

# Lab 3 : Netwerken en Serialisatie

16 maart 2018

Cedric De Boom

(e-mail: {voornaam.familienaam}@ugent.be)

## 1 Inleiding

Sinds de opkomst van het Internet is het gebruik van het netwerk bijna niet meer weg te denken bij softwareapplicaties (al is het maar om updates te downloaden). In deze labsessie introduceren we de basis bouwstenen om applicaties te laten communiceren over een computernetwerk. We breiden het eventsysteem van vorige labsessie uit met een netwerkcomponent, zodat verschillende event brokers met elkaar kunnen communiceren. Hiervoor bouwen we verder op de event-gebaseerde noodoproep-applicatie uit labsessie 1 en de multithreaded event broker uit labsessie 2.

## 2 Sockets

Communicatie tussen twee verschillende processen (Eng. “inter-process communication”, of afgekort IPC) vergt dat deze processen ondubbelzinnig geïdentificeerd kunnen worden. Om een proces uniek te identificeren, hebben we natuurlijk een referentie nodig naar de computer (de host) waarop dit proces loopt. Dit gebeurt via het **IP-adres** van die host. Verder kunnen er vanzelfsprekend verschillende processen op dezelfde host lopen. Daarom beschikt elke host over een reeks **poorten** (elk geïdentificeerd via een geheel getal van 16 bits), en bindt een proces zich aan één of meerdere poorten. Wanneer een communicatie opgezet wordt tussen twee processen, volstaat het dus om per proces het IP-adres en een poortnummer op te geven.

De Java-API beschikt over de klasse `InetAddress` om IP-adressen voor te stellen en er enkele eenvoudige manipulaties op uit te voeren. Zowel IPv4- als IPv6-adressen worden hierbij ondersteund. Belangrijke methoden in deze klasse zijn:

- lokaal IP-adres opvragen (`InetAddress.getLocalHost()`)
- opzoeken van een IP-adres gegeven een computernaam (DNS lookup)  
(`InetAddress.getByName(String hostname)`)
- opzoeken van een computernaam gegeven het IP-adres (reverse DNS lookup)  
(`InetAddress.getByName(String IPaddress)`)

Achterhaal via één van deze methoden het IP-adres van ‘localhost’; dit is altijd 127.0.0.1, en wordt ook wel het *loopback* IP-adres genoemd.

In de praktijk stuurt een proces gegevens naar een poort via een socket. Deze socket handelt de communicatie over het netwerk af, en implementeert het transportprotocol. De meest gebruikte transportprotocollen zijn het UDP- en het TCP-protocol.

### 2.1 UDP-communicatie

Het User Datagram Protocol is bedoeld om individuele pakketten (“datagrammen”) af te leveren tussen twee processen. Het is een onbetrouwbaar protocol, omdat niet expliciet nagegaan wordt of een datagram wel degelijk afgeleverd werd. Daarnaast kunnen datagrammen ook aan de ontvanger afgeleverd worden in een andere volgorde dan waarin ze verstuurd werden. Het voordeel van dit protocol is vooral zijn eenvoud, waardoor kleine vertragingstijden kunnen gehaald worden. Dit protocol wordt dan ook vooral gebruikt in situaties

waarin het afleveren van alle pakketten niet essentieel is, en waarin kleine netwerkvertragingen nagestreefd worden. Voorbeelden zijn het afleveren van streaming video (live video), Skype en VoIP.

Belangrijke Java-classes om UDP-communicatie te realiseren zijn:

- **DatagramSocket**: een object van deze klasse handelt het UDP-protocol af, en laat toe om datagrammen te versturen (**send()**) en te ontvangen (**receive()**)
- **DatagramPacket**: een object van deze klasse stelt een datagram voor. Merk op dat bij het versturen van een bericht het datagram uiteraard met de te versturen boodschap opgevuld wordt. Bij het ontvangen van een datagram fungeert het datagram aan ontvangtzijde als buffer, waarin de gegevens uiteindelijk terechtkomen.

Typisch onderneem je volgende stappen bij het versturen van een datagram:

1. Identificeer de bestemming van het datagram (via de klasse `InetAddress`),
2. Creëer een `DatagramSocket`,
3. Creëer een datagram dat de te versturen boodschap bevat,
4. Verstuur het datagram over de socket,
5. Sluit de socket af als deze niet langer nodig is.

En analoog voor het ontvangen van een datagram:

1. Creëer een `DatagramSocket`,
2. Creëer een datagram dat als buffer voor de te ontvangen boodschap kan fungeren (en met andere woorden voldoende groot is),
3. Ontvang een datagram van de socket,
4. Sluit de socket af als deze niet langer nodig is.

