JNRSync: een synchronisatie framework

Rik Harink Joshua Moerman Nick Overdijk

15 juli 2010

Inhoudsopgave

1 Uiteindelijk Product

1.1 Framework

We hebben een simpel framework gemaakt om games mee te synchroniseren. Het is niet geworden wat we hadden gehoopt, maar de belangrijkste dingen zijn mogelijk. Kort samengevat kan ons framework het volgende (vanuit de cliënt gezien):

- Een verbinding aanleggen met een server
- Data versturen naar de server
- Data ontvangen van de server

De server doet in ons framework niets anders dan alle data doorsturen naar de andere cliënten.

Het framework is geschreven in *C++* op een vrij abstracte manier. We hebben het zo abstract mogelijk gehouden, zodat een gamedeveloper naderhand nog makkelijk zijn spel kan laten synchroniseren. Zonder deze abstractie zou de developer mogelijk zijn softwaredesign moeten aanpassen aan ons framework, maar dit hebben we zoveel mogelijk vermeden.

Het hele framework is terug te vinden in de bijlage. Voor een gebruiker van het framework is slechts een klein deel van belang en van dat deel is er een documentatie.

1.1.1 Een verbinding aanleggen met een server

De verbinding die ons framework aanlegt is een *TCP* verbinding. Er zijn twee veelgebruikte soorten verbindingen, namelijk een TCP verbinding en een *UDP* verbinding. Wij hebben in eerste instantie voor TCP gekozen, omdat deze garandeert dat de verzonden data daadwerkelijk aankomt (tenzij er fysiek geen verbinding mogelijk is natuurlijk). UDP garandeert dit niet en dat is voor de onderzoeksfase niet handig (want als er dan fouten zijn, kun je niet met zekerheid zeggen waar het aan ligt).

TCP is daarentegen wel langzamer dan UDP, maar aangezien we merkten dat de verbinding toch wel snel genoeg was, leek het voor ons geen probleem. Later zagen we ook dat de voorbeeldgame goed speelbaar was en dus was TCP geen verkeerde keus.

De verbinding wordt vanuit de cliënt geïnitialiseerd. Hierbij moet de cliënt het IP-adres van de server weten. Het is in games niet ongebruikelijk om het IP-adres van de server te moeten invullen. Het kan ook geautomatiseerd gebeuren maar dat ligt buiten ons onderzoek. Deze ene verbinding is genoeg om zowel data te kunnen versturen als data te kunnen ontvangen.

1.1.2 Data versturen en ontvangen

Nu we met TCP een verbinding hebben aangelegd kunnen we data versturen. We gebruiken hiervoor standaard *POSIX sockets*. Een socket is een abstract object voor communicatie, dus een object waarnaar je kunt schrijven en waaruit je kunt lezen. Het inlezen en wegschrijven van bestanden wordt op UNIX systemen ook met zulke sockets gedaan.

In ons framework wordt de socket gemaakt bij het aanleggen van de TCP verbinding. Als de cliënt vervolgens schrijft naar de socket, kan de server het vervolgens uitlezen. Andersom geldt precies hetzelfde. Je kunt het vergelijken met 2 bestanden; de cliënt schrijft altijd in bestand A en leest uit bestand B, terwijl de server altijd schrijft in bestand B en leest in bestand A. Als iemand heeft gelezen wordt het bestand geleegd, zodat de andere partij weer nieuwe data kan wegschrijven.

1.1.3 Serialisatie

Naast de verbinden en het schrijven, is er nog een derde probleem, welke taal moeten de cliënten spreken? Het is fijn dat we nu data kunnen versturen, maar als een cliënt uit een sockets leest, dan moet hij er wel iets mee kunnen doen. Hiervoor hebben we in C++ het abstracte type *Serializable* gemaakt. Als jij met het

framework bijvoorbeeld je object playerData wilt versturen, moet je ervoor zorgen dat hij de methodes van Serializable implementeert. Deze methodes zijn (voor een betere beschrijving, zie documentatie):

- serialize
- deserialize
- getSize
- getType

De eerste twee bepalen hoe het object naar het geheugen geschreven dient te worden (serialize) en hoe het uit het geheugen gelezen dient te worden (deserialize). Als je dus op je object de methode serialize oproept, schrijft hij zich weg in het geheugen. Dit geheugen kan je vervolgens weer schrijven in de socket, zodat de server je object kan doorsturen.

Maar je moet in C++ eerst de hoeveelheid geheugen declareren voordat je er zomaar in mag schrijven, daarvoor is de methode getSize. Je kan pas een buffer maken, als je van te voren weet hoe groot iets is, doordat je nu de grootte kan opvragen, weet je precies hoeveel je vrij moet maken om naar te schrijven. Deze methode maakt het ook goed mogelijk om objecten van dynamische grootte te kunnen versturen, zoals lijsten.

Als gamedeveloper wil je ongetwijfeld meerdere soorten objecten doorsturen, daarvoor is de methode getType. Als je een lijst van objecten wilt versturen (of mooier gezegd: een container), dan moet de container bij elk object noteren van welk type het is, zodat ja bij het uitpakken van de container ook de juiste objecten aanmaakt. Door de types dynamisch te houden, is de gamedeveloper niet beperkt door ons framework.

Ons framework heeft al enkele elementaire datatypes geencapsuleerd in serialiseerbare objecten. We bieden de float, integer en array aan. Door operator overloading in C++ kan je als gamedeveloper alle code, op de instantiatie van je variabele na, alles gelijk houden (we hebben niet genoeg tijd gehad om alle operators te overloaden, maar het is zeer makkelijk te doen) en hoef je niet zelf na te denken hoe je een float precies in het geheugen gaat representeren.

1.1.4 Synchronisatie

Met deze bouwstenen is het nu eenvoudig om met ons framework te synchroniseren. Er is een methode *sync* die, bij het oproepen, alle ontvangen data uitpakt en eventueel ook data verstuurt. Van te voren kan de gamedeveloper aangeven welke data hij zou willen versturen (dit hoeft hij maar één keer te doen).

1.1.5 In een notendop

Ons framework is in staat op een simpele manier data te versturen volgens het server-cliënt model. Fysiek gebruikt ons framework een TCP verbinding en POSIX sockets om te lezen en te schrijven. Het framework biedt een abstracte manier om je objecten door te sturen en staat een onbeperkt aantal types toe, zodat de gamedeveloper niet beperkt wordt. De uiteindelijke synchronisatie wordt bereikt met 'e'en enkele methode: *sync*.

1.2 Testgame

Naast het framework hebben we voor ons onderzoek ook een testgame gemaakt. Dit onderdeel is onafhankelijk gemaakt, zodat we in de game geen aannames zouden maken over het framework. We hebben dus zowel het gemak van integreren als de synchronisatie kunnen testen.

Het spel is vrij eenvoudig. Elke cliënt bedient een speler op het scherm. Een speler gaat in een van de vier richtingen: noord, oost, zuid, west. De snelheid van een speler is constant. Er is geen interactie tussen de spelers. Spelers stuiteren aan de rand van het veld. De cliënt laat ook de andere spelers op het scherm zien (in een andere kleur). We konden visueel al goed zien of de synchronisatie wel of niet goed was.

