Projekt Agar.io

Tomasz Sankowski (s193636) | Piotr Sulewski (s192594)

1. Przedstawienie projektu

Agar.io jest to wieloosobowa gra, której rozgrywka polega na kontrolowaniu komórki na mapie. Celem jest uzyskanie jak największej masy poprzez jedzenie małych kulek pojawiających się losowo na mapie, czy też „jedzenie” przeciwników.

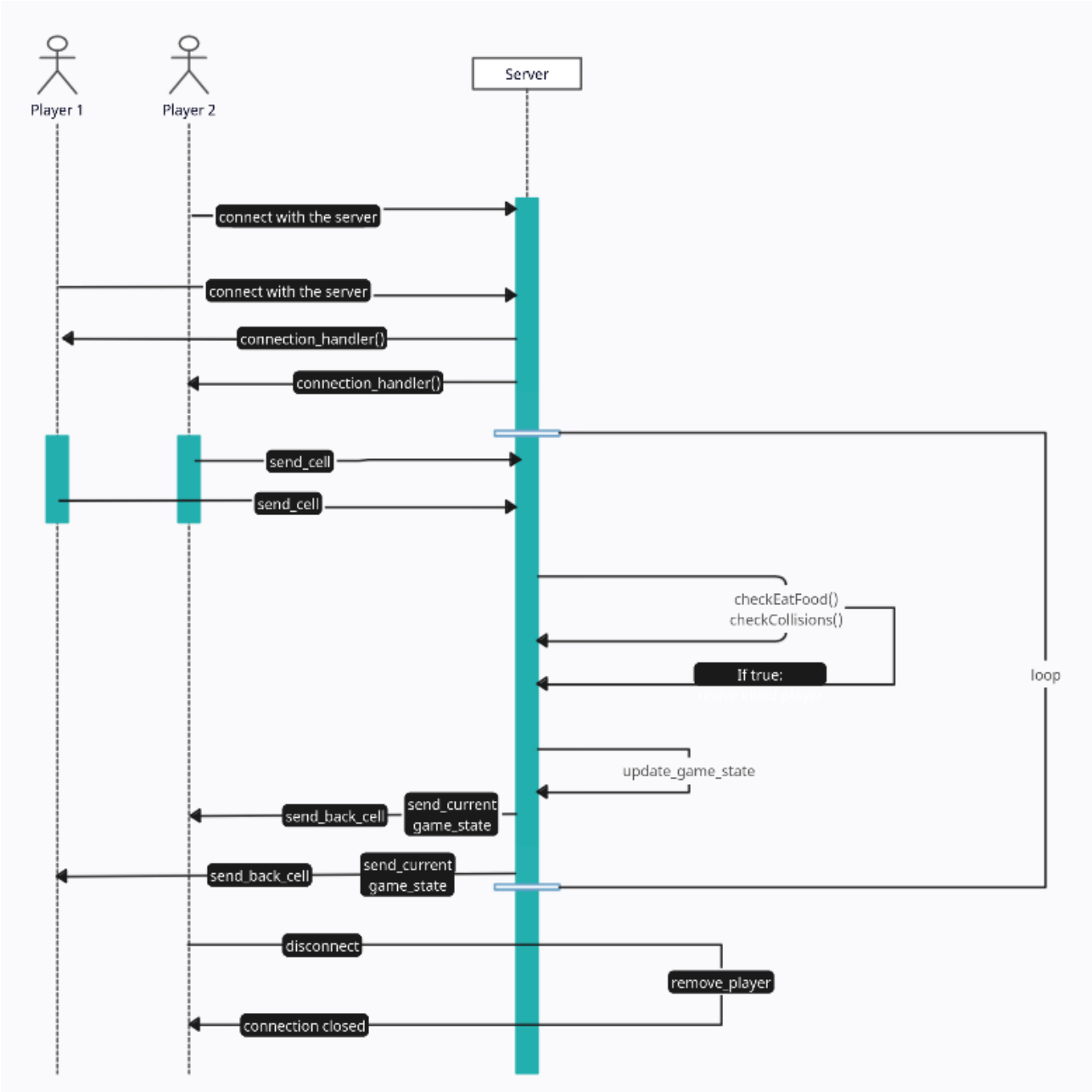
2. Diagram sekwencyjny

W celu przedstawienia sposobu komunikacji serwera z klientem stworzyliśmy diagram sekwencyjny, który to wizualnie obrazuje.

