

Telekommunikation
Liste V
Projekt: UMTS-Simulator
Erstellung des Simulationstools

Ausarbeitung

26.01.2004

Sascha Sadikni

532654

<http://www.informatik.fh-wiesbaden.de/~ssadi001/tk/>

Inhalt

I. Einleitung

II. Implementierungs – Logik

- 1. Aufgabenstellung***
- 2. Versuchsaufbau***
- 3. Bridge – Logik***
- 4. 'Quality Of Service' Bewertungsgrößen***
- 5. Wahrscheinlichkeits-Verteilungen***
- 6. Simulator – Logik***

III. Implementierungs – Details

- 1. Native – C (libpcap – read / libnet – write)***
- 2. JavaNativeInterface***
- 3. Überblick des Gesamtprojekts***
- 4. Übersetzung und Voraussetzungen des Programms***

IV. Zusammenfassung

V. Quellen

I. Einleitung

Ziel des Projektes sollte sein, einen Simulator zu erstellen, der das Verhalten von UDP-Datenverkehr so simuliert, dass sich UMTS-Anwendungen mit ihm im Labor realitätsnah testen lassen.

Dazu gehörte die Definition der simulierten Umgebungsparameter, die Durchführung von erforderlichen Messungen mit realer UMTS-Ausrüstung sowie die Erstellung der Simulationstools. Die Messungen und das Programmieren sollten in getrennten Projekten bearbeitet werden.

Innerhalb dieser Ausarbeitung wird kurz auf die Umgebungsparameter hingewiesen und die Erstellung des Simulationstools besprochen.

Zunächst folgt ein genauerer Blick auf die Aufgabenstellung (II.1.) und den Versuchsaufbau (II.2.). Hierbei zeigt sich, dass man das Problem als die Implementierung einer Bridge mit zusätzlicher Filter bzw. Simulations – Logik ansehen kann.

Ein Rechner fungiert als Bridge, wenn er alle an einem Netzwerk-Interface ankommenden Datenpakete ohne Änderung auf einem anderem Netzwerk-Interface wieder raus schreibt.

Danach wird gezeigt, wie man die Aufgabe als Bridge implementieren kann(II.3.), noch ohne Rücksicht auf eventuelle Störungen bzw. Umgebungsparameter.

In Kapitel II.4. '*Quality Of Service*' *Bewertungsgrößen* werden die Umgebungsparameter aufgeführt, die dann simuliert werden sollen.

Anschliessend werden zwei mögliche Verteilungen diskutiert (II.5.), mit deren Hilfe man aus den Messwerten Simulationswerte bekommt.

Abschliessend wird erklärt, an welchen Punkten der Bridge die Simulations – Logik eingefügt wurde.

In Kapitel III wird dann noch auf Besonderheiten der Implementierung hingewiesen.

Es werden zunächst die C-Bibliotheken 'ibpcap' und 'blnet' vorgestellt, mit deren Hilfe man recht leicht von einem Netzwerk – Interface lesen bzw. auf einem Netzwerk – Interface schreiben kann.

Da das Projekt im Wesentlichen in Java geschrieben ist, war es notwendig für den direkten Zugriff auf die Netzwerkkarten mit dem `JavaNativeInterface` (JNI) zu arbeiten. In Kapitel III.2. wird darauf näher eingegangen.

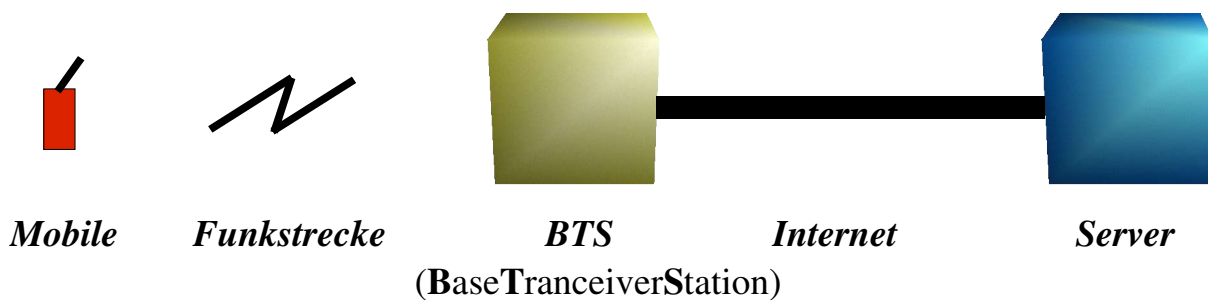
Schliesslich folgen noch eine Gesamtübersicht des Programms sowie eine Anleitung zum Übersetzen und Ausführen.

Eine Auswahl an benutzten Quellen findet sich in Kapitel V.

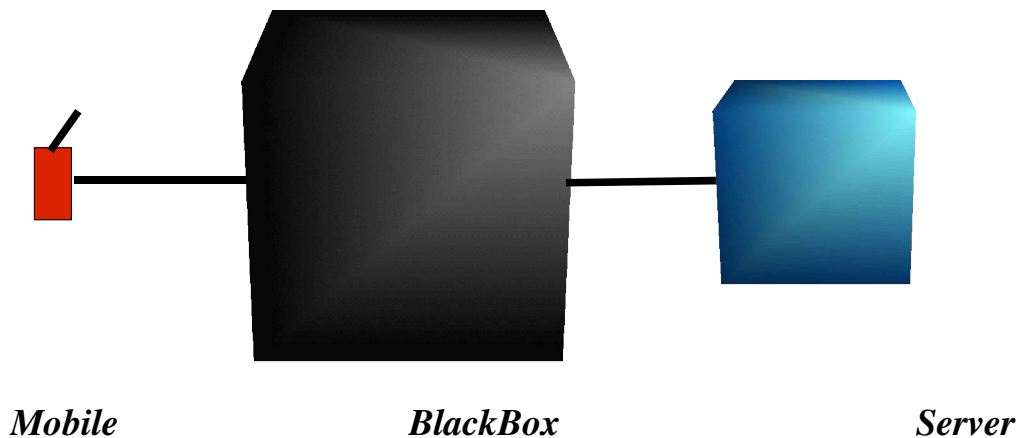
II. Implementierungs – Logik

1. Aufgabenstellung

An einer UMTS-Datenverbindung sind mehrere Komponenten beteiligt. Die Daten kommen von einem normalen *Server*, welcher an das *Internet* angebunden ist. Die *Base-Tranceiver-Station* ist auch noch via Kabel ans Internet angeschlossen und sendet die Daten nun über die *Funkstrecke* zum *Mobilgerät*.



Uns interessiert nur, welche Daten vom Server wann zum Mobilgerät gelangen. Man kann also die *Funkstrecke* und das *Internet* inklusive *Base-Tranceiver-Station* komplett als *BlackBox* ansehen, die simuliert werden soll.