In de sync methode van het framework kunnen we aangeven of we zijn veranderd (en dus kunnen we aangeven of we zelf data willen verturen). Dit stelde ons in staat het spel op twee manieren te testen. Namelijk testen met contstant je eigen data sturen (timer-based), of alleen data versturen als je veranderd bent (event-based).

1.3 Ongehaald doel

We hadden ons oorspronkelijke doel (een framework, die slim uitzoekt wat er vesrtuurd moet worden) niet gehaald. Maar met het uiteindelijke product waren we alsnog in staat veel cliënten zeer goed syncrhoon te houden. Er was ook enige speling in het framework (het feit dat je kan aangeven data te versturen slechts als je veranderd bent) en hiermee kan je goede testen doen. Tevens is het framework makkelijk in gebruik, wat ook belangrijk is.

2 Framework Documentation

Deze documentatie is voor de gebruiker van ons framework. Hierin worden alleen de aspecten genoemd die de gebruiker van ons framework nodig heeft. De aspecten die besproken worden zijn:

- Serialiseerbare objecten
- Framework initialistie
- Framework linken

Merk op dat het hele framework in de namespace JNRSync zit. Als je van de functies van ons framework gebruik moet je ofwel **using namespace** JNRSync; gebruiken of de functies beginnen met ::JNRSync. We zullen dit laatste gebruiken in de voorbeelden.

2.1 Serializable

Het framework is alleen in staat serialiseerbare objecten te synchroniseren. Een object is serialiseerbaar als zijn klasse een subklasse is van de Serializable klasse en de abstracte functies implementeert. Een serialiseerbaar object moet dus de volgende functies implementeren:

```
virtual void serialize(void * const data, size_t & bytePosition) const;
virtual void deserialize(const void * const data, size_t & bytePosition);
virtual size_t getSize() const;
virtual uint16_t getType() const;
```

2.1.1 serialize

Dit is de functie die het object naar een buffer schrijft.

data is de buffer waarin er geschreven moet worden en bytePosition bepaalt waar in de buffer geschreven dient te worden. bytePosition moet vervolgens doortellen, zodat de volgende functie die bytePosition gebruikt, weet waar hij in data moet schrijven.

Als je object uit meerdere serialiseerbare objecten bestaat, is het zeer eenvoudig om deze functie te gebruiken:

```
class SpelerKlasse : public JNRSync::Serializable {
        JNRSync::Integer snelheid;
        JNRSync::Integer levens;

        // we willen de twee member variabelen serialiseren.
        virtual void serialize(void* const data, size_t& bytePosition) const{
                snelheid.serialize(data, bytePosition);
                levens.serialize(data, bytePosition);
        }

        // rest van de klasse ...
}
```

Doordat de bytePosition in beide functies door zal tellen, worden de twee variabelen na elkaar in de buffer geschreven. In het geval van een niet serialiseerbaar object is het iets moeilijker:

```
// we willen een int wegschrijven.
virtual void serialize(void * const data, size_t & bytePosition) const{
    size_t bytes = sizeof(int);
    memcpy((void*)((char*)data + bytePosition), &my_c_int, bytes);
        bytePosition += bytes;
}
```

Hierin schrijf je de binaire representatie van een **int** (het elementaire C type) weg op de plek waar data naar wijst met de verplaatsing die bytePosition aangeeft.

Omdat dit laatste niet zo mooi is, levert het framework de elementaire datatypes van C als serialiseerbare objecten. Zie hoofdstuk 2.3

2.1.2 deserialize

Het uitlezen van een buffer gaat op analoge wijze als het wegschrijven zoals in serialise werd beschreven, zie hoofdstuk 2.1.1. We geven hier slechts een voorbeeld, die aansluit bij het voorbeeld van serializable:

```
// we willen de twee member variabelen deserialiseren.
virtual void deserialize(void* const data, size_t& bytePosition){
        snelheid.deserialize(data, bytePosition);
        levens.deserialize(data, bytePosition);
}
```

Bedenk dat je deserialize alleen op kan roepen op een instantie, dus je zal van te voren een object moeten aanmaken. Meer over het alloceren vind je bij SerializableObjectFactoryFunctions, hoofdstuk 2.2

2.1.3 getSize

De getSize methode geeft terug hoeveel bytes de data inneemt in de buffer. Als we doorgaan op het voorbeeld, wordt de code weer erg simpel, want we kunnen simpelweg de groottes van die objecten aanvragen:

```
// we willen de twee member variabelen deserialiseren.
virtual size_t getSize() const{
        size_t size = 0;
        size += snelheid.getSize();
        size += levens.getSize();
        return size;
}
```

De grootte die getSize teruggeeft moet gelijk zijn aan hoeveel de bytePosition verschuift bij het oproepen van de serialize en deserialize functies. Deze functie is handig voor het alloceren van de buffer waarin je je object vervolgens gaat serialiseren.

Bij elementaire datatypes kan je **sizeof**(type) gebruiken.

2.1.4 getType

Aangezien het mogelijk is meer dan 1 soort data te versturen, moet je het type kunnen opvragen, zodat deze informatie meegestuurd kan worden. Je moet zelf je types binden aan een uint16_t. Je mag elk getal returnen, zolang het maar consequent gebeurt.

We bevelen aan om met een **enum** de types vast te zetten:

```
enum GameSerializeTypes {
    kSerializedTypePlayer = JNRSync::kSerializedTypeLast,
    kSerializedTypeExtraGameInformation,
    ...
};
```

Hierbij moet je rekening houden met het feit dat het framework ook al enkele types kent. Om verder te tellen vanaf het laatste type van het framework moet je de eerste gelijkstellen aan kSerializedTypeLast

2.2 SerializableObjectFactoryFunction

Als het framework data ontvangt, probeert hij dat uit te pakken. Hij maakt hierbij gebruik van SerializableObjectFactoryFu
Deze functies (in de vorm van objecten) maken de serialiseerbare objecten. Elk type object heeft zijn eigen factory functie. Dit object moet de volgende functie implementeren:

```
virtual Serializable* operator()(uint16_t type) const;
```

Deze functie krijgt het type, en kan daarop controleren voordat je het object daadwerkelijk alloceert. Typisch ziet een implementatie er zo uit:

```
class SpelerFactoryFunction : public SerializableObjectFactoryFunction {
        virtual Speler* operator()(uint16_t type) const{
                if (type != kSerializedTypePlayer) {
                        return NULL;
                }
                return new Speler;
        }
};
```

waarbij Speler een serialiseerbaar object is. Nadat deze functie is aangeroepen door het framework, wordt er op het nieuwe object deserialize opgeroepen.

Nu moet deze functie nog toegevoegd worden aan het framework, dit gebeurt bij het initialiseren, zie hoofdstuk 2.4

2.3 Elementaire Datatypes

Ons framework heeft 3 elementaire datatypes:

- Integer
- JNRFloat
- Array

Integer en JNRFloat zijn de serialiseerbare varianten van de standaard datatypes **int** en **float** respectievelijk. Deze kunnen ook alsdien worden gebruikt, er zijn enkele van de operators geoverload, dus ze voelen aan alsof het **int**s en **float**s zijn. Als een operator niet bestaat, kun je altijd de waarde opvragen met getValue en de waarde vervolgens zetten met de assignment operator (de = in C++).

