Dokumentation zum Beleg für Rechnernetze I

Philipp Wenskus - s72902

12. Januar 2016

Inhaltsverzeichnis

1 Belegaufgabe

Die Aufgabe des Belegs besteht darin, zwei Programme (Client + Server) in Java zu erstellen, bei dem der Client eine beliebige Datei sendet und der Server diese empfängt. Als Grundlage für die Übertragung soll das UDP-Protokoll genutzt werden, welches in Java als vorgefertigte Funktion zu Verfügung steht.
Ein Startpaket am Anfang der Übertragung liefert dem Server nötige Informationen zu der Datei die vom Client empfangen wird. Die Datei selbst soll vom Client in kleinere Pakete aufgeteilt werden. Beim Übertragen des letzten Datenpakets muss die Prüfsumme(CRC32) der Datei mitgesendet werden welche der Server dann überprüft um die Datei zu verifizieren. Um Fehler in der Übertragung der einzelnen Pakete zu erkennen und diese ggf. auszubessern sollte ein Stop-And-Wait Protokoll implementiert werden bei dem der Server erfolgreich empfangene Pakete mit einen Acknowledgement(kurz Ack), welches er an Client sendet, bestätigt.

2 Stop-And-Wait Protokoll

Um das Programm besser zu verstehen wird hier die Funktionsweise und der Ablauf des Stop-And-Wait Protokolls beschrieben.

2.1 Ablauf

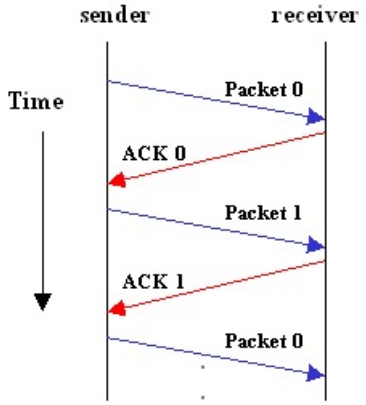


Abbildung 1: Übertragung mit dem Stop-And-Wait Protokoll, Quelle: www.isi.edu

1. Der Client sendet sein vorher zusammengestelltes Paket an den Server
2. Der Server empfängt das Paket erfolgreich und sendet ein Acknowledgement(hier Sessionnr. und Paketnr.) an den Client
3. Der Client empfängt das Ack, überprüft dies und sendet, falls vorhanden, das nächste Paket

2.2 Probleme und Fehlerbekämpfung bei der Übertragung

Der oben beschriebene Ablauf stimmt nur, wenn die Übertragung fehlerfrei funktioniert. Da dies nicht immer gewährleistet werden kann, sind im Protokoll mehrere Kontrollmechanismen, welche Fehler erkennen und ausbessern sollen.
Folgende zwei Szenarien sind möglich welche man auch in Abbildung **??** sehen kann

1. Paket kommt nicht beim Server an (Abbildung **??**, rechte Seite)
   - Der Client sendet ein Paket an den Server. Der Server empfängt das Paket nicht und sendet somit kein Ack. Der im Client eingestellte *Timeout* für das Empfangen des Acks läuft ab. Ist der Timeout abgelaufen sendet der Client das Paket erneut (hier bis zu 10 mal). Sobald der Server das Paket erfolgreich empfangen hat schickt er das Ack zum Client.

2. Acknowledgement kommt nicht beim Client an (Abbildung **??**, linke Seite)
   - Der Client sendet ein Paket an den Server. Der Server empfängt das Paket erfolgreich und schickt ein Ack an den Client. Der Client bekommt das Ack nicht, wodurch der *Timeout* in Kraft tritt und der Client die Datei nochmals sendet (hier bis zu 10 mal). Der Server empfängt das Paket erneut, verwirft es und schickt nochmals das Ack an den Client.