Welche Umgebungsparameter hierbei eine Rolle spielen, war Teil des Mess-Projektes. Dabei stellte sich heraus, dass man sich bei der Simulation auf 3 Grössen beschränken kann:

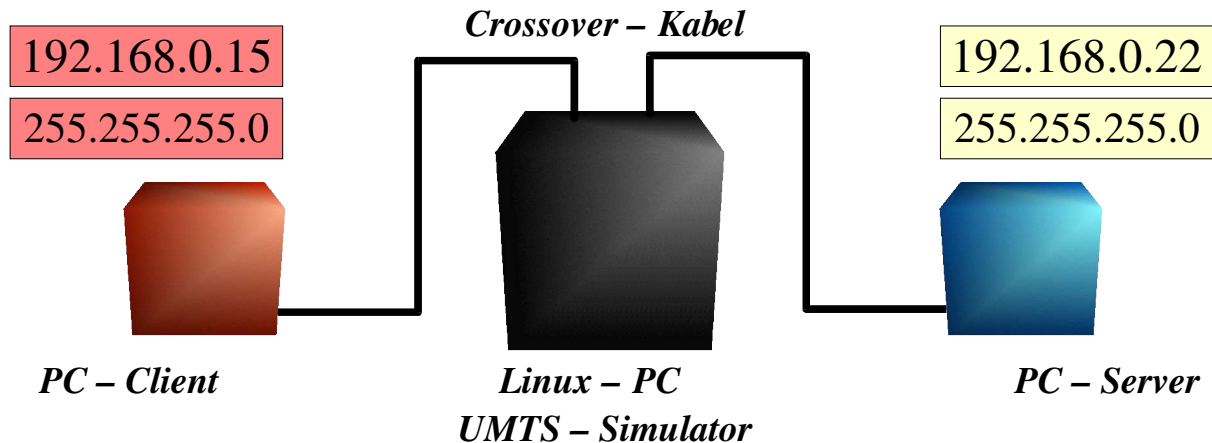
1. BlockLossRate
2. BlockDelaySpread
3. BlockOutOfSequence - Weite

Diese Werte werden in Kapitel II.4. näher erläutert.

2. Versuchsaufbau

Zur Durchführung der Simulation werden also drei Komponenten benötigt.

Zum einen ein Rechner auf dem der UMTS-Simulator läuft und zwei weitere Rechner, welche als Client bzw. Server fungieren können.



Gewählt wurden hier 3 PC' ,swobei Client und Server-PC lediglich über jeweils eine *Ethernet-Netzwerkkarte* verfügen mussten. Der eigentliche UMTS-Simulator dagegen benötigte zwei *Ethernet-Netzwerkkarten*.

Client und Server-PC ist das **Betriebssystem** freigestellt, es muss lediglich *netzwerkfähig* sein und eine *JavaRuntimeEnvironment* dafür existieren. Letzteres wird nur benötigt, wenn man die optionalen Test-Programme nutzen möchte.

Der Simulator-PC muss aufgrund von benötigten C-Bibliotheken mit *Linux* laufen.

Getestet wurden *SuSE Linux* in den Versionen 7.3 sowie 9.0.

Zusätzlich müssen hier noch die C-Bibliotheken *LIBPCAP* (0.7.2) und *LIBNET* (1.1.x) installiert sein.

Auf den Client/Server Rechnern müssen die Netzwerkadressen im gleichen Subnet konfiguriert sein, z. B. könnte die *SUBNET-Maske* 255.255.255.0 und die *IP-Adressen* 192.168.0.15 sowie 192.168.0.22 vergeben werden (so wie in der Abbildung angegeben).

Der Simulator-PC wurde mit der *gleichen SUBNET-Maske* sowie den *Adressen* 192.168.0.11 sowie 192.168.1.11 getestet. Da das Simulatorprogramm aber direkt auf die Netzwerkschnittstellen zugreift, sollte es nicht notwendig sein diese zu konfigurieren bzw. die Konfiguration sollte keine Rolle spielen.

Im Testbetrieb wurden die beiden Client/Server-Rechner normalerweise mit Crossover-Netzwerkkabeln jeweils mit dem Simulator-Rechner verbunden. Es sollte aber auch möglich sein, den Simulator an zwei Switches anzuschliessen. Nur die beiden Client/Server-Rechner sollten nicht zusammen an einem Switch hängen.

3. Bridge – Logik

Zunächst galt es die Kern – Logik zu implementieren, noch ganz ohne die Simulation der Bewertungsgrößen. Client und Server sollen sich miteinander unterhalten können, obwohl sie nicht direkt verbunden sind. Alle Datenpakete gehen zunächst an den Simulator-Rechner. Ohne weitere Software reagiert dieser nicht auf diese Pakete, da sie ja nicht an ihn gerichtet sind. Wir brauchen also ein Programm, welches die Datenpakete vermittelt, d.h. von einer Netzwerkkarte auf die andere kopiert. Hier wurde dies als Bridge implementiert.

Wir können dazu recht einfach auf dem Link-Layer arbeiten. Hier müssen zunächst nur alle an einer Schnittstelle ankommenden Pakete auf die andere kopiert werden.

Hierzu schauen wir uns als Beispiel das einfache Ping-Kommando an:
ping 192.168.0.15

Das Kommando wird auf dem Server (192.168.0.22) ausgeführt. Wir schauen uns nur das zunächst aktive AddressResolutionProtocol an.

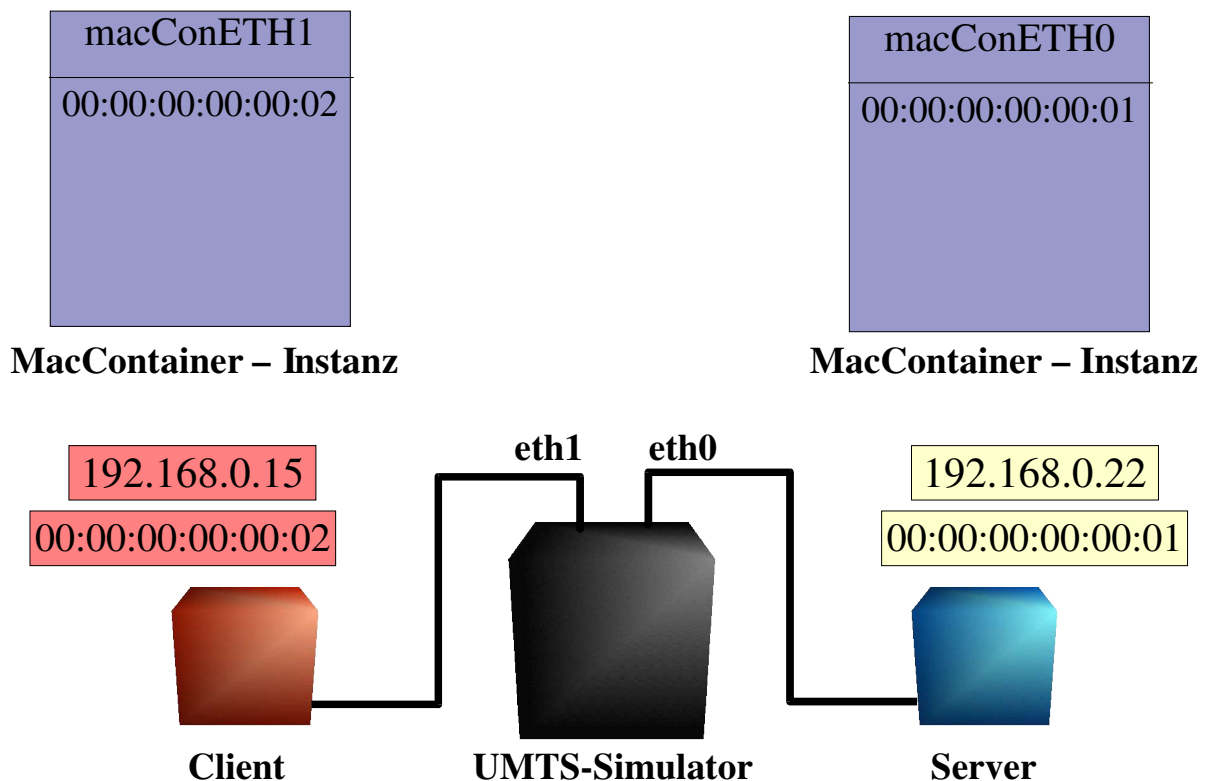
Zunächst erfolgt die Anfrage an alle – Wer hat die IP-Adresse 192.168.0.15? :