Array is een lijst van willekeurige serialiseerbare objecten. Deze kan handig gebruikt worden bij het maken van datacontainers. Voor het hanteren van de lijst kan je de member variabele members gebruiken, van het type std::vector<Serializable*>.

2.4 Framework initialiseren

Bij het opstarten van het programma dien je als eerste initiatie stap met betrekking tot het framework de functie init uit te voeren. Dus:

```
int argc, char *argv[]{
        // ...
        JNRSync::init();
        /// ...
}
```

Vervolgens is het slim om meteen de SerializableObjectfactoryFunctions te binden, zodat het framework de objecten kan aanmaken:

```
JNRSync::SerializableObjectFactory::functions.push_back(new
    SpelerFactoryFunction);
```

Het maakt niet echt uit waar je dit allemaal initialiseert, zolang het maar voor de eerste call naar sync. De rest van de initialisatie is afhankelijk van het feit of het programma de client draait of de server.

2.4.1 Client

Nadat de basis elementen van het framework zijn geïnitialiseerd, kun je nu specifiek de client starten. Dit gebeurt met de constructor van het Client-object:

```
const std::string serverIP = "10.0.2.4";
JNRSync::Client myClient(serverIP);
```

Vervolgens kun je je cliënt vertellen welke data je wilt synchroniseren, door de data toe te voegen aan de cliënt (waarbij er natuurlijk een Serializable object moet worden gegeven):

```
myClient.add(&player);
```

Dat was de gehele initialisatie van de cliënt. Het enige wat resteert is om daadwerkelijk te synchroniseren. De beste plek om dit te doen is in je Runloop van de game (of met een timer, als je OS dat aanbiedt). De functie die synchroniseert is de volgende:

```
void sync(bool shouldSend = true);
\end{lstlisting}

Deze functie heeft \'e\'en parameter: shouldSend. Met deze parameter kun je
    aangeven of de cli\"ent zelf data moet versturen (hiermee kan je
    bijvoorbeeld regelen dat de cli\"ent alleen stuurt als hij zelf beweegt).


\subsubsection{Server}

Na het initialiseren van het framework kun je ook de server starten, dat doe
    je alsvolgt:

\begin{lstlisting}[frame=single, breaklines=true]
JNRSync::Server server;
server.start();
```

2.5 Framework Linken

We raden aan om het framework als een statische library te bouwen, zodat je deze gemakkelijk in je project kan integreren. De enige header die je moet includen is de JNRSync.h header.

