

Założyliśmy, że aby nasz serwer optymalnie wykorzystywał zasoby i maksymalnie wykorzystywał potencjał biblioteki *boost::asio*, powinniśmy zastosować podejście wielowątkowe. Zatem nasza aplikacja składa się z trzech głównych wątków:

- Zgodnie ze wzorcem Monitora, wątek główny klasy Serwer który obsługuje wszelkie dane przychodzące przez protokół TCP. Następnie interpretuje je za pomocą metod serializacyjnych w pakiety klasy Packet i umieszcza na Adapter kolejki pakietów (zabezpieczonej *mutex'em* wielowątkowo kolejki *std::queue<Packet>*).

- Wątek *Packet_handler*, który, jak nazwa wskazuje, obsługuje pakiety przychodzące kierując się polem *Packet::Tag*. To pole jest typu *enum* i informuje ono o typu wiadomości, sygnalizacji czy też rządania jaką niesie pakiet. Przykładowo, informujący, że przyszła nowa wiadomość czatu lub rządanie od klienta o ponowną synchronizację wszelkich zasobów.

- Oraz poboczny wątek w funkcji *main* który symuluje konsolę poleceń umożliwiającą administrację serwerem, czy też jego bezpieczne zakończenie (obecnie jest to pętla czekająca na wciśnięcie klawisza *q*).

W dodatku, chcieliśmy odseparować logikę gry i wszelkie pakiety komunikacyjne związane z konkretną jej realizacją. Dlatego ponownie wykorzystaliśmy wzorzec monitora gdzie każda instancja gry odbywa się w osobnym wątku, a wszelkie pakiety związane z konkretną instancją gry są przekierowywane na prywatną kolejkę danej instancji. Takie podejście umożliwia przyszłe poszerzanie serwera na komunikujące się kilka komputerów, ponieważ przewidujemy iż proces symulacji gry całkowicie po stronie serwera jest limitowana przez zasoby serwera (moc obliczeniową, pamięć).

1.2.3 Opis komunikacji klas i struktura kodu

Wszystkie klasy których obiekty powinny być przesyłane przez internet w ramach komunikacji (takich jak lista pokojów gry, lista klientów na serwerze i podstawowe informacje o nich) dziedziczą po klasie bazowej *Resource*. Jest tak, żebyśmy mogli skorzystać z dobrodziejstwa polimofrizmu przy konstrukcji klasy *Packet* - która się składa z dwóch głównych pól: deskryptora (*Packet::Tag*) i wskaźnika na *Resource*. Korzystamy tutaj też z tego, że biblioteka *boost::serialize* poprawnie serializuje wskazania zapisując też na jakie dane pokazują.

Wszystkie klasy które są zerializowane tylko dla powodu niesienia informacji w zwartej strukturze są podstawowymi klasami z publicznymi składowymi i jedynie metodą serialziacji. Wszelkie klasy dostarczające pewną funkcjonalność, dla przykładu *Client*, dziedziczą po tych podstawowych strukturach. W ten sposób odseparowujemy tworzenie klas funkcjonalnych, których konstruktory mogą zmieniać pewne rzeczy w programie. Dla klas funkcjonalnych jest założenie, że na każdego klienta połączonego z serwerem przypada tylko jedna instancja tej klasy. W dodatku wszystkimi jej instancjami opiekuje się klasa z rodziny *Register*, która umownie jako jedyna ma prawo zarządzania obiektami takich klas.

Przyjmujemy założenie, że klient najlepiej wie jakie informacje chce otrzymywać i kiedy, więc zastosowaliśmy wzorzec obserwatora. Nasz klient jest obserwatorem klas które odpowiadają za zasoby których aktualność klient może chcieć śledzić. Wyróżniamy dwa stany klienta: klient znajdujący się w "lobby" śledzący informacje na temat: listy pokoi (i ich stanów), listy graczy podłączonych do serwera, wymienianych wiadomości czatu, oraz klienta znadującego się w grze - który wyrejestrowuje się z tych źródeł informacji, ponieważ nie są one mu w tym momencie potrzebne. Takie podejście umożliwia łatwe poszerzanie serwera o nowe funkcje (wystarczy dodać nową klasę typu *Register* dziedziczącą po klasie *Subject* i nowe stany pośrednie w jakich klient może się znajdować. Umożliwa też selektywny wybór tego co klient chce otrzymywać od serwera - np w celu zmniejszenia zasobów łącza.