ARP dest: FF:FF:FF:FF:FF:FF src: 00:00:00:00:00:01

Der Empfänger mit der IP-Adresse 192.168.0.15 sollte antworten:

REARP dest: 00:00:00:00:00:01 src: 00:00:00:00:00:02

Damit kennt der Sender die MAC-Adresse des Empfängers und schickt seine Pakete von nun an an diese MAC-Adresse.



Wenn man aber nur jedes ankommende Paket einfach kopiert, bekommt man bald ein Problem. Jedes Paket welches von 'eth0' gelesen und auf 'eth1' geschrieben wird, wird auch wieder von 'eth1' gelesen und wiederum auf 'eth0' geschrieben usw. natürlich.

Um das zu verhindern sieht man auf der oberen Abbildung den MacContainer. Er speichert MAC-Adressen. Ein Paket wird nur geschrieben, wenn die Source-MAC-Adresse dieses Pakets nicht im MacContainer auftaucht, welcher dem aktuellen Schreib-Interface zugeordnet ist. Das Schreib-Interface ist die Netzwerkkarte auf der das Paket geschrieben werden sollte. Falls das Paket geschrieben wird, kommt seine Source-MAC-Adresse vorher in den MacContainer, welcher dem Lese-Interface zugeordnet ist.

NIReaderBridge – Thread:

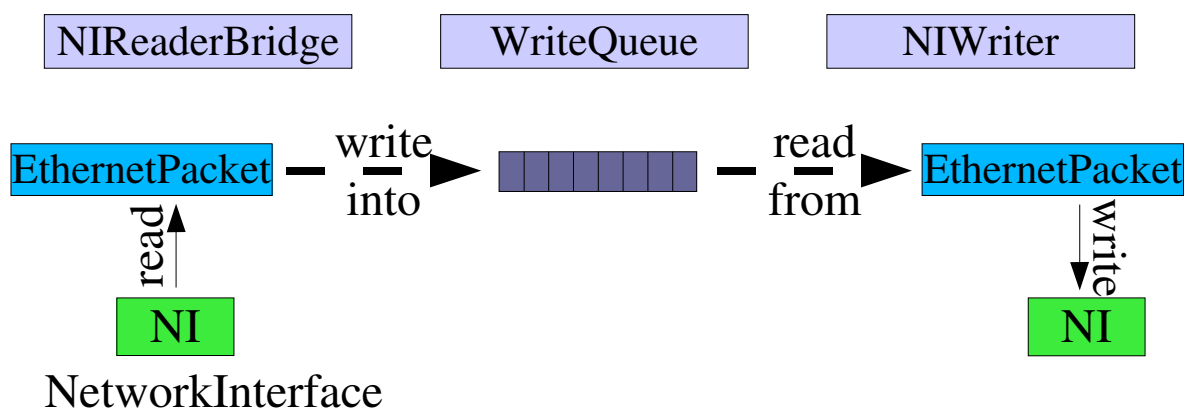
```
// macConETH1 and macConETH0
// should be initialized
while(true) {
    packet = readPacket();
    if(!macConWrite.containsMAC(packet.srcMAC))
    {
        macConRead.addMAC(packet.srcMAC);
        queueWrite.addPacket(packet);
    }
}
```

Implementiert ist die Bridge – Logik mit drei Klassen: NIReaderBridge, WriteQueue und NIWriter.

NIReaderBridge ist als Thread implementiert. Für jede Netzwerkkarte existiert eine Instanz des Threads. Es wird jeweils ein Ethernet – Packet von einer Netzwerkschnittstelle gelesen und anschliessend in die jeweilige WriteQueue platziert.

Von der WriteQueue existiert für jede Netzwerkschnittstelle eine Instanz. Sie bietet get- und set-Methoden an, um Pakete einzustellen bzw. herauszuholen. Normalerweise funktioniert dies nach dem FIFO-Prinzip. Die get-Methode liefert immer das älteste Exemplar der Schlange. Die set-Methode packt ein neues Paket an das Ende der Schlange.

NIWriter ist auch als Thread implementiert mit jeweils einer Instanz pro Netzwerkkarte. Es wird jeweils ein Paket aus der Schlange geholt und anschliessend auf die Netzwerkkarte geschrieben.



Man hätte die Paketvermittlung auch als Gateway implementieren können. Aber das hätte einige Nachteile mit sich gebracht:

- Der Simulatorrechner hätte explizit konfiguriert werden müssen.
 - > Mit 2 IP-Adressen in verschiedenen Subnetzen.
- Auch die Clients hätten dementsprechend in unterschiedlichen Subnetzen liegen müssen. Zusätzlich hätte der Gateway konfiguriert werden müssen.
- Die Implementierung des Simulators wäre komplexer.
 - > Man hätte die Destination-MAC-Adresse immer ändern müssen, für jedes ankommende Paket.

Eine einfache Gateway-Version findet sich in ***NIRreaderGateway***, diese wurde aber nicht weiter getestet und enthält hardcoded MAC-Adressen. Trotzdem lässt dieser sich noch über eine Kommandozeilen-Option (-gw) im Simulator aktivieren.

4. 'Quality Of Service' Bewertungsgrößen

Man kann die zu simulierenden Größen folgendermassen beschreiben:

1. Blockverlustrate: Block Loss Rate – BLR

Definition laut Aufgabenstellung:

Beschreibt den Prozentsatz von verlorengegangenen UDP-Blocks zur Gesamtzahl gesendeter Blocks. Die Größe ist auch von der Streamrate (primär in den Kanal injizierte Datenrate) abhängig.