Serwer po przyjęciu żądania połączenia od klienta, przydziela mu indywidualny wątek do obsługi jego działań oraz wysyła jego pozycję, kolor oraz masę początkową. Następnie w pętli serwer na bieżąco pobiera od klienta dane dotyczące zmiany położenia jego komórki i przetwarza je. Na początku sprawdza czy pozycja gracza pokrywa się z pozycją któregoś z „jedzonek”, jeżeli tak, to zwiększa jego masę. Analogicznie postępuje w przypadku sprawdzania kolizji z innymi graczami, natomiast w tym przypadku, jeżeli nastąpiła kolizja z innym graczem i został on „zabity”, ustawiana jest flaga jego śmierci, tak żeby wątek obsługujący zabitego gracza, mógł go „odżywić”. Następnie serwer przesyła klientowi zaktualizowane dane dotyczące jego komórki oraz aktualny stan całej mapy.

Wyżej wymienione działania wykonują się w pętli do momentu aż klient nie zakończy połączenia z serwerem (np. poprzez zamknięcie aplikacji). Jeżeli taka sytuacja wystąpi, serwer usuwa gracza z listy dostępnych graczy oraz zamyka gniazdo jego połączenia.

Warto wspomnieć, że obsługa wiadomości między graczem a serwerem jest niezależna od obsługi wiadomości między innymi graczami a serwerem. Nie jest wymagane, aby wszyscy gracze wysłali serwerowi swoje aktualne położenie zanim nowy stan mapy zostanie odesłany. Właściwie to w ogóle nie jest sprawdzane, czy następuje regularna wymiana danych między serwerem a innymi graczami. Bardzo dobrze rozwiązuje to problem na przykład nagłej utraty dostępu do internetu – wtedy po prostu gracz, który utracił połączenie z perspektywy innych graczy przestanie się poruszać.


Player 1
Player 2
Server
connect with the server
connect with the server
connection_handler()
connection_handler()
send_cell
send_cell
checkEatFood()
checkCollisions()
If true:
loop
update_game_state
send_back_cell
send_current
game_state
send_back_cell
send_current
game_state
disconnect
remove_player
connection closed

3. Wyścigi oraz sekcje krytyczne

Tak jak w każdej aplikacji wielowątkowej, tak i my musieliśmy poradzić sobie z tytułowymi problemami. Do pozbycia się wcześniej wspomnianych problemów skorzystaliśmy z mechanizmu przydzielania wyłącznego dostępu do zasobu, a konkretniej „mutexów”.

W naszej aplikacji, każdy wątek obsługujący gracza w pewnych momentach potrzebuje wyłączności w dostępie do pewnego zasobu, bądź wyłączności w wykonywaniu jakiejś operacji. W celu bardziej szczegółowego opisu rozwiązanego przez nas problemu, rozbijemy nasz opis, na poszczególne funkcje.

- funkcja checkCollisions

W tym przypadku, wątek który aktualnie sprawdza kolizje, potrzebuje wyłącznego dostępu do funkcji, ponieważ chcemy uniknąć sytuacji w której to dwa wątki wykryją tą samą kolizje i przez to masa gracza większego wzrośnie podwójnie. Zastosowaliśmy tutaj proste użycie mechanizmu bezpośredniego dostępu dla wątku – mutex’u. Dzięki temu kolizje będą wykrywane tylko przez jednego gracza jednocześnie.

