Projekt - Przetwarzanie Rozproszone

Snake Multiplayer

Sebastian Leśniewski 184711 & Piotr Sieński 184297

17.05.2022

1. Wstęp

W sprawozdaniu omówione są problemy dotyczące współbieżności i komunikacji między procesami na podstawie gry wieloosobowej będącej implementacją gry Snake z elementami rywalizacji. Podczas realizacji projektu użyliśmy następujących technologii:

- Aplikacja serwerowa w języku C
- Aplikacja klienta w języku Python
- Komunikacja między serwerem a klientem realizowana jest przy użyciu mechanizmu gniazdek sieciowych i protokołu komunikacji sieciowej TCP/IP

2. Problemy komunikacji i współbieżności

2.1. Protokół komunikacji

Komunikacja odbywa się przy pomocy TCP/IP , gdyż zapewnia gwarancję dotarcia wiadomości do odbiorcy oraz utrzymuje ich kolejność.
Jeśli nie otrzymamy odpowiedzi od klienta, po pewnym ustalonym czasie (33 ms) zapisujemy jego ruch zgodnie z poprzednim wektorem - gracz porusza się po prostej.
W sytuacji otrzymania wielu pakietów na raz poprawność stanu gry jest utrzymana, gdyż możliwe jest ustalenie kolejności pakietów. Dane zawsze docierają w całości, gdyż pakiet danych jest stałej wielkości.

2.2. Dostęp do sekcji krytycznej i przepływ sterowania

Legenda:
T - Tablica wektorów przesunięć.
C - Liczba graczy.
S1 - Semafor binarny początkowo otwarty, kontrolujący dostęp wątków klientów do **T.**
S2 - Semafor uogólniony, który na początku pętli głównej jest zerem, klienci zwiększają go o 1 a wątek główny zmniejsza go o liczbę równą liczbie graczy. Gwarantuje, że wątek główny przesunie graczy o zaktualizowane wektory.
S3 - Tablica semaforów binarnych (jeden semafor dla każdego z graczy) - kontrolujący dostęp do odpowiedzi przesłanej graczom przez odpowiednie wątki. Początkowo ustawiony na zero (wątki graczy czekają na aktualizację jego wartości przez wątek główny), następnie zmniejszany przez każdy wątek gracza który odczyta dane z zasobu.

S4 - Semafor kontrolujący dostęp do finalnej sekcji każdego wątku, zapewnia że żaden z wątków nie podejmie próby wysłania tabeli wyników przed uzupełnieniem jej przez główny wątek.

Zasoby objęte sekcją krytyczną:

- tablica wektorów przesunięć graczy **T**
- zwracana struktura stanu gry

Stan początkowy programu:
Tablica wektorów przesunięć jest pusta, a semafor **S1** otwarty, semafor **S2** zamknięty - czeka na nim wątek główny, w **S3** każdy zamknięty, **S4** zamknięty.

Przepływ sterowania:

1) Gracze czekają w poczekalni wymieniając okresowo dane z serwerem,
otrzymują aktualną ilość graczy w poczekalni

2) Wątki graczy po odczytaniu danych wejściowych modyfikują **T** przechodząc przez **S1**, po czym inkrementują **S2** i czekają na **S3**.

3) Gdy **S2** osiągnie odpowiednią wartość praca wątku głównego jest wznawiana. Przesunięcia wszystkich graczy są nanoszone na mapę, a następnie wyznaczane są kolizje - taki mechanizm zapewnia, że jeśli dwóch graczy wejdzie na to samo pole w tym samym czasie obaj zginą.

4) Na podstawie **T** i wyznaczonych kolizji kompletowana jest paczka odpowiedzi do klientów, po czym semafor **S3** jest podnoszony i wątek główny z powrotem czeka na **S2**.

5) Wątki graczy przechodzą przez **S3**, wysyłają paczkę danych do klientów i czekają na odebranie danych wejściowych.