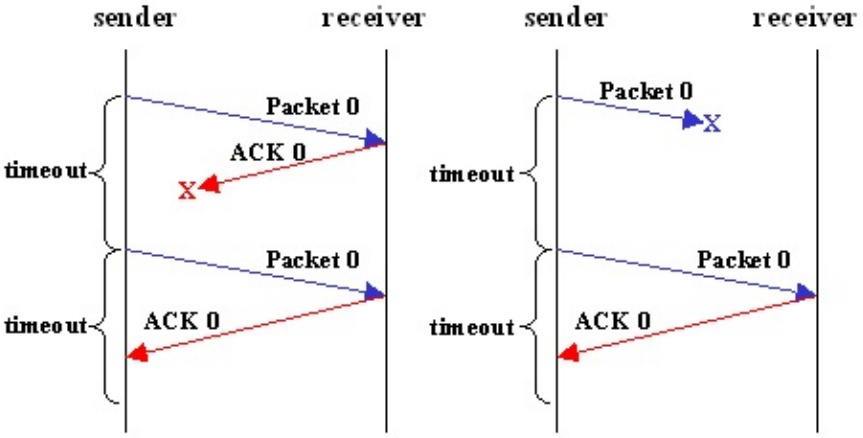


Abbildung 2: Fehlerszenarien des Stop-And-Wait-Protokolls, Quelle: www.isi.edu

3 Funktionsweise

In Folgenden werden die beiden Programme und deren Funktionen detaillierter beschrieben.

3.1 Client

Der Client ist für das Senden einer Datei zum Empfänger zuständig. Dabei splittet der Client die Datei in mehrere kleine Pakete auf und verschickt diese dann. Um den Server auf die eingehende Datei vorzubereiten schickt der Client ein Startpaket mit allen nötigen Informationen an den Empfänger.

3.1.1 Startpaket

Das Startpaket wird am Anfang der Verbindung an den Server gesendet um diesen eine neue Verbindung zu signalisieren und nötige Informationen zu liefern. Nachfolgend wird auf die einzelnen Bestandteile des Startpakets genauer eingegangen.

Startpaket
- Sessionnr. (2 Byte) - Paketnr. (1 Byte, im Startpaket immer 0) - Kennung 'Start' (5 Byte, ASCII) - Dateigröße (8 Byte) - Länge des Dateinamens (2 Byte) - Dateiname (0-255 Byte, UTF-8) - CRC32 über gesamtes Startpaket (4 Byte)

Sessionnummer Die Sessionnr. ist eine zufällig generierte, 2 Byte große positive Zahl welche durch die Random-Funktion von Java erzeugt wird:

```
1 public static short randomSession()
2 {
3         Random randSession = new Random();
4
5         short randomNumber = (short)randSession.nextInt(Short.MAX_VALUE + 1);
6         return randomNumber;
7 }
```

Short.MAX_VALUE gibt dabei die möglichst große Positive Zahl eines einer Short-Variable an. Die +1 steht dabei für die Null, die noch nicht in Short.MAX_VALUE enthalten ist.

Paketnummer Die Paketnummer ist im Startpaket immer 0 und kann in den folgenden Paketen entweder 1 oder 0 annehmen (Packetnr mod(2))

```
int i = 0; //Globale Variable der Client-Klasse

public static byte[] getnewPackageNr()
{
        byte[] packageNr = new byte[1];
        packageNr[0] = (byte)(i%2);
        Client.i++;
        return packageNr;
}
```

Die Funktion *getnewPackageNr* gibt die Paketnummer für das kommende Paket zurück. Die globale Variable i, die in der Funktion modulo 2 gerechnet wird, ist die dabei die echte Paketnummer (z.B. für das Startpaket gleich $0 \Rightarrow 0 \bmod(2) = 0$)

Kennung Die Kennung ist ein weiteres Merkmal an dem der Server das Startpaket erkennen soll. Sie ist immer 'Start' und als ASCII formatiert.

Dateigröße Der Server benötigt die Dateigröße um festzustellen wann die Daten der Datei vollständig angekommen sind um dann am Ende des letzten Datenpakets den CRC-Code der Datei auszulesen zu können

```
File file = new File(filepathString);
//Dateilänge (64 Bit = 8 Byte)
byte[] filelength = ByteBuffer.allocate(8).putLong(file.length()).array();
```

Dateinamenlänge Der Empfänger braucht die Dateilänge für das Auslesen des Dateinamens aus dem Startpaket, da dieser Teil eine variable Größe hat.

```
File file = new File(filepathString);
....
String filenameString = file.getName();
short filenamelengthShort = (short)filenameString.length();
byte[] filenamelength =
                ByteBuffer.allocate(2).putShort(filenamelengthShort).array();
```

filenamestring ist hier der Dateiname der zu sendenden Datei, *filenamelengthshort* die Länge des Namens.