Główna klasa Serwera jest singletonem i dziedziczy po strukturze grupującej zasoby serwera (wszystkie klasy rodziny *Register*), żeby każdy zainteresowany miał do nich dostęp. W zasobach serwera znajduje się też lista subskrybcji tematów dla obserwatora.

Funkcja *notify* w klasie typu temat jest przekazywana obserwatorowi w argumencie funkcji *update* i zawiera informację którą klient powinien przesłać na sobie znany adres.

Ustawienia gry są odszyfrowywane przez klasę dziedziczącą po ustawieniach *LuaInterpreter*, która potrafi je poprawnie zinterpretować w języku *LUA*

Rozdział 2

Niepowodzenia i wnioski z projektu

Niewiele było przetestowane jako całość przez różne opóźnienia. Stosowaliśmy jedynie testy jednostkowe, przez co nie jesteśmy w stanie zaprezentować scenariusza testowego na chwilę obecną.

2.0.4 Organizacja pracy

- Uważamy, że jedną z głównych przyczyn niepowodzeń w naszym projekcie była niezdrowa organizacja pracy. Przez długi czas nie mogliśmy ustalić wspólnego *interface'u* i częste konieczne zmiany były czasochłonne i nie wnosiły wiele do funkcjonalności projektu. W przyszłości poświęcimy więcej czasu na dokładne przeanalizowanie i ustalenie komunikacji naszych klas. Przełożyło się to też na to, że pomimo wielu godzin poświęconych projektowi (szacuję je na około 500 godzin wspólnych wysiłków) efekty były mizerne.
- Powinniśmy byli o wiele wcześniej zabrać się za przetestowanie podstawowej funkcjonalności łącznej niż dążyć do uzyskania kompletnego zestawu klas (testowanych osobno w testach jednostkowych) który potem będziemy testować jako całość.
- Gdybyśmy zastosowali *test driven development* lub metodologię *SCRUM* to prawdopodobnie uniknęlibyśmy powyższych problemów.
- Przez wiele czasu nie mielśmy działającego serwera realizującego połączenie przez internet, by móc je sensownie przetestować. Wynikało to z problemami ze zrozumieniem biblioteki *boost::asio*

2.0.5 Problemy związane z projektowaniem

- Bardzo wiele kłopotów sprawiło nam zbyt płytkie pojęcie na temat działania biblioteki *boost::serialize*. Oprócz licznych niewyjaśnionych błędów, okazuje się, że napotkaliśmy problemy które grożą niewykonalnością naszego podejścia dotyczącego polimofrizmu klasy *Resource*.
- Na początku projektu przez długi czas nie było konsensusu co do tego jak ma wyglądać nasza współpraca, przez co straciliśmy ponad miesiąc zanim zabraliśmy się zgodnie do pracy.

Rozdział 3

Indeks hierarchiczny

3.1 Hierarchia klas

Ta lista dziedziczenia posortowana jest z grubsza, choć nie całkowicie, alfabetycznie:

Rozdział 4

Indeks klas

4.1 Lista klas

Tutaj znajdują się klasy, struktury, unie i interfejsy wraz z ich krótkimi opisami:

Rozdział 5

Dokumentacja klas

5.1 Dokumentacja struktury Address

Obiekty tej klasy reprezenują w serwerze adresy, z których łączą się klienci.

`#include <Address.hpp>`

Metody publiczne

- **Address** (AddressIP Ip="non-defined", AddressPort Port=~0)
- void **change_owner** (ClientID newOwner) const
- void **update_connection** (const TcpPointer &x) const
- bool **operator<** (const Address &latter) const

Atrybuty publiczne

- AddressIP **ip**
- AddressPort **port**
- ClientID **owner**
- TcpPointer connection

  przechowuje identyfikator klienta, który łączy się z tego adresu; pole jest inicjowane dopiero po dodaniu klienta do rejestru klientów

5.1.1 Opis szczegółowy

Obiekty tej klasy reprezenują w serwerze adresy, z których łączą się klienci.