6) Po zakończeniu rozgrywki wątek główny podnosi **S4** o liczbę graczy umożliwiając im wysłanie do swoich klientów tabeli wyników

3. Przebieg Komunikacji

KLIENT:

Klient próbuje połączyć się z serwerem. Jeśli otrzyma connection refused error to znaczy, że gra już trwa / serwer jest wyłączony. W przeciwnym razie wszystko przebiegło pomyślnie a klient w pętli zaczyna na zmianę otrzymywać strukturę lobby data informującą o liczbie graczy w poczekalni i wysyłać wiadomość potwierdzającą że nadal czeka na rozgrywkę. Nie wysłanie takiej wiadomości przez określony czas (2s) skutkuje odłączeniem od poczekalni.Następnie po otrzymaniu wiadomości w której liczba graczy w poczekalni = oczekiwanej liczbie graczy i oczekuje na otrzymanie paczki inicjalizacyjnej.

Struktura danych o stanie poczekalni

```
struct lobby_data{
	int current; // aktualna liczba połączonych graczy
	int target; // oczekiwana liczba połączonych graczy
};
```

SERWER:

Serwer akceptuje połączenie od gracza, przypisuje mu ID i tworzy nowy wątek dla każdego z graczy. Wątki oczekują na dołączenie wszystkich graczy (otrzymując w międzyczasie od serwera informację o liczbie graczy w poczekalni), gdy wszyscy są połączeni każdy z wątków wysyła do odpowiedniego gracza paczkę inicjalizacyjną zawierającą informacje o wszystkich graczach oraz ID połączonego klienta.

Inicjalizacyjna struktura danych przypisywana każdemu graczowi:

```
struct player_init_data{
	int ID; // ID gracza
	int Color[3]; // składowe RGB koloru gracza
	double start_x; // współrzędna startowa x
	double start_y; // współrzędna startowa y
	double size; // rozmiar startowy gracza
};
```

Skład inicjalizacyjnej paczki danych:

```
struct initialization_data{
	// Dane o wszystkich graczach
	struct player_init_data[PLAYERS_COUNT];
	// numer ID wskazujący na danego gracza w tablicy
	int session_ID;
};
```

KLIENT:

Odebrane informacje o pozycjach początkowych wszystkich graczy zapisywane są u klienta i rozpoczyna się gra. Tworzone są instancje graczy, a gracz o otrzymanym session_ID tworzony jest jako gracz aktywny, który będzie przekazywał swoje ruchy z powrotem do serwera. Reszta utworzonych obiektów graczy jest jedynie rysowana na podstawie danych z

serwera.Po otrzymaniu paczki inicjalizacyjnej klient wysyła do serwera informację, że inicjalizacja przebiegła pomyślnie.

SERWER:

Gdy serwer otrzyma od wszystkich graczy informacje że inicjalizacja przebiegła pomyślnie rozpoczyna się pętla gry, wszyscy gracze dostają status alive.Każdy z wątków otrzymuje od swojego klienta informację o zmianie kąta ruchu i wyliczany jest wektor przesunięcia. Zmiany pozycji graczy wprowadzane są do wektorów przesunięć, dostęp do kórej kontrolowany jest przez semafor zapewniający że do tablicy zapisuje tylko jeden wątek.

Potem wątek przechodzi przez kolejny semafor i inkrementuje go a serwer może przejść dalej jedynie wtedy kiedy wartość semafora będzie równa liczbie wątków, po przejściu przez niego semafor jest ponownie zerowany. Następuje sprawdzanie kolizji przez wątek główny - jeśli wykryto kolizję (z przeszkodą lub bonusem) modyfikowany jest status gracza. Główny wątek kompletuje paczkę zmian pozycji graczy razem ze statusami i z pozycjami bonusów na mapie, która jest wysyłana przez poszczególne wątki do każdego z klientów (każdy klient otrzymuje tę samą paczkę danych).

Struktura zawierająca informację o zmianie stanu danego gracza:

```
struct player_state{
	int ID; // ID gracza
	bool alive; // czy gracz nadal żyje
	double dx; // przesunięcie w poziomie
	double dy; // przesunięcie w pionie
	double size; // aktualny rozmiar
	double bonus_modifier; // aktualny stan odnośnie bonusów
};
```

Struktura stanu graczy używana wewnętrznie przez serwer:

```
struct player_state_2{
	struct player_state state;
	double velocity; // do obliczania przesunięć
	double invincible; // do bonusu nieśmiertelności
};
```

