Tanks - dokumentacja

Stanisław Nieradko, Bartłomiej Krawisz, Jakub Bronowski

1. Wprowadzenie

Celem projektu było stworzenie gry sieciowej czasu rzeczywistego, która będzie symulacją bitwy czołgów obsługującą kilku graczy. Gracze sterują czołgami, które poruszają się po planszy i strzelają do siebie. Gra kończy się zwycięstwem gracza który zdoła pokonać resztę graczy.

Projekt zaimplementowaliśmy w języku Python bez wykorzystynia zewnętrznych bibliotek. Gra działa w trybie serwer-klient, gdzie serwer zarządza grą, a klienci sterują czołgami. Zarówno serwer jak i klient wykorzystują natywną Pythonowi współbieżność opartą o wątki (wbudowana biblioteka `threading`) w celu płynnej i jednoczesnej komunikacji sieciowej oraz obsługi gry.

2. Model Komunikacji

Gra wykorzystuje protokół TCP do komunikacji między klientami a serwerem. Wybór ten został dokonany ze względu na wystarczającą wydajność względem UDP i prostotę implementacji. Podczas testowania eksperymentalnej implementacji opartej o UDP, zauważyliśmy że różnice w opóźnieniach były na tyle małe, że nie wpływały one znacznie na rozgrywkę.

Do serializowania zdarzeń wybraliśmy format JSON z uwagi na czytelność danych oraz nieograniczone możliwości rozbudowy. W celu odseparowania poszczególnych zdarzeń, każde zdarzenie jest zakończone znakiem `NULL`.

Nazwa	Typ	Opis
`event_type`	str	Typ zdarzenia (możliwe typy zdarzenia: `connect`, `refuse`, `ping`, `pong`, `setPlayerId`, `gameState`, `disconnect`)
`time`	int	Czas zdarzenia [unix timestamp]
`data`	object	Dane zdarzenia (zależne od `event_type`)

Table 1: Struktura zdarzenia

Nazwa	Typ	Opis
`serverTime`	int	Czas serwera [unix timestamp]

Table 2: Struktura obiektu `data` dla zdarzenia `pong`

Nazwa	Typ	Opis
`playerId`	int	Identyfikator gracza

Table 3: Struktura obiektu `data` dla zdarzenia `setPlayerId`

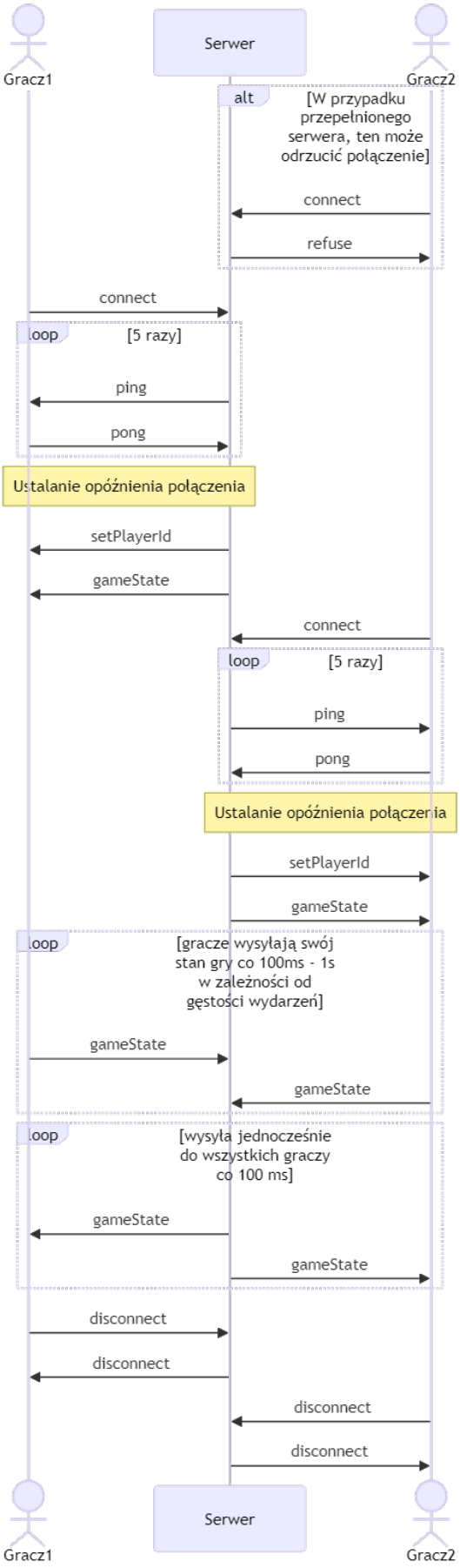
Nazwa	Typ	Czy opcjonalne	Opis
`tanks`	dic[int, Tank]	Tak	Obiekty czołgów
`bullets`	list[Bullet]	Tak	Tablica obiektów pocisków
`map`	list[Obstacle]	Tak	Tablica przeszkód na mapie
`isGameOver`	bool	Nie	Czy gra się zakończyła?

