Bombowe roboty

Pytania proszę wysyłać na adres agluszak@mimuw.edu.pl.

Historia zmian: - **10.06.2022** - doprecyzowanie jak przebiega generacja początkowych bloków - **08.06.2022** - doprecyzowanie jak wygląda koniec gry, dodanie skryptu `verifier.sh` - **06.06.2022** - nowe pytania - **25.05.2023** - Doprecyzowanie, kiedy wysyłane są komunikaty do GUI: Po Turn - Game Po AcceptedPlayer, GameEnded i Hello - Lobby Po GameStarted - nic

A wszystkie pozycje początkowe graczy i bloków są wysyłane w turze 0. - **24.05.2022** - Wycofanie poniższego (nie będziemy osobno oceniać jakości kodu po pierwszej części) - **23.05.2022** - Przy oddawaniu klienta pliki (lub ich części) dotyczące serwera zostaną uznane za zbędne - **20.05.2022** - WAŻNE: zmiana jak wysyłane są informacje o rozgrywce po dołączeniu w trakcie. Doprecyzowanie, w jaki sposób obliczany jest wybuch bomby. - **18.05.2022** - nowe pytania - **16.05.2022** - obsługa IPv6 w GUI, doprecyzowanie jak projekt ma się budować - **13.05.2022** - zmiana display na gui, dodanie pytań - **10.05.2022** - doprecyzowanie jak identyfikowani są klienci - **09.05.2022** - poprawki w GUI, nowe pytania w FAQ - **08.05.2022** - doprecyzowanie jak obliczać wybuch kilku bomb, zmiana generatora liczb losowych, zmiana flag kompilatora

0. Dostarczone programy

Do uruchomienia programów potrzeba kompilatora Rusta, a także pewnych bibliotek systemowych.

Po zainstalowaniu kompilatora należy wykonać komendę: `cargo run --bin <gui/verifier>` i uzupełnić parametry.

Skompilowany serwer (bynajmniej nie wzorcowy) jest dostępny tutaj. Został on skompilowany na maszynie `students`. Aby wyświetlały się komunikaty, należy uruchomić go ze zmienną środowiskową `RUST_LOG=debug`.

0.1. GUI

Interfejs graficzny dla gry Bombowe Roboty. GUI prawdopodobnie będzie jeszcze aktualizowane.

Sterowanie

```
W, strzałka w górę -  porusza robotem w górę.
S, strzałka w dół - porusza robotem w dół.
A, strzałka w lewo - porusza robotem w lewo.
D, strzałka w prawo - porusza robotem w prawo.
Spacja, J, Z - kładzie bombę.
K, X - blokuje pole.
```

0.2. Weryfikator

Ten program pozwala sprawdzić, czy wiadomości są poprawnie serializowane. Innymi słowy, jest to wzorcowy deserializator. Można łączyć się z nim zarówno po TCP, jak i UDP (z parametrem `-u`). Przy uruchamianiu należy podać, jakiego rodzaju wiadomości mają być sprawdzane.

Przykładowo, jeśli chcemy sprawdzić, czy klient wysyła prawidłowe wiadomości do serwera, wykonać: `cargo run --bin verifier -- -p <port, na którym klient myśli, że serwer nasłuchuje> -m client`

1. Gra Bombowe roboty

1.1. Zasady gry

Tegoroczne duże zadanie zaliczeniowe polega na napisaniu gry sieciowej - uproszczonej wersji gry Bomberman. Gra rozgrywa się na prostokątnym ekranie. Uczestniczy w niej co najmniej jeden gracz. Każdy z graczy steruje ruchem robota. Gra rozgrywa się w turach. Trwa ona przez z góry znaną liczbę tur. W każdej turze robot może:

- nic nie zrobić

- przesunąć się na sąsiednie pole (o ile nie jest ono zablokowane)
- położyć pod sobą bombę
- zablokować pole pod sobą

Gra toczy się cyklicznie - po uzbieraniu się odpowiedniej liczby graczy rozpoczyna się nowa rozgrywka na tym samym serwerze. Stan gry przed rozpoczęciem rozgrywki będziemy nazywać `Lobby`.

1.2. Architektura rozwiązania

Na grę składają się trzy komponenty: serwer, klient, serwer obsługujący interfejs użytkownika. Należy zaimplementować serwer i klienta. Aplikację implementującą serwer obsługujący graficzny interfejs użytkownika (ang. *GUI*) dostarczamy.

Serwer komunikuje się z klientami, zarządza stanem gry, odbiera od klientów informacje o wykonywanych ruchach oraz rozsyła klientom zmiany stanu gry. Serwer pamięta wszystkie zdarzenia dla bieżącej partii i przesyła je w razie potrzeby klientom.

Klient komunikuje się z serwerem gry oraz interfejsem użytkownika.

Zarówno klient jak i serwer mogą być wielowątkowe.

Specyfikacje protokołów komunikacyjnych, rodzaje zdarzeń oraz formaty komunikatów i poleceń są opisane poniżej.

