NVS Programmierprojekt

Frauenschuh Florian, Lindner Peter, Weilert Alexander

5. Juli 2023

1 Messreihen: Version 1 - Keine Kontrollnachrichten

Für die Messung wurde eine Datei mit genau 10MB verwendet, welche mittels „fsutil file createnew 10MB 10000000“ erstellt wurde. Für die Messungen wurde jeweils die Sende- bzw. Empfangszeit des Initialisierungs- sowie des Finalisierungspakets verwendet.
Die drei Messreihen wurden mit Paketlänge 100, 1400, 60000 Bytes durchgeführt.
Sowohl Receiver als auch Transmitter wurden über localhost ausgeführt.
Im Weiteren werden wir für diese und auch die darauf folgenden Versionen die wichtigsten Ideen der Implementierung erläutern. Dabei werden lediglich kurze Code-Snippets der Java Implementierung dargestellt, da die Python Varianten diesen Code identisch abbilden.

Implementierung des Transmitters

Der Transmitter bietet die Funktionalität um Init-, Daten und auch Finalize-Pakete zu versenden.

```
...
private void sendPacket(Packet packet) throws IOException {
    ...
    // convert packet to byte[]
    byte[] bytes = packet.serialize();

    DatagramPacket udpPacket =
        new DatagramPacket(bytes, bytes.length, receiver, port);
    socket.send(udpPacket);
}
...
```

Dabei wird der Payload dieser Pakete in „Chunks“ aus dem jeweiligen File ausgelesen, der MD-5 Hash berechnet, in ein Datenpaket verpackt und zum Receiver gesendet.

```
...
try (FileInputStream input =
        new FileInputStream(fileToTransfer)) {
    for (ChunkedIterator it =
            new ChunkedIterator(input, chunkSize); it.hasNext(); ) {
        byte[] chunk = it.next();
        Packet dataPacket = new DataPacket(transmissionId,
                                    sequenceNumber++, chunk);
        sendPacket(dataPacket);
        md.update(chunk);
        ...
    }
}
...
```

Implementierung des Receivers

Im Receiver wird nun auf einen bestimmten Port gehorcht.

```
...
DatagramPacket udpPacket = new DatagramPacket(buffer, buffer.length);
...
socket.receive(udpPacket);
...
```

Dabei ist wichtig, dass das ankommende Paket richtig geparsed wird.

```
...
if (PacketInterpreter.isInitializationPacket(udpPacket)) {
    packet = new InitializePacket(udpPacket.getData(),
        udpPacket.getLength());
    ...
} else {
    ...
    // check for finalize or data packet
    if (sequenceNumber == (transmissions.get(transmissionId) + 1)) {
        packet = new FinalizePacket(udpPacket.getData(),
            udpPacket.getLength());
    } else {
        packet = new DataPacket(udpPacket.getData(),
            udpPacket.getLength());
    }
}
...
```

Über die „PacketDigest“ Klasse wird dann abschließend die Datei geschrieben und der im Finalize-Paket übermittelte MD-5 Hash auf Korrektheit geprüft.

Die Usage über die Commandline sieht dabei wie folgt aus

```
TX: <operatingMode> <transmissionId> <ip> <port> <fileName>
<packetSize>
RX: <operatingMode> <transmissionId> <port> <targetFolder>
```

Der Transmitter wurde in Java beziehungsweise Python über

```
java edu.plus.cs.Main 0 1 127.0.0.1 9999 file1 <p-length>
py .\main.py 0 1 127.0.0.1 9999 .\file1 <p-length>
```

und der Receiver jeweils über

```
java edu.plus.cs.Main 0 1 9999 ./
py .\main.py 0 1 9999 ./
```

gestartet.

1.1 Messreihe 1: 100 Byte

Man kann beobachten dass Python → Python die kürzeren und Java → Java die längeren Übertragungszeiten hat. Generell ist die Differenz zischen den einzelnen Messungen nie größer als 304ms. Außerdem wurden alle Dateien erfolgreich übertragen.

1.2 Messreihe 2: 1400 Byte

Wenn man die Paketlänge erhöht, wird die Übertragungszeit wesentlich kürzer. Die Differenz zwischen Python → Python und Java → Java ist hier relativ betrachtet größer als in der Messreihe 1. Man kann auch erkennen dass bei zwei Übertragungen die Datei nicht korrekt übertragen wurde. Das könnte daran liegen, dass der Receiver mit den hohen Übertragungsraten Schwierigkeiten haben könnte. Unabhängig von den angegebenen Messreihen, haben wir Tests mit Delay beim Versenden der Pakete durchgeführt und dabei wurden alle Dateien immer korrekt übertragen. Diese Tests wurden hier nicht angegeben, da diese Verzögerung die Messungen verfälschen würden.