3 Framework code

```
/*
 *  Array.h
 *  JNRSync
 *
 *  Created by Joshua Moerman on 5/31/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#ifndef ARRAY_HEADER
#define ARRAY_HEADER

#include <vector>
#include "SerializableObjectFactoryFunction.h"

namespace JNRSync {
        class Serializable;

        class Array : public Serializable {
        public:
                std::vector<Serializable*> members;

                virtual uint16_t getType() const;
                virtual void serialize(void * const data, size_t &
                   bytePosition) const;
                virtual void deserialize(const void * const data, size_t &
                   bytePosition);
                virtual size_t getSize() const;
                virtual ~Array() {}
        };

        class ArrayFactoryFunction : public SerializableObjectFactoryFunction
           {
                virtual Serializable* operator()(uint16_t type) const{
                        if (type != kSerializedTypeArray) {
                                return NULL;
                        }
                        return new Array;
                }
        };

}

#endif
```

```
/*
 *  Buffer.h
 *  JNRSync
 *
 *  Created by Joshua Moerman on 5/11/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#ifndef BUFFER_HEADER
#define BUFFER_HEADER

#include "MutexLock.h"

namespace JNRSync {
        class Buffer : public Lockable {
        public:
                // ctors & dtors
                Buffer(size_t size);
                Buffer(const Buffer &copyBuffer);
                ~Buffer();

                // members
                MutexLock locky;
                size_t bytesRead;

                // methods
                void print() const;


                const void * getBuffer() const;
                void * getBuffer();

                size_t getSize() const;
                void reserve(size_t size);
                void clear();

                void removeBytesAtBeginning(size_t bytes);

                virtual int lock();
                virtual int unlock();

        private:
                void* buffer;
                size_t size;
        };
}

#endif
```

```
/*
 *  Client.h
 *  JNRSync
 *
 *  Created by Nick Overdijk on 5/10/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#ifndef CLIENT_HEADER
#define CLIENT_HEADER

#include <string>

#include "Player.h"
#include "Server.h"
#include "Connection.h"
#include "Array.h"

namespace JNRSync {
        class Client {
                public:
                Client(const std::string &ipAdres, const int &
                    incomingConnectionFD);
                Client(const std::string &serverIpAdres);

                Connection incoming;

                Array objectsToSync;

                void connect(const std::string &serverIpAdres);

                void add(Serializable * object);
                void sync(bool shouldSend = true);
        };
}

#endif
```

```
/*
 *  Connection.h
 *  JNRSync
 *
 *  Created by Joshua Moerman on 5/11/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#ifndef CONNECTION_HEADER
```

```
#define CONNECTION_HEADER

#include <ctype.h>

#include <string>

#include "Buffer.h"

namespace JNRSync {
        class Connection {
        public:
                static unsigned int verboseLevel;
                Connection();
                Connection(int connectionFD);
                int connectionFD;

                Buffer incomingBuffer;
                Buffer outgoingBuffer;

                int read(size_t bytes);

                int write(void* buffer, size_t bytes);
                int write(const Buffer& bufferToWrite, size_t bytes);

                void printBuffer();

                void setNonBlock(const bool nonBlock);
        };

}

#endif
```

```
/*
 *  Connector.h
 *  JNRSync
 *
 *  Created by Joshua Moerman on 5/25/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */


#ifndef CONNECTOR_HEADER
#define CONNECTOR_HEADER

#include <string>
using std::string;

namespace JNRSync {
```

```
        class Connector{
        public:

                static int connect(const string& ipAdres);

        };
}

#endif
```

```
/*
 *  Functor.h
 *  JNRSync
 *
 *  Created by Nick Overdijk on 6/14/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#ifndef FUNCTOR_HEADER
#define FUNCTOR_HEADER

namespace JNRSync {
        class Serializable;
        class Functor {
        public:
                virtual void operator() (Serializable const * const object) =
                    0;
        };
}

#endif
```

```
/*
 *  Globals.h
 *  JNRSync
 *
 *  Created by Nick Overdijk on 5/25/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

namespace JNRSync {
        extern unsigned int incomingConnectionsPort;
}
```

```
/*
 *  Informant.h
 *  JNRSync
 *
 *  Created by Nick Overdijk on 6/11/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#ifndef INFORMANT_HEADER
#define INFORMANT_HEADER

#include <string>

namespace JNRSync {
        class Informant {
        public:
                enum {
                        kVerboseLevelNone,
                        kVerboseLevelUseful,
                        kVerboseLevelDebug,
                        kVerboseLevelDebugUltra,
                        kVerboseLevelRedundant
                };

                Informant();
                Informant(const unsigned int &level);

                void print(const std::string &message, const unsigned int
                    levelNeeded);
                void perror(const std::string &message, const unsigned int
                    levelNeeded);

        private:
                unsigned int verboseLevel;

        };
}

#endif
```

```
/*
 *  Integer.h
 *  JNRSync
 *
 *  Created by Joshua Moerman on 5/31/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */
```

```
#ifndef INTEGER_HEADER
#define INTEGER_HEADER

#include <iostream>
#include <cstdlib>

#include "SerializableObjectFactoryFunction.h"
#include "Serializable.h"

namespace JNRSync {
        class UniqueInteger;
        class Integer : public Serializable {
        public:
                Integer();
                Integer(UniqueInteger const &uniqueInteger);
                Integer(int const &value);

                friend std::ostream& operator<<(std::ostream& out, Integer
                    const &integer){
                        out << integer.value;
                        return out;
                }

                int getValue() const;

                //This is the PREfix operator ++Integer
                Integer& operator++ ();
                Integer& operator-- ();
                //Integer& operator++ (int) //this'd be the POSTfix operator
                    Integer++

                Integer& operator= (const Integer& copy);
                Integer& operator= (const int &copy);

                virtual uint16_t getType() const;
                virtual void serialize(void * const data, size_t &
                    bytePosition) const;
                virtual void deserialize(const void * const data, size_t &
                    bytePosition);
                virtual size_t getSize() const;
                virtual ~Integer() {}

        protected:
                int value;
        };

        class IntegerFactoryFunction : public
            SerializableObjectFactoryFunction {
                virtual Serializable* operator()(uint16_t type) const{
                        if (type != kSerializedTypeInteger) {
```

```
                                return NULL;
                        }
                        return new Integer;
                }
        };
}


#endif
```

```
/*
 *  JNRFloat.h
 *  JNRSync
 *
 *  Created by Nick Overdijk on 5/10/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#ifndef JNRFLOAT_HEADER
#define JNRFLOAT_HEADER

#include <cstdlib>
#include <iostream>

#include "Serializable.h"
#include "SerializableObjectFactoryFunction.h"

namespace JNRSync {
        class JNRFloat : public Serializable {
        public:
                float value;

                friend std::ostream& operator<<(std::ostream& out, const
                    JNRFloat& _float){
                        out << _float.value;
                        return out;
                }

                JNRFloat& operator=(const JNRFloat& rhs){
                        value = rhs.value;
                        return *this;
                }

                JNRFloat& operator=(const float& rhs){
                        value = rhs;
                        return *this;
                }

                virtual uint16_t getType() const;
```

```
                virtual void serialize(void * const data, size_t &
                    bytePosition) const;
                virtual void deserialize(const void * const data, size_t &
                    bytePosition);
                virtual size_t getSize() const;
                virtual ~JNRFloat() {}
        };

        class FloatFactoryFunction : public SerializableObjectFactoryFunction
            {
                virtual Serializable* operator()(uint16_t type) const{
                        if (type != kSerializedTypeFloat) {
                                return NULL;
                        }
                        return new JNRFloat;
                }
        };
}

#endif
```

```
/*
 *  JNRSync.h
 *  JNRSync
 *