Początkowo odradzanie gracza zabitego odbywało się w tej funkcji i odpowiadał za to „gracz zabijający”, natomiast występował problem z nadpisaniem stanu przez wątek, który obsługiwał gracza zabitego – tuż po tym jak ustawiliśmy graczowi zjedzonemu nowe położenie i masę startową, ten mógł zaktualizować swój stan wysyłając swoją poprzednią pozycję oraz masę przez co znowu dochodziło do kolizji, a większy ponownie zwiększał swoją masę zjadając przeciwnika.

Problem rozwiązaliśmy dzięki zastosowaniu dodatkowej tablicy flag, która sprawdza czy gracz jest żywy. W przypadku kiedy dany wątek dowie się, że jego gracz został pokonany, to on odpowiada za jego odrodzenie i dzięki temu nie występuje już problem z nadpisywaniem stanu.

- funkcja checkEatFood

Metoda ta sprawdza dla danego gracza, czy koliduje on z jakimś jedzeniem. Jest to również sekcja krytyczna aplikacji, która wymaga zastosowania mechanizmu blokady. Tylko jeden gracz może w danym momencie działania programu sprawdzać, czy koliduje z którymś z jedzeń. Niezastosowanie blokady spowodowałoby, że wszyscy gracze który będą próbowali zjeść to samo jedzenie w podobnym czasie, otrzymaliby za nie punkty, a w końcu trzeba wybrać tylko jednego, któremu należy się bonusowa masa za spożycie konkretnego jedzenia.

- funkcja create_new_player

Podczas tworzenia nowego gracza, musimy zastosować blokadę. Każdy gracz otrzymuje unikalny ID, będący indeksem tablicy graczy, pod którym trzymane są jego dane takie jak położenie czy masa. ID jest ustawiane na najmniejszy, aktualnie nie zajęty indeks tablicy stanu gry. Jeśli blokada nie byłaby używana, a weszłoby do gry dwóch graczy jednocześnie, to otrzymaliby oni ten sam numer ID, co skończyłoby się nadpisaniem pozycji jednego z graczy i sterowaniem jedną kulką przez kilku graczy jednocześnie. Zastosowanie mutexa pozwala na uniknięcie takiej sytuacji - kolejny gracz jest dodawany do gry oraz jego ID jest wybierane dopiero wtedy kiedy poprzedni gracz zostanie już dodany, a jego ID zarezerwowane poprzez ustawienie flagi running[ID] w strukturze stanu gry na True.

- funkcje zmieniające stan obiektu game_state i dlaczego nie potrzebują one mechanizmów blokady

Gracze przez cały okres trwania połączenia na zmianę modyfikują strukturę stanu gry oraz otrzymują jej zaktualizowaną wersję z powrotem. Zauważyć można, że w kodzie w funkcjach operujących na game_state nie stosowane są na nich blokady (wyjątkiem jest tylko funkcja create_new_player opisana wyżej). Wynika to z tego, że każdy z graczy modyfikuje inny indeks tablicy. Wszystkie operacje na naszym indeksie z tablicy game_state są wykonywane w naszym wątku. Żaden inny gracz w żadnym momencie naszego programu nie modyfikuje naszego gracza, więc nie ma możliwości na wystąpienie wyścigów w tym aspekcie gry. Ponadto nowe pozycje graczy są wysyłane i aktualizowane tak często, że nie musimy się przejmować tym, że podczas wysyłania zaktualizowanego stanu gry z serwera do klienta położenie któregoś z graczy będzie właśnie modyfikowane – zostanie to uwzględnione w kolejnej aktualizacji stanu gry, dosłownie klika milisekund później. Zastosowanie blokad nie zmieniłoby działania programu, ale mogłoby mocno go spowolnić, bo każdy gracz musiałby czekać aż inny wpisze i otrzyma z powrotem swój stan gry.