1.3 Messreihe 3: 60000 Byte

Hier ist die relative Differenz zwischen Python → Python und Java → Java ungefähr so groß wie in der zweiten Messreihe. Auch hier ist eine der zehn Übertragungen fehlgeschlagen (vermutlich wegen den selben Gründen wie in der zweiten Messreihe).

Tabelle 1: Paketlänge: 100

Python → Python	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	945	84.66	J
	952	84.03	J
	904	88.50	J
Java → Python	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	1081	74.00	J
	1055	75.83	J
Python → Java	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	1182	67.68	J
	1174	68.14	J
Java → Java	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	1208	71.38	J
	1143	70.00	J
	1104	72.46	J

Tabelle 2: Paketlänge: 1400

Python → Python	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	111	720.72	J
	99	808.08	J
	90	888.89	N
Java → Python	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	159	503.14	J
	155	516.13	N
Python → Java	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	107	747.66	J
	119	672.27	J
Java → Java	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	188	425.53	J
	217	368.66	J
	211	379.15	J

Tabelle 3: Paketlänge: 60000

Python → Python	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	29	2758.62	J
	42	1904.76	N
	19	4210.53	J
Java → Python	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	84	952.38	J
	62	1290.32	J
Python → Java	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	20	4000.00	J
	19	4210.53	J
Java → Java	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	61	1311.48	J
	73	1095.89	J
	64	1250.00	J

2 Messreihen: Version 2 - Stop & Wait

Die zweite Messreihe wurde analog zur ersten Messreihe aufgenommen.

Implementierung des Transmitters

Bei dem Transmitter musste nun durch den neuen Operating-Mode auf ein Acknowledgement nach jedem gesendeten Paket gewartet werden.

```
...
// only check for acknowledgement if the operating mode requires to
if (operatingMode == OperatingMode.STOP_WAIT
        && !handleAcknowledgementPacket()) {
    return;
}
...
```

Implementierung des Receivers

Weiters muss nun im Receiver eine Sende-Funktionalität implementiert werden.

```
...
private void sendAcknowledgementPacket(short transmissionId,
            int sequenceNumber) throws IOException {
    AcknowledgementPacket ackPacket =
        new AcknowledgementPacket(transmissionId, sequenceNumber);

    byte[] bytes = ackPacket.serialize();
```

```
    DatagramPacket udpPacket =
        new DatagramPacket(bytes, bytes.length, this.ackIp, this.ackPort);
    this.socket.send(udpPacket);
}
...
```

Nach dem Erhalt des Acknowledgements kann der Transmitter somit das nächste Paket übertragen.
Die Usage über die Commandline sieht dabei wie folgt aus

```
TX: <operatingMode> <transmissionId> <ip> <port> <fileName>
<packetSize>
RX: <operatingMode> <transmissionId> <ip> <port> <fileName>
<packetSize> <ackPort>
```

Dabei wurde der Transmitter jeweils in Java beziehungsweise Python über

```
java edu.plus.cs.Main 1 1 127.0.0.1 9999 file1 <p-length> 9998
py .\main.py 1 1 127.0.0.1 9999 .\file1 <p-length> 9998
```

und der Receiver jeweils über

```
java edu.plus.cs.Main 1 1 9999 ./ 127.0.0.1 9998
py .\main.py 1 1 9999 ./ 127.0.0.1 9998
```

gestartet.

2.1 Messreihe 1: 100 Byte

In den Messdaten ist ersichtlich, dass in dieser Version mit 100 Byte großen Paketen fast alle Kombinationen in etwa gleich hohe Übertragungsraten erreichen, wobei Java → Java die höchste Übertragungsrate erreicht. In keiner der aufgezeichneten Messungen ist ein Paketverlust aufgetreten. Allgemein fällt natürlich auf, dass die Übertragungsrate durch die Acknowledgements rapide gesunken ist.

2.2 Messreihe 2: 1400 Byte

Auch in dieser Messreihe sind fast alle Kombinationen in etwa gleich schnell, wobei diesmal die Kombination Python → Java die höchste Übertragungsrate erreicht. In keiner der aufgezeichneten Messungen ist ein Paketverlust aufgetreten. Analog zu der Messreihe mit 100 Byte großen Paketen ist auch hier die Übertragungsrate im Vergleich zur NO_ACK Variante stark gesunken.

