
Rechnernetze/Kommunikationssysteme

Dokumentation zur Belegarbeit

vorgelegt von

Leonard Hecker

Inhaltsverzeichnis

1	Funktionsweise des Servers	3
2	Funktionsweise des Clients	7
3	Theoretische und praktische Geschwindigkeiten	10
3.1	Theoretische Geschwindigkeit	10
3.2	Praktische Geschwindigkeit.	12
4	Verbesserungsvorschläge für das Protokoll	14

1 Funktionsweise des Servers

Nach der Erstellung eines UDP Sockets befindet sich der Server im sog. *main loop* und wartet dort zu Beginn bis ein Paket empfangen wird in der *receive* Methode des Sockets. Wie im Protokoll festgelegt muss das erste Paket, einem Handshake entsprechen, weshalb alle Felder auf Konformität geprüft werden. Neben einem Test der die Einhaltung der Grenzwerte von jedem Feld prüft muss auch die CRC32 Summe über alle Handshake-Felder, abgesehen vom CRC32 Feld selbst berechnet werden. Ist die Prüfsumme die gleiche wie die im CRC32 Feld des Handshakes, so sendet der Server nun ein ACK an den Client. Ist eines der Felder jedoch invalide, oder aber die Prüfsumme ungleich, so wird kein ACK gesendet und an den Anfang des Programms gesprungen, wo der Server wieder auf einen validen Handshake wartet.. Zunächst erstellt der Server nun eine Datei mit dem im Handshake spezifizierten Namen. Sollte die Datei bereits existieren, dann werden wie in der Aufgabenstellung gefordert alternative Namen getestet. Wurde ein Name gefunden, so erstellt der Server damit eine neue, leere Datei, sendet dem Client ein ACK und springt in den *data loop*. Tritt ein Fehlerfall im *data loop* ein, so wird die aktuelle Verbindung abgebrochen, die bereits erstellte Datei gelöscht und wieder zu dem Beginn des *main loop* gesprungen. Da zum Beginn des *main loop* auf einen validen Handshake getestet wird und für invalide kein ACK gesendet wird, wird der Client nach 10 weiteren *send*-Versuchen ohne ACKs durch einen Timeout abbrechen.

Auch am Anfang des *data loop* wartet der Server wieder in der *receive* Methode des Sockets, bis ein Paket empfangen wird, oder ein Timeout eintritt. Ein Timeout entspricht hierbei einem Fehlerfall. Wurde ein Paket vom aktuellen Client, mit der selben IP, dem selben Port und der selben

Sessionnummer, empfangen, so wird es wieder auf Konformität zum Protokoll getestet. Stammt es von einem anderen Client, dann wird es ignoriert, jedoch nicht das Timeout zurückgesetzt. Würde man dies tun, könnte man eine abgebrochene Übertragung beliebig lange mit Paketen anderer Clients in der *data loop* Schleife festhalten. Um die Validität des Paketes festzustellen, muss zunächst getestet werden ob die Sessionnummer die gleiche wie die im Handshake vereinbarte ist. Ist sie nicht gleich, so ist dies ein Fehlerfall. Wenn der Client ein ACK des Servers nicht erhalten hat, sendet dieser ein Datenpaket noch einmal, weshalb an dieser Stelle getestet werden muss, ob es sich bei der Paketnummer des Pakets um eine abgelaufene hält. Falls dies der Fall ist ignoriert der Server den Paketinhalt, sendet das ACK für das vergangene Paket noch einmal und setzt am Anfang des *data loop* fort. Andernfalls kann man davon ausgehen, dass es sich um das nächste, neue Paket handelt, weshalb der Server nun das Timeout zurücksetzt. Sollte das Paket mehr Daten enthalten, als noch geschrieben werden müssen, dann ist dies auch ein Fehlerfall. Diese Menge der noch fehlenden Bytes lässt sich leicht aus der im Handshake angegebene Dateigröße und der Menge an bereits geschriebenen Bytes ermitteln. Die Daten des Paketes, sofern welche vorhanden sind, werden nun in die bereits erstellte Datei geschrieben. Ist die Menge der im Handshake vereinbarten Daten bereits geschrieben worden, so müssen in diesem Paket exakt 4 weitere Byte vorhanden sein. Diese 4 Byte entsprechen dem CRC32 Feld im Protokoll und müssen mit der CRC32 Prüfsumme aller geschriebenen Daten übereinstimmen. Eine ungleiche Prüfsumme gilt als Fehlerfall. Ansonsten sendet der Server nun für das aktuelle Paket ein ACK und setzt falls noch Daten oder das CRC32 Feld fehlen im *data loop* und andernfalls im *main loop* fort.