Dokumentacja dla tej struktury została wygenerowana z plików:

- src/server/Address.hpp
- src/server/Address.cpp

5.2 Dokumentacja klasy AddressRegister

Rejestr adresów, z których nawiązano połączenie z serwerem.

`#include <AddressRegister.hpp>`

Metody publiczne

- const Address * **register_address** (const Address &x)
- ClientID get_address_owner (const Address &x) const

  rejestracja adresu, z którego połączono się z serwerem
- const Address * get_address_pointer (const Address &x) const

  ta metoda "tłumaczy" id klienta na jego adres

5.2.1 Opis szczegółowy

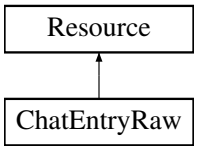
Rejestr adresów, z których nawiązano połączenie z serwerem.

Dokumentacja dla tej klasy została wygenerowana z plików:

- src/server/AddressRegister.hpp
- src/server/AddressRegister.cpp

5.3 Dokumentacja struktury ChatEntryRaw

Diagram dziedziczenia dla ChatEntryRaw



Metody publiczne

- **ChatEntryRaw** (const std::string &nick__, const std::string &message__)
- virtual Resource::Tag **get_tag** ()
- template<class Archive >
  void **serialize** (Archive &ar, const unsigned int)
- virtual std::string **show_content** ()

Atrybuty publiczne

- std::string **nick_**
- std::string **message_**

Przyjaciele

- class **boost::serialization::access**

Dodatkowe Dziedziczone Składowe

Dokumentacja dla tej struktury została wygenerowana z pliku:

- src/shared/ChatEntryRaw.hpp

5.4 Dokumentacja struktury ChatHistoryRaw

Przyjaciele

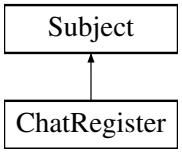
- class **boost::serialization::access**

Dokumentacja dla tej struktury została wygenerowana z pliku:

- src/shared/ChatHistoryRaw.hpp

5.5 Dokumentacja klasy ChatRegister

Diagram dziedziczenia dla ChatRegister



Metody publiczne

- void **register_message** (ChatEntryRaw &message)

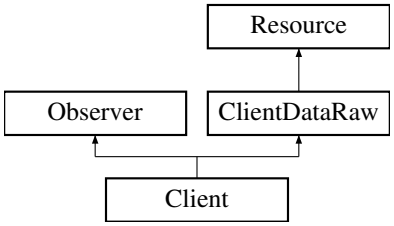
Dodatkowe Dziedziczone Składowe

Dokumentacja dla tej klasy została wygenerowana z plików:

- src/shared/ChatRegister.hpp
- src/shared/ChatRegister.cpp

5.6 Dokumentacja klasy Client

Diagram dziedziczenia dla Client



Metody publiczne

- **Client** (const Address ∗address, TcpPointer pointer=nullptr, std::string nick="UNREGISTERED")
- **Client** (const Client &c)
- ClientID **get_client_id** () const
- ClientState **get_state** () const

- std::string **get_nickname** () const
- const Address * **get_address** () const
- void **set_state** (ClientState s)
- bool **operator**< (const Client &) const
- virtual void update (Resource *updateInfo, Packet::Tag *tag)

  metoda wywoływana przez temat obserwacji, gdy nastąpi zmiana w środowisku gracza

Dodatkowe Dziedziczone Składowe

Dokumentacja dla tej klasy została wygenerowana z plików:

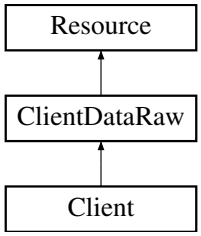
- src/server/Client.hpp
- src/server/Client.cpp

5.7 Dokumentacja struktury ClientDataRaw

To klasa zawierająca dane klienta.

`#include <ClientDataRaw.hpp>`

Diagram dziedziczenia dla ClientDataRaw



Metody publiczne

- ClientDataRaw (ClientID clientID, std::string nickname, ClientState state)

  konstruktor dla boost::serialization
- virtual Tag **get_tag** ()
- virtual std::string **show_content** ()

Atrybuty publiczne

- ClientState **state_**
- const ClientID **clientID_**
- std::string nickname_

  unikalne ID dla każdego gracza

Przyjaciele

- class **boost::serialization::access**

Dodatkowe Dziedziczone Składowe

5.7.1 Opis szczegółowy

To klasa zawierająca dane klienta.