1.3. Parametry wywołania programów

Serwer:

```
 -b, --bomb-timer <u16>
-c, --players-count <u8>
-d, --turn-duration <u64, milisekundy>
-e, --explosion-radius <u16>
-h, --help                                  Wypisuje jak używać programu
-k, --initial-blocks <u16>
-l, --game-length <u16>
-n, --server-name <String>
-p, --port <u16>
-s, --seed <u32, parametr opcjonalny>
-x, --size-x <u16>
-y, --size-y <u16>
```

Klient:

```
 -d, --gui-address <(nazwa hosta):(port) lub (IPv4):(port) lub (IPv6):(port)>
-h, --help                                Wypisuje jak używać programu
-n, --player-name <String>
-p, --port <u16>                          Port na którym klient nasłuchuje komunikatów od GUI
-s, --server-address <(nazwa hosta):(port) lub (IPv4):(port) lub (IPv6):(port)>
```

Interfejs graficzny:

```
 -c, --client-address <(nazwa hosta):(port) lub (IPv4):(port) lub (IPv6):(port)>
-h, --help                              Wypisuje jak używać programu
-p, --port <u16>                        Port na którym GUI nasłuchuje komunikatów od klienta
```

Do parsowania parametrów linii komend można użyć funkcji `getopt` z biblioteki standardowej C lub modułu `program_options` z biblioteki Boost.

Wystarczy zaimplementować rozpoznawanie krótkich (`-c`, `-x` itd.) parametrów.

2. Protokół komunikacyjny pomiędzy klientem a serwerem

Wymiana danych odbywa się po TCP. Przesyłane są dane binarne, zgodne z poniżej zdefiniowanymi formatami komunikatów. W komunikatach wszystkie liczby przesyłane są w sieciowej kolejności bajtów, a wszystkie napisy muszą być zakodowane w UTF-8 i mieć długość krótszą niż 256 bajtów.

Napisy (String) mają następującą reprezentację binarną: `[1 bajt określający długość napisu w bajtach][bajty bez ostatniego bajtu zerowego]`.

Listy są serializowane w postaci `[4 bajty długości listy][elementy listy]`. Mapy są serializowane w postaci `[4 bajty długości mapy][klucz][wartość][klucz][wartość]...`.

Pola w strukturze serializowane są bezpośrednio po bajcie oznaczającym typ struktury.

Należy wyłączyć algorytm Nagle'a (tzn. ustawić flagę TCP_NODELAY).

2.1. Komunikaty od klienta do serwera

```
enum ClientMessage {
    [0] Join { name: String },
    [1] PlaceBomb,
    [2] PlaceBlock,
    [3] Move { direction: Direction },
}
```

Typ Direction ma następującą reprezentację binarną:

```
enum Direction {
    [0] Up,
    [1] Right,
    [2] Down,
    [3] Left,
}
```

Wiadomość od klienta `Join(“Żółć!”)` zostanie zserializowana jako ciąg bajtów `[0, 9, 197, 187, 195, 179, 197, 130, 196, 135, 33]`, gdzie:

```
 0 - rodzaj wiadomości
9 - długość napisu
197, 187 - 'Ż'
195, 179 - 'ó'
197, 130 - 'ł'
196, 135 - 'ć'
33 - '!'
```

Natomiast wiadomość `Join(“□□ □□ □□ □□□□”)` zostanie zserializowana jako ciąg bajtów `[0, 49, 240, 159, 145, 169, 240, 159, 143, 188, 226, 128, 141, 240, 159, 145, 169, 240, 159, 143, 188, 226, 128, 141, 240, 159, 145, 167, 240, 159, 143, 188, 226, 128, 141, 240, 159, 145, 166, 240, 159, 143, 188, 240, 159, 135, 181, 240, 159, 135, 177]`.

Wiadomość `Move(Down)` zserializowana zostanie jako ciąg bajtów `[3, 2]`.

Klient po podłączeniu się do serwera zaczyna obserwować rozgrywkę, jeżeli ta jest w toku. W przeciwnym razie może zgłosić chęć wzięcia w niej udziału, wysyłając komunikat `Join`.

Serwer ignoruje komunikaty `Join` wysłane w trakcie rozgrywki. Serwer ignoruje również komunikaty typu innego niż `Join` w `Lobby`.

2.2. Komunikaty od serwera do klienta

```
enum ServerMessage {
    [0] Hello {
        server_name: String,
        players_count: u8,
        size_x: u16,
        size_y: u16,
        game_length: u16,
        explosion_radius: u16,
        bomb_timer: u16,
    },
    [1] AcceptedPlayer {
        id: PlayerId,
        player: Player,
    },
    [2] GameStarted {
            players: Map<PlayerId, Player>,
    },
    [3] Turn {
            turn: u16,
            events: List<Event>,
    },
    [4] GameEnded {
            scores: Map<PlayerId, Score>,
    },
}
```

Wiadomość od serwera typu `Turn`

```
ServerMessage::Turn {
        turn: 44,
        events: [
            Event::PlayerMoved {
                id: PlayerId(3),
                position: Position(2, 4),
            },
            Event::PlayerMoved {
                id: PlayerId(4),
                position: Position(3, 5),
            },
            Event::BombPlaced {
                id: BombId(5),
                position: Position(5, 7),
            },
        ],
```

będzie miała następującą reprezentację binarną:

```
[3, 0, 44, 0, 0, 0, 3, 2, 3, 0, 2, 0, 4, 2, 4, 0, 3, 0, 5, 0, 0, 0, 0, 5, 0, 5, 0, 7]

3 - rodzaj wiadomości od serwera (`Turn`)
0, 44 - numer tury
0, 0, 0, 3 - liczba zdarzeń
2 - rodzaj zdarzenia (`PlayerMoved`)
3 - id gracza
0, 2 - współrzędna x
0, 4 - współrzędna y
2 - rodzaj zdarzenia (`PlayerMoved`)
4 - id gracza
0, 3 - współrzędna x
0, 5 - współrzędna y
0 - rodzaj zdarzenia (`BombPlaced`)
0, 0, 0, 5 - id bomby
0, 5 - współrzędna x
0, 7 - współrzędna y
```

Dostarczymy program do weryfikowania poprawności danych.