Dateiname Übergibt den Namen der Datei, UTF-8 formatiert, an den Server. Dieser Speichert die Datei unter dem gleichen Namen ab (bei schon vorhanden Dateinamen ⇒ Dateiname + 1)

```
File file = new File(filepathString);
....
String filenameString = file.getName();
short filenamelengthShort = (short)filenameString.length();
//Dateiname als UTF8 String (0-255 Byte) -> länge kann von filenamelength abhängig ge
byte[] filename = new byte[filenamelengthShort];
filename = filenameString.getBytes("UTF-8");
```

CRC32 (Startpaket) Sobald alle Daten des Startpakets vorbereitet sind werden diese über einen ByteBuffer in ein Bytearray geschoben um die Prüfsumme für das Startpaket zu berechnen.

```
Checksum checksumStart = new CRC32();
checksumStart.update(startPackage_withoutCRC.array(),
        0, startPackage_withoutCRC.array().length);
long crcStartNumber= checksumStart.getValue();
byte[] crcStart = longtoByteCRC(crcStartNumber);

/*Startpaket mit CRC-Code zusammenfügen*/
ByteBuffer startPackageBuff = ByteBuffer.allocate
                (startPackage_withoutCRC.array().length + crcStart.length);
//Add CRC Code
startPackageBuff.put(startPackage_withoutCRC.array());
startPackageBuff.put(crcStart);
byte[] startPackage = startPackageBuff.array();
```

Sobald die Prüfsumme berechnet wurde wird sie an das Startpaket angefügt, welches somit versendet werden kann. Problem dabei ist, den CRC-Code, welcher von der Java-Funktion als *long* zurückgegeben wird in ein 4-Byte-Aaray zu schreiben. Dieses Problem löst die hier verwendete Funktion longtoByteCRC:

```
public static byte[] longtoByteCRC(long checksum)
{
        byte[] checksumBytes=new byte[4];
        for(int j=4-1;j>=0;j--)
        {
                for(int i=0;i<8;i++)
                {
```

```
8                           if((checksum%2)==1) {
9                                   checksumBytes[j] |=(1 << i);
10                          } else {
11                                  checksumBytes[j] |=(0 << i);
12                          }
13                          checksum=checksum/2;
14                  }
15          }
16          return checksumBytes;
17  }
```

Durch Bitoperationen werden die Bytes des errechneten *long* Wertes, in ein 4-Byte-Array konvertiert.

3.1.2 Datenpakete

Die Datenpakete werden gesendet sobald das Startpaket erfolgreich gesendet wurde. Sie enthalten die Sessionnr. (die gleiche wie das Startpaket) eine Paketnummer (abwechselnd 0 oder 1) und die Daten der Datei, die vom Client gesplittet wurde. Außerdem enthält das letzte Datenpaket den CRC Code der gesendeten Datei um diese zu verifizieren.

Datenpaket
- Sessionnr. (2 Byte, gleiche wie im Startpaket) - Paketnr. (1 Byte, 0 oder 1) - Daten - CRC32 der Datei (4 Byte, nur im letzten Paket)

3.1.3 Start des Clients

Zum Start des Clients müssen Adresse des Empfängers, der Port mit dem dieser Daten empfängt und der Pfad der Datei die gesendet werden soll, angegeben werden. Um den Client einfacher zu starten wird er über ein Script aufgerufen:

./client-udp <Server Adresse> <Port> <Dateiname/Dateipfad>

Sollten zu wenig Parameter übergeben worden sein wird der Client nicht gestartet.

3.1.4 Programmablauf des Clients

Nach dem Starten mit den erforderlichen Parametern wird das unter **??** angegebene Paket erstellt. Das Zusammenstellen des Paketes übernimmt die Funktion *getStartPackage*, welche die Datei und die zuvor generierte Sessionnr. übergeben bekommt, und das fertige Byte-Array mit den Daten des Startpaketes zurückgibt. Danach wird das Startpaket an dem Server übermittelt. Dies geschieht über das UDP-Socket (DatagramSocket).

```
DatagramSocket socket = new DatagramSocket();    //öffne den UDP-Socket
socket.setSoTimeout(TIMEOUT);

InetAddress serverAddress = InetAddress.getByName(host);
DatagramPacket startPackage =
                    new DatagramPacket(startPackageByte, startPackageByte.length, serv

long timeStart = System.nanoTime();         //starte Zeitabnahme
socket.send(startPackage);
```