Table 4: Struktura obiektu `data` dla zdarzenia `gameState`

Przykładowe zdarzenia:

```
// Zdarzenie connect
{"eventType":"connect","time":1711291958}

// Zdarzenie refuse
{"eventType":"refuse","time":1711291958}

// Zdarzenie ping
{"eventType":"ping","time":1711293829}

// Zdarzenie pong
{"eventType":"pong","time":1711295934,"data":{"serverTime":1711293829}}

// Zdarzenie setPlayerId
{"eventType":"setPlayerId","time":1711295934,"data":{"playerId":2}}

// Zdarzenie gameState
{
  "eventType": "gameState",
  "time": 1711293829,
  "data": {
      "tanks": {
        "1": {
            "x": 115.2,
            "y": 254.32,
            "direction": 1,
            "speed": 100,
            "score": 50,
            "alive": true
        },
        "2": {
            "x": 34.2,
            "y": 74.32,
            "direction": 0,
            "speed": 0,
            "score": 0,
            "alive": false
        }
      },
      "bullets": [
        {
            "x": 115.2,
            "y": 254.32,
            "direction": 1,
            "speed": 100,
            "playerId": 2
        },
        {
            "x": 34.2,
            "y": 74.32,
            "direction": 0,
            "speed": 0,
            "playerId": 1
        }
      ],
      "map": [{"x":51,"y":24,"type":1},{"x":23,"y":11,"type":3},{"x":0,"y":5,"type":0}],
      "isGameOver": false
  }
}
```

2.2 Diagram Sekwencji



2.3 Opis działania aplikacji

Gracze dołączają do serwera poprzez wysłanie zdarzenia `connect`. Następuje 5 krotna wymiana zdarzeń `ping` (ze strony serwera) oraz `pong` (ze strony użytkonika) w celu ustalenia opóźnienia (z ang. *latency*) połączenia. Następnie serwer przesyła zdarzenie `setPlayerId` (z przypisanym identyfikatorem gracza) oraz zdarzenie `gameState` (z aktualnym stanem oraz mapą gry) do użytkownika. Jeżeli okazałoby się że serwer jest przepełniony zamiast tego zostanie przesłane zdarzenie `refuse` zamykające połączenie.

Podczas gry gracze przesyłają lokalny stan swojego czołgu co 100ms - 1s (w zależności od ilości lokalnych zmian) w formie zdarzenia `gameState`. Serwer odbiera zdarzenia od wszystkich graczy, rozwiązuje konflikty, łączy je w jeden spójny stan i przesyła do wszystkich graczy w formie zdarzenia `gameState`. Ta operacja wykonywana jest co 100ms.

Po spełnieniu warunków zakończenia potyczki (pozostanie jeden gracz na planszy), serwer wysyła zdarzenie `gameState` z `isGameOver` ustawionym na `true`, co informuje klientów iż gra się zakończyła i po 3s serwer rozpoczyna nową grę pozwalając graczom na dalszą jej kontynuację.