2.3. Definicje użytych powyżej rekordów

```
Event:
[0] BombPlaced { id: BombId, position: Position },
[1] BombExploded { id: BombId, robots_destroyed: List<PlayerId>, blocks_destroyed: List<Position> },
[2] PlayerMoved { id: PlayerId, position: Position },
[3] BlockPlaced { position: Position },

BombId: u32
Bomb: { position: Position, timer: u16 },
PlayerId: u8
Position: { x: u16, y: u16 }
Player: { name: String, address: String }
Score: u32
```

Pole `address` w strukturze `Player` może reprezentować zarówno adres IPv4, jak i adres IPv6.

Liczba typu `Score` informuje o tym, ile razy robot danego gracza został zniszczony.

2.4. Generator liczb losowych

Do wytwarzania wartości losowych należy użyć poniższego deterministycznego generatora liczb 32-bitowych. Kolejne wartości zwracane przez ten generator wyrażone są wzorem:

```
r_0 = (seed * 48271) mod 2147483647
r_i = (r_{i-1} * 48271) mod 2147483647
```

gdzie wartość `seed` jest 32-bitowa i jest przekazywana do serwera za pomocą parametru `-s`. Jeśli ten parametr nie jest zdefiniowany, można jako wartości domyślnej użyć dowolnej liczby, która będzie zmieniać się przy każdym uruchomieniu, np.

`unsigned seed = time(NULL)` (C) lub `unsigned seed = std::chrono::system_clock::now().time_since_epoch().count()` (C++).

Powyższy generator odpowiada generatorowi `std::minstd_rand`.

Należy użyć dokładnie takiego generatora, żeby umożliwić automatyczne testowanie rozwiązania (uwaga na konieczność wykonywania pośrednich obliczeń na typie 64-bitowym).

Przykłady użycia generatora zostały podane w plikach `c/random.c` oraz `cpp/random.cpp`.

2.5. Stan gry

Serwer jest „zarządcą" stanu gry, do klientów przesyła informacje o zdarzeniach. Klienci je agregują i przesyłają zagregowany stan do interfejsu użytkownika. Interfejs nie przechowuje w ogóle żadnego stanu.

Serwer powinien przechowywać następujące informacje:

- lista graczy (nazwa, adres IP, numer portu)
- stan generatora liczb losowych (innymi słowy stan generatora NIE restartuje się po każdej rozgrywce)

Oraz tylko w przypadku toczącej się rozgrywki:

- numer tury
- lista wszystkich tur od początku rozgrywki
- pozycje graczy
- liczba śmierci każdego gracza
- informacje o istniejących bombach (pozycja, czas)
- pozycje istniejących bloków

Lewy dolny róg planszy ma współrzędne `(0, 0)`, odcięte rosną w prawo, a rzędne w górę.

Klient powinien przechowywać zagregowany stan tak, aby móc wysyłać komunikaty do GUI. W szczególności klient powinien pamiętać, ile razy dany robot został zniszczony (aby móc wysłać tę informację w polu `scores`).

2.6. Podłączanie i odłączanie klientów

Klient wysyła komunikat `Join` do serwera po otrzymaniu dowolnego (poprawnego) komunikatu od GUI, o ile klient jest w stanie `Lobby` (tzn. nie otrzymał od serwera komunikatu `GameStarted`).

Po podłączeniu klienta do serwera serwer wysyła do niego komunikat `Hello`. Jeśli rozgrywka jeszcze nie została rozpoczęta, serwer wysyła komunikaty `AcceptedPlayer` z informacją o podłączonych graczach. Jeśli rozgrywka już została rozpoczęta, serwer wysyła komunikat `GameStarted` z informacją o rozpoczęciu rozgrywki, a następnie wysyła wszystkie dotychczasowe komunikaty `Turn`.

Jeśli rozgrywka nie jest jeszcze rozpoczęta, to wysłanie przez klienta komunikatu `Join` powoduje dodanie go do listy graczy. Serwer następnie rozsyła do wszystkich klientów komunikat `AcceptedPlayer`.

Graczom nadawane jest ID w kolejności podłączenia (tzn. odebrania komunikatu `Join` przez serwer). Dwoje graczy może mieć taką samą nazwę. Ponieważ klienci łączą się z serwerem po TCP, wiadomo który komunikat przychodzi od którego klienta.

Odłączenie gracza w trakcie rozgrywki powoduje tylko tyle, że jego robot przestaje się ruszać. Odłączenie klienta-gracza przed rozpoczęciem rozgrywki nie powoduje skreślenia go z listy graczy. Odłączenie klienta-obserwatora nie wpływa na działanie serwera.

