Bas Weelinck (5985498), Merlijn Wajer (5948940), Koos van Strien (5783437)

18 mei 2010

# 1 Inleiding – probleembeschrijving

Volgens de specificaties gegeven in het opdrachtdocument moet een gedistribueerde chatserver worden geïmplementeerd. De gedistribueerde chatserver moet volgens gegeven specificaties communiceren met clients, andere servers en een controlserver die de verhoudingen tussen de verschillende servers vaststelt.

# 2 Structuur van de server

Om de server op te zetten hebben we gekozen voor een heldere structuur, door middel van een Socket Multiplexer en Managed Sockets.

## 2.1 Managed Sockets

Op het eerste gezicht is een Managed Socket niet meer dan een abstractie om makkelijk met een non-blocking socket om te gaan. Echter, doordat Managed Sockets uitgebreid zijn naar Managed DCP sockets wordt de functionaliteit opeens veel groter.

### 2.1.1 Managed DCP Sockets

In Managed DCP Sockets worden binnenkomende messages op een bepaald socket automatisch gecontroleerd op correctheid, en wordt een bijbehorende handler overeenkomstig het DCP-protocol aangeroepen. Door deze Managed DCP Sockets weer uit te breiden en de handlers te overriden, kan op een eenvoudige manier een implementatie van een controlserver-, een server- of een clientverbinding worden gemaakt.

Enkele events zijn standaard geïmplementeerd - denk hierbij bijvoorbeeld aan onPing en onPong-handlers. Andere events zijn wel beschikbaar, maar moeten nog afgehandeld worden. Te denken valt hier aan het verzenden van berichten - wanneer een bericht ván een bepaald socket komt en náár een ander socket gebracht moet worden, kan dit niet binnen de socket afgehandeld worden. Ten slotte mogen sommige events niet beschikbaar zijn in alle soorten sockets - een client mag bijvoorbeeld niet als client zijnde tegelijk registreren als server.

### 2.2 Socket Multiplexer

De socket multiplexer beheert grote hoeveelheden sockets binnen een thread. De socket multiplexer biedt mogelijkheden om te "port listener" uit te zetten of om handmatig connecties te openen. Aan deze connecties wordt een Managed Socket gehangen. Dit is voor de server voldoende - wanneer er een Managed Socket-implementatie bestaat voor de communicatie met de control-server en een implementatie voor de communicatie met clients en servers kan een server handmatig een connectie openen met de control-server (en bijbehorend socket er aan koppelen) en aan de port listener de tweede socket hangen.

In de multiplexer wordt tevens de data opgeslagen die door de hele server heen gebruikt wordt. Denk hierbij aan lijsten van verbonden clients en servers, de huidige parent-server en dergelijke. Ook biedt de multiplexer methoden om opdrachten uit te voeren op alle client- of alle server-sockets. Wanneer een Managed Socket een opdracht binnenkrijgt die niet binnen het huidige socket afgehandeld kan worden (wat in negen van de tien gevallen het geval is, aangezien er voor de meeste berichten communicatie met andere sockets nodig is), dan wordt de opdracht doorgespeeld naar de Socket Multiplexer en daar verder afgehandeld.

### 2.3 Voorbeeld: ChatServer, ControlServerJob en ChatSocket

De ChatServer-klasse start een multiplexer voor de chatserver. Deze multiplexer maakt verbinding met de controlserver via de ControlServerJob-klasse en beheert alle andere verbindingen via de ChatSocket-klasse. Wanneer de controlserver echter een parent toewijst aan de server, moet deze verbonden worden met een ChatSocket. Dit handelt de ChatServer-klasse af. Ook de lijsten met clients en servers worden bijgehouden in de ChatServer-klasse. In de praktijk zullen dus calls uit de ChatSocket- of ControlServerJob-klasse vaak doorgegeven worden aan de ChatServer-klasse, die er vervolgens de nodige acties mee uitvoert.

## 3 Tools

Om het ontwikkelen en testen van onze server wat eenvoudiger te maken, hebben we enkele tools geschreven die we hierbij kunnen gebruiken. Deze tools maken gebruik van de bovengenoemde DCP library.

### 3.1 Eenvoudige control-server

Aangezien er in de eerste week van de opdracht nog geen control-server beschikbaar was, hebben we er zelf één geschreven. Deze control-server biedt ons de mogelijkheid uitgebreid te loggen en gegarandeerd alleen onze eigen server te testen.