-> *Prozentzahl der verlorengegangenen Pakete.*

2. Block Delay Spread – BDS

Definition laut Aufgabenstellung:

Erfasst man die Ankunftszeiten empfangener Blöcke, hängt die Zeit zwischen Blöcken (Blockstart zu Blockstart oder Blockende zu Blockende) zum einen natürlicherweise von der Blockgröße ab, zum anderen von Verzugszeiten im Transfer. BDS ist eine Größe, die die Delayverteilung charakterisiert, die also mehrere Komponenten besitzt. Nimmt man eine Verteilung an, die einer mathematisch definierbaren Grundform entspricht, lässt sich BDS durch deren Koeffizienten bestimmen.

Plausibel wäre z.B. eine von einem Minimalwert ausgehende, zu größeren Zeiten exponentiell abfallende Verteilungskurve, so dass BDS als Minimaldelay und Exponentialkoeffizient des Abfalls beschreibbar wäre.

Anzustreben ist für den hier benötigten Anwendungsfall im weiteren Definitionsprozess eine von der konkreten Datenrate und Blockgröße unabhängige, d.h. normierte Beschreibungsform.

-> *Abstand von zwei aufeinanderfolgenden Paketen (beim Empfänger) in Millisekunden.*

3. Block Out of Sequence-Weite – BOSW

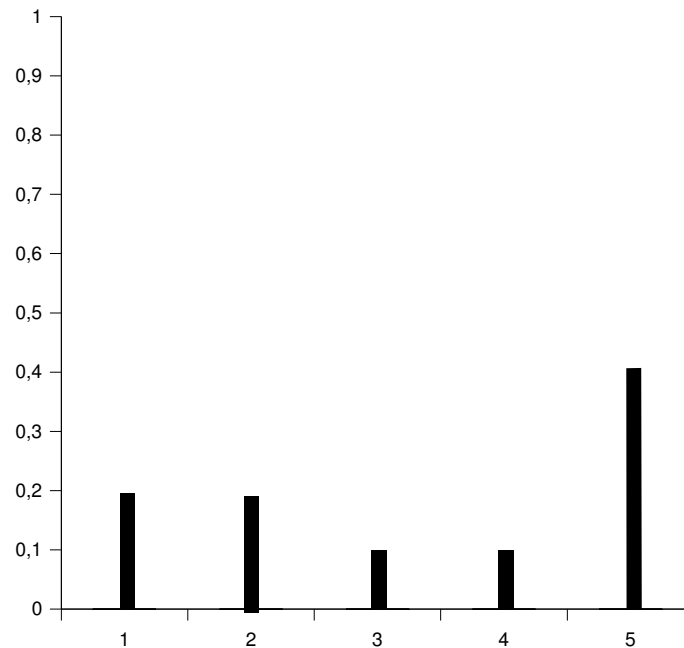
Definition laut Aufgabenstellung:

Aufgrund wechselnden Routings können bei UDP Blocks, die in Sequenz injiziert wurden, außer Sequenz empfangen werden. Nimmt man an, dass jeder Block eine blockweise inkrementierte Sequenznummer enthält, ist BOSW als (1 – max. Differenz von Sequenznummern im Verlauf) definiert. Werden alle Blöcke in Sequenz empfangen, ist BOSW also 0. Die konkrete Umsetzung muss berücksichtigen, dass bei Blockverlust automatisch BOSW-Werte größer Null entstehen; dieser Effekt muss ggf. auskorrigiert werden.

-> *Pakete kommen in falscher Reihenfolge an.*

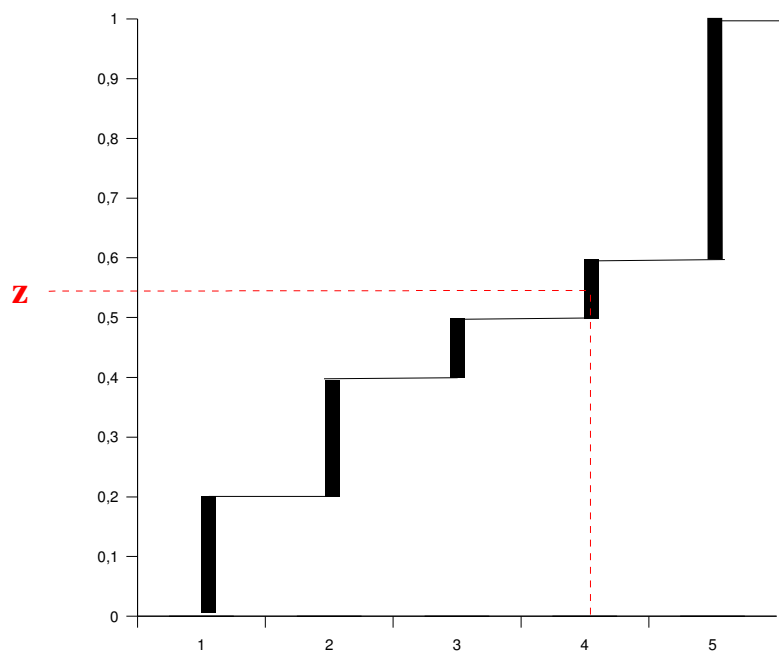
5. Wahrscheinlichkeits-Verteilungen

Diskrete Verteilung

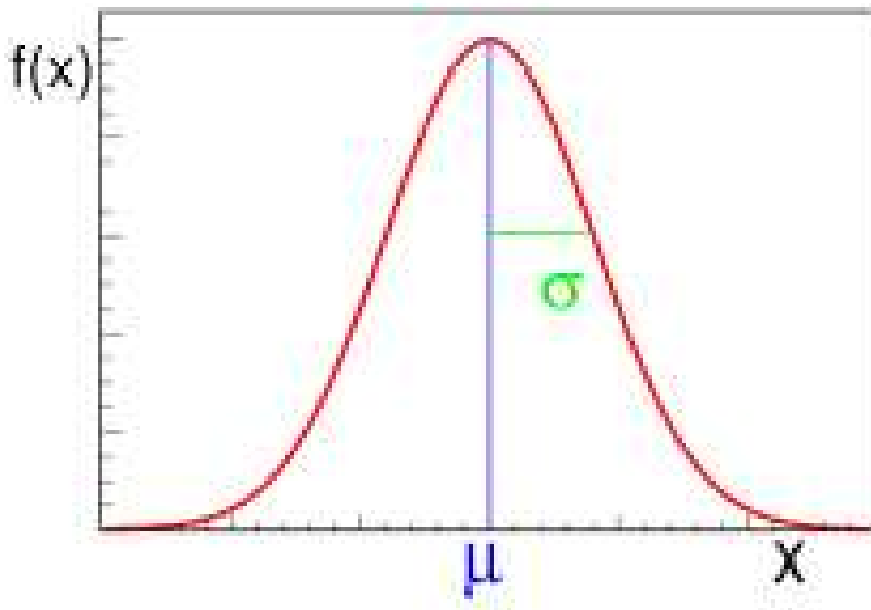


Die obere Abbildung zeigt eine diskrete Verteilung, wie man sie erhält, wenn man die Häufigkeitswerte von Messwerten bildet. Man sieht das die Werte 1 und 2 jeweils in 20% der Fälle, die Werte 3 und 4 in jeweils 10% der Fälle auftraten und der Wert 5 in 40% aller Fälle auftrat.