Jest klasą bazową dla Client. To obiekty tej klasy są serializowane i przesyłane - dlatego jest oddzielona od Clienta i ma wszystkie pola public.

Dokumentacja dla tej struktury została wygenerowana z pliku:

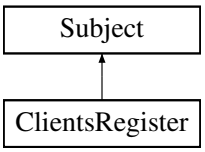
- src/server/ClientDataRaw.hpp

5.8 Dokumentacja klasy ClientsRegister

Ta klasa jest rejestrem klientów - jest wykorzystywana do zarządzania graczami podłączonymi do serwera.

`#include <ClientsRegister.hpp>`

Diagram dziedziczenia dla ClientsRegister



Metody publiczne

- ClientID **register_client** (const Address *address, TcpPointer pointer=nullptr, std::string nickname="UNR↩ EGISTERED")
- const ClientPtr look_up_with_id (ClientID id) const

  metoda rejestrująca nowego klienta na serwerze
- void change_state (ClientID id, ClientState state)

  wyszukiwanie klientów po zadanym id
- void remove_client (ClientID id)

  metoda zmieniająca stan (miejsce "pobytu") klienta
- ClientState get_state (ClientID id) const

  usuwanie klienta z serwera
- int get_size () const

Dodatkowe Dziedziczone Składowe

5.8.1 Opis szczegółowy

Ta klasa jest rejestrem klientów - jest wykorzystywana do zarządzania graczami podłączonymi do serwera.

Pozwala dodawać, usuwać, zmieniać stan klientów.

5.8.2 Dokumentacja funkcji składowych

5.8.2.1 int ClientsRegister::get_size ( ) const [inline]

mówi ile jest graczy na serwerze

Dokumentacja dla tej klasy została wygenerowana z plików:

- src/server/ClientsRegister.hpp
- src/server/ClientsRegister.cpp

5.9 Dokumentacja struktury ClientState

ClientState reprezentuje stan (miejsce), w którym obecnie jest klient - np.

`#include <ClientDataRaw.hpp>`

Typy publiczne

- enum **Location** { **LOBBY**, **GAMEROOM**, **GAME** }

Metody publiczne

- **ClientState** (Location l=LOBBY, int lIdentifier=0)

Atrybuty publiczne

- Location **location**
- int **locationIdentifier**

Przyjaciele

- class **boost::serialization::access**

5.9.1 Opis szczegółowy

ClientState reprezentuje stan (miejsce), w którym obecnie jest klient - np.

w grze.

Dokumentacja dla tej struktury została wygenerowana z pliku:

- src/server/ClientDataRaw.hpp

5.10 Dokumentacja klasy GameLogic

Dokumentacja dla tej klasy została wygenerowana z pliku:

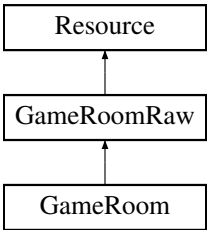
- src/shared/GameLogic.hpp

5.11 Dokumentacja klasy GameRoom

Klasa reprezentująca pokój gry na serwerze.

`#include <GameRoom.hpp>`

Diagram dziedziczenia dla GameRoom



Metody publiczne

- **GameRoom** (ClientID host, std::string gameRoomName)
- void **add_player** (ClientID newPlayer)
- void remove_player (ClientID player)

  dodawanie gracza
- GameRoomID **get_id** ()
- std::string **get_name** ()
- unsigned int **get_number_of_players** ()
- unsigned int **get_max_number_of_players** ()
- ClientID **get_host_id** () const
- GameRoomRaw **get_raw_data** ()

Przyjaciele

- class **boost::serialization::access**

Dodatkowe Dziedziczone Składowe

5.11.1 Opis szczegółowy

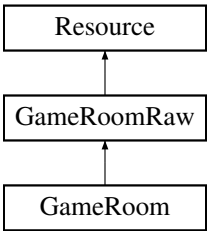
Klasa reprezentująca pokój gry na serwerze.