Listing 1 toont voorbeeldcode van een UDP-client. Na het aanmaken van `DatagramSocket` op poort 1300, wordt een datagram aangemaakt. Naast de inhoud van het bericht zelf, bevat het datagram ook het IP-adres van de bestemming (hier dezelfde computer, namelijk "localhost") en de bestemmingspoort (namelijk poort 1234). De bijhorende servercode wordt in Listing 2 weergegeven. Het gaat om een server die de naam van de client ontvangt, en een begroeting terugstuurt. De client blijft namen doorsturen tot de tekst "stop" ingegeven wordt. De server loopt in een oneindige lus, en maakt als eerste stap een buffer aan waarin een boodschap kan terechtkomen. De oproep `receive()` blokkeert tot een datagram ontvangen wordt. Vervolgens wordt het bericht aangepast en terug in een datagram verpakt, dat uiteindelijk naar de originele zender verstuurd wordt. Bemerk dat de server verschillende datagrammen afzonderlijk behandelt, die in principe van verschillende bronnen afkomstig zouden kunnen zijn.

Deze code is beschikbaar voor download. Start eerst het Server-programma op, en nadien één of meerdere Clients. Verifieer of de code correct werkt, ook als er meerdere client-processen tegelijk verzoeken naar de server sturen. Vergeet zeker het Server-programma niet af te sluiten na afloop.

## 2.2 TCP-communicatie

In tegenstelling tot UDP is het TCP-protocol (Transmission Control Protocol) **stroom-georiënteerd**: eens een verbinding gemaakt is tussen twee processen via een poort, zal de TCP-socket het mogelijk maken om gegevens tussen die processen uit te wisselen via een openstaande connectie. Elke socket is daarom voorzien van twee stroom-objecten, namelijk een `InputStream` en een `OutputStream` waarlangs respectievelijk gegevens ontvangen en verstuurd worden. Daarom moet de server (in tegenstelling tot een UDP-server) per bron een socket openhouden, zolang de communicatie loopt! Aan de clientzijde verloopt de communicatie vrij gelijkaardig met UDP. Het enige verschil is dat niet via datagrammen maar via stromen gecommuniceerd wordt. Daarom zal na het aanmaken van de TCP-socket, via

```
Socket socket=new Socket(host,port);
```

typisch de `InputStream` en `OutputStream` van de socket opgevraagd worden, bijvoorbeeld via de constructie:

```
InputStream inputStream=socket.getInputStream();
```

```
OutputStream outputStream=socket.getOutputStream();
```

Versturen en ontvangen van berichten gebeurt dan via de klassieke bewerkingen op stromen.

---

```

package greeter;
public class Client {
    public static void main(String[] args) {
        InetAddress host = null;
        int serverPort = 1234;
        DatagramSocket socket = null;
        BufferedReader consoleInput = new BufferedReader(new InputStreamReader(
            System.in));
        String name = "";
        try {
            host = InetAddress.getLocalHost();
            socket = new DatagramSocket(1300);
            do {
                // read in a name
                System.out.println("Enter a name : ");
                name = consoleInput.readLine();

                if (!(name.equals("stop"))) {
                    // send the name
                    byte[] messageBytes = name.getBytes();
                    DatagramPacket request = new DatagramPacket(messageBytes,
                        messageBytes.length, host, serverPort);
                    socket.send(request);

                    // receive reply
                    byte[] buffer = new byte[50];
                    DatagramPacket reply = new DatagramPacket(buffer,
                        buffer.length);
                    socket.receive(reply);

                    // print the greeting
                    System.out.println("Reply from server = "
                        + (new String(reply.getData())));
                }
            } while (!(name.equals("stop")));
        } catch (UnknownHostException e) {
            System.err.println(e);
        } catch (SocketException e) {
            System.err.println(e);
        } catch (IOException e) {
            System.err.println(e);
        } finally {
            if (socket != null)
                socket.close();
        }
    }
}