Diese diskreten Werte kann man nun wie in der unteren Abbildung gezeigt auftragen. Man erhält eine unstetige Funktion mit Funktionswerten zwischen 0 und 1. Wenn man sich eine gleichverteilte Zufallszahl z besorgt, kann man den zugehörigen Messwert an der x -Achse ablesen.



Gauss Verteilung



In der Abbildung ist eine Standard-Normal-Verteilung zu sehen.

Die Normal-Verteilung wird charakterisiert durch die folgenden Kenngrößen:

- Mittelwert: μ
- Standardabweichung: σ

Dabei gilt $\sigma^2 = \Sigma [(x_i - \mu)^2] / n$.

x_i - sind die einzelnen Messwerte;

n - ist die Anzahl der Messwerte;

Eine **Standard** – Normalverteilung liegt vor für:

- $\mu = 0$
- $\sigma = 1$

Wenn man eine **standard-normalverteilte Zufallszahl** z hat, kann man diese mit Hilfe von **Mittelwert** und **Standardabweichung** leicht in eine beliebige Normalverteilung umrechnen.

Die Zufallszahl z bekommt man in Java z.B. mit Hilfe der Klasse **Random**:

```
 $z = \text{Random.nextGaussian}();$ 
```

```
 $x = z * \sigma + \mu$ 
```

6. Simulator – Logik

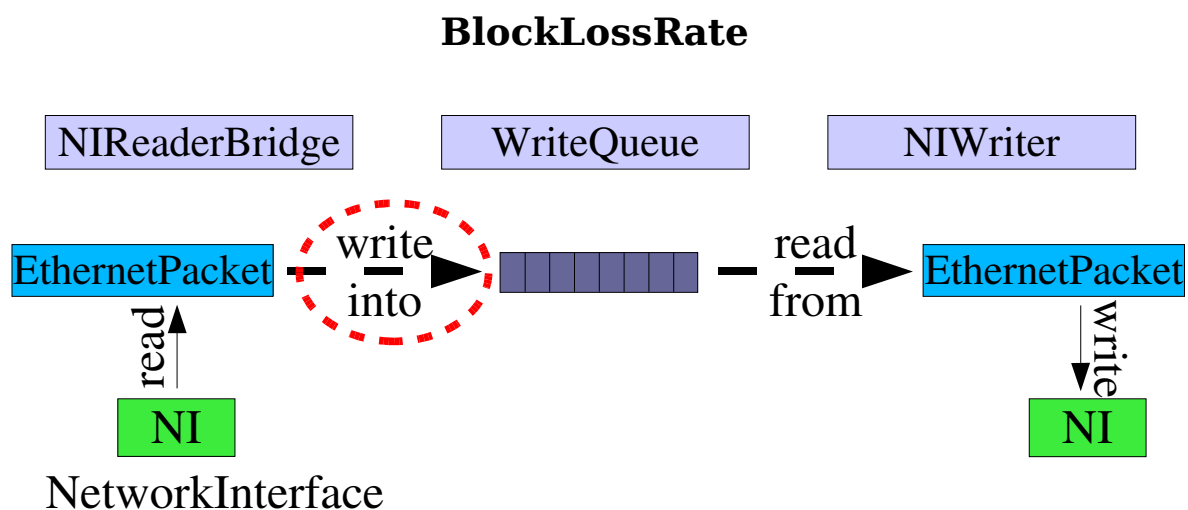
Simuliert werden die drei Grössen *BlockLossRate (BLR)*, *BlockDelaySpread (BDS)* und *BlockOutOfSequence (OS)*. Zunächst wird für jede Grösse gezeigt, an welcher Stelle sie im Bridge-Code auftauchen. Anschliessend wird der *Filter* beschrieben, dieser berechnet aus den Messwerten und mit Hilfe einer Verteilung den jeweiligen aktuellen Wert.

Die Klasse *Filter* bietet dafür die drei Methoden *blrFilter()*, *bdsFilter()* und *osFilter()*.

Die Messwerte stellt die Klasse *ConfigValues* zur Verfügung. Diese werden beim Programmstart aus den von dem Mess-Projekt-Team erstellten *.usim*-Dateien ausgelesen.

Die Filter-Methoden benötigen ausschliesslich einen *Key* um zu wissen, welche Messwerte benutzt werden sollen. Dieser *Key* ist der Name des ReadDevices, damit wird also bestimmt, welche Richtung gerade simuliert wird.

Es existieren zwei Instanzen der Klasse *ConfigValues* die jeweils über diesen *Key* verfügbar sind. Da die Klasse *NIRedReaderBridge* als einzige den Namen des ReadDevices (*key*) kennt, werden alle Filter schon in dieser Klasse berechnet.



Ob ein Paket verschickt wird oder nicht wird in der Klasse *NIRedReaderBridge* entschieden. Nachdem das *EthernetPacket* eingelesen wurde, wird an Hand des Rückgabewertes des *BLR-Filters* entschieden, ob das Paket in die *WriteQueue* eingestellt wird oder nicht.

BLR Filter:

```
if(Filter.blrFilter() == true) {  
    writeIntoQueue(EthernetPacket);  
}
```

Die Filter-Methode *blrFilter()* gibt entweder true oder false zurück. Sie holt sich eine gleichverteilte Zufallszahl *randBLR* und den aktuell gültigen BLR-Wert in Prozent *pBLR*. Falls *randBLR* grösser als *pBLR/100* ist, gibt sie true zurück:

Filter.blrFilter(key):

```
ConfigValues cv=ConfigValues.getInstance(key);
double randBLR =randomBLR.nextDouble();

int pBLR = cv.getActBLR();

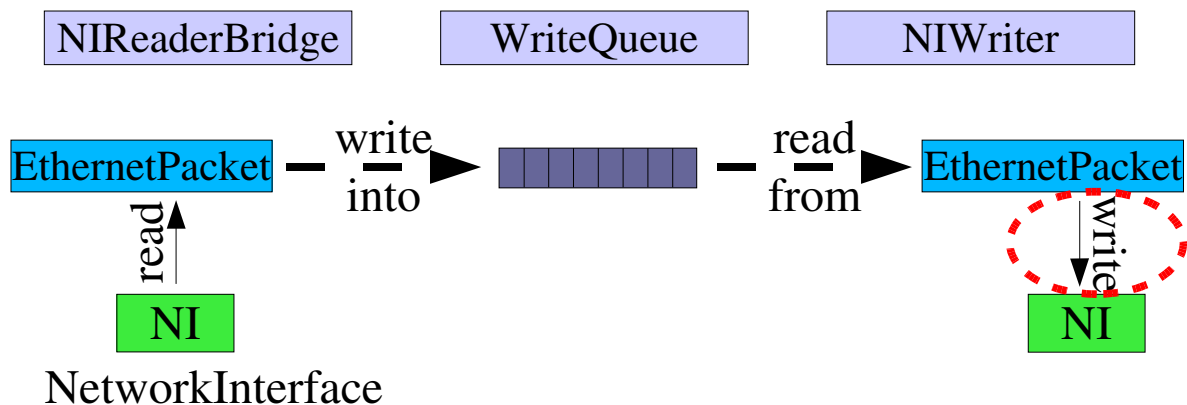
// *** if pBLR==-1(error, cv not init)
// *** => return true (no filter)
if(randBLR>(((double)pBLR)/100.0))return true;
return false;
```

