Zadanie 2: transakcyjny protokół SKJ (2015)

1 Wstęp

Zadanie polega na zaprojektowaniu niezawodnego protokołu transakcyjnego bazującego na protokole UDP. Protokół ten ma być realizowany przez klasy implementujące interfejs *ITransactionalSocket* (gniazda komunikacyjne), oraz *IWaitingSocket* (gniazda serwerowe czekające na zgłoszenia od klientów).

Należy zwrócić szczególną uwagę na to, aby prawidłowo dostarczone zostały nie tylko pakiety z danymi, ale także pakiety informacyjne: np. te o prośbie ponownego dostarczenia pakietów, czy o rozpoczęciu/zakończeniu transakcji. Protokół ma być zoptymalizowany pod kątem ilości danych przesyłanych przez sieć.

2 Interfejsy

Interfejs *ITransactionalSocket* udostępnia operacje do komunikacji pomiedzy dwoma transkcyjnymi gniazdami:

```
public interface ITransactionalSocket extends Closeable {
   public byte[] read() throws IOException; // blocking
   public void write(byte[] buff);

   public void openTransaction() throws IOException;
   public void closeTransaction() throws IOException;
   public void abortTransaction() throws IOException;

   public void close() throws IOException;
```

```
    public void setTimeout(int readTimeout);
    public int getTimeout();

    public void setSegmentSize(int size);
    public int getSegmentSize();

    public int getLocalPort();
    public InetAddress getLocalAddress();

    public int getRemotePort();
    public InetAddress getRemoteAddress();
}
```

1. Metoda *byte[] read()* służy do przeczytania wszystkich danych wysłanych z kolejnej jednej transakcji. Ta metoda blokuje wątek, który ją wywołał, tak długo, aż pojawią się dane z całej transakcji, bądź minie wyznaczony *timeout* dla czytania. Jeżeli nastąpi problem z połączeniem (w tym zamknięcie gniazda z drugiej strony), to zostanie podniesiony wyjątek *IOException*.

2. Metoda *void write(byte[] buff)* służy do wysłania danych w obrębie otwartej transakcji. Metoda ta może być wołana tylko jeżeli transakcja jest otwarta. Wysłanie danych w obrębie transakcji powinno zawsze kończyć się powodzeniem — jeżeli istnieje problem komunikacji z drugą stroną, zostanie to zasygnalizowane dopiero podczas zamykania transakcji. W szczególności, poprawną metodą implementacji *void write(byte[] buff)* jest buforowanie wszystkich zapisywanych danych i wysyłanie ich przez sieć dopiero podczas zamykania transakcji — nie jest to jednak najlepsze rozwiązaniem z punku widzenia ruchu sieciowego i dlatego nie będzie maksymalnie punktowane.

3. Metoda *void openTransaction()* otwiera transakcję. Metoda ta może być wołana tylko, jeżeli transakcja nie jest aktualnie otwarta. Jeżeli nie powiodło się otwarcie transakcji (o ile protokół przewiduje taką możliwość), ma być rzucony wyjątek *IOException*.

4. Metoda *void closeTransaction()* zamyka transakcję. Metoda ta może być wołana tylko, jeżeli transakcja jest aktualnie otwarta. Jeżeli nie

powiodło się zamknięcie transakcji ma być rzucony wyjątek *IOException*

5. Metoda *void abortTransaction()* przerywa i wycofuje bieżącą transakcję. Z punktu widzenia drugiej użytkownika drugiej strony (tj. oczekującej na dane), przerwana transakcja ma mieć dokładnie taki sam efekt jakby nigdy się nie rozpoczęła. Jeżeli nie powiodło się przerwanie transakcji ma być rzucony wyjątek *IOException*

6. Metoda *void setTimeout(int readTimeout)* i *int getTimeout()* odpowiednio ustawiają i odczytują *timeout* dla czytania danych.

7. Metoda *int getLocalPort()* i *InetAddress getLocalAddress()* zwracają odpowiednio port UDP i adres IP lokalnego gniazda służącego do komunikacji.

8. Metoda *int getRemotePort()* i *InetAddress getRemoteAddress()* zwracają odpowiednio port UDP i adres IP gniazda strony, z którą się komunikujemy.

Interfejs *IWaitingSocket* udostępnia jedyną operację *ITransactionalSocket listen()* służącą do czekania na połączenia od klientów i zwracającą w wyniku transakcyjne gniazdo do komunikacji:

```
public interface IWaitingSocket extends Closeable {
   // blocking
   public ITransactionalSocket listen() throws IOException;

   public int getPort();
   public InetAddress getAddress();

   public void close() throws IOException;
}
```

oraz dwie metody: *int getPort()* i *InetAddress getAddress()* zwracające odpowiednio port UDP i adres IP gniazda nasłuchującego na przychodzące połączenia.