Der Timeout wird hier auf einen Anfangswert festgelegt und während des Sendens je nach Verbindung entweder erhöht oder verringert. Da sich über ein DatagramSocket nur ein DatagramPacket versendet lässt, wird ein DatagramPacket erstellt, dass die Daten des Pakets, die Länge der Daten, die Adresse des Empfängers und den Port den der Empfänger benutzt enthält.
Nun ist der Ablauf wie im Kapitel **??** beschriebenen Stop-And-Wait Protokoll.
Der Client wartet auf das Ack(siehe **??**). Sollte er das Paket empfangen überprüft er ob die Paket- und Sessionnr. mit dem gesendeten Paket übereinstimmen. Ist das der Fall setzt der Client mit dem Senden der Datenpakete fort. Wenn ein Timeout beim Empfangen eintritt oder das Ack fehlerhaft ist wird das Startpaket bis zu 10 mal erneut gesendet. Sollte dann immer noch kein richtiges Ack angekommen sein wird eine Fehlermeldung ausgegeben und der Client beendet.

Zum übertragen der Datei wird immer ein Stück der Datei in einen Bytebuffer geschrieben. Dann wird das unter Kapitel **??** beschriebene Datenpaket erstellt und an den Server gesendet. Es wird wie auch beim Startpaket auf das Ack gewartet. Sollte dies erfolgreich ankommen wird das Einlesen, Senden und Empfangen in einer while-Schleife so lange wiederholt, bis die Datei ganz eingelesen wurde.

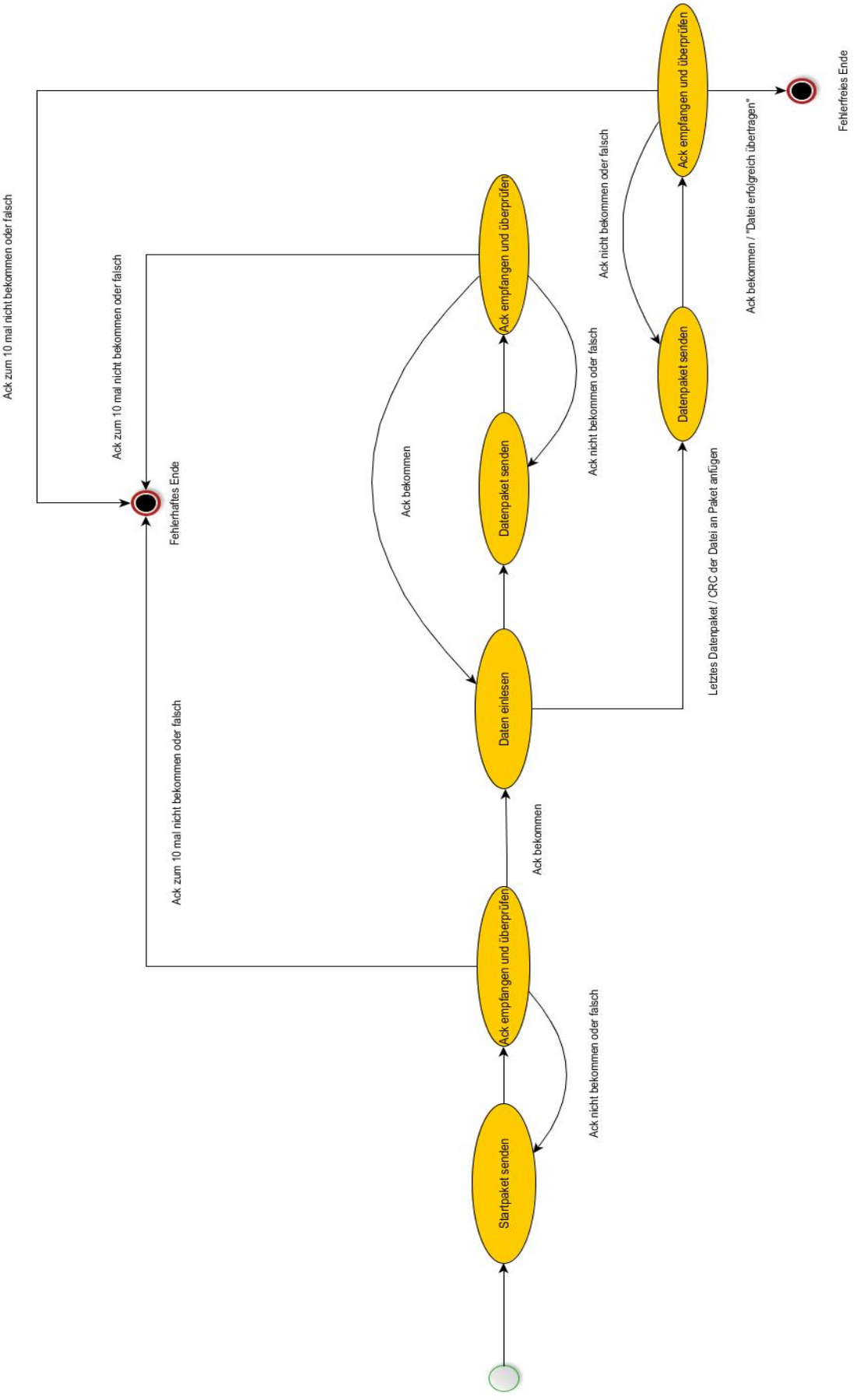
```
FileInputStream fileStream = new FileInputStream(file);
FileChannel fileChnl = fileStream.getChannel();
ByteBuffer filebuffer = ByteBuffer.allocate(PACKAGESIZE);
while(fileChnl.read(filebuffer) > 0)
{
        long current;
        DatagramPacket dataPackage;
        filebuffer.flip();
        sendedBytes += PACKAGESIZE;
        if (sendedBytes < file.length())
        {
                byte[]dataPackageBytes = getDataPackage(filebuffer.array(), sessionNr
                dataPackage =
                        new DatagramPacket(dataPackageBytes, dataPackageBytes.length,
```

```
                current = dataPackageBytes.length;
                timeStart = System.nanoTime();
                socket.send(dataPackage);
                filebuffer.clear();
        }
        /******Falls letztes Paket gesendent wird*****/
        else
        {
                socket.setSoTimeout(1500); //Timeout wird erhöht da berechnen von CRC
                int restBytes = (int)(PACKAGESIZE -(sendedBytes-filesize));
                byte[]dataPackageBytes =
                        getDataEndPackage(filebuffer.array(),
                        sessionNr, checksumFile, restBytes);
                dataPackage =
                        new DatagramPacket(dataPackageBytes,
                        dataPackageBytes.length,serverAddress, port);
                current = dataPackageBytes.length;
                timeStart = System.nanoTime();
                socket.send(dataPackage);
                filebuffer.clear();
        }
        .....
}
```

