Zadanie UDP: Rozproszony System Uśredniający

SKJ (2024)

Wstęp 2

Budowa 2

Klasa `Logger` 2

Klasa `Message` 2

Klasa `Accumulator` 3

Klasa `Client` 4

Klasa `Server` 5

Klasa `Das` 6

Logika 7

Uruchamianie 7

Serwer 7

Klient 8

Wiadomość 8

Wyjątki 8

Komunikacja 9

Ograniczenia 9

Spadek Wydajności 9

Dodatkowe Funkcjonalności 9

Logowanie 9

Potwierdzanie i retransmisja 10

Wstęp

Dokumentacja zawiera opis budowy oraz logiki aplikacji realizującej Rozproszony System Uśredniający.

Budowa

Klasa `Logger`

Służy do zarządzania systemem logowania w aplikacji.

- **Pola:**

Nazwa	Typ	Opis
`state`	`boolean`	Statyczne pole, które określa, czy logowanie jest aktywne.

- **Metody:**

Nazwa	Typ zwracany	Parametry	Opis
`log(String text)`	`void`	`String text`	Jeśli `state == false`, metoda nie wykonuje żadnych operacji. Jeśli `state == true`, metoda wyświetla tekst `text` w konsoli.

Klasa `Message`

Reprezentuje wiadomość.

- **Pola:**

Nazwa	Typ	Opis

`size`	`int`	Rozmiar wiadomości w bajtach, obliczany na podstawie zmiennej `number`.
`number`	`String`	Wartość wiadomości przechowywana jako ciąg znaków.

- **Metody:**

Nazwa	**Typ zwracany**	**Parametry**	**Opis**
`Message(String number)`	`-`	`String text`	Tworzy wiadomość na podstawie ciągu znaków i oblicza jej rozmiar.
`getMessageFromInt(int number)`	`Message`	`int number`	Tworzy wiadomość na podstawie zmiennej `number`.
`getMessageFromBytes(byte[] buffer)`	`Message`	`byte[] buffer`	Tworzy wiadomość na podstawie tablicy bajtów `buffer`.
`getBytes()`	`byte[]`	`-`	Zwraca wiadomość w formie tablicy bajtów: pierwszy bajt to rozmiar, kolejne to znaki wiadomości.
`getNumber()`	`int`	`-`	Zwraca zawartość wiadomość.

Klasa `Accumulator`

Pełni funkcję kontenera do przechowywania obiektów klasy `Message`.

- **Pola:**

Nazwa	**Typ**	**Opis**
`list`	`List<Message>`	Lista przechowuje obiekty klasy `Message`.

- **Metody:**

Nazwa	Typ zwracany	Parametry	Opis
`Accumulator()`	`-`	`-`	Tworzy pustą listę obiektów klasy `Message`.
`put(Message message)`	`void`	`Message message`	Dodaje obiekt klasy `Message` do listy `list`.
`getAverage()`	`int`	`-`	Oblicza średnią arytmetyczną wartości z wiadomości w liście `list`, biorąc pod uwagę liczby większe od 0.

Klasa `Client`

Służy do wysyłania wiadomości w sieci za pomocą protokołu UDP.

- **Pola:**

Nazwa	Typ	Opis
`socket`	`DatagramSocket`	Gniazdo sieciowe używane do wysyłania pakietów UDP.

- **Metody:**

Nazwa	Typ zwracany	Parametry	Opis
`Client(DatagramSocket socket)`	`-`	`DatagramSocket socket`	Inicjalizuje obiekt klienta z podanym gniazdem sieciowym.
`sendMessage(int port, int number)`	`void`	`int port, int number`	Wysyła wiadomość z podaną wartością `number` na określony

			port `port` do adresu localhost.
`waitForAcknowledgement (int port, int number)`	`Void`	`int port, int number`	Metoda czeka na otrzymanie potwierdzenia od serwera, jeżeli go nie otrzyma próbuje wysłać komunikat jeszcze raz. Metoda próbuje podejmuje próbę retransmisji maksymalnie 3 razy.
`close()`	`void`	`-`	Zamyka gniazdo `socket`.