### 3.2 DCPNet-client

De DCPNet-client is een soort Telnet-client die op een handige manier kan omgaan met de messages zoals die gespecificeerd zijn in het DCP-protocol. Dit is handig voor o.a. het testen van de server en van het DCP-protocol zelf.

## 4 Server internals

In deze paragraaf willen we toelichten welke keuzes we gemaakt hebben in de implementatie van onze server. De keuzes die we willen toelichten zijn:

- Events
- Non-blocking sockets
- Meerdere processen
- Implementatie van het protocol
- Logging

### 4.1 Events

Het volledige systeem zoals we dat implementeren is event-driven. Sockets hebben binnen ons systeem callbacks, die geactiveerd worden wanneer een event zich voordoet.

Wanneer een nieuw soort socket wordt aangemaakt binnen de server, wordt deze gerepresenteerd door een uitbreiding van de ManagedSocket class. De standaard methoden (openen van socket e.d.) zijn daarmee al geïmplementeerd, sommige andere events worden op het moment van extension gedefinieerd.

### 4.2 Non-blocking sockets

We hebben ervoor gekozen om de gehele server in een enkel proces te gieten, dat via non-blocking sockets de communicatie verzorgt. Door non-blocking sockets te gebruiken is het niet nodig om meerdere processen te creeëren om beschikbaarheid van de server te kunnen garanderen. De overhead van het processen creëren voor elke connectie vervalt hierdoor.

Om met een enkele thread meerdere sockets te kunnen beheren maken we een SocketMultiplexer. Deze houdt bij welke sockets in de gaten gehouden moeten worden voor lezen en/of schrijven.

### 4.3 Meerdere processen

Zoals hierboven beschreven is het mogelijk de hele server in één enkel proces te draaien. Echter, dit levert enkele significante nadelen op.

### 4.3.1 Redenen om meerdere threads te gebruiken

De meeste machines op de UvA zijn multi-core. Door alles in een enkele thread te draaien, wordt een groot deel van de capaciteit van de machine niet gebruikt. Door het proces in meerdere threads te draaien kunnen we een groter deel van de capaciteit van de machine gebruiken.

Daarnaast heeft Linux een standaardlimiet van 1024 file descriptors per proces. Naast dat dit een beperking geeft aan het aantal clients op de server, waarbij de beperking niet voortkomt uit de capaciteit van de server maar uit de beperking van een protocol (TCP/IP), maakt dit de server een nogal makkelijk doelwit voor DoS-attacks, omdat 1024 connecties al genoeg zijn zijn om de server geen clients meer te laten accepteren en zo "op slot" te gaan. Wanneer we meer dan 1024 clients op onze server willen toelaten, dan zullen we meerdere processen moeten gebruiken per server.

### 4.3.2 Problemen met Python

Python biedt standaard een library aan voor multi-processing, maar deze werkt niet op de UvA. Daarnaast biedt Python ondersteuning voor threading, maar houdt daarbij alle threads binnen één proces. Daardoor worden de meerdere cores van een processor nog steeds niet gebruikt.

Om ervoor te zorgen dat onze server daadwerkelijk uit meerdere processen zou bestaan èn op de UvA zou werken, hebben we zelf een RPC-bridge library geschreven. Door middel van fork()-calls verdeelt deze library een klasse over meerdere processen.

## 4.4 Implementatie van het protocol

We hebben ervoor gekozen ons zo strikt mogelijk aan het protocol te houden. Om door alle klassen heen dezelfde implementatie te verzekeren hebben we een DCP library geschreven. Deze library is een verzameling van klassen, losse functies en gedefinieerde constanten die het werken met het DCP-protocol eenvoudig en in alle klassen eender maken. De kern van de library is de ManagedDCPSocket-klasse, die in een socket voorziet waarin functionaliteit voor een bepaalde gespecificeerde call geïmplementeerd wordt d.m.v. on**-methoden.

Hoewel we de werking van onze eigen server kunnen verifiëren en tot op zekere hoogte ook garanderen, kunnen we over de werking van andere servers weinig zeggen. Ten opzichte van deze servers, waarmee we toch zullen moeten samen werken, stellen we ons vrij lenient op.