2.3 Messreihe 3: 60000 Byte

Die Messreihe mit 60000 Byte großen Paketen jedoch weißt eine große Varianz bezüglich der Übertragungsraten auf. Es kann beobachtet werden, dass die Kombinationen in denen der Java Transmitter involviert war die geringsten Übertragungsraten erreicht. In keiner der aufgezeichneten Messungen ist ein

Tabelle 4: Paketlänge: 100

Python → Python	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	10075	7.94	J
	10580	7.56	J
	10478	7.64	J
Java → Python	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	9836	8.13	J
	10072	7.94	J
Python → Java	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	10689	7.48	J
	10154	7.88	J
Java → Java	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	8796	9.10	J
	9191	8.70	J
	9356	8.55	J

Paketverlust aufgetreten. Die Differenz der Übertragungsraten zur NO_ACK Variante sind nun nicht mehr so ausgeprägt, was auch zu erwarten war, da durch die große Paketgröße nicht mehr allzu viele Pakete mit einem Acknowledgement bestätigt werden müssen.

Tabelle 5: Paketlänge: 1400

Python → Python	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	817	97.92	J
	814	98.28	J
	808	99.01	J
Java → Python	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	869	92.06	J
	902	88.69	J
Python → Java	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	749	106.81	J
	775	103.23	J
Java → Java	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	971	82.39	J
	821	97.44	J
	814	98.28	J

Tabelle 6: Paketlänge: 60000

Python → Python	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	56	1428.57	J
	54	1481.48	J
	50	1600.00	J
Java → Python	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	123	650.41	J
	132	606.06	J
Python → Java	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	50	1600.00	J
	56	1428.57	J
Java → Java	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Korrekt?**
	125	640.00	J
	134	579.01	J
	128	625.00	J

3 Messreihen: Version 3 - Sliding Window

Die dritte und somit letzte Messreihe wurde wieder analog zu den beiden vorhergehenden Messreihen aufgenommen.

Implementierung des Transmitters

Bei dem Transmitter war es nun notwendig, über den neuen Operating-Mode auf ein cumulative Acknowledgement oder auf eines oder mehrere duplicate Acknowledgments nach Übertragung eines Windows zu achten. Somit wurde nach jeder Übertragung eines Pakets auf solche geprüft, wenn ein Window komplett übertragen wurde

```
...
// check for SLIDING_WINDOW acknowledgement
if (operatingMode == OperatingMode.SLIDING_WINDOW
    && windowBuffer.size() == windowSize) {
    while(!handleSlidingWindowAcknowledgement());
}
...
```

Implementierung des Receivers

Die Erweiterung des Receivers war jedoch etwas aufwendiger. Hierbei musste der Socket über ein Timeout erweitert werden, welcher erreicht wird, sobald der Transmitter mit der Übertragung eines Windows fertig geworden ist und somit keine neuen Pakete mehr am Receiver ankommen.

```
...
try {
    socket.receive(udpPacket);
} catch (SocketTimeoutException ex) {
    // transmission of window is complete
    if (operatingMode == OperatingMode.SLIDING_WINDOW) {
        // check if window is ready for processing
        if ( ... ) {
            processWindow();
        } else {
            // check for missing packets inside the window
            checkForMissingPackets();

            if (!missingPackets.isEmpty()) {
                // request a lost packet
                requestMissingPacket(missingPackets.pop());
            }
        }
```

```
            continue;
        }
    }
...
```

Dadurch weiß der Receiver nun, dass er das Window vollständig erhalten hat. Sollten Pakete dieses Windows fehlen, dann werden diese über ein duplicate Acknowledgement angezeigt und vom Transmitter angefordert.

Die Usage über die Commandline sieht dabei wie folgt aus

TX: <operatingMode> <transmissionId> <ip> <port> <fileName> <packetSize> <ackPort> <windowSize>
RX: <operatingMode> <transmissionId> <port> <targetFolder> <ackIp> <ackPort> <windowSize> <windowTimeout> <dupAckDelay>

Der Transmitter wurde beispielsweise in Java beziehungsweise Python über

```
java edu.plus.cs.Main 2 1 127.0.0.1 9999 file1 <p-length> 9998 100
py .\main.py 2 1 127.0.0.1 9999 .\file1 <p-length> 9998 100
```

und der Receiver jeweils über

```
java edu.plus.cs.Main 2 1 9999 ./ 127.0.0.1 9998 100 5 2
py .\main.py 2 1 9999 ./ 127.0.0.1 9998 100 5 2
```

gestartet.