 *  Created by Nick Overdijk on 6/10/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#ifndef JNRSYNC_HEADER
#define JNRSYNC_HEADER

#include "Serializable.h"
#include "SerializableObjectFactory.h"

#include "Integer.h"
#include "JNRFloat.h"
#include "Array.h"
#include "UniqueInteger.h"

#include "Informant.h"

namespace JNRSync {
        /**
         Initializes JNRSync: adds factory functions for JNRSync primitive
            types
        */
        void init();
```

```
        class Functor;
        extern Functor * uniqueIntegerReceivedFunction;

        void uniqueIntegerReceived(const UniqueInteger &uniqueInteger);
}

#endif
```

```
/*
 *  Listener.h
 *  JNRSync
 *
 *  Created by Joshua Moerman on 5/11/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#ifndef LISTENER_HEADER
#define LISTENER_HEADER

namespace JNRSync {

        class Listener {
        public:
                Listener();
                static void* listen(void* delegate);

        private:
                int socketFD;
                int port;
        };


}

#endif
```

```
/*
 *  ListenerDelegateInterface.h
 *  JNRSync
 *
 *  Created by Joshua Moerman on 5/25/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#ifndef LISTENERDELEGATEINTERFACE_HEADER
#define LISTENERDELEGATEINTERFACE_HEADER
#include <string>
```

```
namespace JNRSync {

        class Listener;

        class ListenerDelegateInterface {

        public:

                virtual void setListener(Listener* listener) = 0;
                virtual void incomingConnection(const int connectFD, const std
                    ::string &ipAdress) = 0;

        };
}

#endif
```

```
/*
 *  Lockable.h
 *  JNRSync
 *
 *  Created by Nick Overdijk on 5/11/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#ifndef LOCKABLE_HEADER
#define LOCKABLE_HEADER

namespace JNRSync {
        class Lockable  {
        public:
                virtual int lock() = 0;
                virtual int unlock() = 0;
        };
}

#endif
```

```
/*
 *  MutexLock.h
 *  JNRSync
 *
 *  Created by Nick Overdijk on 5/11/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */
```

```
#ifndef MUTEXLOCK_HEADER
#define MUTEXLOCK_HEADER

#include <pthread.h>

#include "Lockable.h"

namespace JNRSync {
        class MutexLock : public Lockable {
        public:
                MutexLock();
                ~MutexLock();

                virtual int lock();
                virtual int unlock();

        private:
                pthread_mutex_t locky;

                static void handleMutexInitError(int status);
                static void handleMutexDestroyError(int status);
        };

}

#endif
```

```
/*
 *  NMLog.h
 *  FastTypeModelController
 *
 *  Created by Nick Overdijk and Maurice Bos of http://dotsimplcity.net on
    2/21/10.
 *  Copyright 2010. All rights reserved.
 *
 */

#ifndef NMLOG_H
#define NMLOG_H

#define NMLog(s, ...) \
nmstd::Log::print(__FILE__, __LINE__, (s), ##__VA_ARGS__)

#include <cstdlib>
#include <cstdio>

namespace nmstd {
        class Log {
        public:
                static void print(const char * filename, const unsigned int
                    lineNumber, const char * format, ... ){
```

```
                        va_list ap;

                        if(loggingOn == false) return;

                        printf("%s:%d :: ", filename, lineNumber);
                        printf(format, ap);
                        putchar('\n');
                }

                const static bool loggingOn = true;
        };
}

#endif
```

```
/*
 *  PosixUtils.h
 *  JNRSync
 *
 *  Created by Nick Overdijk on 6/10/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */
#ifndef POSIXUTILS_HEADERS
#define POSIXUTILS_HEADERS

int set_nonblock_flag (int desc, const bool value);

#endif
```

```
/*
 *  Serializable.h
 *  JNRSync
 *
 *  Created by Nick Overdijk on 5/10/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#ifndef SERIALIZABLE_HEADER
#define SERIALIZABLE_HEADER

#include <ctype.h>
#include <cstdlib>
#include <stdint.h>

namespace JNRSync {
        enum {
                kSerializedTypeInteger,
```

```
                kSerializedTypeFloat,
                kSerializedTypeArray,
                kSerializedTypeCommand,
                kSerializedTypeUniqueInteger,
                kSerializedTypeLast
        };

        class Serializable {
        public:
                virtual uint16_t getType() const = 0;
                virtual void serialize(void * const data, size_t &
                    bytePosition) const = 0;
                virtual void deserialize(const void * const data, size_t &
                    bytePosition) = 0;
                virtual size_t getSize() const = 0;
                virtual ~Serializable() {}
        };
}

#endif
```

```
/*
 *  SerializableObjectFactory.h
 *  JNRSync
 *
 *  Created by Joshua Moerman on 6/1/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#ifndef SERIALIZABLEOBJECTFACTORY
#define SERIALIZABLEOBJECTFACTORY

#include <vector>
#include <stdint.h>

namespace JNRSync {

        class Serializable;
        class SerializableObjectFactoryFunction;

        class SerializableObjectFactory {

        public:

                //Returns a created object
                static Serializable* makeObject(uint16_t type);
                static std::vector<SerializableObjectFactoryFunction*>
                    functions;
```

```
        };
}

#endif
```

```
/*
 *  SerializableObjectFactoryFunction.h
 *  JNRSync
 *
 *  Created by Joshua Moerman on 6/1/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#ifndef SERIALIZABLEOBJECTFACTORYFUNCTION_HEADER
#define SERIALIZABLEOBJECTFACTORYFUNCTION_HEADER


#include "Serializable.h"

namespace JNRSync {
        class SerializableObjectFactoryFunction {
        public:
                virtual Serializable* operator()(uint16_t type) const = 0;
        };
}


#endif
```

```
/*
 *  Server.h
 *  JNRSync
 *
 *  Created by Nick Overdijk on 5/10/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#ifndef SERVER_HEADER
#define SERVER_HEADER

#include <list>

#include "MutexLock.h"
#include "ClientDataParser.h"
#include "ListenerDelegateInterface.h"
#include "UniqueInteger.h"
```

```
namespace JNRSync {
        class Client;
        class Listener;

        class Server : virtual public ListenerDelegateInterface {
                public:
                MutexLock clientsLock;

                Server();
                ~Server();

                std::list<Client*> clients;

                pthread_t listenerThread;

                ClientDataParser parser;

                UniqueInteger clientID;

                void start();
                virtual void setListener(Listener *listener);
                virtual void incomingConnection(const int connectFD, const std
                    ::string &ipAdress);
                void addClient(Client& client);
                void removeClient(Client& client);
        };
}

#endif
```

```
/*
 *  UniqueInteger.h
 *  JNRSync
 *
 *  Created by Joshua Moerman on 14-06-10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#ifndef UNIQUEINTEGER_HEADER
#define UNIQUEINTEGER_HEADER

#include <iostream>
#include <cstdlib>

#include "Integer.h"

namespace JNRSync {

        class UniqueInteger : public Integer {
```

```
        public:
                UniqueInteger (const int &value);

                virtual void deserialize(const void * const data, size_t &
                    bytePosition);

                virtual uint16_t getType() const;
        };

        class UniqueIntegerFactoryFunction : public
            SerializableObjectFactoryFunction {
                virtual Serializable* operator()(uint16_t type) const{
                        if (type != kSerializedTypeUniqueInteger) {
                                return NULL;
                        }
                        return new UniqueInteger(0);
                }
        };
}


#endif
```

```
/*
 *  Array.cpp
 *  JNRSync
 *
 *  Created by Joshua Moerman on 5/31/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
     reserved.
 *
 */

#include "Array.h"
#include <cstring>
#include <iostream>
#include "Serializable.h"
#include "SerializableObjectFactory.h"

namespace JNRSync {
        uint16_t Array::getType() const {
                return kSerializedTypeArray;
        }

        void Array::serialize(void * const data, size_t & bytePosition) const
            {
                unsigned int size = members.size();
                memcpy((void*)((char*)data + bytePosition), &size, sizeof(
                    unsigned int));
                bytePosition += sizeof(unsigned int);
```

```
        for (std::vector<Serializable*>::const_iterator it = members.
            begin(); it != members.end();it++) {
                const Serializable* object = *&*it;
                // type
                size_t bytes = sizeof(uint16_t);
                uint16_t type = object->getType();
                memcpy((void*)((char*)data + bytePosition), &type,
                    bytes);
                bytePosition += bytes;
                // data
                object->serialize(data, bytePosition);
        }
}

void Array::deserialize(void const * const data, size_t &bytePosition)
    {
        unsigned int size = 0;
        memcpy(&size, (void const * const)((const char * const)data +
            bytePosition), sizeof(unsigned int));