Sollte das letzte Paket gesendet werden wird der else-Zweig des Codes ausgeführt, der neben den Daten noch den CRC-Code der Datei an das Datenpaket anfügt.
Wenn der Client alle Pakete an den Server geschickt hat und ein Ack erhält ist das die Bestätigung für die korrekte Übertragung der Datei, da der Server den vom Client erhaltenen CRC-Code mit dem selbst berechneten CRC-Code der Datei vergleicht und nur bei Übereinstimmung ein Ack sendet.

Zusätzliche Funktionen Während der Übertragung wird dem Benutzer eine Fortschrittsleiste angezeigt, welche aufweist wie viele Daten schon gesendet worden sind und wie viele noch fehlen. Zudem wird die aktuelle Datenrate, mit der die Pakete übertragen werden neben der Fortschrittsleiste angegeben.
Am Ende der Übertragung wird außerdem die durchschnittliche Datenübertragungsrate und die Gesamtdauer angegeben.

3.1.5 Zustandsdiagramm



3.2 Server

Der Server ist zum Empfang der gesendeten Pakete zuständig und fügt nach erfolgreichem Empfangenen die Daten in einer Datei zusammen, so dass ein Abbild der vom Client(**??**) geschickten Datei auf dem Server entsteht. Dabei wird jedes erfolgreich empfangene Paket mit einen Ack bestätigt.

3.2.1 Acknowledgement (Ack)

Mit dem Acknowledgement bestätigt der Server dem erfolgreichen Empfang eines einzelnen Pakets. Dabei enthält das Ack die Session- und Paketnummer des empfangenen Paketes

Acknowledgement
- Sessionnr. (2 Byte, gleiche wie im zuletzt empfangenen Paket) - Paketnr. (1 Byte, gleiche wie im zuletzt empfangenen Paket)

3.2.2 Start des Servers

Zum Start des Servers muss der Port angegeben werden, auf dem der Server auf eine Verbindung eines Clients warten soll. Aufgerufen wird der Server wie der Client über ein Script:

./server-udp <Port>

Sollten zu wenig Parameter übergeben werden oder wird der Port schon verwendet(z.B weil Server schon gestartet wurde) kann der Server nicht gestartet werden.