```

---

Listing 1: Communicatie via UDP: Client-programma

---

```

package greeter;
import java.net.*;
import java.io.*;

public class Server {
    public static void main(String[] args) {
        DatagramSocket socket=null;
        int serverPort=1234;
        int bufferSize=100;

        try {
            socket=new DatagramSocket(serverPort);
            while(true) {
                byte[] buffer=new byte[bufferSize];
                DatagramPacket request=new DatagramPacket(buffer,buffer.length);
                socket.receive(request);
                String rMess="Hello, "+(new String(request.getData()));
                DatagramPacket reply =new DatagramPacket(rMess.getBytes(),rMess.length(),
                    request.getAddress(),request.getPort());
                socket.send(reply);
            }
        } catch(SocketException e) {
            System.err.println(e);
        } catch(IOException e) {
            System.err.println(e);
        } finally {
            if(socket!=null) socket.close();
        }
    }
}

```

---

Listing 2: Communicatie via UDP: Server-programma

Aan serverzijde moet, zoals gezegd, per openstaande stroom een socket aangemaakt worden. Dit gebeurt via de constructie:

```
ServerSocket listen=new ServerSocket(serverPort);
while(true) {
    Socket client=listen.accept();
    // server code
}
```

De `ServerSocket` wacht, op de gespecificeerde poort, tot een client zich aanbiedt. De `accept()`-methode blokkeert hierbij, i.e. de server wacht op clients. Wanneer dan wel een verzoek binnenkomt, creëert de `ServerSocket` een gewone `Socket` voor die specifieke client, waarlangs de communicatie dan kan verlopen.

## Opgave 1 - Begroeting

Pas de gegeven klassen `greeter.Client` en `greeter.Server` aan, zodat deze programma's via TCP met elkaar communiceren in plaats van UDP. Gebruik poort 1024 voor de TCP communicatie. De Client leest een persoonsnaam in van de console, en stuurt die naar de server. De server ontvangt die persoonsnaam, en stuurt een begroeting terug, namelijk die naam voorafgegaan door "Hello, ". De Client ontvangt de begroeting, en drukt deze af. De Client herhaalt dit proces tot de naam "stop" ingegeven wordt. Wanneer dit laatste gebeurt, stopt de Client, en sluit de Server de connectie naar die Client. De Server wacht vervolgens op nieuwe inkomende connecties. Het is dus niet de bedoeling om tegelijkertijd meerdere clients te ondersteunen. Aangezien er met Strings gewerkt wordt, kan gebruik gemaakt worden van de daarvoor bestemde Reader en Writer decorators: een `InputStreamReader` decorated met `BufferedReader` om te lezen, en een `OutputStreamWriter` decorated met een `PrintWriter` om te schrijven (zie cursus hoofdstuk 7).

## 3 Multithreading

Wanneer je het client-serverprogramma uit Opgave 1 test, dan kan de server slechts één client tegelijk afhandelen. Dit is doorgaans niet hoe je wil dat een server werkt (denk bijvoorbeeld maar aan een webserver). Om dit op te lossen, maken we gebruik van multithreading.

### Opgave 2: Parallele begroeting

In deze opgave maken we een meer realistische versie van de Begroeting uit opgave 1. Programmeer de klasse `parallelgreeter.Server` zodanig dat deze verschillende inkomende connecties in parallel kan behandelen. Daartoe splits je de code die specifiek de afhandeling van één Client op zich neemt, af in een afzonderlijke draad. Test dat je code inderdaad werkt wanneer meerdere Clients tegelijk actief zijn.

## 4 Serialisatie

Het ingebouwde Java-serialisatiemechanisme maakt het mogelijk om een reeks objecten, samen met hun relaties, naar een binair formaat om te zetten. Deze gedaante kan dan bijvoorbeeld naar een bestand geschreven worden, of over een netwerk verstuurd worden. De conversie tussen een object en zijn binaire, seriële vorm wordt mogelijk gemaakt door de Java-klassen `ObjectInputStream` en `ObjectOutputStream`. Schrijven naar en lezen uit een dergelijke stroom laat toe om een object als geheel te reconstrueren uit resp. om te zetten naar zijn seriële vorm.

### Opgave 3

In deze opgave maken we een client-servertoepassing, waarbij de client gegeven de naam van een persoon, het telefoonnummer van die persoon kan vinden. Aan de server worden immers een reeks personen met hun telefoonnummer bijgehouden. Wanneer de server een `Person`-object ontvangt, vult de server de ontbrekende velden (in casu het telefoonnummer) aan. Vul de gegeven klassen in de `serialization` package aan, waarbij je veronderstelt dat slechts één client zich tegelijk aanbiedt (dus zonder multithreading) en de connectie meteen gesloten mag worden zodra de server geantwoord heeft.