Um Netzwerke zu simulieren enthält der Server Funktion um Delay, Delay-Variation und Loss einzustellen. Delay entspricht hierbei der mittleren Verzögerung in jeweils Sende- und Empfangsrichtung. Mittels Delay-Variation kann man hierbei dem Delay eine scheinbare, um den Delay-Wert normalverteilte, Zufälligkeit beimischen. Diese Verzögerung

wird hierbei mittels einem Aufruf von *sleep* auf dem Main-Thread erreicht und betrifft sowohl Sende- als auch Empfangsrichtung. Ein Delay von *100ms* führt somit zu einer RTT von *200ms*. Zuletzt ist es mittels Loss möglich Pakete mit einer bestimmten Wahrscheinlichkeit nicht zu empfangen, bzw. zu senden. Wurde entschieden ob ein Paket als Loss gilt wird dieses einfach ignoriert und im Falle eines zu sendenden ACKs nichts unternommen, bzw. beim empfangen auf das nächste gewartet.

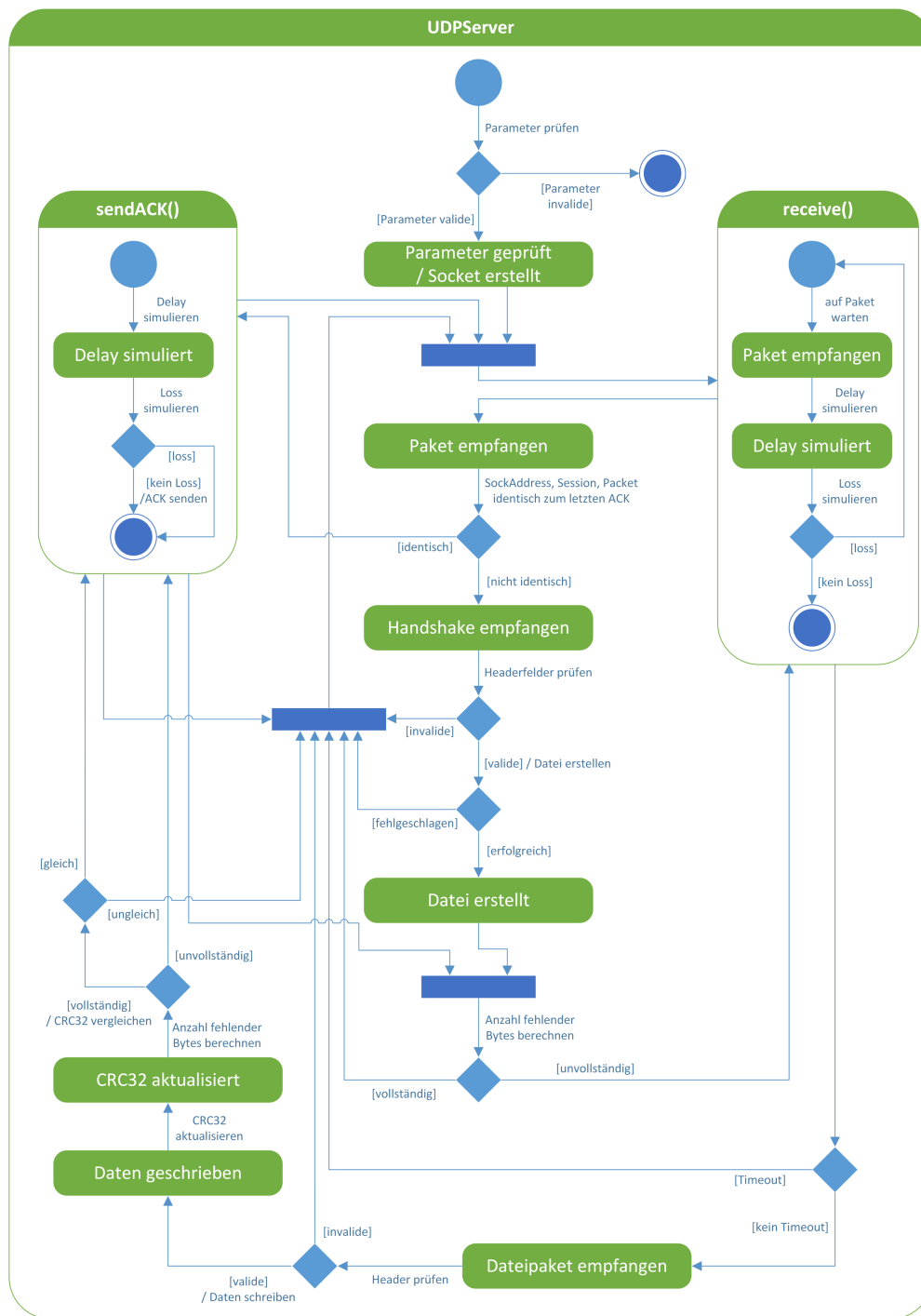


Abbildung 1.1: Zustandsdiagramm einer Dateiübertragung

2 Funktionsweise des Clients

Zunächst ist festzulegen, dass der Client bei einem Fehlerfall die Ausführung abbricht. Sollte der angegebene Host und Port nicht auflösbar, oder die Datei unter dem angegebenen Namen nicht zu finden sein, dann ist dies ein Fehlerfall. Da es für eine optimale Geschwindigkeit nötig ist die MTU bis zum Server zu kennen, wäre eine MTU Path Discovery nötig. Da dies jedoch unter Java nahezu unmöglich ist wird die MTU hier angenähert. Dies ist möglich indem man das minimum der MTU aller Network Interfaces findet. Sollte dies jedoch fehl schlagen oder eine ungewöhnlich kleine MTU gefunden werden, so wird eine MTU von 576 Bytes angenommen. Dies entspricht der im RFC 1122 als EMTU_R festgelegten für Hosts kleinstmögliche MTU. Da jedoch eine MTU die Größen von IPv4/v6-Headern