2.7. Rozpoczęcie partii i zarządzanie podłączonymi klientami

Partia rozpoczyna się, gdy odpowiednio wielu graczy się zgłosi. Musi być dokładnie tylu graczy, ile jest wyspecyfikowane przy uruchomieniu serwera.

Inicjacja stanu gry przebiega następująco:

```
 nr_tury = 0
zdarzenia = []

dla każdego gracza w kolejności id:
    pozycja_x_robota = random() % szerokość_planszy
    pozycja_y_robota = random() % wysokość_planszy

    dodaj zdarzenie `PlayerMoved` do listy

tyle razy ile wynosi parametr `initial_blocks`:
    pozycja_x_bloku = random() % szerokość_planszy
    pozycja_y_bloku = random() % wysokość_planszy

    jeśli nie ma bloku o pozycji (pozycja_x_bloku, pozycja_y_bloku) na liście:
        dodaj zdarzenie BlockPlaced do listy

wyślij komunikat `Turn`
```

2.8. Przebieg partii

Zasady:

- Nie ma ograniczenia na liczbę bloków i bomb.
- Gracze nie mogą wchodzić na pole, które jest zablokowane. Mogą natomiast z niego zejść, jeśli znajdą się na nim, wskutek zablokowania go lub „odrodzenia" się na nim.
- Gracze nie mogą wychodzić poza planszę.
- Wielu graczy może zajmować to samo pole.
- Bomby mogą zajmować to samo pole.
- Gracze mogą położyć bombę, nawet jeśli stoją na zablokowanym polu (czyli na jednym polu może być blok, wielu graczy i wiele bomb)
- Na danym polu może być maksymalnie jeden blok

```
zdarzenia = []

dla każdej bomby:
    zmniejsz jej licznik czasu o 1
    jeśli licznik wynosi 0:
        zaznacz, które bloki znikną w wyniku eksplozji
        zaznacz, które roboty zostaną zniszczone w wyniku eksplozji
        dodaj zdarzenie `BombExploded` do listy
        usuń bombę

dla każdego gracza w kolejności id:
    jeśli robot nie został zniszczony:
        jeśli gracz wykonał ruch:
            obsłuż ruch gracza i dodaj odpowiednie zdarzenie do listy
    jeśli robot został zniszczony:
        pozycja_x_robota = random() % szerokość_planszy
        pozycja_y_robota = random() % wysokość_planszy

        dodaj zdarzenie `PlayerMoved` do listy

zwiększ nr_tury o 1
```

W wyniku eksplozji bomby zostają zniszczone wszystkie roboty w jej zasięgu oraz jedynie najbliższe bloki w jej zasięgu. Eksplozja bomby ma kształt krzyża o długości ramienia równej parametrowi `explosion_radius`. Jeśli robot stoi na bloku, który zostanie zniszczony w wyniku eksplozji, to taki robot również jest niszczony.

Intuicyjnie oznacza to, że można się schować za blokiem, ale położenie bloku pod sobą nie chroni przed eksplozją.

Przykłady:

```
@ - blok
A, B, C... - bomby
1, 2, 3... - gracze
x - eksplozja
```

```
.@2..
..1..
@@A.@
..@..
.....
```

Pola oznaczone jako eksplozja po wybuchu A z promieniem równym 2:

```
.@x..
..x..
@xxxx
..x..
.....
```

A zatem zniszczone zostaną 3 bloki i oba roboty.

Jeśli na polu jest bomba, blok i jacyś gracze, to wybuch bomby zniszczy blok i wszystkich graczy na tym polu stojących.

```
 @@@@@
@@AB@
.@@@@
```

Jednoczesna eksplozja A i B z promieniem równym 2:

```
 @@xx@
@xxxx
.@xx@
```

Po eksplozji:

```
 @@..@
@....
.@..@
```

Eksplozja jednej bomby nie powoduje eksplozji bomb sąsiednich. Jeśli kilka bomb wybucha w jednej turze, to skutki eksplozji są sumą teoriomnogościową pojedynczych eksplozji rozpatrywanych osobno. W powyższym przykładzie widać że blok o współrzędnych (0, 1) nie został zniszczony.

2.9. Wykonywanie ruchu

Serwer przyjmuje informacje o ruchach graczy w następujący sposób: przez `turn_duration` milisekund oczekuje na informacje od graczy. Jeśli gracz w tym czasie nie wyśle odpowiedniej wiadomości, to w danej turze jego robot nic nie robi. Jeśli w tym czasie gracz wyśle więcej niż jedną wiadomość, to pod uwagę brana jest tylko ostatnia.

To serwer decyduje o tym, czy dany ruch jest dozwolony czy nie. Jeśli gracz stojący na krawędzi planszy wyśle komunikat, który spowodowałby wyjście robota poza planszę, to serwer komunikat ignoruje. Podobnie jeśli spróbuje wejść na zablokowane pole.

2.10. Kończenie rozgrywki

Po `game_length` turach serwer wysyła do wszystkich klientów wiadomość `GameEnded` i wraca do stanu `Lobby`. Klienci, którzy byli do tej pory graczami, przestają nimi być, ale oczywiście mogą się z powrotem zgłosić przy pomocy komunikatu `Join`. Wszystkie komunikaty otrzymane w czasie ostatniej tury rozgrywki są ignorowane.