Dokumentacja dla tej klasy została wygenerowana z plików:

- src/shared/GameRoom.hpp
- src/shared/GameRoom.cpp

5.12 Dokumentacja struktury GameRoomRaw

Diagram dziedziczenia dla GameRoomRaw



Metody publiczne

- **GameRoomRaw** (std::string name, ClientID hostID, GameRoomID GRId)
- virtual Tag **get_tag** ()
- virtual std::string **show_content** ()
- template<class Archive >
  void **serialize** (Archive &ar, const unsigned int)

Atrybuty publiczne

- std::string **gameRoomName**
- ClientID **host**
- std::list< ClientID > **players**
- unsigned int **numOfPlayers**
- unsigned int **maxNumOfPlayers**
- const GameRoomID **id**

Przyjaciele

- class **boost::serialization::access**

Dodatkowe Dziedziczone Składowe

Dokumentacja dla tej struktury została wygenerowana z plików:

- src/shared/GameRoomRaw.hpp
- src/shared/GameRoomRaw.cpp

5.13 Dokumentacja klasy GameRoomsRegister

Rejestr pokojów gry - służy do zarządzania pokojami gry na serwerze.

`#include <GameRoomsRegister.hpp>`

Diagram dziedziczenia dla GameRoomsRegister

Metody publiczne

- GameRoomPtr **add_game_room** (ClientID host, std::string name)
- void remove_game_room (GameRoomID id)

  dodawanie nowego pokoju do rejestru
- GameRoomPtr look_up_with_id (GameRoomID id)

  usuwanie z serwera pokoju o zadanym id
- unsigned int get_size ()

  wyszukiwanie pokoju o zadanym id
- void **notify** (Resource *, const Packet::Tag *)

Dodatkowe Dziedziczone Składowe

5.13.1 Opis szczegółowy

Rejestr pokojów gry - służy do zarządzania pokojami gry na serwerze.

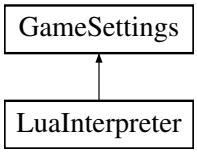
Pozwala na dodawanie, usuwanie pokojów.

Dokumentacja dla tej klasy została wygenerowana z plików:

- src/server/GameRoomsRegister.hpp
- src/server/GameRoomsRegister.cpp

5.14 Dokumentacja struktury GameSettings

Diagram dziedziczenia dla GameSettings



Atrybuty publiczne

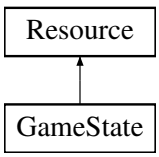
- Map **map**
- int **numberOfPlayers**

Dokumentacja dla tej struktury została wygenerowana z pliku:

- src/shared/GameSettings.hpp

5.15 Dokumentacja klasy GameState

Diagram dziedziczenia dla GameState



Metody publiczne

- virtual std::string **show_content** ()
- virtual Resource::Tag **get_tag** ()

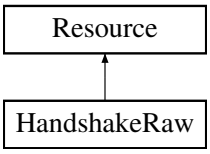
Dodatkowe Dziedziczone Składowe

Dokumentacja dla tej klasy została wygenerowana z plików:

- src/shared/GameState.hpp
- src/shared/GameState.cpp

5.16 Dokumentacja struktury HandshakeRaw

Diagram dziedziczenia dla HandshakeRaw



Metody publiczne

- **HandshakeRaw** (std::string nick)
- virtual Resource::Tag **get_tag** ()
- template<class Archive >
  void **serialize** (Archive &ar, const unsigned int)
- virtual std::string **show_content** ()

Atrybuty publiczne

- std::string **nick_**

Przyjaciele

- class **boost::serialization::access**

Dodatkowe Dziedziczone Składowe

Dokumentacja dla tej struktury została wygenerowana z pliku:

- src/shared/HandshakeRaw.hpp

5.17 Dokumentacja klasy Logger

Statyczne metody publiczne

- static Logger & **getInstance** ()
- static log4cpp::CategoryStream **log_debug** ()
- static log4cpp::CategoryStream **log_main** ()

Atrybuty publiczne

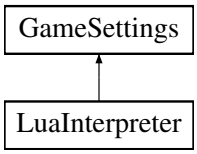
- log4cpp::Category & **main**
- log4cpp::Category & **debug**

Dokumentacja dla tej klasy została wygenerowana z plików:

- src/server/Logger.hpp
- src/server/Logger.cpp

5.18 Dokumentacja klasy LuaInterpreter

Diagram dziedziczenia dla LuaInterpreter



Metody publiczne

- void **set_settings** ()
- void **load_settings** (const char ∗)
- void **registerFunctions** ()