usw. nicht einbezieht, müssen wir noch z.B. 48 Byte abziehen um die maximale Nutzdatengröße zu erhalten, was der Größe eines UDP Headers in einem IPv6 Header entspricht. Zunächst wird ein Handshake-Paket gemäß dem vorgegebenen Protokoll erstellt. Die Felder für die Länge des Dateinamens und der Dateiname selbst werden selbstverständlich mit dem Basename der bereits zu Beginn gefundenen Datei ausgefüllt. Das Feld für den Dateinamen hat hierbei eine dynamische Größe und passt sich der Länge des Dateinamens an. Das für die CRC32 Prüfsumme wird hingegen gemäß Protokoll mit der Prüfsumme über alle Handshake-Felder außer dem des CRC32-Feld's selbst ausgefüllt. Ist das Handshake-Paket fertig so wird dieses an den Server geschickt und in den sog. *data loop* gesprungen.

Die *send* Methode des Clients handhabt hierbei das Versenden von Daten. In dieser Methode ist abstrahiert dabei das geforderte

Verhalten, dass ein Paket bis zu 10 mal gesendet werden soll, falls kein ACK eintrifft. Dies bedeutet, dass die *send* Methode somit auch das empfangen von ACKs und die Berechnung von geeigneten Timeouts handhabt. Das Timeout wird mittels einem Algorithmus berechnet, der dem des Retransmission Timer aus dem TCP Protokoll ähnlich ist. Hier wird jedoch mit einer feineren Granulität von Millisekunden statt ganzen Sekunden gearbeitet. Der minimale Timeoutwert liegt des Weiteren bei nur *10ms* und der maximale nur bei *3000ms*. Dies ergibt sich daraus, dass dieses Projekt nur zu Demonstrationszwecken verwendet wird. Tritt 10 mal hintereinander ein Timeout ein, so ist dies ein Fehlerfall. Enthält das ACK des Servers eine invalide Sessionnummer, so ist das ein Fehlerfall. Falsche Paketnummern werden hingegen als ein Timeout bewertet.