        bytePosition += sizeof(unsigned int);

        members.clear();
        members.reserve(size);

        for (unsigned int i = 0; i < size; i++) {
                // reading type
                size_t bytes = sizeof(uint16_t);
                uint16_t type;
                memcpy(&type, (void*)((char*)data + bytePosition),
                    bytes);
                bytePosition += bytes;
                // making object of type
                Serializable* object = SerializableObjectFactory::
                    makeObject(type);
                if (object !=NULL) {
                        object->deserialize(data, bytePosition);
                        members.push_back(object);
                } else {
                        std::cerr << "Could not resolve dataType " <<
                            type <<  " of serializable object" << std
                            ::endl;
                }

        }
}

size_t Array::getSize() const{
        //std::cout << "Elements in array = " << members.size() << std
            ::endl;
        size_t bytes = 0;
        bytes += sizeof(unsigned int);          // counter
```

```
                for (std::vector<Serializable*>::const_iterator it = members.
                    begin(); it != members.end();it++) {
                        const Serializable* object = *&*it;
                        bytes += object->getSize();
                        bytes += sizeof(uint16_t);              // it also
                            serializes the type of the object!
                }
                return bytes;
        }
}
```

```
/*
 *  Buffer.cpp
 *  JNRSync
 *
 *  Created by Joshua Moerman on 5/11/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#include "Buffer.h"

#include <cstdlib>
#include <cstdio>
#include <cstring>

namespace JNRSync {

        Buffer::Buffer(size_t size):
        size(size),
        bytesRead(0)
        {
                buffer = calloc(1, size);
        }

        Buffer::Buffer(const Buffer &copyBuffer):
        size(copyBuffer.getSize()),
        bytesRead(0)
        {
                buffer = calloc(1, copyBuffer.size);
                memmove(buffer, copyBuffer.buffer, copyBuffer.size);
        };


        Buffer::~Buffer(){
                free(buffer);
        }

        void Buffer::print() const{
                printf("Buffer of size %ld\n", bytesRead);
```

```
                for (size_t i = 0; i < bytesRead; i++) {
                        printf("%c", ((char*)buffer)[i]);
                }
        }

        const void * Buffer::getBuffer() const{
                return buffer;
        }

        void * Buffer::getBuffer() {
                const Buffer * constThis = this;
                void * nonConstBuffer = const_cast<void*>(constThis->getBuffer
                    ());
                return nonConstBuffer;
        }

        void Buffer::reserve(size_t size){
                this->size = size;
                buffer = realloc(buffer, this->size);
        }

        void Buffer::clear(){
                memset(buffer, 0, size);
        }

        void Buffer::removeBytesAtBeginning(size_t bytes){
                size -= bytes;
                memmove(buffer, reinterpret_cast<char*>(buffer)+bytes, size);
                buffer = realloc(buffer, size);
        }

        size_t Buffer::getSize() const{
                return size;
        }

        int Buffer::lock(){
                return locky.lock();
        }

        int Buffer::unlock(){
                return locky.unlock();
        }

}
```

```
/*
 *  Client.cpp
 *  JNRSync
 *
 *  Created by Nick Overdijk on 5/10/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#include "Client.h"
#include "NMLog.h"

#include <string>

#include "Connector.h"
#include "Listener.h"

namespace JNRSync {
        //Server side client constructor
        Client::Client(const std::string &ipAdres, const int &
            incomingConnectionFD) :
        incoming(incomingConnectionFD)
        {
        }

        //Client side client constructor
        Client::Client(const std::string &serverIpAdres){
                connect(serverIpAdres);
        }

        void Client::connect(const std::string& ipAdres){
                incoming.connectionFD = Connector::connect(ipAdres);
                incoming.setNonBlock(true);
        }

        void Client::add(Serializable * object){
                objectsToSync.members.push_back(object);
        }

        void Client::sync(bool shouldSend){
                if (shouldSend) {
                        void * data = calloc(sizeof(char), objectsToSync.
                            getSize());
                        size_t offset = 0;

                        objectsToSync.serialize(data, offset);

                        incoming.write(data, offset);

                        free(data);
```

```
                }

                incoming.read(1024*1024);

                Array array;
                size_t offset2 = 0;
                array.deserialize(incoming.incomingBuffer.getBuffer(), offset2
                    );

        }


}
```

```
/*
 *  Connection.cpp
 *  JNRSync
 *
 *  Created by Joshua Moerman on 5/11/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#include "Connection.h"

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <errno.h>

#include <limits>

#include "Informant.h"
#include "PosixUtils.h"

namespace JNRSync {
        Informant informant(Informant::kVerboseLevelNone);

        Connection::Connection() :
                connectionFD(0),
                outgoingBuffer(1024),
                incomingBuffer(1024)
        {

        }

        Connection::Connection(int connectionFD = 0) :
        connectionFD(connectionFD),
```

```
outgoingBuffer(1024),
incomingBuffer(1023)
{
        //set nonblock to true
        //!0 === (int) 1;
        set_nonblock_flag(connectionFD, !0);
}

int Connection::read(size_t bytesToRead){
        if (bytesToRead > incomingBuffer.getSize()) {
                incomingBuffer.reserve(bytesToRead);
        }

        incomingBuffer.clear();

        int readReturn = 0;
        incomingBuffer.lock();
        readReturn = ::read(connectionFD, incomingBuffer.getBuffer(),
            bytesToRead);
        incomingBuffer.unlock();


        if(0 > readReturn){
                informant.print("Could not read from socket: ",
                    Informant::kVerboseLevelUseful);
                switch(errno) {
                        case EAGAIN:
                                informant.print("No data to read from
                                    socket", Informant::
                                    kVerboseLevelDebug);
                        break;

                        default:
                                informant.perror("Connection:: Should
                                    never see this! (Default case
                                    entered)", Informant::
                                    kVerboseLevelDebugUltra);
                        break;
                }

                incomingBuffer.bytesRead = 0;
        } else {
                informant.print("Succesfully read data from socket",
                    Informant::kVerboseLevelRedundant);
                incomingBuffer.bytesRead = readReturn;
        }

        return readReturn;
}

int Connection::write(void* buffer, size_t bytes){
```

```
                return ::write(connectionFD, buffer, bytes);
        }

        int Connection::write(const Buffer& bufferToWrite, size_t bytes){
                return ::write(connectionFD, bufferToWrite.getBuffer(), bytes)
                    ;
        }

        void Connection::printBuffer(){
                incomingBuffer.print();
        }

        void Connection::setNonBlock(const bool nonBlock){
                set_nonblock_flag(connectionFD, nonBlock);
        }


}
```

```
/*
 *  Connector.cpp
 *  JNRSync
 *
 *  Created by Joshua Moerman on 5/25/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#include "Connector.h"

#include <iostream>

#include <cstdlib>
#include <cstdio>
#include <cerrno>
#include <cstring>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string>
using std::string;

#include "NMLog.h"

#include "Globals.h"

namespace JNRSync {
```

```
        int Connector::connect(const string& ipAdres){
                std::cout << "Connector: Trying to connect with: " << ipAdres
                    << std::endl;
                struct sockaddr_in stSockAddr;
                int Res;

                memset(&stSockAddr, 0, sizeof(struct sockaddr_in));

                stSockAddr.sin_family = AF_INET;
                stSockAddr.sin_port = htons(incomingConnectionsPort);
                Res = inet_pton(AF_INET, ipAdres.c_str(), &stSockAddr.sin_addr
                    );


                int SocketFD = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

                if (-1 == SocketFD)
                {
                        perror("cannot create socket");
                        exit(EXIT_FAILURE);
                }

                if (0 > Res)
                {
                        perror("error: first parameter is not a valid address
                            family");
                        close(SocketFD);
                        exit(EXIT_FAILURE);
                }
                else if (0 == Res)
                {