**Belangrijk!** – bij het creëren van de `ObjectInputStream` en `ObjectOutputStream` van een gegeven `Socket` MOET je onderstaande volgorde van bewerkingen volgen (anders bestaat het risico op blokkeren):

1. creëren van de `ObjectOutputStream`,
2. flushen van deze stroom,
3. creëren van de `ObjectInputStream`.

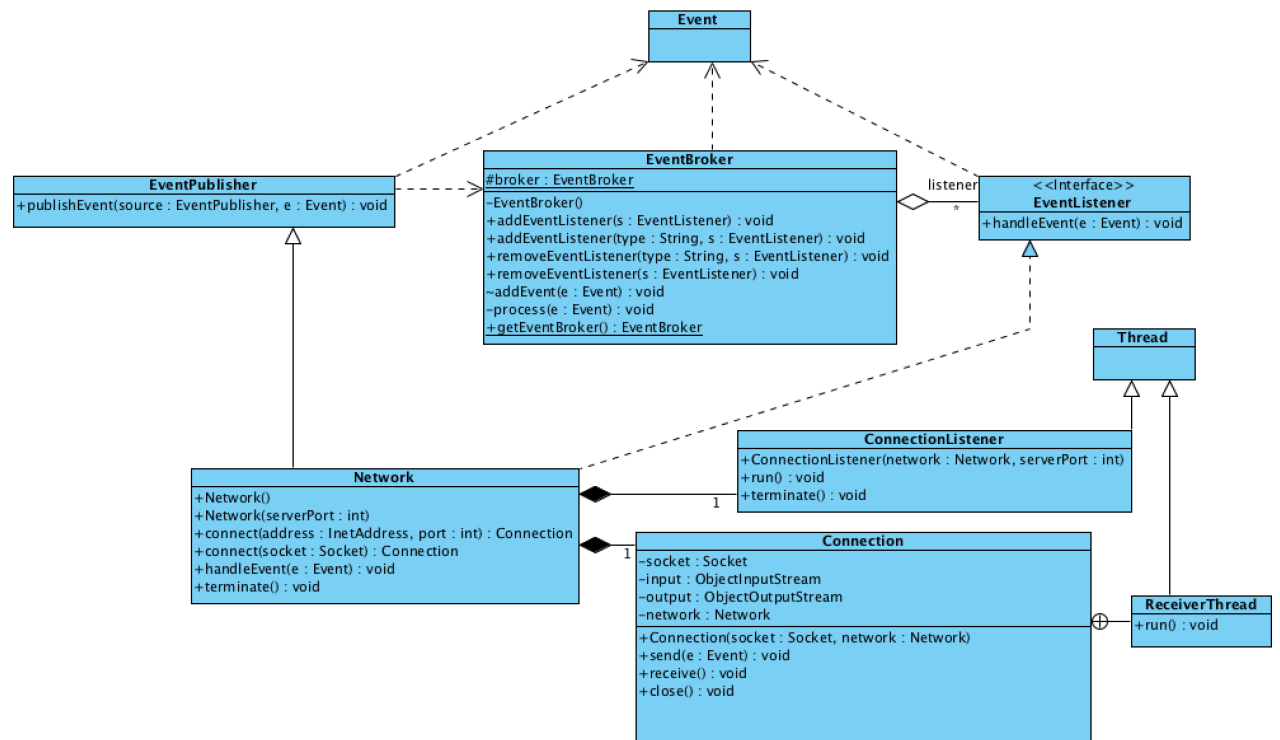
## 5 Netwerk events

In deze sectie breiden we de event broker uit de vorige labsessie uit, zodat Events over het netwerk kunnen verstuurd worden. Het is hierbij de bedoeling dat de `EventBroker` logisch één entiteit vormt, hoewel deze fysiek op verschillende processen aanwezig is. Op die manier wordt de toepassingsontwikkelaar, die van deze verdeelde `EventBroker` gebruik maakt, afgeschermd van de netwerkaspecten.

Het ontwerp van dit systeem wordt in onderstaande figuur weergegeven.