Sollte der Client schneller senden, als der Server die Pakete aktuell empfängt, so würden sich diese aufstauen. Da der Server für jedes empfangene Paket ein ACK sendet, so wird der Client nun für jedes aufgestaute Paket ein ACK erhalten. Da der Client für jedes invalide Paket nun ein ACK senden würde, würde dieser wiederum genausoviele Pakete senden. Man könnte so durch einen schnell sendenden Client also eine immer größere Anzahl an aufgestauten Paketen provozieren.

Im *data loop* wird nun zunächst der Header eines Datenpaketes gemäß dem Protokoll erstellt. Dem Datenpaket wird nun die größtmögliche Menge an Daten, welche sequentiell aus der Datei ausgelesen wurden, angehängen. Dies geschieht unter Beachtung der bereits ermittelten maximalen Payloadgröße und der Bedingung des Protokolls, dass die CRC32 Prüfsumme am Ende der Datei nicht auf mehrere Pakete aufgeteilt werden darf. Die jeweilige Menge an Daten werden nun auf eine CRC32 Prüfsumme aufgerechnet. Wurde die gesamte Datei in Datenpakete aufgeteilt wird nun die berechnete CRC32 Prüfsumme an das Ende des Paketes angehängen — es ist hierbei auch möglich dass das finale Paket nur mit CRC32 und ohne Daten gesendet wird.

