# Technologie sieciowe
# Sprawozdanie do listy 4

Szymon Wojtaszek 236592

June 2018

## TCP/IP

### Wymagania

1. Przykładowe programy: **Z2Forwarder.java**, **Z2Packet.java**, **Z2Receiver.java**, **Z2Sender.java**. Plik **plik.txt** zawiera przykładowe dane.

2. Program Z2Sender wysyła w osobnych datagramach po jednym znaku wczytanym z wejścia do portu o numerze podanym jako drugi parametr wywołania programu. Jednocześnie drukuje na wyjściu informacje o pakietach otrzymanych w porcie podanym jako pierwszy parametr wywołania. Program Z2Receiver drukuje informacje o każdym pakiecie, który otrzymał w porcie o numrze podanym jako pierwszy parametr wywołania programu i odsyła go do portu podanego jako drugi paramer wywołania programu. Klasa Z2Packet, umożliwia wygodne wstawianie i odczytywanie czterobajtowych liczb całkowitych do tablicy bajtów przesyłanych w datagramie - metody: public void setIntAt(int value, int idx) oraz public int getIntAt(int idx). Wykorzystane jest to do wstawiania i odczytywania numerów sekwencyjnych pakietów.

   Po skompilowaniu, można je uruchomić w terminalu w następujący sposób: java Z2Receiver 6001 6000 java Z2Sender 6000 6001 ¡ plik.txt W tej konfiguracji Z2Receiver powinien otrzymywać wszystkie pakiety w odpowiedniej kolejności i bez strat, i odsyłane prze niego potwierdzenia, dochodzą do Z2Sender w taki sam (niezadwodny) sposób. Program Z2Sender po zakończeniu transmisji musi być ręcznie przerwany (CTRL+C), bo wątek odbierający oczekuje na kolejne pakiety. Program Z2Receiver można zatrzymać przy użyciu poleceń ps (aby odczytać nr procesu) i kill (aby wysłać do procesu sygnał zakończenia). W Internecie każdy pakiet przesyłany jest niezależnie i w miarę dostępnych możliwości. W związku z tym pakiety wysyłane przez nadawcę mogą być tracone, przybywać z różnymi opóźnieniami, w zmienionej kolejności, a nawet mogą być duplikowane. Program Z1Forwarder symuluje tego typu połączenie. Aby go użyć można wykonać (po zabiciu innych programów korzystających z portów 6000, 6001, 6002, 6003) następujące polecenia: java Z2Receiver 6002 6003 java Z2Forwarder 6001 6002 java Z2Forwarder 6003 6000 java Z2Sender 6000 6001 ¡ plik.txt W tej konfiguracji pierwszy Z2Forwarder przekazuje pakiety od Z2Sender do Z2Receiver, a drugi - w przeciwnym kierunku. (Może wystąpić pewne opóźnienie, zanim zaczną się pojawiać wyniki drukowane przez Z2Receiver i Z2Sender.)

3. Zadanie polega na takim wykorzystaniu potwierdzeń i numerów sekwencyjnych przez nadawcę i odbiorcę, aby odbiorca wydrukował wszystkie pakiety w kolejności ich numerów sekwencyjnych nawet jeśli połączenie w obie strony odbywa się przez Z2Frowarder. Nadawca może przypuszczać, że pakiet nie dotarł do celu jeśli przez długi czas nie otrzyma potwierdzenia od odbiorcy Może wtedy ten pakiet ponownie wysłać (retransmitować). Odbiorca może wykorzystywać numery sekwencyjne pakietów aby się zorientować czy ma prawo drukować dany pakiet, czy też musi czekać na brakujące wcześniejsza pakiety albo dany pakiet już był drukowany (np. jest duplikatem).

## Realizacja

Po użyciu klasy **Z2Forwarder** w komunikacji datagramami pojawiają się trzy problemu z którymi trzeba się zmierzyć w zadaniu:

- Kolejność wypisywania otrzymanych elementów;
- Eliminacja powtórek;
- Zapewnienie retransmisji w przypadku utraty datagramu.