- de klasse `Network`: deze klasse implementeert de interface `EventListener`, en registreert zich als luisteraar bij de `EventBroker`. Op die manier wordt het netwerk van alle lokale Events op de hoogte gebracht, en wordt het versturen van de Events in de `handleEvent()`-methode verzorgd. Een proces kan ook Events ontvangen via het netwerk, en zal deze via de `Network` klasse aan de `EventBroker` doorgeven. De klasse `Network` fungeert dus ook als bron van Events, en erft dus tevens over van de klasse `EventPublisher`. Het versturen van de Events zelf wordt overgelaten aan een instantie van de klasse `Connection`.
- de klasse `Connection`: deze abstractie staat in voor het versturen en ontvangen van Events, en aggregeert daartoe een object van het type `Socket`, waarmee zowel een `ObjectInputStream` als een `ObjectOutputStream` geassocieerd zijn. De constructor van deze klasse neemt o.a. een `Socket` als argument. Deze `Socket` kan gegenereerd worden door de `accept()`-methode uitgevoerd op een `ServerSocket` of door expliciete constructie via IP-adres en poort. De methode `send()` verstuurt een Event over het netwerk, terwijl de `receive()`-methode Events ontvangt. Omdat we niet wensen dat bij het oproepen van de `receive()`-methode de toepassing blokkeert, start deze methode een afzonderlijke thread, namelijk een instantie van de klasse `ReceiverThread`. De `ReceiverThread` wacht continu op een binnenkomend Event en geeft dit door aan de `EventBroker` door een callback naar de methode `publishEvent()` van de klasse `Network`. Voor de eenvoud veronderstellen we dat een `Network` instantie slechts 1 `Connection` kan hebben, m.a.w. er kan met slechts één andere `EventBroker` berichten uitgewisseld worden.
- de klasse `ConnectionListener`: bij communicatie tussen twee processen, zal één van beide (typisch de server) wachten op inkomende verzoeken van het andere proces. Dit wachten wordt in een object van de klasse `ConnectionListener` gerealiseerd. Omdat we niet wensen dat de toepassing blokkeert, wordt deze geïmplementeerd als afzonderlijke thread.

De klassen `Network` en `ConnectionListener` beschikken over een methode `terminate()` die de actieve draaden afsluit.



Figuur 1: Ontwerp van een verdeelde eventbroker.

Aan serverzijde worden volgende stappen genomen:

- aanmaken van een `Network`-instantie, met opgave van de poort waarop gewacht wordt. Deze instantie registreert zich als luisteraar bij de `EventBroker` (niet getoond op Figuur 2!).
- aanmaken van een `ConnectionListener`-object, dat op de afgesproken poort wacht op inkomende verzoeken
- wanneer een inkomende verbinding gedetecteerd wordt, wordt een `Connection`-object aangemaakt, waarbij de `Socket` gebruikt wordt die door de `accept()`-methode toegeleverd werd.

Deze logica wordt verder verduidelijkt in Figuur 2.





van wie de `connect` methode oproept. Merk ook op dat er slechts 1 connectie nodig is om te communiceren in beide richtingen, aangezien je met eenzelfde `Connectie` zowel berichten kan sturen als ontvangen.

Opmerking: indien opgaven 4 en 5 niet afraken binnen het tijdsbestek van de labsessie, werk je die best thuis verder af. In volgende labsessies bouwen immers verder op de gedistribueerde `EventBroker`. Mocht je hierbij problemen ondervinden of hulp wensen, contacteer dan zeker de begeleiders van de labsessie per e-mail.

## Opgave 4

Bouw een genetwerkte `EventBroker` in overeenstemming met het hierboven beschreven ontwerp. Implementeer hiervoor de `network` package met daarin de klassen `Network`, `Connection` en `ConnectionListener`.

## Opgave 5

Bouw gebruik makend van de verdeelde eventbroker een verdeelde versie van de alarmtoepassing uit labsessie 1. Test het programma met behulp van bijgeleverde testcode in package `alarmevent`. Vul hierbij de code voor `Client` en `Server` verder aan. Hierbij genereert de `Client` alarmen, meer bepaald zijn er twee alarmcentrales actief, namelijk de centrale “112” en de centrale “101”. Achtereenvolgens worden volgende alarmen gegenereerd:

- type: crash, locatie: Plateastraat
- type: assault, locatie: Veldstraat
- type: fire, locatie: Zwijnaardse Steenweg
- type: assault, locatie: Overpoortstraat

Aan serverzijde worden als luisteraars geregistreerd (bij constructie):

- `Hospital UZ`
- `Hospital AZ`
- `PoliceDepartment`
- `FireDepartment`

Zowel aan client- als aan serverzijde dient uiteraard ook een instantie van de `EventBroker` te worden gestart, alsook een `Network`-instantie.

## 6 Tot slot

Dien je code in via Indianio, waarbij dezelfde codeerconventies gelden als voor labsessie 1. De ingediende zip-file bevat aangevulde \*.java bestanden, waarvoor telkens een startversie tevens op Indianio beschikbaar is. Zorg dat je dezelfde package- en klassenamen gebruikt als in de startcode.