3.2.3 Programmablauf des Servers

Nach dem Start eröffnet der Server ein UDP-Socket auf dem als Startparameter angegebenen Port. Er fängt an auf eine Verbindung eines Clients zu warten und da im Server kein Timeout eingestellt ist, wartet der Server so lange, bis Daten empfangen werden.

```
1 DatagramPacket receiveStartPackage =
2         new DatagramPacket(receiveStart, receiveStart.length);
3 serverSocket.receive(receiveStartPackage);
4 InetAddress IPAddress = receiveStartPackage.getAddress();
5 int port_receive = receiveStartPackage.getPort();
```

Nachdem der Server ein Datagram-Paket empfangen hat extrahiert er über *.getAdress()* die IP-Adresse des Clients und durch *.getPort()* den Port den der Client verwendet. Der Port ist zufällig und ändert sich bei jeder Ausführung des Clients.

Ab hier läuft das Programm in einer äußeren Schleife, welche für die Daten des Startpakets zuständig ist und eine, in dieser Schleife enthaltenen inneren Schleife, welche zum Empfang der Datenpakete dient. Folgend werden die Funktionen der Schleifen beschrieben:

Äußere Schleife Sobald der Server das erste Paket bekommen hat konvertiert er die im Startpaket(siehe **??**) enthalten Daten mit Hilfe eines Bytebuffers zurück in die Ursprungsform

```
1 //Beispiel für die Konvertierung der Sessionnr. zurück zu einen short.
2 short sessionNr = startpackageBuffer.getShort(0);
```

Danach überprüft der Server den CRC-Code des Startpakets indem er die Prüfsumme vom Startpaket abschneidet und dann selbst einen CRC-Code von dem restlichen Startpaket erstellt. Sind diese beiden Codes gleich wurde das Startpaket richtig empfangen.

```
1 if (crcStartCalc != checksumStartReceived || packageNr != 0 ||
2                 !Kennung.equals("Start"))
3 {
4         System.out.printf("Fehlerhaftes Startpaket.\n");
5         //Warte auf neues Paket
6         serverSocket.receive(receiveStartPackage);
7         IPAddress = receiveStartPackage.getAddress();
8         port_receive = receiveStartPackage.getPort();
9 }
```

Außerdem wird geprüft ob die Paketnummer wie vorgeschrieben 0 ist, und die Kennung dem String 'Start' entspricht. Sollte eins dieser Vorgaben nicht zutreffen wird das Startpaket verworfen, und auf ein neues Paket gewartet, welches dann erneut die äußere Schleife durchläuft.
Sollte alles richtig sein sendet der Server ein Ack an den Client

```
1  ByteBuffer ACKStart = getACK(sessionNr, packageNr);
2 //Sende ACK
3 DatagramPacket sendStartACK = new DatagramPacket(ACKStart.array(), ACKStart.array().l
4 serverSocket.send(sendStartACK);
```

Dabei erstellt *getACK()* das zu sendende Paket. Dieses wird dann über den UDP-Socket welches zum Start intialisiert wurde an den Client gesendet.
Dann überprüft der Server ob der Dateiname schon im Ordner enthalten ist.

```
File file = new File(filename);
while (file.exists())
{
        String newFilename = file.getName() + "1";
        file = new File(newFilename);
}
FileOutputStream fileOutputStream = new FileOutputStream(file);
FileChannel fileChnl = fileOutputStream.getChannel();
```

Ist der Dateiname schon vorhanden wird eine '1' an den Dateinamen angefügt und erneut getestet. Dies geschieht bis ein freier Dateiname gefunden wurde. Folgend wird ein *FileOutputStream* für die Datei geöffnet. Außerdem wird ein *FileChannel* geöffnet durch den in der inneren Schleife die Daten in die Datei geschrieben werden.

Innere Schleife Nachdem das Startpaket erfolgreich empfangen wurde und der *FileChannel* geöffnet ist fängt die innere Schleife an.
Am Anfang dieser Schleife wird auf ein neues Paket vom Server gewartet. Wird das Paket empfangen wird aus dem Paket die Sessionnummer, die Paketnummer und, um auszuschließen, dass es kein Startpaket ist, die Kennung des Paktes ausgelesen.

```
short sessionNrData = dataPackageBuffer.getShort(0);
packageNr = dataPackageBuffer.get(2);
byte[] KennungBytesData = Arrays.copyOfRange(receiveData, 3, 8);
String KennungData = new String(KennungBytesData, StandardCharsets.US_ASCII);

//Überprüfen ob Startpaket
if (sessionNrData != sessionNr && KennungData.equals("Start"))
{
        System.out.print("If 1\n");
        port_receive = receiveDataPackage.getPort();
        IPAddress = receiveDataPackage.getAddress();
        receiveStart = Arrays.copyOfRange(receiveData, 0, 277);
        break;
}
```