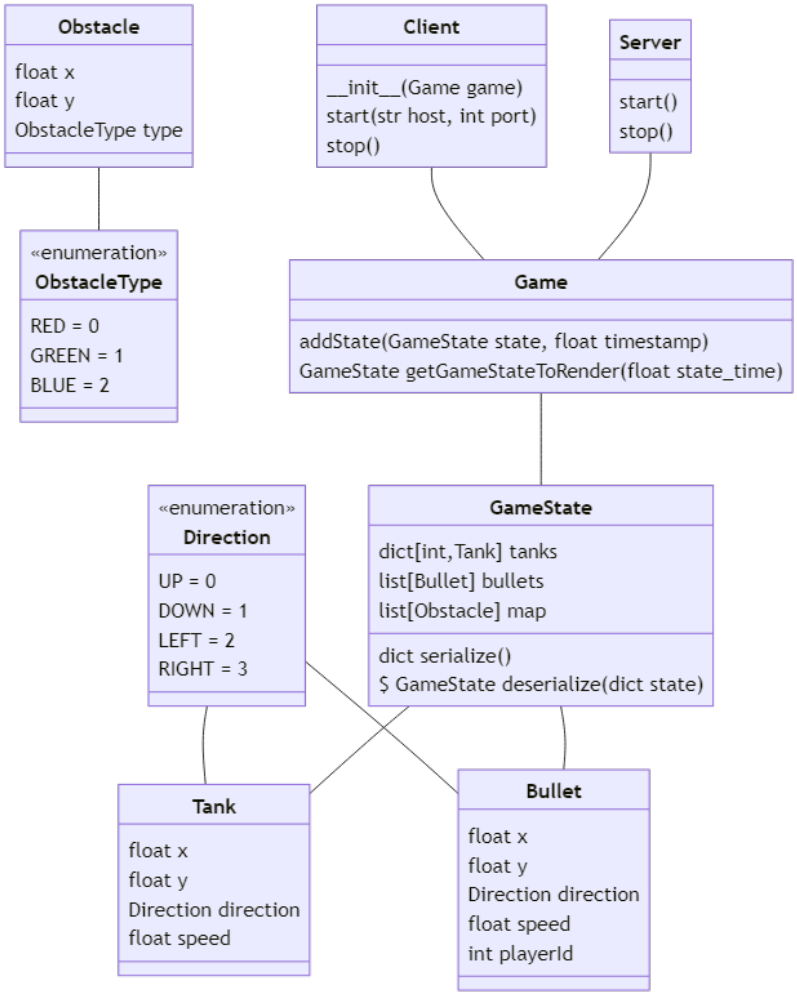
W celu rozłączenia się z serwerem gracz wysyła zdarzenie `disconnect` (w wyniku którego otrzymuje on także zwrotne zdarzenie `disconnect` od serwera jako potwierdzenie). Możliwe jest także otrzymanie takiego zdarzenia w przypadku awarii lub zakończenia pracy serwera (zostaje ono wysłane do wszystkich graczy).

2.3 Potencjalne elementy krytyczne

Najważniejszym pod względem spójności elementem gry jest obiekt `GameState` przechowujący jej stan. W celu ochrony jego spójności zastosowaliśmy mechanizm blokady Read-Write, umożliwiający jednoczesny odczyt przez wiele wątków oraz wyłączny zapis przez jeden wątek. Dzięki temu zapewniamy, że stan gry jest zawsze spójny i niezmienny podczas odczytu.

Kolejnym elementem krytycznym jest obsługa zdarzeń. W celu zapewnienia spójności, zdarzenia są przetwarzane w kolejności ich otrzymania w sposób synchroniczny (po jednym zdarzeniu na raz) przez główny wątek serwera po pobraniu z kolejki zdarzeń. Dzięki temu zapewniamy, że zdarzenia są przetwarzane w kolejności ich otrzymania i nie dochodzi do konfliktów.