Ponadto należy zaimplementować statyczne metody służące do tworzenia instancji zaimplementowanych klas:

- w interfejsie *ITransactionalSocket*:

```
public static ITransactionalSocket
    createTransactionalSocket(remotePort, remoteAddress,
    localPort, localAddress) throws IOException;
```

tworzącą gniazdo na porcie *localPort* i adresie *localAddress*, które ma komunikować się z gniazdem o porcie *port* i adresie *address*

- w interfejsie *IWaitingSocket*:

```
public static IWaitingSocket createWaitingSocket(port,
    address) throws IOException;
```

tworzącą gniazdo nasłuchowe na porcie *port* i adresie *address*.

3 Schemat komunikacji

Z punktu widzenie użytkownika protokołu, wyróżniamy dwie strony: inicjującą połączenie i czekającą na połączenia.

Inicjowanie połączenia odbywać się ma poprzez stworzenie instancji klasy implementującej interfejs *ITransactionalSocket*. Podczas tworzenia instancji ma powstać gniazdo UDP na zadanym adresie i porcie, oraz przeprowadzony ma zostać protokół przywitania ze stroną z którą chcemy się skomunikować (identyfikowaną także przez port UDP i adres IP). Po udanej inicjacji połączenia, wysyłanie danych polega na wielokrotnym:

- otwarciu transakcji
- pisaniu do gniazda
- zamknięciu transakcji

natomiast czytanie polega na odebraniu wszystkich danych wysłanych w obrębie jednej transakcji.

Czekanie na połączenia odbywa się poprzez utworzenie instancji klasy implementującej *IWaitingSocket* na zadanym adresie i porcie, oraz wołaniu metody *ITransactionalSocket listen()*. Metoda ta ma blokować się tak długo, aż pojawi się klient inicjujący połączenie. Wynikiem zwracanym przez tę metodę ma być instancja *ITransactionalSocket*, z którą dany klient będzie się dalej porozumiewał.

Typowy schemat komunikacyjny po stronie inicjującej połączenia wyglądać ma następująco:

```
ITransactionalSocket socket =
   new TransactionalSocket(serverAddress, serverPort);

// read some data
buff = socket.read()

// first transaction
socket.openTransaction();
socket.write(buff1);
// do something
socket.write(buff2);
// do something else
socket.closeTransaction();

// second transaction
socket.openTransaction();
socket.write(buff3);
socket.abortTransaction();

socket.close();
```

natomiast po stronie oczekującej na połączenia:

```
IWaitingSocket server = new WaitingSocket(address, port);

while(true) {
   ITransactionalSocket client = server.listen();
   new Thread(() -> {
      // perform some communication with client
      // ...

      client.close();
   }).start();
}
```

Protokół ma zapewnienia niezawodności przesyłanych danych opierając się na następującym schemacie:

- strona chcąca wysłać dane inicjuje rozpoczęcie nowej transakcji
- strona, która wysłała wszystkie dane w obrębie jednej transakcji, informuje o zakończeniu transakcji
- strona, która odebrała wiadomość o zakończeniu transakcji, wysyła informację o tych pakietach, które nie zostały jej dostarczone w obrębie transakcji
- strona wysyłająca dane, ponownie wysyła te fragmenty
- jeżeli zachodzi taka potrzeba, ostatnie dwa powyższe punkty są wykonywane wiele razy

Implementacja nie musi zapewniać poprawności współbieżnego dostępu do instancji definiowanych klas.

4 Zadanie

1. Za poprawny i pełny projekt student może otrzymać do 6 punktów.
2. Wszystkie aplikacje piszemy w języku Java zgodnie ze standardem Java 8 (JDK 1.8), a do komunikacji pomiędzy nimi wykorzystujemy standardowe gniazda UDP omawiane na zajęciach.
3. Projekty powinny zostać zapisane do odpowiednich katalogów w systemie EDUX do 13.12.2016 (termin może zostać zmieniony przez prowadzącego grupę).
4. Spakowany plik projektu powinien obejmować:
   - opis protokołu komunikacyjnego
   - opis klas wraz z przykładami użycia
   - pliki źródłowe (dla JDK 1.8)
   - plik *Readme.txt* z opisem i uwagami autora
5. Prowadzący oceniać będą w pierwszym rzędzie organizację komunikacji po sieci — tj. poprawność i efektywność protokołu, ale na ocenę wpływać będzie także zgodność wytworzonego oprogramowania z zasadami inżynierii oprogramowania i jakość implementacji.

6. JEŚLI NIE WYSZCZEGÓLNIONO INACZEJ, WSZYSTKIE NIEJASNOŚCI NALEŻY PRZEDYSKUTOWAĆ Z PROWADZĄCYM ZAJĘCIA POD GROŹBĄ NIEZALICZENIA PROGRAMU W PRZYPADKU ICH NIEWŁAŚCIWEJ INTERPRETACJI.