Skład paczki ze zaktualizowanym stanem gry:

```
struct state_update{
	bool running; // czy gra nadal trwa
	// zmiany stanów wszystkich graczy
	struct player_state player_states[PLAYERS_COUNT];
	// pozycje bonusów na mapie
	struct bonus bonuses[PLAYERS_COUNT];
};
```

Skład paczki zawierającej informacje o bonusie:

```
struct bonus bonuses{
	int x; // współrzędna x bonusu
	int y; // współrzędna y bonusu
	int type; // typ bonusu
	int active; // czy bonus nadal trwa
};
```

KLIENT:

Na podstawie odebranych danych po stronie klienta następnie są rysowani wszyscy gracze i bonusy. W odpowiedzi przesyła dane wejściowe pobrane od gracza.

```
typedef enum key_state {
	UP, // Klawisz został podniesiony
	NO_CHANGE, // Brak zmiany stanu klawisza
	DOWN // Klawisz został wciśnięty
};
```

Struktura paczki wysyłanej w odpowiedzi przez klienta:

```
struct player_input{
	key_state left; // status wciśnięcia lewej strzałki
	key_state right; // status wciśnięcia prawej strzałki
};
```

Jeśli klient otrzymał paczkę ze zaktualizowanym stanem gry, w której stan alive gracza aktywnego u danego klienta jest ustawiony na false przestaje wysyłać odpowiedzi, a gdy w odebranej paczce danych running jest ustawiony na false to zaczyna nasłuchiwać na paczkę z wynikami, po otrzymaniu której są one wyświetlane i kończy się gra i połączenie klienta z serwerem zostaje zakończone.

SERWER:

W przypadku, gdy zostanie tylko jeden żywy gracz (lub zero) to do klientów rozsyłana jest paczka z running ustawione na false. Następnie nie czekając na odpowiedzi od klientów do każdego z nich wysyłany jest ustalony ranking graczy w postaci tablicy zawierającej ID każdego z nich na odpowiednim miejscu.

4. Rozgrywka

Uruchamianie

Serwer uruchamiany jest poprzez uruchomienie skryptu **compile.sh** znajdującego się w folderze *'server'*. Skrypt można uruchomić z domyślnymi ustawieniami poprzez polecenie **./compile.sh** lub modyfikując domyślne parametry jako: **./compile.sh -e N_GRACZY IP**

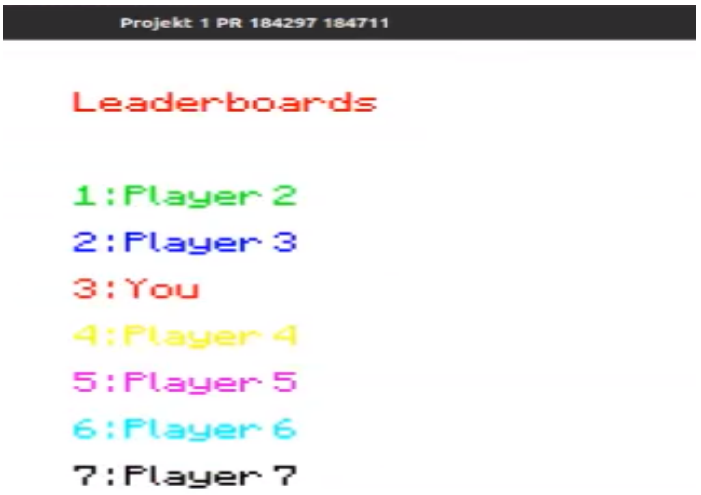
Klient uruchamiany jest poprzez wywołanie main.py z pierwszym argumentem będącym IP (domyślnie localhost):
python3 main.py IP

Przebieg Rozgrywki

Początkowo wszyscy gracze mają identyczne statystyki (rozmiar, prędkość), mogą one zostać zmodyfikowane przez trzy bonusu dostępne w grze: bonus do prędkości, pogrubienie lub nieśmiertelność. Wymienione wyżej bonusy wpływają na statystyki okresowo i generowane są w sposób losowy. Dodatkowo każdy gracz w stałych interwałach czasu pozostawia na swojej ścieżce przerwę pozwalającą na przedostanie się innego gracza.

Dwóch graczy i dwa bonusy obecne na mapie

Gracz po zebraniu bonusu zwiększającego wielkość



Tablica wyników widoczna po zakończeniu gry