Klasa `Server`

Działa jako serwer UDP, który obsługuje komunikację sieciową.

- **Pola:**

Nazwa	**Typ**	**Opis**
`port`	`int`	Port, na którym serwer nasłuchuje wiadomości.
`socket`	`DatagramSocket`	Gniazdo sieciowe używane do odbierania i wysyłania pakietów UDP.
`accumulator`	`Accumulator`	Obiekt do przechowywania i operacji na wiadomościach `Message`.

- **Metody:**

Nazwa	Typ zwracany	Parametry	Opis
`Server(int port, int number)`	`-`	`int port, int number`	Tworzy serwer nasłuchujący na podanym porcie, inicjalizuje akumulator i dodaje pierwszą wiadomość.
`getMessage()`	`boolean`	`-`	Odbiera wiadomość UDP, dodaje ją do akumulatora, wykonuje oraz wykonuje broadcast, zamknięcie, obliczenie średniej.
`broadcastMessage(int number)`	`void`	`int number`	Wysyła wiadomość rozgłoszeniową z liczbą `number` na całą sieć lokalną.
`sendAcknowledgement (InetAddress address, int port)`	`void`	`InetAddress address, int port`	Wysyła wiadomość potwierdzającą otrzymanie wiadomości od klienta z wartością `-2`.
`close()`	`void`	`-`	Zamyka serwer.

Klasa DAS

Odpowiada za działanie serwera oraz klienta.

- **Pola:**

Nazwa	Typ	Opis
`args`	`String[]`	Argumenty wejściowe przekazane do programu z wiersza poleceń.

- **Metody:**

Nazwa	Typ zwracany	Parametry	Opis

`main(String[] args)`	`Void`	`String[] args`	Weryfikuje argumenty, a następnie uruchamia serwer lub klienta w zależności od sytuacji.
`checkRequirements(String[] arr)`	`boolean`	`String[] arr`	Weryfikuje poprawność argumentów wejściowych - sprawdza liczbę argumentów, zakres portu i liczbę.

Logika

Uruchamianie

Aby uruchomić program wpierw trzeba skompilować klasę `DAS.java` (razem z nią automatycznie wszystkie inne klasy powinny się skompilować).

Program uruchamiany jest z argumentami wejściowymi:

- `PORT` – numer portu, na którym serwer będzie nasłuchiwać lub na który klient wyśle wiadomość
- `NUMBER` – liczba, która jest przekazywana jako wiadomość

Jeśli port jest wolny, uruchamiany jest serwer, jeśli port jest zajęty, uruchamiany jest klient, który wysyła wiadomości na wskazany port.

Serwer

Serwer nasłuchuje wiadomości do klientów na podanym porcie `PORT`. Wiadomość odczytywana jest jako liczba całkowita przy pomocy metody `getMessageFromBytes()`. W zależności od wartości liczby podejmowana jest odpowiednia akcja:

- Jeśli liczba jest równa `-1`: serwer wyświetla wiadomość na konsoli, przesyła broadcastowo wiadomość `-1`, po czym zamyka się

- Jeśli liczba jest równa `0`: serwer oblicza średnią ze wszystkich dotychczas odebranych liczb większych od zera, wyświetla wynik na konsoli oraz przesyła go broadcastowo
- Jeśli liczba jest różna od `-1` i `0`: serwer dodaje liczbę do akumulatora oraz wyświetla ją na konsoli

Klasa `Accumulator` przechowuje i przetwarza dane liczbowe, które serwer odbiera w postaci obiektów `Message`.

- `put(Message message)` umożliwia dodawanie nowej liczby do listy przechowywanej w `Accumulator`
- `getAverage()` oblicza średnią ze wszystkich liczb w liście większych od zera

Klient

Klient tworzy wiadomość z podaną liczbą `NUMBER`, tworzy obiekt klasy `Message`, a następnie tworzy bufor danych przy pomocy metody `getBytes()`. Następnie wysyła bufor danych na wskazany port do serwera.