                        perror("char string (second parameter does not contain
                            valid ipaddres)");
                        printf("\n%s\n",ipAdres.c_str());
                        close(SocketFD);
                        exit(EXIT_FAILURE);
                }

                if (-1 == ::connect(SocketFD, (const struct sockaddr *)&
                    stSockAddr, sizeof(struct sockaddr_in)))
                {
                        perror("connect failed");
                        close(SocketFD);
                        exit(EXIT_FAILURE);
                }

                return SocketFD;

        }
}
```

```
/*
 *  Globals.cpp
 *  JNRSync
 *
 *  Created by Nick Overdijk on 5/25/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#include "Globals.h"

namespace JNRSync {
        unsigned int incomingConnectionsPort = 6554;
}
```

```
/*
 *  Informant.cpp
 *  JNRSync
 *
 *  Created by Nick Overdijk on 6/11/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#include "Informant.h"
#include <iostream>
#include <cstdio>

namespace JNRSync {
        Informant::Informant() :
        verboseLevel(0){}
        Informant::Informant(const unsigned int &level) :
        verboseLevel(level) {}

        void Informant::print(const std::string &message, const unsigned int
            levelNeeded) {
                if(verboseLevel >= levelNeeded){
                        std::cout << message << std::endl;
                }
        }

        void Informant::perror(const std::string &message, const unsigned int
            levelNeeded) {
                if(verboseLevel >= levelNeeded){
                        std::perror(message.c_str());
                }
        }
}
```

```
/*
 *  Integer.cpp
 *  JNRSync
 *
 *  Created by Joshua Moerman on 5/31/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#include "Integer.h"
#include "UniqueInteger.h"
#include <cstring>

namespace JNRSync {
#pragma mark -
#pragma mark Constructors
        Integer::Integer(const UniqueInteger &uniqueInteger) :
        value(uniqueInteger.value) {
        }

        Integer::Integer(const int &value) :
        value(value)
        {
        }

        Integer::Integer() :
        value(0)
        {
        }

        int Integer::getValue() const {
                return value;
        }

#pragma mark -
#pragma mark Operators
        //This is the PREfix operator ++Integer
        Integer& Integer::operator++ () {
                ++value;
                return *this;
        }

        Integer& Integer::operator-- () {
                --value;
                return *this;
        }

        Integer& Integer::operator= (const Integer& copy) {
                value = copy.value;
                return *this;
```

```
        }

        Integer& Integer::operator= (const int &copy) {
                value = copy;
                return *this;
        }

        uint16_t Integer::getType() const{
                return kSerializedTypeInteger;
        }

#pragma mark -
#pragma mark Serializable implementation
        void Integer::serialize(void * const data, size_t & bytePosition)
            const{
                size_t bytes = getSize();
                memcpy((void*)((char*)data + bytePosition), &value, bytes);
                bytePosition += bytes;
        }

        void Integer::deserialize(const void * const data, size_t &
            bytePosition){
                size_t bytes = getSize();
                memcpy(&value, (void*)((char*)data + bytePosition), bytes);
                bytePosition += bytes;
        }

        size_t Integer::getSize() const{
                return sizeof(int);
        }

}
```

```
/*
 *  JNRFloat.cpp
 *  JNRSync
 *
 *  Created by Nick Overdijk on 5/10/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
     reserved.
 *
 */

#include "JNRFloat.h"
#include <cstring>

namespace JNRSync {
        uint16_t JNRFloat::getType() const {
                return kSerializedTypeFloat;
        }
```

```
        void JNRFloat::serialize(void * const data, size_t & bytePosition)
            const{
                size_t bytes = getSize();
                memcpy((void*)((char*)data + bytePosition), &value, bytes);
                bytePosition += bytes;
        }

        void JNRFloat::deserialize(const void * const data, size_t &
            bytePosition){
                size_t bytes = getSize();
                memcpy(&value, (void*)((char*)data + bytePosition), bytes);
                bytePosition += bytes;
        }

        size_t JNRFloat::getSize() const{
                return sizeof(float);
        }

}
```

```
/*
 *  JNRSync.cpp
 *  JNRSync
 *
 *  Created by Nick Overdijk on 6/10/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#include "JNRSync.h"

#include "Functor.h"

namespace JNRSync {

        Functor * uniqueIntegerReceivedFunction;

        void init() {
                SerializableObjectFactory::functions.push_back(new
                    IntegerFactoryFunction);
                SerializableObjectFactory::functions.push_back(new
                    FloatFactoryFunction);
                SerializableObjectFactory::functions.push_back(new
                    ArrayFactoryFunction);
                SerializableObjectFactory::functions.push_back(new
                    UniqueIntegerFactoryFunction);
        }

        void uniqueIntegerReceived(const UniqueInteger &uniqueInteger){
                (*uniqueIntegerReceivedFunction)(&uniqueInteger);
```

```
        }

}
```

```
/*
 *  Listener.cpp
 *  JNRSync
 *
 *  Created by Joshua Moerman on 5/11/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#include "Listener.h"
#include "Server.h"
#include "ListenerDelegateInterface.h"

#include <iostream>

//Should be in namespace POSIX
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netdb.h>

#include <cstdio>
#include <cstdlib>
#include <cstring>

using std::cout;
using std::cerr;
using std::endl;

#include "Globals.h"
#include "NMLog.h"

namespace JNRSync {
        Listener::Listener(){
                struct sockaddr_in sockAddr;
                memset((void*)&sockAddr, 0, sizeof(sockaddr_in));
                sockAddr.sin_family = AF_INET;
                sockAddr.sin_port = htons(port = incomingConnectionsPort);
                sockAddr.sin_addr.s_addr = INADDR_ANY;

                socketFD = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

                if(-1 == socketFD){
                        cerr << "Listener: Failed to create socket" << endl;
```

```
                exit(EXIT_FAILURE);
        }

        int bindResult = bind(socketFD, (const struct sockaddr *) &
            sockAddr, sizeof(struct sockaddr_in));

        if(-1 == bindResult){
                cerr << "Listener: Failed to bind socket" << endl;
                close(socketFD);
                exit(EXIT_FAILURE);
        }

        int listenResult = ::listen(socketFD, 10);
        if(-1 == listenResult){
                cerr << "Listener: Failed to listen" << endl;
                close(socketFD);
                exit(EXIT_FAILURE);
        }

        cout << "Listener: Succesfully created listening socket" <<
            endl;
}

void* Listener::listen(void* voidDelegate){
        ListenerDelegateInterface* delegate = reinterpret_cast<
            ListenerDelegateInterface*>(voidDelegate);
        Listener self;

        delegate->setListener(&self);

        do {
                struct sockaddr addr;
                socklen_t sizeOfSocket = sizeof(struct sockaddr);
                int ConnectFD = accept(self.socketFD, &addr, &
                    sizeOfSocket);

                if(0 > ConnectFD)
                {
                        perror("error accept failed");
                        close(self.socketFD);
                        exit(EXIT_FAILURE);
                }

                printf("Connection has been made!\n");

                char* ipAdress = (char*)calloc(1, INET_ADDRSTRLEN);
                if(getnameinfo(&addr, sizeof(struct sockaddr),
                    ipAdress, INET_ADDRSTRLEN, NULL, 0, NI_NUMERICHOST
                    )){
                        perror("Could not resolve hostname");
                        close(self.socketFD);
```

```
                                exit(EXIT_FAILURE);
                        }

//                      if(NULL == inet_ntop(AF_INET, &addr, ipAdress,
    INET_ADDRSTRLEN)){
//                              perror("Converting IP to string failed");
//                              exit(EXIT_FAILURE);
//                      }

                        std::cout << "Established connection to ipAdress(" <<
                            ipAdress << ")" << std::endl;
                        std::string ipAdressString(ipAdress);
                        delegate->incomingConnection(ConnectFD, ipAdressString
                            );

                } while (true);

                return NULL;
        }


}
```

```
/*
 *  MutexLock.cpp
 *  JNRSync
 *
 *  Created by Nick Overdijk on 5/11/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
     reserved.
 *
 */

#include "MutexLock.h"

#include <pthread.h>
#include <errno.h>

#include "NMLog.h"

namespace JNRSync {
        MutexLock::MutexLock(){
                int result = pthread_mutex_init(&locky, 0);
                handleMutexInitError(result);
        }

        MutexLock::~MutexLock(){