Besonders wichtig hierbei sind das Window Delay mit je 5ms und die Window Size mit 100.

3.1 Messreihe 1: 100 Byte

In den Messdaten ist ersichtlich, dass in dieser Version mit 100 Byte großen Paketen die Übertragungsrate zwischen den unterschiedlichen Implementierungen stark variiert. Dabei ist Python → Python die schnellste Variante und übertrifft die Übertragungsrate von Version 2. Hierbei ist auch zu beachten, dass bei einer solchen Paketgröße und der übertragenen Datei über 1000 Windows übertragen werden, und diese somit mit einem Window-Delay von 5ms alleine schon 5 Sekunden der Übertragungszeit einnehmen. Ergo ist die Übertragung selbst somit um einiges schneller wie in der Stop & Wait Version der Implementierung. In keiner der aufgezeichneten Messungen ist ein Paketverlust aufgetreten.

3.2 Messreihe 2: 1400 Byte

Bei dieser Messreihe ist auffällig, dass die Messungen mit dem Java Receiver bereits langsamer sind. Zusätzlich treten bei der Java → Java Variante auch Paketverluste auf, die jedoch logischerweise dem Protokoll zufolge nachgeladen

Tabelle 7: Paketlänge: 100

Python → Python	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Retransmitted Pakete**	**Korrekt?**
	8828	9.06	0	J
	8593	9.31	0	J
	8693	9.20	0	J
Java → Python	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Retransmitted Pakete**	**Korrekt?**
	28796	2.78	0	J
	14249	5.61	0	J
Python → Java	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Retransmitted Pakete**	**Korrekt?**
	16448	4.86	0	J
	16466	4.86	0	J
Java → Java	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Retransmitted Pakete**	**Korrekt?**
	16453	4.86	0	J
	16476	4.86	0	J
	16457	4.86	0	J

werden. Die Übertragungsraten der Python Varianten sind vergleichbar mit denen der Stop & Wait Versio, dabei muss wieder in Betracht gezogen werden, dass bei einer Window Size von 100 und einer Paketgröße von 1400 Byte über 70 Windows übertragen werden, sich also das Window Delay dementsprechend aufsummiert. Ergo kann wieder festgehalten werden, dass die Übertragungsrate ohne diesem Delay somit um einiges höher ist wie bei der Stop & Wait Version.

3.3 Messreihe 3: 60000 Byte

In der letzten Messreihe kann beobachtet werden, dass wieder die Messungen mit dem Java Receiver zu einer langsamen Übertragungsrate führten, da unter Anderem einige Pakete verloren gegangen sind und somit nachgeladen werden mussten. Interessanterweise liefern die Python Implementierungen unter Einbeziehung des Window Delay eine ähnliche Übertragungsrate wie in der Stop & Wait Version, eine Annäherung war aber aufgrund der sinkenden Paketanzahl zu erwarten.

Tabelle 8: Paketlänge: 1400

Python → Python	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Retransmitted Pakete**	**Korrekt?**
	767	104.30	0	J
	760	105.26	0	J
	697	114.78	0	J
Java → Python	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Retransmitted Pakete**	**Korrekt?**
	1112	71.94	0	J
	705	113.48	0	J
Python → Java	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Retransmitted Pakete**	**Korrekt?**
	1144	69.93	0	J
	1138	70.30	0	J
Java → Java	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Retransmitted Pakete**	**Korrekt?**
	1319	60.65	10	J
	1290	62.02	9	J
	2005	39.90	54	J

Tabelle 9: Paketlänge: 60000

Python → Python	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Retransmitted Pakete**	**Korrekt?**
	78	1025.64	0	J
	68	1176.47	0	J
	65	1230.77	0	J
Java → Python	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Retransmitted Pakete**	**Korrekt?**
	63	1269.84	0	J
	68	1176.47	0	J
Python → Java	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Retransmitted Pakete**	**Korrekt?**
	331	241.69	15	J
	345	231.88	16	J
Java → Java	**Zeit (ms)**	**Datenrate (Mbit/s)**	**Retransmitted Pakete**	**Korrekt?**
	543	147.33	27	J
	529	151.23	27	J
	327	244.65	15	J