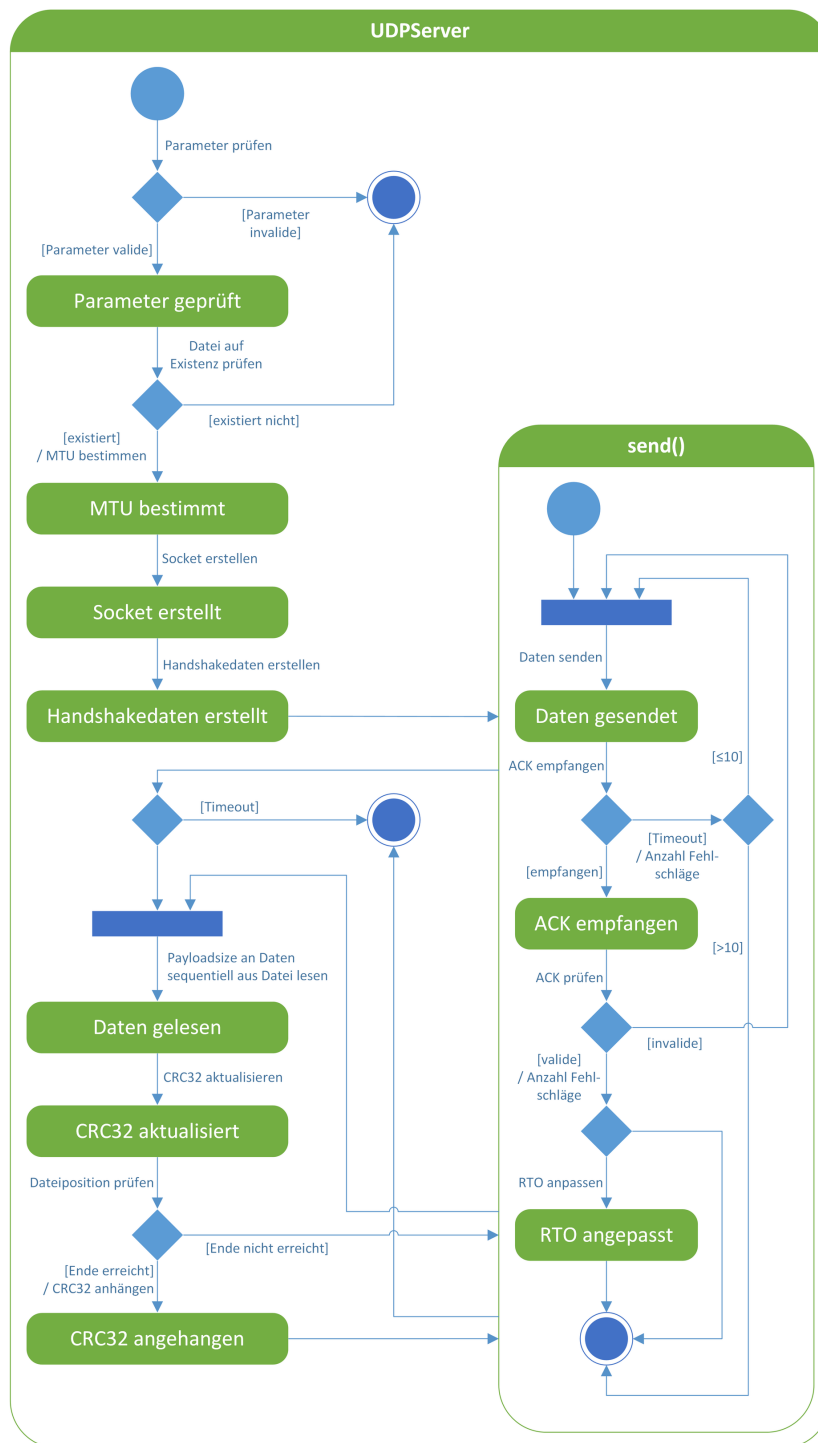


Abbildung 2.1: Zustandsdiagramm einer Dateiübertragung

3 Theoretische und praktische Geschwindigkeiten

3.1 Theoretische Geschwindigkeit

In den folgenden Berechnungen wird eine Verbindung zwischen *ilux150* und dem 10%/10ms Server auf *idefix* der HTW Dresden mit folgenden Eigenschaften angenommen:

- MTU $L = 1500Byte$
- Länge des IP-Headers $L_{IP} = 20Byte$
- Länge des UDP-Headers $L_{UDP} = 8Byte$
- Länge des Datenpaket-Headers $L_{DATA} = 3Byte$
- Datenrate $r_b = 1GBit/s$
- Ausbreitungsverzögerung $T_a = 10ms$ (in beide Richtungen)
- Verlustwahrscheinlichkeit $P_v = 10\%$ (in beide Richtungen)

Daraus ergeben sich folgende Parameter:

$$\begin{aligned}
T_p &= \frac{L}{r_b} = 12\mu s \\
T_{ACK} &= \frac{L_{ACK}}{r_b} = \frac{31Byte}{1GBit/s} = 248ns \\
k &= L - L_{IP} - L_{UDP} - L_{DATA} \\
&= 1500Byte - 20Byte - 8Byte - 3Byte = 1469Byte \\
R &= \frac{k}{L} = \frac{1469Bytes}{1500Byte} \approx 0,98
\end{aligned}$$

Die Effizienz η_{sw} und die Übertragungsrate r_{sw} können daraus wie folgt berechnet werden:

$$\begin{aligned}
\eta_{sw} &= \frac{T_p \times R \times (1 - P_v)^2}{T_p + 2 \times T_a + T_{ACK}} \\
&= \frac{12\mu s \times 0,98 \times (1 - 10\%)^2}{12\mu s + 2 \times 10ms + 248ns} \\
&\approx 4,757 \times 10^{-4} \\
r_{sw} &= r_b \times \eta_{sw} \approx \underline{\underline{475,7kBit/s}}
\end{aligned}$$