Sollte sich die Sessionnr des Datenpakets von der des Startpakets unterscheiden und der String, der ausgelesen wurde der Kennung 'Start' entsprechen werden die Daten in das Bytearray des Startpakets geschrieben, die innere Schleife beendet und die äußere Schleife mit den empfangenen Daten ausgeführt. Dann wird überprüft ob die Sessionnr. der des Startpaketes entspricht und ob die neue Paketnummer und die Paketnummer des zuvor empfangenen Pakets gleich sind (Kann z.B. vorkommen wenn Client Ack nicht bekommen hat, siehe Abbildung **??**)

```
if (packageNr == packageNrOld && sessionNrData == sessionNr)
{
        ByteBuffer ACKData = getACK(sessionNrData, packageNr);
        DatagramPacket sendDataACK =
                new DatagramPacket(ACKData.array(),
                ACKData.array().length, IPAddress, port_receive);
        serverSocket.send(sendDataACK);
}
```

Sollte dies der Fall sein werden die neu empfangenen Daten verworfen, und erneut das Ack des davor empfangenen Pakets verschickt.
Andernfalls werden die Daten über dem FileChanel in die Datei geschrieben, und ein Ack an den Client geschickt. Dabei wird dir Größe des Pakets zu einer Variable addiert und dann geprüft ob das letzte Paket empfangen wurde.

```
receivedBytes += PACKAGESIZE;
byte[] data = new byte[PACKAGESIZE];
if (receivedBytes < fileLength)
{
        data = Arrays.copyOfRange(receiveData, 3, 3 + PACKAGESIZE);
        ByteBuffer fileDataBuffer = ByteBuffer.allocate(data.length);
        fileDataBuffer.put(data);
        fileDataBuffer.flip();
        fileChnl.write(fileDataBuffer);
        fileDataBuffer.clear();
        //Sende ACK
        ByteBuffer ACKData = getACK(sessionNrData, packageNr);
        DatagramPacket sendDataACK = new DatagramPacket(ACKData.array(),
                        ACKData.array().length, IPAddress, port_receive);
        serverSocket.send(sendDataACK);
        dataPackageBuffer.clear();
}
/********Falls letztes Datenpaket empfangen wird**************/
else
{
......
if (checksumFile != checksumFileReceived )
{
System.out.print("Fehler im CRC Code der Datei..\n");
sessionNr = 0; //verhindert das für die Wiederholungspakete ein Ack gesendet wird

}
else
```