                int result = pthread_mutex_destroy(&locky);
                handleMutexDestroyError(result);
        }
```

```
        int MutexLock::lock(){
                return pthread_mutex_lock(&locky);
        }

        int MutexLock::unlock(){
                return pthread_mutex_unlock(&locky);
        }

        void MutexLock::handleMutexInitError(int status){
                switch (status) {
                        case 0:
                                NMLog("Mutex succesfully made");
                                break;
                        case EAGAIN:
                                NMLog("Not enough resources");
                                break;
                        case ENOMEM:
                                NMLog("No memory");
                                break;
                        case EPERM:
                                NMLog("No permission");
                                break;
                        case EBUSY:
                                NMLog("Is already made");
                                break;
                        case EINVAL:
                                NMLog("Invalid attributes parameter");
                                break;
                        default:
                                NMLog("Unknown error: %x", status);
                                break;
                }
        }

        void MutexLock::handleMutexDestroyError(int status){
                switch (status) {
                        case 0:
                                NMLog("Mutex succesfully destroyed");
                                break;
                        case EBUSY:
                                NMLog("Is locked");
                                break;
                        case EINVAL:
                                NMLog("Invalid attributes parameter");
                                break;
                        default:
                                NMLog("Unknown error: %x", status);
                                break;
                }
        }
}
```

```
/*
 *   PosixUtils.c
 *   JNRSync
 *
 *   Created by Nick Overdijk on 6/10/10.
 *   Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
     reserved.
 *
 */


#include "PosixUtils.h"
#include <fcntl.h>

int set_nonblock_flag (int filedesc, const bool value){
        int oldflags = fcntl (filedesc, F_GETFL, 0);

        /* If reading the flags failed, return error indication now. */
        if (oldflags == -1)
                return -1;

        /* Set just the flag we want to set. */
        if (value) {
                oldflags |= O_NONBLOCK;
        } else {
                oldflags &= ~O_NONBLOCK;
        }

        /* Store modified flag word in the descriptor. */
        return fcntl (filedesc, F_SETFL, oldflags);
}
```

```
/*
 *   SerializableObjectFactory.cpp
 *   JNRSync
 *
 *   Created by Joshua Moerman on 6/1/10.
 *   Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
     reserved.
 *
 */

#include "SerializableObjectFactory.h"

#include "SerializableObjectFactoryFunction.h"
#include "Serializable.h"

namespace JNRSync {
        std::vector<SerializableObjectFactoryFunction*>
            SerializableObjectFactory::functions;
```

```
        Serializable* SerializableObjectFactory::makeObject(uint16_t type){
                Serializable* object = NULL;

                //In this loop, we try to find what object we just received
                //There's a list of functions we call with the object of an
                    unknown type
                //If one of the functions "recognizes" that object, it
                    deserializes it
                //and returns the constructed type
                //TODO: Find out how we can determine the type at runtime
                //              Possible solutions:
                //              - dynamic_cast<>
                //              + Checking the enum value
                for(std::vector<SerializableObjectFactoryFunction*>::
                    const_iterator it = functions.begin();
                        it != functions.end();
                        it++){

                        SerializableObjectFactoryFunction& function = **it;
                        object = function(type);
                        if (object != NULL) { //If the function created the
                            object
                                break;
                        }
                }

                //Return the, now probably created, object
                return object;
        }

}
```

```
/*
 *  Server.cpp
 *  JNRSync
 *
 *  Created by Nick Overdijk on 5/10/10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#include "Server.h"
#include "NMLog.h"

#include <iostream>

#include <pthread.h>
#include <errno.h>
#include <list>
```

```
#include "Client.h"
#include "Listener.h"
#include "Connector.h"

namespace JNRSync {

        Server::Server():
        parser(this),
        clientID(0)
        {
        }

        Server::~Server(){
        }

        void Server::start(){
                pthread_create(&listenerThread, 0, &Listener::listen, this);

                do {
                        clientsLock.lock();

                        for (std::list<Client*>::iterator it = clients.begin()
                            ; it != clients.end(); it++) {
                                Client &readClient = **it;

                                //parser.parse(client.incoming.buffer);

                                // de server is momenteel slechts een
                                    doorstuurserver
                                // dus eerst lezen, zodat de buffer opgeslagen
                                    is in de connection
                                // eerst maar even 1 MB lezen
                                size_t bytesRead = readClient.incoming.read(
                                    sizeof(char)*1024*1024);

                                if ((int)bytesRead <= 0) {
                                        //Als er niets is gelezen hoeven we
                                            ook niets te sturen
                                        continue;
                                }

                                // naar elke client sturen (ongelijk zichzelf,
                                    neem ik aan)
                                for (std::list<Client*>::iterator it2 =
                                    clients.begin(); it2 != clients.end(); it2
                                    ++) {
                                        if (*it == *it2) {
                                                continue;
                                        }
                                        Client &writeClient = **it2;
```

```
                                        // aantal bytes versturen
                                        writeClient.incoming.write(readClient.
                                            incoming.incomingBuffer, bytesRead
                                            );

                                }
                        }

                        clientsLock.unlock();
                } while (true);
        }


        void Server::setListener(Listener *listener){
        }

        void Server::incomingConnection(const int connectFD, const std::string
            &ipAdress){
                std::cout << "Received connection from(" <<  ipAdress << ")"
                    << std::endl;
                clientsLock.lock();
                NMLog("Pushing back new client!");
                Client * newClient = new Client(ipAdress, connectFD);
                clients.push_back(newClient);
                clientsLock.unlock();

                // sending the unique ID of the client
                Array container;
                container.members.push_back(&clientID);
                void * buffer = calloc(1, container.getSize());
                size_t numberOfBytes = 0;
                container.serialize(buffer, numberOfBytes);
                newClient->incoming.write(buffer, numberOfBytes);

                free(buffer);

                ++clientID;
        }
}
```

```
/*
 *   ServerMain.cpp
 *   JNRSync
 *
 *   Created by Nick Overdijk on 5/25/10.
 *   Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
     reserved.
 *
 */

#include <iostream>
```

```
using std::cout;
using std::endl;

#include "JNRSync.h"


#include "Server.h"

int main(int argc, char *argv[]){
        cout << "Server launched!" << endl;

        JNRSync::init();
        JNRSync::Server server;
        server.start();

        return 0;
}
```

```
/*
 *  UniqueInteger.cpp
 *  JNRSync
 *
 *  Created by Joshua Moerman on 14-06-10.
 *  Copyright 2010 Nick Overdijk, Joshua Moerman, Rik Harink. All rights
    reserved.
 *
 */

#include "UniqueInteger.h"
#include "JNRSync.h"
#include <cstring>

namespace JNRSync {
        UniqueInteger::UniqueInteger (int const &value) :
        Integer(value) {
        }

        void UniqueInteger::deserialize(const void * const data, size_t &
            bytePosition){
                        Integer::deserialize(data, bytePosition);
                JNRSync::uniqueIntegerReceived(*this);
        }

        uint16_t UniqueInteger::getType() const{
                return kSerializedTypeUniqueInteger;
        }

}
```