2.11. Błędy w komunikacji

Co jeśli klient prześle komunikat o nieprawidłowym formacie? Czy należy wtedy uznać go za odłączonego? Tak, bo ponieważ protokół jest binarny i po napotkaniu jakichkolwiek nieprawidłowych danych nie da się dowiedzieć, od którego momentu dane z powrotem są prawidłowe, jedyne co można zrobić to odłączyć klienta.

3. Protokół komunikacyjny pomiędzy klientem a interfejsem użytkownika

Komunikacja z interfejsem odbywa się po UDP przy użyciu komunikatów serializowanych tak jak wyżej.

Klient wysyła do interfejsu graficznego następujące komunikaty:

```
enum DrawMessage {
    [0] Lobby {
        server_name: String,
        players_count: u8,
        size_x: u16,
        size_y: u16,
        game_length: u16,
        explosion_radius: u16,
        bomb_timer: u16,
        players: Map<PlayerId, Player>
    },
    [1] Game {
        server_name: String,
        size_x: u16,
        size_y: u16,
        game_length: u16,
        turn: u16,
        players: Map<PlayerId, Player>,
        player_positions: Map<PlayerId, Position>,
        blocks: List<Position>,
        bombs: List<Bomb>,
        explosions: List<Position>,
        scores: Map<PlayerId, Score>,
    },
}
```

Explosions w komunikacie `Game` to lista pozycji, na których robot by zginął, gdyby tam stał.

Klient powinien wysłać taki komunikat po każdej zmianie stanu (tzn. otrzymaniu wiadomości `Turn` jeśli rozgrywka jest w toku lub `AcceptedPlayer` jeśli rozgrywka się nie toczy).

Interfejs wysyła do klienta następujące komunikaty:

```
enum InputMessage {
    [0] PlaceBomb,
    [1] PlaceBlock,
    [2] Move { direction: Direction },
}
```

Są one wysyłane za każdym razem, gdy gracz naciśnie odpowiedni przycisk.

Można założyć, że komunikaty zmieszczą się w jednym datagramie UDP. Każdy komunikat wysyłany jest w osobnym datagramie.

4. Ustalenia dodatkowe

Program klienta w przypadku błędu połączenia z serwerem gry lub interfejsem użytkownika powinien się zakończyć z kodem wyjścia 1, uprzednio wypisawszy zrozumiały komunikat na standardowe wyjście błędów.

Program serwera powinien być odporny na sytuacje błędne, które dają szansę na kontynuowanie działania. Intencja jest taka, że serwer powinien móc być uruchomiony na stałe bez konieczności jego restartowania, np. w przypadku kłopotów komunikacyjnych, czasowej niedostępności sieci, zwykłych zmian jej konfiguracji itp.

Serwer nie musi obsługiwać więcej niż 25 podłączonych klientów (graczy + obserwatorów) jednocześnie.

Programy powinny umożliwiać komunikację zarówno przy użyciu IPv4, jak i IPv6.

Można korzystać z biblioteki `Boost`, w szczególności z modułu `asio`.

Rozwiązanie ma kompilować się i działać na serwerze students.

Rozwiązania należy kompilować z flagami `-Wall -Wextra -Wconversion -Werror -O2`.

Rozwiązania napisane w języku C++ powinny być kompilowane z flagą `-std=gnu++20`, a w języku C z flagą `-std=gnu17` przy użyciu `GCC 11.2` lub nowszego (na students w katalogu `/opt/gcc-11.2/bin`.

Rozwiązanie powinno być odpowiednio sformatowane (można użyć np. `clang-format`).

Dodatkowo polecamy używanie lintera (np. `clang-tidy`, który jest zintegrowany z `CLionem`) i/lub kompilowanie z flagą `-fanalyzer`.

5. Oddawanie rozwiązania

Jako rozwiązanie można oddać tylko klienta (część A) lub tylko serwer (część B), albo obie części.

Termin oddawania części A to 23.05, a termin oddawania części B to 07.06 (siódmy czerwca).

Jako rozwiązanie należy dostarczyć pliki źródłowe oraz plik `makefile` ALBO `CMakeLists.txt`, które należy umieścić jako skompresowane archiwum w Moodle. Archiwum powinno zawierać tylko pliki niezbędne do zbudowania programów. Nie wolno w nim umieszczać plików binarnych ani pośrednich powstających podczas kompilowania programów.

Po rozpakowaniu dostarczonego archiwum, w wyniku wykonania w jego głównym katalogu polecenia `make` (`cmake . && make` jeśli używa się `CMake`), dla części A zadania ma powstać w tym katalogu plik wykonywalny `robots-client` a dla części B zadania – plik wykonywalny `robots-server`.

`makefile` powinien obsługiwać cel `clean`, który po wywołaniu kasuje wszystkie pliki powstałe podczas kompilowania.