Wiadomość

Pierwszy bajt wiadomości zawiera rozmiar tablicy bajtowej przechowującej wartość przesyłanej liczby, kolejne bajty zawierają wartość liczby jako ciąg znaków ASCII.

Maksymalna wartość typu `int` w Javie to `2147483647`. Konwertując te liczbę na typ `String`, otrzymuje się `„2147483647”`. Tablice bitów przechowująca taki ciąg znaków będzie mieć długość 10. Oznacza to, że bufor przesyłany za pomocą protokołu UDP będzie mógł mieć co najwyżej wielkość 11 (1 bajt na długość liczby, oraz maksymalnie 10 bajtów na ciąg znaków reprezentujących daną liczbę).

Wyjątki

`Serwer` zamyka swoje gniazdo w przypadku wyjątku.

`Klient` zamyka swoje gniazdo w przypadku wyjątku.

DAS wyłącza się w przypadku braku lub niepoprawności argumentów, a jeśli serwer nie może się uruchomić przez zajęty port, automatycznie uruchamia klienta.

Komunikacja

Serwer oraz klient mogą generować obiekty klasy `Message` w celu przesyłania wiadomości.

- `getMessageFromInt(int number)` tworzy wiadomość na podstawie liczby całkowitej (np. do broadcastowania średniej lub sygnału zakończenia)
- `getMessageFromBytes(byte[] buffer)` tworzy wiadomość na podstawie odebranego bufora

Przed wysłaniem wiadomości, serwer oraz klient korzystają z metody `getBytes`, aby zamienić wiadomość na tablicę bajtów.

Po odebraniu pakietu UDP serwer wywołuje metodę `Message.getMessageFromBytes`, aby przekonwertować bajty z bufora na obiekt `Message`. Po odebraniu wiadomości, serwer przekazuje ją do obiektu `Accumulator`, co pozwala zapamiętać liczbę.

Kiedy serwer odbiera liczbę `0`, wywoływana jest metoda `getAverage`. Wynik średniej przesyłany jest w odpowiedzi do klientów jako broadcast.

Ograniczenia

Spadek wydajności

Zauważony znaczy spadek wydajności aplikacji podczas jednoczesnego uruchomienia 2000 klientów (z przerwami o długości 0 – 2 sekundy między uruchamianiem klientów problem znika).

Dodatkowe funkcjonalności

Logowanie

Klasa `Logger` jest wykorzystywana do rejestrowania operacji: tworzenia obiektów, przetwarzania danych, komunikacji oraz diagnozy błędów.

Logowanie można włączać i wyłączać globalnie poprzez zmianę wartości pola `state` (domyślnie logowanie jest wyłączone: `Logger.state = false`). Aby umożliwić rejestrację zdarzeń, należy ustawić `Logger.state = true`.

Potwierdzanie i retransmisja

Została podjęta próba implementacji mechaniki potwierdzeń i retransmisji.

Klasa `Server` została wyposażona w możliwość wysłania potwierdzeń otrzymanych wiadomości do nadawców. Wysyła wtedy wiadomość `-2` do nadawcy. Jeżeli `Server` przesłał broadcastowo wiadomość to również on sam ją odbierze. W takim przypadku, za każdym razem sprawdzane jest czy treść wiadomości nie jest równa `-2`. Jeżeli jest równa `-2`, wtedy `Server` kończy odbieranie wiadomości i zwraca wartość `true`.

Klasa `Client` po wysłaniu wiadomości będzie czekać na otrzymanie wiadomości od `Servera` zawierającej potwierdzenie otrzymania wiadomości. `Client` może podjąć maksymalnie 3 próby retransmisji wiadomości. Za każdym razem wpierw czeka 5 sekund na odpowiedź, a jeżeli takowa nie nadejdzie rozpoczyna retransmisję. Po 3 próbie `Client` zamyka swoje gniazdo.