Dodatkowe Dziedziczone Składowe

Dokumentacja dla tej klasy została wygenerowana z plików:

- src/luacore/LuaInterpreter.hpp
- src/luacore/LuaInterpreter.cpp

5.19 Dokumentacja klasy Map

Metody publiczne

- void **map_resize** (int x, int y)
- void **set_base_map_unit** (const MapUnit &a)
- void **set_field** (const int &x, const int &y, const MapUnit &character)

Atrybuty publiczne

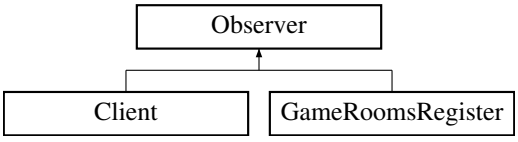
- std::vector< std::vector< MapUnit > > **field_**
- int **width**
- int **height**
- MapUnit **baseMapUnit_**

Dokumentacja dla tej klasy została wygenerowana z pliku:

- src/shared/Map.hpp

5.20 Dokumentacja klasy Observer

Diagram dziedziczenia dla Observer



Metody publiczne

- **Observer** (unsigned id)
- virtual void **update** (Resource ∗, const Packet::Tag ∗)

Atrybuty publiczne

- unsigned **observerID**

Statyczne atrybuty publiczne

- static unsigned **observerNextID** = 0

Dokumentacja dla tej klasy została wygenerowana z plików:

- src/shared/Observer.hpp
- src/shared/Observer.cpp

5.21 Dokumentacja klasy Packet

Typy publiczne

- enum Tag {
  UPDATED_RESOURCE, REMOVE_RESOURCE, REGISTER_REQUEST, CHAT_ENTRY_MESSAGE_↩
  REQUEST,
  GAMEROOM_CREATE_REQUEST, GAMEROOM_JOIN_REQUEST, GAMEROOM_LEAVE_REQUEST,
  GAMEROOM_UPDATE_REQUEST,
  GAMEROOM_START_REQUEST, GAME_START_FAILURE_INFO, SYNCHRONISE_REQUEST, CLO↩
  CK_SYNCHRONISE,
  GAME_STATE, GAME_ACTION, KEEP_ALIVE }

  Enum będący flagą stanowiącą podstawę komunikacji w naszym systemie.
- typedef std::string **StreamBuffer**

Metody publiczne

- Packet ()

  boost::serialization potrzebuje bezparametrowego konstruktora, można go przenieść do "private"
- **Packet** (Tag tag__, const Address ∗ad__=nullptr, Resource ∗content__=nullptr)
- **Packet** (Tag tag__, const Address ∗ad__, ResourcePtr content__=nullptr)
- template<class Archive >
  void serialize (Archive &ar, const unsigned int)

  Serializacja Pakietu dotyczy tagu pakietu i zawartości (wskaźnika na Resource)
- StreamBuffer **get_data_streambuf** ()
- const Address ∗ **get_address** () const
- Tag **get_tag** () const
- ResourcePtr **get_content** () const
- std::string **show_resource_content** ()

Przyjaciele

- class **boost::serialization::access**

5.21.1 Dokumentacja składowych wyliczanych

5.21.1.1 enum Packet::Tag

Enum będący flagą stanowiącą podstawę komunikacji w naszym systemie.

Ten enum moze urosnac duzy, poniewaz do kazdej dodanej funkcjonalnosci, bedzie nowy tag pakietu.

Wartości wyliczeń

UPDATED_RESOURCE dane aktualizacyjne dla klienta, mozna rozpoznac zawartosc po tagu Resource

REMOVE_RESOURCE dane aktualizacyjne dla klienta, mozna rozpoznac zawartosc po tagu Resource

REGISTER_REQUEST w srodku pakietu Handshake przedstawiajacy dane o kliencie.