6. Ocena

Za rozwiązanie części A zadania można dostać maksymalnie 10 punktów. Za rozwiązanie części B zadania można dostać maksymalnie 15 punktów. Każda część zadania będzie testowana i oceniana osobno. Ocena każdej z części zadania będzie się składała z trzech składników:

1. ocena wzrokowa i manualna działania programu (20%)
2. testy automatyczne (50%)
3. jakość kodu źródłowego (30%)

6.1. Ocena wzrokowa i manualna działania programu

- jak program reaguje, gdy zostanie wywołany z bezsensownymi argumentami? (Najlepiej jeśli wypisuje jakiś komunikat o błędzie; ważne żeby nie było segfaulta)
- czy w grę rzeczywiście da się grać

6.2. Testy automatyczne

Testy będą obejmowały m.in.: - bardzo proste scenariusze testowe (czy podłączenie gracza do serwera powoduje wysłanie odpowiedniego komunikatu do klientów, czy otrzymanie wiadomości od interfejsu powoduje wysłanie wiadomości do serwera, czy otrzymanie wiadomości od serwera powoduje wysłanie wiadomości do klienta itd., czy programy prawidłowo resolvują nazwy domenowe (np. localhost), czy można się połączyć zarówno po IPv4 jak i IPv6) - proste scenariusze testowe (symulacja krótkiej rozgrywki z jednym graczem, czy generowanie planszy odbywa się zgodnie z powyższym opisem; czy wybuch bomby jest prawidłowo obliczany, czy prawidłowo obsługiwane są znaki spoza zakresu ASCII) - złożone scenariusze testowe (symulacja kilku rozgrywek z wieloma graczami)

6.3. Jakość kodu źródłowego

- absolutne podstawy: kod powinien być jednolicie sformatowany (najlepiej użyć do tego clang-format lub formatera wbudowanego w cliona), nie wyciekać pamięci, po skompilowaniu z parametrami `-Wall -Wextra` nie powinno być

żadnych ostrzeżeń. Dodatkowo można sprawdzić sobie program przy użyciu lintera `clang-tidy`
- kod powinien być sensownie podzielony na funkcje, nazwy funkcji i zmiennych powinny być znaczące (a nie np. a, b, x, y, temp) i w jednym języku
- komentarze powinny być w jednym języku
- „magiczne stałe” powinny być ponazywane
- „Parse, don’t validate”
- jeśli kod napisany jest w C++, to należy przestrzegać konwencji programowania w tym języku

7. FAQ

- P: Klient może wysłać do serwera bardzo dużo ruchów (bo np. gracz wciska szybko różne strzałki), zatem nawet jak na bieżąco odczytujemy dane z socketu, to po upływie tych turn-duration milisekund, w sockecie wciąż mogą zalegać ruchy. Czy przechodzą one na następną turę? Dla przykładu, robię ruchy LPDLLPDGGLPDG, więc też takie trafią do socketu po stronie serwera, i przed upływem turn-duration ms, serwer przetworzył LPDL, więc przyjmuejmy, że w tej turze gracz robi ruch L. Czy pozostałe ruchy zalegające w sockecie (LPDGGLPDG) przechodzą na następną turę?
- O: Możemy założyć, że zależy to od implementującego, bo testy automatyczne będziemy uruchamiać z dostatecznie długimi turami (rzędu 1s), żeby to się na pewno nie zdarzyło
- P: Jak rozumiem, gra się zaczyna po tym jak serwer dostanie players-count komunikatów Join. Co jeśli przyjdzie więcej komunikatów Join? Mamy je zignorować?
- O: Tak, serwer ignoruje komunikaty Join w momencie, gdy rozgrywka jest w trakcie
- P: Odłączanie graczy rozpoznajemy po tym, że read/write z socketu TCP zwróci 0?
- O: Tak
- P: Kiedy mamy zapomnieć o istnieniu danego klienta? Jeśli dobrze rozumiem, to jeśli obserwator (czyli ktoś, kto nawiązał połączenie TCP z serwerem, ale nie wysłał jeszcze komunikatu Join) się odłączy to możemy zapomnieć o nim. Jeśli gracz się odłączy to ślad po nim (tj. pozycja robota itp.) istnieje do końca obecnej gry, ale po jej zakończeniu, możemy o nim zapomnieć?
- O: Dokładnie tak
- P: Jeśli gra się jeszcze nie rozpoczęła i podłączy się nowy klient, to jak rozumiem, należy wysłać do niego komunikat Hello i serię komunikatów AcceptedPlayer, by poinformować o tym jacy są obecnie gracze w Lobby. Jeśli w odpowiedzi na to, klient prześle Join to należy do wszystkich obserwatorów i graczy wysłać AcceptedPlayer, żeby wszyscy się dowiedzieli o nowym graczu. Dobrze rozumiem?
- O: Tak właśnie
- P: Co jeśli wybuchnie bomba, a na jej "drodze wybuchu" będzie znajdować się inna bomba?
- O: Nic (to znaczy wybuch jednej bomby nie powoduje wybuchu innych bomb ani ich nie niszczy)
- P: Rekord Player: { name: String, address: String }. Czy jest jakaś specyfikacja jak powinien wyglądać adres IPv4/IPv6? Czy można założyć, że dopuszczalny będzie po prostu output z funkcji inet_ntop?
- O: Tak
- P: Co zrobić, gdy GUI wyśle komunikat, którego nie da się sparsować, do klienta?
- O: Zignorować
- P: Co zrobić, gdy serwer wyśle komunikat, którego nie da się sparsować, do klienta?
- O: Rozłączyć się, bo po niepoprawnym komunikacie nie wiadomo, kiedy miałby zacząć się poprawny komunikat
- P: Co zrobić, gdy serwer wyśle komunikat, który da się sparsować, ale nie ma sensu? (np. wybucha bomba, która miała jeszcze 10 tur na timerze lub gracz zostaje przeniesiony nagle na drugi koniec planszy)
- O: Serwer zawsze ma rację
- P: Co ma robić klient jak jest w trakcie gry a dostanie od serwera komuikat AcceptedPlayer/GameStarted?
- O: Zależy od implementacji
- P: Obliczanie score w kliencie to nie jest tak proste, że się sprawdza ile razy przyszedł komunikat o zabiciu gracza, tylko score to ilość tur, gdzie występuję przynajmniej jeden taki komunikat?
- O: Tak
- P: Czy id graczy się resetują przy nowej grze?
- O: Tak
- P: Czy dwa bloki o takich samych współrzędnych są traktowane jako jeden blok, czy jako dwa różne?
- O: Na danym polu może stać tylko jeden blok.
- P: Czy jeśli w trakcie tury klient wyśle wiele komunikatów i część z nich jest poprawna, część nie, ale ostatni jest niepoprawny (wykonuje niedozwolony ruch), to serwer ma wziąć pod uwagę ostatni poprawny ruch wysłany w tej turze, czy zignorować wszystkie, bo ostatni wysłany był niepoprawny?
- O: Wysłanie komunikatu niepoprawnego składniowo powoduje rozłączenie klienta. Komunikat poprawny składniowo, ale niemający sensu (np. join w czasie gry) jest ignorowany. Komunikat sensowny może oznaczać chęć wykonania niedozwolonego ruchu (wyjścia poza planszę, wejścia na blok, zablokowania zablokowanego pola), ale nie zmienia to faktu, że jest sensowny. W czasie gry liczy się ostatni nadesłany sensowny komunikat, niezależnie od tego, czy spowoduje poprawny ruch czy nie.
- P: Czy możemy być pewni, że wiadomość od GUI przyszła z podanego adresu i wiadomości do GUI są wysyłane z podanego portu? Innymi słowy, czy wiadomości od GUI mamy odbierać przez receive, czy receive_from (i analogicznie wysyłać przez send, czy send_to)?
- O: Adres i port GUI, które podaje się w kliencie, służą do wysyłania wiadomości od klienta do GUI. GUI może wysyłać komunikaty z portów efemerycznych. Ale ogólnie najlepiej nic nie zakładać o adresie GUI i być gotowym na odbieranie