3.2 Praktische Geschwindigkeit

In der Praxis ließen sich stark schwankende Geschwindigkeiten beobachten. Bei Tests zwischen einem Client auf *ilux150* und dem 10%/10ms Server auf *idefix* schwankten so die Ergebnisse zwischen rund 80kBit/s und 800kBit/s . Am häufigsten ließ sich jedoch das folgende Ergebnis mit ca. 789kBit/s reproduzieren:

```
Thu Jan 15 12:59:47 CET 2015
Server v1.05: Waiting for new connection...
Server: Start packet received
IP / Port: /141.56.132.150 48501
Start ID:-24587, Length: 4932736 Bytes, Filename: sublime_text
CRC Data StartPacket:c9de66ce CRC field:c9de66ce
Thu Jan 15 13:18:03 CET 2015
Server: All data received
Server: CRC (received data): 8f7290c3
Server: CRC (source data) : 8f7290c3
CRC OK
Server: Transmission time: 50.0s, data rate: 789.23776 kbit/s
Server Info: Server receive timed out - Finish...
```

Eine Erklärung für solch hohen praktischen Ergebnissen über den theoretischen Maximum kann leider nicht ohne Weiteres geliefert werden. Da man aber annehmen kann, dass der Client bei einer *vollständigen* Übertragung die simulierte Paketverzögerung des Servers nicht austricksen können sollte, könnte dies durch einen Bug in der Serverimplementierung verursacht worden sein.

Eigentlich sollten jedoch die Werte etwas unter dem theoretischen Maximum liegen. Dies könnte man dann z.B. dadurch begründen, dass der *idefix* Server mit einer Paketverzögerung von 10ms konfiguriert ist, jedoch noch zusätzlich eine Verzögerung durch das Ethernet zwischen *idefix* und *ilux150* existiert. Des Weiteren sollte die Berechnung des Retransmission Timeouts (RTO) bei Paketverlust inoptimal sein. Die Paketverzögerung ist nämlich bei den hier durchgeführten Tests relativ stabil – der RTO wird jedoch bei Timeouts verdoppelt. Bei 2 oder mehr aufeinanderfolgenden Paketverlusten führt dies somit zu starken Verzögerungen in der Erkennung des Verlustes,

obwohl die Paketverzögerung weiterhin stabil bei z.B. $10ms$ bleibt. Die für diese Belegarbeit entwickelte Software addiert deshalb bei den ersten beiden Timeouts nur $RTTVAR$ auf RTO , wodurch eine kleine Performanceverbesserung eintrat. Zuletzt wurde hier ein im Ethernet zwischen *ilux150* und *idefix* auftretender Paketverlust ebenfalls nicht betrachtet.

4 Verbesserungsvorschläge für das Protokoll

Die Header für den Handshake als auch für Datenpakete sollten mit der selben Struktur beginnen und ein Feld für den Typ des Pakets enthalten. Um zu Verhindern dass eine Übertragung von einem fremden Client gestohlen wird, muss getestet werden ob es sich um den gleichen Absender handelt, der auch den Handshake durchgeführt hat. Da 2^{16} parallele Sessions eines Client unwahrscheinlich sind könnte man also die Sessionnummer auf *8Bit* limitieren. Die nun übrigen *8Bit* könnte man dann als Typ-Feld verwenden. Dieses würde nun die Art der Nachricht identifizieren, womit man nun ein Handshake eindeutig von einem Datenpaket unterscheiden kann. Das ist umso wichtiger wenn man bedenkt, dass theoretisch ein Datenpaket existiert, welches gleichzeitig ein valider Handshake ist.

Des Weiteren wäre es günstig, wenn das Feld für die Paketnummer mindestens 4 verschiedene Zustände erlauben würde. Dadurch wäre der Server in der Lage sowohl ein vorheriges, also ein vom Client mehrfach gesendetes Paket, sowie das aktuelle Paket zu erkennen. Eine vorvorherige, sowie eine zukünftige Paketnummer könnten somit als Fehler zuverlässlich erkannt werden.

Darüber hinaus könnte man selbstverständlich viele Ideen aus dem TCP Protokoll übernehmen. Hierunter fällt unter anderem, dass ein Sliding Window eine höhere Performance erreichen würde, als Stop-and-Wait.