Z dwoma pierwszymi trudnościami można się uporać dzięki wykorzystaniu kolejki priorytetowej. Po otrzymaniu każdego datagramu pakujemy go do kolejki, a później bierzemy elementy z wierzchu. Po określeniu odpowiedniej relacji mniejszości na elementach datagramu (porównujemy indeksy nadawane przy tworzeniu) właściwości kolejki priorytetowej zapewniają nam kolejność ich wypisywania. Dzięki odpowiedniemu zaimplementowaniu funkcji **Z2Packet getCorrectPacket()** klasy **DiagramQueue** datagramy, które zostały już przetworzone są po prostu wyrzucane z kolejki:

```
1     public Z2Packet getCorrectPacket(){
2
3         while(queue.peek() != null && queue.peek().getIntAt(0) <
          ↪  NumberOfDatagramProcessed)
4             queue.poll();
5
6         if(queue.peek() != null && queue.peek().getIntAt(0) ==
          ↪  NumberOfDatagramProcessed){
7             NumberOfDatagramProcessed++;
8             return queue.poll();
9         }else
10            return null;
11    }
```

Dzięki tej klasie prosta zmiana w **ReceiverThread** w klasach **Z2Receiver** i **Z2Sender** zapewnia nam dwie pierwsze właściwości.

```
1        class Z2Receiver.ReceiverThread extends Thread
2    {
3        public void run(){
4            try{
5                while(true)
6                {
7                    byte[] data=new byte[datagramSize];
8                    DatagramPacket packet=
9                            new DatagramPacket(data, datagramSize);
10                   socket.receive(packet);
11                   Z2Packet p=new Z2Packet(packet.getData());
12                   System.err.println("Received:"+p.getIntAt(0)
13                           +": "+(char) p.data[4]);
14                   datagramQueue.addPacket(p);
15                   while ((p = datagramQueue.getCorrectPacket()) != null){
16                       System.out.println("R:"+p.getIntAt(0)
17                               +": "+(char) p.data[4]);
18                   }
19
20                   // WYSLANIE POTWIERDZENIA
21                   packet.setPort(destinationPort);
22                   socket.send(packet);
23               }
24           }
25           catch(Exception e)....
26       }
27   }
```

Ostatnia właściwość dotyczy głównie klasy Z2Sender. Została ona zapewniona przez przechowywanie ArrayListy już wysłanych obiektów oraz testowanie czy została otrzymana taka sama ilość datagramów jak wysłana. W przeciwnym wypadku po odpowiedniej szczelinie czasowej zostaje wysłany datagram ponownie. Program czeka kilka szczelin czasowych, aby się upewnić, czy za chwile nie dostanie potwierdzenia.

```
1       class SenderThread extends Thread {
2          public void run() {
3              try {
4                  final int MAX_NUMBER_OF_ATTEMPTS = 40;
5                  int actual_number_of_attemps = 0;
6                  Z2Packet unfindedPacket = null;
7
8                  while (true) {
9                      if (!packetsToSendList.isEmpty()) {
10
11                         Z2Packet p = packetsToSendList.poll();
12                         DatagramPacket packet =
13                                 new DatagramPacket(p.data, p.data.length,
14                                         localHost, destinationPort);
15                         socket.send(packet);
16                         System.err.println("Sending: " + p);
17                     } else if(datagramQueue.getNumberOfDatagramProcessed() <
                       ↪  packetsPreparedList.size()){
18                         if(unfindedPacket == null ||
                           ↪  !unfindedPacket.equals(packetsPreparedList.get(datagramQueue.getNumberOf
19                             unfindedPacket =
                               ↪  packetsPreparedList.get(datagramQueue.getNumberOfDatagramProcessed()
20                             actual_number_of_attemps = 0;
21                         }
22                         if ((actual_number_of_attemps \% MAX_NUMBER_OF_ATTEMPTS) ==
                           ↪  0) {
23                             packetsToSendList.add(unfindedPacket);
24                             System.err.println("Repeated prepare: " + unfindedPacket);
25                         }
26                         actual_number_of_attemps++;
27                     }
28                     sleep(sleepTime);
29
30                 }
31             } catch (Exception e) ...
32         }
33     }
```

## Wnioski

Ćwiczenie pozwoliło zaznajomić się z mechanizmami służącymi do zapewnienia niezawodności w sieci opartej na protokół UDP (and User Datagram Protokół) oraz odkryć, że można tworzyć aplikacje sieciowe oparte na innych schematach, niż połączeniowe TCP.