2.3 Diagramy klas



3. FAQ

TCP vs UDP
Po przetestowaniu obu protokołów, zdecydowaliśmy się na TCP ze względu na mniejszą ilość problemów związanych z przesyłem danych oraz wystarczającą wydajność do naszych zastosowań.

Mimo, iż UDP jest szybszy, to implementacja gry w tym protokole wymagałaby dodatkowego nakładu pracy. W przypadku TCP, odnotowane opóźnienia względem implementacji korzystającej z UDP były na tyle małe, że nie miały one wpływu na rozgrywkę.

W jaki sposób radzimy sobie z sytuacją w której klient, przestał przesyłać informacje?
W przypadku gdy klient przestaje przesyłać informacje, serwer po 5s od ostatniego zdarzenia wysyła zdarzenie `disconnect` w celu rozłączenia klienta. W momencie wysyłania zdarzenia czołg klienta zostaje natychmiast usunięty z gry.

W międzyczasie serwer będzie "symulował" zachowanie czołgu poprzez ekstrapolację przez co najwyżej 0,5s. Po tym czasie czołg klienta zatrzyma się w miejscu.

W jaki sposób radzimy sobie z sytuacją w której serwer, przestał przesyłać informacje?
Klient symuluje zachowania czołgów innych graczy dzięki ekstrapolacji (przez maks. 0,5s). Jeżeli do tego czasu nie uda się przywrócić połączenia z serwerem, klient zakończy grę (lokalnie) i pokaże komunikat o problemach z połączeniem.

W jaki sposób radzimy sobie z opóźnieniami przesyłu i/lub zakolejkowanymi wiadomościami?
Serwer wykorzystuje asynchroniczny model przetwarzania zdarzeń, co pozwala na obsługę kolejnych zdarzeń podczas wysyłania i oczekiwania na ukończenie połączenia. Wszystkie zdarzenia są przetwarzane w kolejności

ich otrzymania, także w przypadku opóźnień po stronie klienta, tak szybko jak nadejdą opóźnione zdarzenia, zostaną one obsłużone.

Podczas gry, gdy zdarzenia klienta nie dochodzą do serwera, ten ekstrapoluje ruch czołgu (przez 0,5s) klienta na podstawie ostatnich danych otrzymanych od klienta i przesyła je do pozostałych graczy. Po oknie ekstrapolacji ale przed wyrzuceniem gracza z powodu nieaktywności, czołg klienta nie porusza się. W przypadku gdy klient znów zacznie przesyłać dane, serwer cofnie czołg klienta do pozycji sprzed maks. 0,5s i zaakceptuje dane od klienta tak długo jak nie są one starsze niż 0,5s. W innym wypadku pozycja czołgu klienta będzie taka sama jak po cofnięciu.

W jaki sposób werfyikujemy, że dane zostały przesłane w całości?
Dzięki wykorzystaniu JSON'a jako medium przesyłu danych, serwer jest w stanie sprawdzić czy dane zostały przesłane w całości. W przypadku błędu w przesłaniu danych, serwer odrzuca zdarzenie i oczekuje na kolejne. Każda wiadomość musi być zakończona znakiem `NULL` w celu odseparowania poszczególnych JSON'ów.

W jaki sposób radzimy sobie z sytuacją gdy pakiet "zagubi się" i nie dotrze poprawnie do/z serwera?
TCP zapewnia, że dane zostaną dostarczone w całości i w odpowiedniej kolejności. W przypadku gdy pakiet zostanie "zagubiony", TCP ponownie wyśle dane. W przypadku gdy pakiet nie dotrze do serwera, klient ponownie wyśle dane.

Nawet gdy w magiczny sposób było to możliwe lub dokonalibyśmy zmiany na protokół UDP, to i tak dane przesyłane są na tyle często (co 100ms) że nie ma to wpływu na rozgrywkę.