Die gesamte Logik der *NIReaderBridge*-Klasse (inkl. Bridge und Simulator – Logik):

NIReaderBridge – Thread:

```
// macConETH1 and macConETH0
// should be initialized
while(true) {
    packet = ni.readPacket();
    if(!macConWrite.containsMAC(packet.srcMAC) {
        macConRead.addMAC(packet.srcMAC);
        if(Filter.blrFilter(devRead)) {
            queueWrite.addPacket(packet,
                Filter.osFilter(devRead),
                Filter.bdsFilter(devRead));
        }
    }
}
```

BlockDelaySpread



Der *NIWriter* holt sich das *EthernetPacket* aus der *WriteQueue* und legt sich kurz vor dem schreiben des Pakets für die im BDS-Filter berechnete Zeit schlafen. Anschliessend wird das Paket gesendet.

BDS Filter:

```
readFromQueue();
sleep( Filter.bdsFilter() );
writeOnNI(EthernetPacket);
```

Die Filter-Methode *bdsFilter()* berechnet den aktuellen Delay. Sie berechnet eine gaussverteilte Zufallszahl x mit Hilfe der Messwerte *bdsAvg* und *bdsSigma*. Da sich bei Tests mit den konkreten Messwerten gezeigt hat, dass sehr häufig negative Werte für x berechnet wurden, wird zuvor noch der *bdsSigma*-Wert angepasst.

Filter.bdsFilter(key):

```
ConfigValues cv=ConfigValues.getInstance(key);

long bdsAvg    = cv.getActBDSAvg();
long bdsSigma  = cv.getActBDSSigma();

// *** to avoid negative-values
if(bdsSigma>bdsAvg/3) bdsSigma=bdsAvg/3;

// *** normal-distribution
z = randomBDS.nextGaussian();
x = (z * bdsSigma) + bdsAvg;

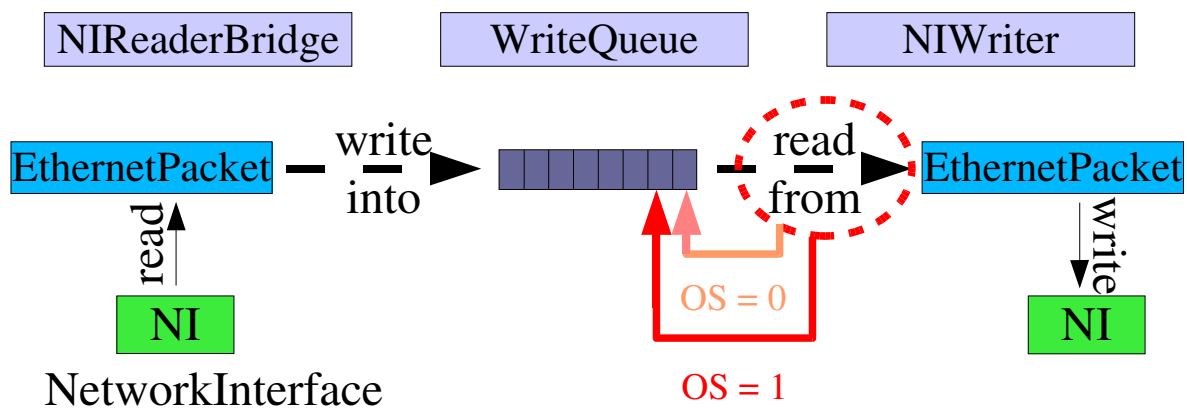
return x;
```

Die gesamte Logik der *NIWriter*-Klasse (inkl. Bridge und Simulator – Logik):

NIWriter – Thread:

```
while(true) {  
    packetInfo = queue.getPacket();  
    delay      = packetInfo.getDelay();  
    packet     = packetInfo.getPacket();  
  
    this.sleep(delay);  
  
    ni.writePacket(packet);  
}
```

BlockOutOfSequence



Ein OutOfSequence wird simuliert, indem die Methode *getPacket()* der *WriteQueue* nicht immer das letzte Paket aus der Queue zurückliefert. Es wird anhand des *OS-Filters* entschieden, welches Paket zurückgeliefert wird.

BOSW Filter:

```
Queue.get(Filter.osFilter());
```

Die Filter-Methode *osFilter()* berechnet den OutOfSequence-Wert. Sie holt sich zunächst eine gleichverteilte Zufallszahl *randOS* und den aktuell gültigen OS-Wert in Prozent *pOS*. Falls *randOS* grösser als *pOS/100* ist, ist ein OutOfSequence aufgetreten und es wird zusätzlich noch eine gaussverteilte Zufallszahl *x* berechnet mit Hilfe der Messwerte *osAvg* und *osSigma*. Auch hier wieder die sigma-Korrektur zur Vermeidung von negativen Werten:

Filter.osFilter(key):

```

ConfigValues cv=ConfigValues.getInstance(key);
double  randOS = randomOS.nextDouble();
int  pOS      = cv.getActOS();
int  osAvg    = cv.getActOSAvg();
int  osSigma  = cv.getActOSSigma();
// *** if pOS=-1(error,cv not init)=>no filter
if( randOS < (((double)pOS)/100.0) ) {
    // *** to avoid negative-values
    if(osSigma>osAvg/3) osSigma=osAvg/3;

    // *** normal-distribution
    double z = randomOS.nextGaussian();
    double x = (z * osSigma) + osAvg;
    return x;
}
// *** else: no out of sequence
return 0;

```

Die gesamte Logik der *WriteQueue*-Klasse (inkl. Bridge und Simulator – Logik):

WriteQueue – getPacket():

```

// *** try to find the first packet with os==0
for(i=0;i<packetInfos.size();i++) {
    pi = packetInfos.get(i);
    if(pi.getOSValue()<=0) {
        packetInfos.removeElementAt(0); break;
    }
    else {
        pi.decrOSValue();
        pi=null;
    }
}
return pi;

```

III. Implementierungs – Details

1. Native – C (*libpcap* – read / *libnet* – write)

Ein wichtiger Teil des Projekts war direkten Zugriff auf die Netzwerkkarte zu bekommen. Es sollten direkt über das Device EthernetPackets verschickt bzw. empfangen werden.

Nach längeren Recherchen und Tests direkt mittels RawSockets zu arbeiten, habe ich mich für zwei etablierte C-Bibliotheken entschieden.

Mit Hilfe der **LIBPCAP** kann man EthernetPackets direkt von der Netzwerkkarte lesen.