(poprawnych) wiadomości od kogokolwiek

- P: Czy możemy założyć, że rozmiar planszy będzie zawierał się w praktycznych wymiarach? Plansza o maksymalnych wymiarach ma kilka miliardów pół co z punktu widzenia gry jest zupełnie niepraktyczne, a utrudnia implementacje logiki gry, gdy musimy założyć, że powinna działać dla takich wymiarów. Ujmując problem inaczej: czy możemy założyć, że deklaracja `T board[size_x][size_y]` , gdzie T jest typem o rozsądnej wielkości będzie poprawna?
- O: Nie wydaje mi się, żeby tworzenie takiej tablicy dwuwymiarowej było do czegokolwiek potrzebne.
- P: Co się dzieje, kiedy ktoś podłączy się w trakcie gry, jest to dozwolone? W treści jest zdanie: `Po podłączeniu klienta do serwera serwer wysyła do niego komunikat Hello. Jeśli rozgrywka jeszcze nie została rozpoczęta, serwer wysyła komunikaty AcceptedPlayer z informacją o podłączonych graczach. Jeśli rozgrywka już została rozpoczęta, serwer wysyła komunikat GameStarted z informacją o rozpoczęciu rozgrywki, a następnie wysyła komunikat Turn z informacją o aktualnym stanie gry. Numer tury w takim komunikacie to 0` . Czy jeśli rozgrywka trwa, a podłączy się klient-obserwator, to a) dostaje komunikat Hello, Game Started, a później kolejne tury (tak jak gracze) b) komunikat Hello, później kolejne Tury (jak gracze) c) komunikat Hello, Game Started i tury numerowane od 0?
- O: a)
- P: Czy klient-obserwator może wysyłać jakieś komunikaty w trakcie gry?
- O: Może, ale będą ignorowane
- P: Komunikat Game do GUI w polu explosions powinien przekazywać tylko wybuchy z poprzedniej tury, tak? Czyli odebranie komunikatu bomb exploded między innymi dla klienta oznacza "zapomnienie" o danej bombie i wrzucenie jej pozycji do explosions?
- O: Tak
- P: Klient powinien niezależnie od serwera kontrolować timer bomb i co turę zmniejszać go o 1, nawet patrząc na to, że dostanie komunikat od serwera, gdy bomba wybuchnie?
- O: Tak. Jak wybuchnie bomba, która nie powinna wybuchnąć, to jest UB (ale można założyć, że serwer ma zawsze rację)
- P: Mam mały problem z gui - roboty się w nim nie wyświetlają. Przesyłam przykład, plansza na której powinien być tylko robot. Ostatnia wiadomość otrzymana przez gui: `2022-05-12T14:04:23.246721Z INFO gui: {"Game":{"server_name":"zabawownia","size_x":10,"size_y":10,"game_length":1000,"turn":10,"players":{"0":{"name":"michal","socket_addr":"127.0.0.1:42704"}},"player_positions":{"0":[3,3]},"blocks":[],"bombs":[],"explosions":[],"scores":{}}}`
    - O: W scores musi być player.
    - P: Czy klient może połączyć się z serwerem zanim otrzyma wiadomość od gui?
    - O: Klient łączy się z serwerem od razu po uruchomieniu
    - P: Czy klient może wysyłać Join wielokrotnie?
    - O: Może, ale to bez sensu
    - P: Jak powinna zachowywać się bomba wybuchająca w bloku - niszczy ten blok i nie propaguje eksplozji dalej, czy niszczy blok i rozszerza eksplozję do swojego maksymalnego zasięgu?(Oczywiście z pominięciem ingerencji innych bloków)
    - O: Niszczy i nie propaguje (ale roboty stojące na tym bloku są niszczone)
    - P: Czy po zakończeniu rozgrywki klient ma wyświetlić lobby, czy planszę, a jeśli lobby, to jakie jest zastosowanie mapy scores w wiadomości GameEnded?
    - O: Lobby, wiadomość jest potrzebna do testowania, bo inaczej nie da się dowiedzieć jakie były wyniki po ostatniej turze
    - P: Co robi klient jeżeli otrzyma wiadomość której się nie spodziewał (np. GameEnded zanim otrzymał GameStarted, Turn przed GameStarted, Hello po otrzymaniu początkowego, pierwszego Hello)?
    - O: UB
    - P: Jak mamy postępować z wiadomościami które zostały zbudowane poprawnie, ale zawierają ewidentnie niepoprawne wartości (np punkt leżący poza mapą, odwołanie do id gracza lub bomby która nie istnieje)?
    - O: UB, można zignorować
    - P: Co zrobić z wiadomością GameStarted/GameEnded, które zawierają id graczy od których nie otrzymaliśmy komunikatu AcceptedPlayer?
    - O: UB
    - P: Czy klient może wysyłać do serwera w stanie lobby wiadomości nie będące join?
    - O: Może, ale zostaną zignorowane (chodzi o to, że mogą np. dojść z opóźnieniem z ostatniej tury, kiedy serwer wróci już do stanu lobby)
    - P: Jak klient ma postępować z bombami które zostały mu przesłane, ale nie wybuchły, mimo tego, że ich timer spadł poniżej zera?
    - O: UB