## 4.5 Logging

Om de serverlogs netjes bij te houden hebben we een module geschreven die de logging afhandelt. De PyLogger biedt verschillende niveaus van verbosity, messages in drie categorieën (Information, Warning en Error) en mogelijkheden

om naar stdout uit te voeren (al dan niet in kleur). Bovendien biedt de PyLogger log-rotation, zodat het formaat van de log-files altijd binnen de perken blijft.

# 5 Evaluatie server: verbeteringen en uitbreidingen

Hier willen we enkele mogelijke verbeteringen bespreken en evalueren in hoeverre ze het waard zouden zijn te implementeren.

- Problemen
- Verbeterpunt: Andere programmeertaal?

## 5.1 Problemen

Bij het uitwerken van de server liepen we tegen enkele kleine problemen aan. In deze sectie noemen we enkel de problemen die nu nog spelen.

### 5.1.1 Localhost

Bij het opvragen van het IP-adres van sommige machines (niet op de UvA) wordt het IP-adres '127.0.0.1' geretourneerd, ongeacht of er een internetverbinding is of niet. Dit kan natuurlijk aan de configuratie van de machine liggen, maar het zorgt er wel voor dat de server zich niet normaal kan aanmelden bij de control-server.

### 5.1.2 Vrijmaken van ontbonden clients en servers

Na het verbreken van een verbinding blijft ergens nog een reference staan naar de bijbehorende ChatSocket. Gevolg is dat geheugen onnodig bezet gehouden wordt.

## 5.2 Andere programmeertaal

Hoewel Python een stabiele programmeertaal is en snel genoeg om een redelijke server draaiend te houden, kan het altijd efficiënter. Het porten van de code naar C++ zou tot een verhoging van performance en capaciteit kunnen leiden.

Hoewel het porten van de code waarschijnlijk tot enige mate van verhoging van performance zal leiden, is de vraag hoe groot dit effect zal zijn. Veel van de performance-gevoelige code wordt namelijk uitbesteed aan de kernel. De performance-winst zal vermoedelijk daarom zo laag zijn dat het de moeite niet is om de code te porten terwille van de performance.

# 6 Team

## 6.1 Communicatie en code delen

We hebben aan het begin van het project een GIT-repository ingericht waarin we alle code door het project heen plaatsten. Naast de code bevonden zich hier ook een TODO en een PROGRESS-bestand. Wat er per commit veranderde stond logischerwijs beschreven in de GIT commit-message. Daarnaast hebben we het grootste gedeelte van de communicatie gedaan via een eigen IRC-channel.

## 6.2 Problemen

Bij aanvang van het project was het grootste gedeelte van de server vrij snel geschreven. Bas had al een duidelijk plan in zijn hoofd en leverde daar erg snel een basis mee. Merlijn en Koos moesten zich daardoor eerst in de code inwerken en de bestaande concepten helder krijgen voordat ze er verder mee konden programmeren. Met name Koos had hier wat moeite mee. Sommige toevoegingen uit de een-na-laatste week maakten dit ook duidelijk, aangezien toen bleek dat grote delen van de server die hij geprogrammeerd had, niet netjes in elkaar bleken te steken.

Aan het eind van het project vertaalde dit zich in het feit dat Merlijn en Bas de server grotendeels afgemaakt hebben, waarin met name Merlijn het voortouw nam.

## 6.3 Wie deed wat?

Visualisatie en stats zijn te vinden `http://wizzup.org/dcs/` Naast een Gource-visualisatie, volgt hier een kort - GLOBAAL - overzicht van wie wat deed.

- Opzet server – Bas
- Schrijven control-server – Bas
- Socket Multiplexer – Bas
- DCP Library – Bas (en Merlijn)
- DCPNet client – Bas
- Vervolg server – Merlijn en Koos
- Aanpassingen/Fixes DCP Library - Merlijn en Koos
- Afronding server – Merlijn en Bas
- Client – Bas en Merlijn
- Verslag – Koos

# 7 Code

De GIT repository waar de code zich in bevindt kan gevonden worden op `http://vila.villavu.com/cgi-bin/gitweb/gitweb.cgi?p=dsc.git;a=summary` Op de UvA is een checkout te realiseren d.m.v. het commando

```
git clone git://vila.villavu.com/dsc.git
```