Die **LIBNET** ergänzt diese Funktionalität mit der Möglichkeit direkt auf der Netzwerkkarte zu schreiben.

Die Wahl viel auf diese beiden Bibliotheken, da diese leichter anzuwenden sind als direkte RawSocket-Programmierung und beide weit verbreitet sind. Zusätzlich existieren diese Bibliotheken für verschiedene Betriebssysteme.

Um mit der **LIBPCAP** von einem Device zu lesen, muss sie zuerst initialisiert werden:

```
// *** init the LIBPCAP:
pcap_t *pcap_descr;
// *** init the ErrorBuffer
char pcap_errbuf[PCAP_ERRBUF_SIZE];
memset(pcap_errbuf, '\0', PCAP_ERRBUF_SIZE);
// *** init PCAP with: ReadDevice, max. CaptureLength,
// *** Promisc on(1)/off(0), timeout, Errbuffer;
pcap_descr = pcap_open_live(dev_read, BUFSIZ,
                           promisc, -1, pcap_errbuf);

// *** Error-Handling
if(pcap_descr == NULL) {
    printf("pcap_open_live():%s\n", pcap_errbuf);
    return NULL;
}
return pcap_descr;
```

Um das Device zu schliessen:

```
pcap_close(pcap_descr);
```

Das eigentliche Lesen funktioniert folgendermassen:

```
// *** read from device with LIBPCAP:
const u_char* packet;
// *** init PCAP-PacketHeader: contains the captured length
struct pcap_pkthdr pcap_packetHeader;
memset(&pcap_packetHeader, 0, sizeof(struct pcap_pkthdr));
// *** reads one packet
packet = pcap_next(pcap_descr, &pcap_packetHeader);
// *** Error-Handling:
if(packet == NULL) {
    printf("%s\n", pcap_geterr(pcap_descr));
    return -1;
}
// *** return the captured length
return pcap_packetHeader.caplen;
```

Um mit der *LIBNET* auf einem Device zu schreiben, muss sie zuerst auch initialisiert werden (dies geschieht analog zur *LIBPCAP*):

```
// *** init the LIBNET:
libnet_t *libnet_descr;
// *** init the ErrorBuffer
char libnet_errbuf[LIBNET_ERRBUF_SIZE];
memset(libnet_errbuf, '\0', LIBNET_ERRBUF_SIZE);
// *** init LIBNET with: injection-type: LinkLayer-Advanced,
                        WriteDevice, ErrBuffer
libnet_descr=libnet_init(LIBNET_LINK_ADV, dev_write, libnet_errbuf);
// *** Error-Handling
if(libnet_descr==NULL) {
    printf("Error while opening link interface:%s\n",libnet_errbuf);
    return NULL;
}
return libnet_descr;
```

Um das Device zu schliessen:

```
// *** close LIBNET:
libnet_destroy(libnet_descr);
```

Das eigentliche Schreiben funktioniert folgendermassen:

```
// *** write to device with LIBNET:
int write_status=-1;

// *** writes one packet
write_status = libnet_adv_write_link(libnet_descr, packet, len);
// *** Error-Handling
if(write_status<0) {
    printf("%s\n", libnet_geterror(libnet_descr));
}
// *** return the really written count of bytes
return write_status;
```

2. *JavaNativeInterface*

Da der Hauptteil des Projekts in Java implementiert wurde und der Zugriff auf die Netzwerkkarten in C erfolgen musste, war es notwendig für die Kommunikation zwischen Java und C das *JavaNativeInterface* einzusetzen.

In Java werden alle Methoden, die in C implementiert werden müssen, als *native* deklariert. Die Klasse *NetworkInterface* kapselt in dem Projekt alle native-Methoden.

```
public native int readPacket(int pcapDescr, byte[] packet,  
                             byte[] srcMAC, byte[] destMAC);
```

Mit dem im JDK enthaltenen Programm *javah* wird eine C-Header Datei erzeugt, welche alle benötigten JNI-konformen Prototypen enthält.

Diese C-Methoden werden nun implementiert, darin müssen hauptsächlich die Java-Datentypen in C-konforme umgewandelt werden.

Als Beispiel fungiert hier die native Implementierung der readPacket()-Methode:

```
JNIEXPORT jint JNICALL  
Java_fhw_telekommunikation_umlssim_NetworkInterface_readPacket  
    (JNIEnv *env, jobject obj, jint pcapDescr,  
     jbyteArray packet, jbyteArray srcMAC, jbyteArray destMAC) {  
  
    ...  
  
    /* *** translating java-byte[]: packet */  
    jsize packetLength = (*env)->GetArrayLength(env, packet);  
    jbyte *packetBody=(*env)->GetByteArrayElements(env, packet, 0);  
  
    ...  
  
    /* *** calls the wrapped method */  
    capLength = readPacket((pcap_t *)pcapDescr,  
                           (u_char *)packetBody,  
                           (unsigned char *)srcMACBody,  
                           (unsigned char *)destMACBody);  
  
    /* *** releasing the memory */  
    (*env)->ReleaseByteArrayElements(env, packet, packetBody, 0);  
  
    ...  
  
    return capLength;  
}
```

Der Prototyp der Funktion ist in dem generierten Header-File festgelegt. Ansonsten müssen in diesem Fall nur die Java-Byte-Arrays umgewandelt werden. Anschliessend wird die normale C-Funktion aufgerufen. Am Ende muss noch aufgeräumt werden.

3. Überblick des Gesamtprojekts

Das Projekt bestand aus drei Teilen:

1. Der native C-Teil – Kommunikation mit den Netzwerkkarten

Der gesamte native Code steckt in der Datei ' `networkinterface.c` '. Diese wird als shared-object kompiliert (siehe *III.4.*). Weitere Details hierzu finden sich in *III.1. Native-C* und *III.2. JNI*.

2. Die Bridge – und Simulator – Logik

Simulator:

Die Klasse mit der *main*-Methode; steuert das gesamte Programm.

NIReader / NIReaderBridge / NIReaderGateway:

Liest Pakete vom NetworkInterface in die WriteQueue.

NIWriter:

Schreibt Pakete auf das NetworkInterface; liest sie vorher von der WriteQueue.

WriteQueue:

Verwaltet die EthernetPackets.

Filter:

Stellt die drei in *II.6 Simulations – Logik* besprochenen Filter zur Verfügung.

MACContainer:

Verwaltet MACAdressen (siehe *II.3 Bridge – Logik*).

NetworkInterface:

Stellt die Java-Schnittstelle zu den native-Methoden dar.

3. Einlesen und Bereitstellen der Messwerte

ConfigValues:

Verwaltet die Messwerte; von dieser Klasse existiert je eine Instanz pro Netzwerkkarte.

ConfigValues hält zusätzlich zu den Messwerten noch die Dauer (wie lange eine Messwert-Reihe gültig ist) und einen Index (welcher Messwert jeweils zurückgegeben wird).

ConfigFileReader:

Liest die .usim-Datei mit den Messwerten ein.