06.06.2022

- P: Czy można użyć biblioteki PFR https://github.com/boostorg/pfr) z Boosta nowszego niż jest na students?
- O: Tak, można założyć, że `#include "boost/pfr.hpp"` będzie działać w środowisku testowym. Nie należy dołączać biblioteki do paczki z rozwiązaniem.
- P: Czy można użyć jakichś innych bibliotek header-only, nie wchodzących w skład Boosta 1.74?
- O: Nie

08.06.2022

- P: Jak wygląda koniec gry?
- O: Serwer wysyła Turn z turn=game_length-1 czeka na ruchy użytkowników, przetwarza, wysyła Turn z turn=game_length i od razu (nic się już nie zmienia) wysyła GameEnded z wynikami. Wówczas wyniki wysłane przez serwer powinny być takie jak obliczone w kliencie.
- P: Czy można prosić o przykład, jak powinna wyglądać jakaś krótka rozgrywka?
- O: Załóżmy, że serwer został uruchomiony z parametrami `-b 1 -c 1 -l 3` . Do serwera w stanie lobby podłącza się klient. Otrzymuje on komunikat `Hello`. Serwer nadal jest w stanie lobby, dopóki klient nie wyśle komunikatu `Join`. Wówczas serwer wysyła kolejno komunikat `AcceptedPlayer`, przechodzi w stan rozgrywki i wysyła kolejno: `GameStarted`, `Turn 0` (z początkową pozycją gracza i bloku). Zaczyna się tura 1. Załóżmy, że w czasie jej trwania gracz kładzie bombę. Serwer wówczas wyśle turę o numerze 1 ze zdarzeniem `BombPlaced`. W i-tej (gdzie i >= 1) turze najpierw jest faza otrzymywania wiadomości od klientów, a później podsumowanie tej fazy poprzez wysłanie komunikatu do klientów. Rozpoczyna się druga tura. Załóżmy, że teraz dołącza drugi klient, który staje się obserwatorem. Serwer wyśle do niego kolejno: `Hello`, `GameStarted`, `Turn 0`, `Turn 1` (tury wysyłane są z identycznymi zdarzeniami jak do pierwszego klienta - żeby obserwator mógł "nadrobić" stan gry). Teraz serwer wyśle do obu klientów turę o numerze 2 ze zdarzeniami `BombExploded`, `PlayerMoved` (bo gracz się "odrodzi" na innym polu - jeśli w czasie trwania drugiej tury chciał wykonać jakiś ruch, to zostanie on zignorowany, bo robot w tej turze został zniszczony) oraz `BlockDestroyed` (o ile blok został zniszczony). Rozpoczyna się trzecia tura. Załóżmy, że teraz gracz się poruszył. Wówczas serwer wyśle do obu klientów turę (o numerze 3) ze zdarzeniem `PlayerMoved` i od razu `GameEnded`.