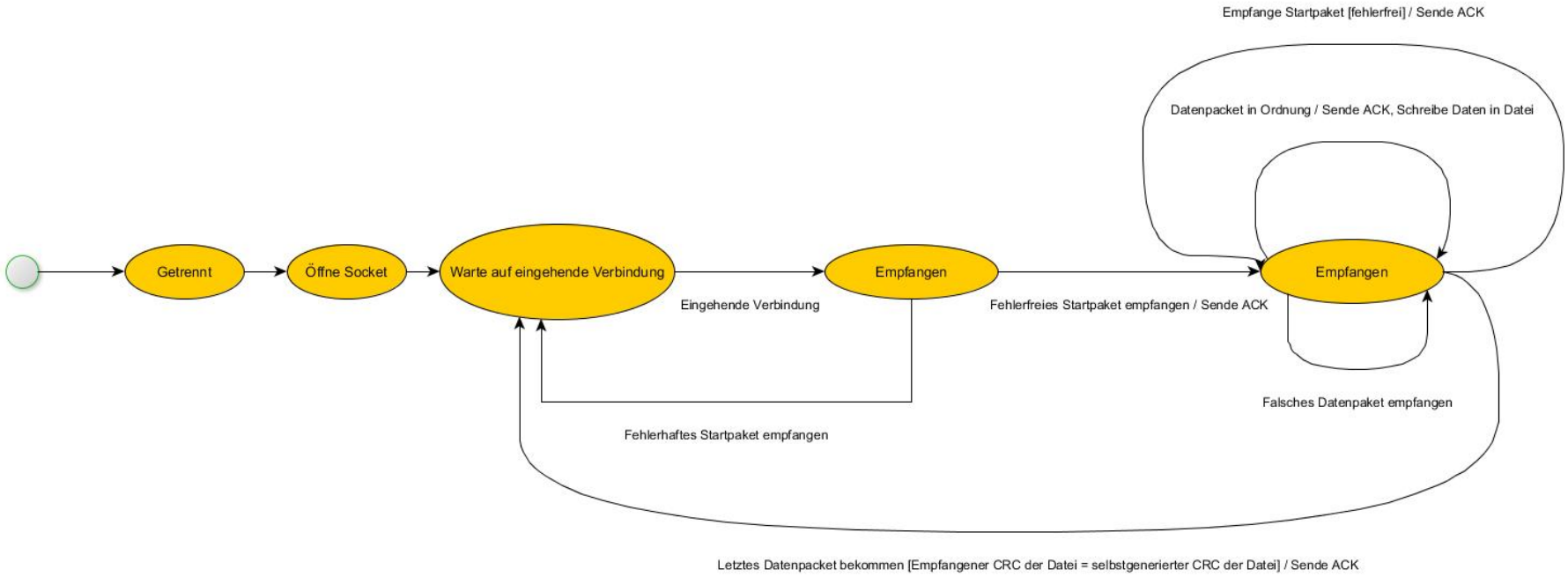
```
29 {
30 //Sende ACK
31 ....
```

Wenn das letzte Paket empfangen wird, wird die else-Anweisung ausgeführt in der, neben dem schreiben der Datei, auch der CRC-Code der empfangenen Datei erzeugt. Wenn die erstellte Prüfsumme mit der im letzen Datenpaket enthaltenen Prüfsumme übereinstimmt wird ein Ack an den Client gesendet. Sollte sie nicht übereinstimmen wird kein Ack gesendet und durch setzen der Sessionnr. auf Null verhindert, für die Wiederholungspakete ein Ack zu senden.

3.2.4 Bekannte Probleme

Der Server ist nur in der Lage mit der Paketgröße des unter **??** beschriebenen Clients zu arbeiten. Um das Empfangen verschieden großer Datenpakete zu ermöglichen und dabei einen Bufferoverflow zu vermeiden müsste der Server ein Bytearray mit der Maximalgröße des UDP-Pakets (65.535 Bytes) erstellen, in dem er die empfangen Daten speichert.

3.2.5 Zustandsdiagramm



4 Verbesserungsvorschläge

1. Prüfsumme für jedes Datenpaket
   - Mit einer Prüfsumme am Ende jedes Datenpakets könnte man Fehler in Paketen frühzeitig erkennen. Bei Fehlern in dem CRC-Code könnte der Server sich das Paket erneut senden lassen und würde keinen falschen Daten in die Datei schreiben.
   - Wenn die Prüfsumme erst bein letzten Paket überprüft wird muss die ganze Datei erneut gesendet werden was besonders bei größeren Dateien sehr lange dauern kann.

Aufgabe 8

Berechnung

Bestimmung des maximalen erzielbaren Durchsatz bei 10% Paketverlust und 10ms Verzögerung mit dem SW-Protokoll.
Gegeben: $P_L = 10$, $RTT = 10ms$, Packetgröße $\equiv$ L = 8192 KBit, $T_w = 10ms$,
Datenrate $\equiv r_d = 10^9$Bit
Berechnung:

$$\eta_{sw} = \frac{T_p}{T_p + T_w} (1 - P_L)^2 * R$$

$$T_p = \frac{L}{r_b}$$

$$R = \frac{1024}{1024 + \underbrace{3}_{\text{Paketheader}} + \underbrace{20}_{\text{IP-Header}}}$$

$$T_p = \frac{8192 + \overbrace{3 \cdot 8}^{\text{Größe des Paketheaders}} + \overbrace{8 \cdot 8}^{\text{Größe es UDP-Headers}} + \overbrace{20 \cdot 8}^{\text{Größe des IP-Headers}}}{\underbrace{1 \cdot 10^9}_{\text{Bei GBit-Lan}}} = 8,376\mu s$$

$$\Rightarrow \eta_{sw} = \frac{8.376\mu s}{8.376\mu s + 10000\mu s} \cdot (1 - 10)^2 \cdot R = 0,00080223$$

Datenübertragungsrate $\equiv r_b = \eta_{sw} \cdot r_d = 8,0223 \cdot 10^{-4} \cdot 10^9 = \underline{\underline{802,226\text{KBit/s}}}$

Bezug zur Praxis

Da im Client einen Timeout benutzt und diesen bei jedem verlorenen Pakete erhöht sind die 337 KBit/s nicht zu erreichen.