ConfigTimer:

Zählt den Index von ConfigValues hoch, nachdem die gültige Dauer einer Messwert-Reihe überschritten wurde.

4. Übersetzung und Voraussetzungen des Simulators

Die usim-Files müssen folgendermassen aufgebaut sein:

bdsAvg;bdsMin;bdsMax;blr;os;osAvg;osMin;osMax;bdsVarianz;bdsSigma;osVarianz;osSigma

Betriebssystem:

SuSE Linux getestet in den Versionen 7.3 und 9.0

Bibliotheken:

LIBPCAP getestet in der Version 0.7.2

LIBNET getestet in den Version 1.1.0 und 1.1.1 (ältere Versionen (1.0) funktionieren nicht)

JDK getestet in der Version 1.4.2 von **SUN**

Übersetzung der Java-Klassen (telekommunikation/java/compileAllJava.sh):

```
javac -d classes fhw/telekommunikation/umtssim/helper/*.java
```

```
javac -d classes fhw/telekommunikation/umtssim/config/*.java
```

```
javac -d classes fhw/telekommunikation/umtssim/*.java
```

```
javac -d classes fhw/telekommunikation/umtssim/test/*.java
```

Erzeugung des C-Headers:

```
javah -jni -d ../c/jni/ fhw.telekommunikation.umtssim.NetworkInterface
```

Übersetzung des nativen Codes – networkinterface.c (telekommunikation/c/jni/compile.sh):

```
# compile
```

```
gcc -fPIC -c `libnet-config --defines --cflags` -I/usr/lib/java/include -I/usr/lib/java/include/linux networkinterface.c
```

```
# link as shared-object
```

```
ld -shared -lc networkinterface.o -o libnetworkinterface.so -lpcap `libnet-config --libs`
```

```
# copy in the right directory (optional)
```

```
cp libnetworkinterface.so ../../java/
```

```
# compile as executable (optional: only for testing)
```

```
gcc `libnet-config --defines --cflags` -I/usr/lib/java/include -I/usr/lib/java/include/linux -o networkinterface networkinterface.c -lpcap `libnet-config --libs`
```

set CLASSPATH (Aktuelles Verzeichnis muss telekommunikation/java sein) :

```
export CLASSPATH=`pwd`/classes # telekommunikation/java/setClasspath.sh
```

Start des Simulators (telekommunikation/java/runSimBridge.sh):

```
# possible options: --help, -debug, -dev0 eth0, -dev1 eth1, -simFile0 test.usim, -simFile1 test.usim
```

```
# -Djava.library.path=' Pfad zum SharedObject'
```

```
java -Djava.library.path=. fhw.telekommunikation.umtssim.Simulator
```

Start der TestTools (telekommunikation/java/runSimTestServer.sh | runSimTestClient.sh):

```
# possible options: --help, -port portNumber, -delay delayInMillis
```

```
java fhw.telekommunikation.umtssim.test.DatagramServer
```

```
# options: --help, -port portNumber, -debug, -gui
```

```
java fhw.telekommunikation.umtssim.test.DatagramClient
```

IV. Zusammenfassung

In dieser Ausarbeitung wurde gezeigt, wie man mit Hilfe der **LIBPCAP** direkt von einer Netzwerkkarte liest, sowie mittels der **LIBNET** direkt auf einer Netzwerkkarte schreibt. Eine Implementierungsmöglichkeit in **C** wurde in *III.1 Native-C* gezeigt.

Wie man auf diesen nativen C-Code via **JavaNativeInterface** zugreift, wurde anschliessend in *III.2 JNI* gezeigt.

Kapitel *II.3 Bridge – Logk* beinhaltet die Logik zur Programmierung einer Bridge. Zusammen mit *II.5 Wahrscheinlichkeitsverteilungen / II.6 Simulations – Logk* bilden diese drei Kapitel den Schwerpunkt dieser Ausarbeitung. Es wurde gezeigt, wie man mittels einer Gauss – Verteilung die in *II.4* genannten Grössen simulieren könnte und an welchen Stellen man die Bridge – Logik erweitern muss.

Abschliessend bleibt zu sagen, dass die Recherchen sowie die Implementierung zum direkten Zugriff auf Netzwerkkarten einen erheblichen Teil der zur Verfügung stehenden Zeit beanspruchte. Es fanden sich dutzende verschiedene Möglichkeiten zur Kommunikation mit den Netzwerkkarten, von denen die meisten aber nicht mit den zur Verfügung stehenden Linux-Versionen funktionierten. Auch der Versuch mit der **LIBNET** in der Version 1.0.x (welche z.B. bei SuSE Linux 8.2 enthalten ist) die **LIBPCAP** zu ergänzen, funktionierte nicht wunschgemäss. Mit der Umstellung auf SuSE 9.0 und damit **LIBNET** 1.1.0 stand ein neues Interface bereit, mit dessen Hilfe es dann möglich war, das mit **LIBPCAP** gelesene Paket direkt wieder rauszuschreiben.

Auf Grund der dann recht knappen Zeit, wurde dann eine einfache Gauss – Verteilung gewählt zur Simulation der Bewertungsgrössen. Ausserdem hatte man sich zu früh auf die Werte im .usim-File geeinigt. Zu dem Zeitpunkt wussten die Programmierer noch gar nicht, was auf sie zukommen wird. So standen nur min, max, average und Standardabweichung zur Verfügung.

Im Nachhinein würde ich die in *II.5* diskutierte **diskrete Verteilung** favorisieren. Dafür bräuchte man aber alle Messwerte bzw. deren Häufigkeitsverteilung. Ich denke das wäre sicher eine bessere und genauere Abbildung der realen Messwerte gewesen.

V. Quellen

LIBPCAP – 0.7.2

[http://www.tcpdump.org/
man-pages](http://www.tcpdump.org/man-pages)

LIBNET – 1.1.x

[http://www.packetfactory.net/projects/libnet/
man-pages](http://www.packetfactory.net/projects/libnet/man-pages)

<http://www.securityfocus.com/infocus/1386>
<http://cs1.mcm.edu/tutorial/doc/libnet-1.0.1b/html/6.html>

JAVA – 1.4.2

JNI-Tutorial – <http://java.sun.com/docs/books/tutorial/index.html>
API – <http://java.sun.com/j2se/1.4.2/docs/api/>

Shutdownhook – <http://www.ibm.com/developerworks/ibm/library/i-signalhandling/>

Verteilungen

<http://www.uni-konstanz.de/FuF/wiwi/heiler/os/vt-index.html>
<http://mathworld.wolfram.com/topics/StatisticalDistributions.html>
<http://thesaurus.maths.org/dictionary/map/word/996>
<http://www.xycoon.com/continuousdistributions.htm>

C – Programmierung

<http://www-106.ibm.com/developerworks/library/l-shobj/>

Raw – Sockets

<http://docs.sun.com/db/doc/806-1017/6jab5di3g?a=view>
<http://www.lowtek.com/sockets/>
<http://mixter.void.ru/rawip.html>
<http://www.ecst.csuchico.edu/~beej/guide/net/>