CHAT_ENTRY_MESSAGE_REQUEST prosba o nadanie wiadomosci czatu

GAMEROOM_CREATE_REQUEST prosba o stworzenie nowego pokoju

GAMEROOM_JOIN_REQUEST prosba o dolaczenie do pokoju

GAMEROOM_LEAVE_REQUEST prosba o opuszczenie pokoju

GAMEROOM_UPDATE_REQUEST prosba o zmiane ustawien pokoju (np. ustawien gry dla hosta, a dla gracza wyrazenie gotowosci)

GAMEROOM_START_REQUEST prosba o rozpoczecie rozgrywki

GAME_START_FAILURE_INFO informacja dla klienta o niespelnionym rzadaniu, np. z powodu niegotowosci wszystkich graczy

SYNCHRONISE_REQUEST prosba o wyslanie wszystkich zasobow... podejrzewam, że mozna sie bez tego obyc i zamiast tego zrobic timeouty

CLOCK_SYNCHRONISE prosba o okreslenie czasu wzgledem serwera (sluzy tez do obliczenia round trip time)

GAME_STATE pakiet zawierajacy stan rozgrywki - do przekierowania dla konkretnej instancji gry

GAME_ACTION byc moze sie przyda?

KEEP_ALIVE ping! do ustalenia czy ktos stracil polaczenie.

Dokumentacja dla tej klasy została wygenerowana z plików:

- src/shared/Packet.hpp
- src/shared/Packet.cpp

5.22 Dokumentacja klasy Packet_handler

Ta klasa jest odpowiedzialna za przekazywanie wszystkich przychodzących pakietów w odpowiednie miejsca.

`#include <Packet_handler.hpp>`

Metody publiczne

- **Packet_handler** (PacketQueue ∗inQueue, bool ∗running)
- void **operator()** ()

5.22.1 Opis szczegółowy

Ta klasa jest odpowiedzialna za przekazywanie wszystkich przychodzących pakietów w odpowiednie miejsca.

Dzięki temu, że każdy pakiet jest oznaczony przez Tag mówiący, jakiego typu jest przychodząca informacja, można ją przesłać w odpowiednie miejsce. Przykład: informacja przenoszona przez pakiet o tagu REGISTER_REQUEST zostanie wykorzystana do dodania nowego gracza do rejestru klientów.

Packet_handler działa w oddzielnym wątku.

Dokumentacja dla tej klasy została wygenerowana z plików:

- src/server/Packet_handler.hpp
- src/server/Packet_handler.cpp

5.23 Dokumentacja klasy PacketQueueAdapter

Metody publiczne

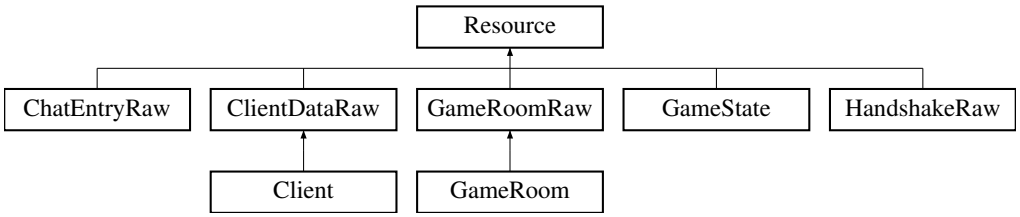
- void **push** (const Packet &)
- void **pop** ()
- Packet & **front** ()
- bool **empty** ()

Dokumentacja dla tej klasy została wygenerowana z plików:

- src/server/PacketQueueAdapter.hpp
- src/server/PacketQueueAdapter.cpp

5.24 Dokumentacja struktury Resource

Diagram dziedziczenia dla Resource



Typy publiczne

- enum **Tag** { **CHAT_ENTRY**, **CLIENT_DATA**, **GAMEROOM**, **HANDSHAKE** }

Metody publiczne

- virtual Tag **get_tag** ()=0
- template<typename Archive >
  void **serialize** (Archive &, const unsigned int)
- virtual std::string **show_content** ()=0

Przyjaciele

- class **boost::serialization::access**

Dokumentacja dla tej struktury została wygenerowana z pliku:

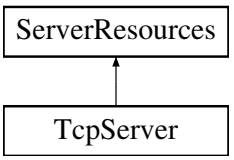
- src/shared/Resource.hpp

5.25 Dokumentacja struktury ServerResources

Zawiera instancje wszystkich głównych klas serwera.

`#include <ServerResources.hpp>`

Diagram dziedziczenia dla ServerResources



Metody publiczne

- const std::vector< Subject * > & get_subscrible_list ()

  lista tematów obserwacji, które uaktualniają swój stan w czasie pracy serwera
- void **init** ()

Atrybuty publiczne

- std::thread * **packetHandler**
- PacketQueue **received**
- Packet_handler lobbyManager

  kolejka wszystkich, które przyszły do serwera
- AddressRegister registeredAddresses

  ten funktor zarządza przychodzącymi z zewnątrz pakietami - przesyła je w odpowiednie miejsca
- ClientsRegister connectedClients

  rejestr adresów, z których nawiązano połączenie z serwerem
- ChatRegister registeredChat

  klienci aktualnie podłączeni do serwera
- GameRoomsRegister registeredRooms

  rejestr czatu - przechowuje wiadomości czatu
- std::vector< Subject * > SubscriptionList

  pokoje gry zarejestrowane na serwerze

Atrybuty chronione

- bool **running_**

5.25.1 Opis szczegółowy

Zawiera instancje wszystkich głównych klas serwera.

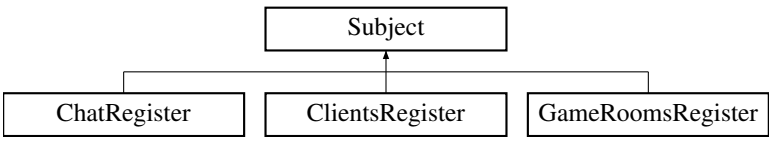
Jest bazą dla klasy TcpServer.

Dokumentacja dla tej struktury została wygenerowana z pliku:

- src/server/ServerResources.hpp

5.26 Dokumentacja klasy Subject

Diagram dziedziczenia dla Subject



Metody publiczne

- void **addObserver** (Observer ∗o) const
- void **eraseObserver** (Observer ∗o)
- virtual void **notify** (Resource ∗, const Packet::Tag ∗)=0

Atrybuty chronione

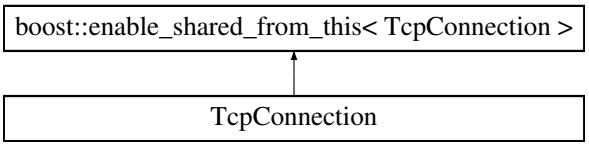
- std::vector< Observer ∗ > **obs_**

Dokumentacja dla tej klasy została wygenerowana z pliku:

- src/shared/Subject.hpp

5.27 Dokumentacja klasy TcpConnection

Diagram dziedziczenia dla TcpConnection



Metody publiczne

- tcp::socket & **socket** ()
- void **write** (std::string)
- std::string **read** () const
- std::string **ip_address** () const
- void **wait_data** ()
- unsigned short **port** () const

Statyczne metody publiczne

- static TcpPointer **create** (boost::asio::io_service &)

Dokumentacja dla tej klasy została wygenerowana z plików:

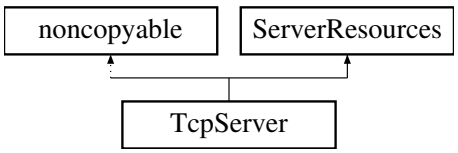
- src/shared/TcpConnection.hpp
- src/shared/TcpConnection.cpp

5.28 Dokumentacja klasy TcpServer

Ta klasa pośredniczy w wymianie informacji pomiędzy tym, co znajduje się w ServerResources, a graczami podłączonymi przez internet.

`#include <Server.hpp>`

Diagram dziedziczenia dla TcpServer



Metody publiczne

- void **start** ()
- void stop ()

  metoda kończąca pracę serwera

Statyczne metody publiczne

- static TcpServer & getInstance ()

  serwer zaczyna pracować po wywołaniu tej metody

Statyczne atrybuty publiczne

- static TcpServer * **pointer**

Dodatkowe Dziedziczone Składowe

5.28.1 Opis szczegółowy

Ta klasa pośredniczy w wymianie informacji pomiędzy tym, co znajduje się w ServerResources, a graczami podłączonymi przez internet.

Jest singletonem i to jej instancja jest uruchamiana przy włączeniu programu.

Dokumentacja dla tej klasy została wygenerowana z plików:

- src/server/Server.hpp
- src/server/Server.cpp

Skorowidz