Beleg Dateitransfer

Dokumentation

Lenny Reitz

15. Januar 2021

Inhaltsverzeichnis

1 Überblick

Diese Java-Konsolenanwendung ermöglicht den Austausch von Dateien zwischen zwei Hosts mittels UDP. Ein verlustfreier Datenverkehr wird über das Stop-And-Wait Protokoll gewährleistet. Die Anwendung ist zweigeteilt in Server und Client. Je nach Art des Aufrufs lässt sich das Programm entweder als Server oder als Client starten.

Startet man die Anwendung als Server, so ist nur die Angabe einer Portnummer notwendig und das Programm wartet auf diesem Port auf mögliche Dateianfragen. Wird die Anwendung als Client gestartet, muss die Zieladresse des Servers (Hostname oder IP), Portnummer und ein Dateipfad angegeben werden. Der Client wird dann versuchen, die angegebene Datei an den Server zu schicken.

Das Programm entstand im Rahmen des Rechnernetze Belegs.

2 Installation

1. Kompilieren:
   - *make.sh* ausführen, um Binärdateien zu erstellen: `./make.sh`
2. Server ausführen:
   - *filetransfer* als Server ausführen:
     `./filetransfer server <port> [<loss_rate> <avg_delay>] ['debug']`
   * `port` - Integer-Wert über (üblicherweise über 1000) gibt an, auf welchem Port der Server auf Anfragen wartet
   * `loss_rate` - (optional für Testumgebung) Double-Wert zwischen 0.0 - 1.0 gibt an, mit welcher Wahrscheinlichkeit der Server ein verloren gegangenes Paket simuliert.
   * `avg_delay` - (optional für Testumgebung) Integer-Wert in ms gibt an, mit welcher mittleren Verzögerung der Server die Ankunft und Antwort von Paketen simuliert
   * `'debug'` - (optional) startet den Server im Debug-Modus. Es werden mehr Ausgaben in der Konsole angezeigt
3. Client ausführen:
   - *filetransfer* als Client ausführen:
     `./filetransfer client <host_name/ip> <port> <file_name> ['debug']`
   * `host_name/ip` - Adresse des Servers (Hostname oder IP), an die eine Datei gesendet werden soll
   * `port` - Integer-Wert über (üblicherweise über 1000) gibt an, auf welchem Port der Server auf Anfragen wartet
   * `file_name` - Dateiname bzw. Dateipfad der Datei, die versendet werden soll
   * `'debug'` - (optional) startet den Client im Debug-Modus. Es werden mehr Ausgaben in der Konsole angezeigt

2.1 Beispielaufruf

Ein besipielhafter Aufruf des Programms, um die Datei `text.txt` auf demselben Computer an den Ausführungsort des Servers zu schicken:

1. Server: `./filetransfer server 1234 0.1 150 debug`
2. Client: `./filetransfer client localhost 1234 test.txt debug`

3 Funktionsweise

Die Anwendung ist zweigeteilt in Server und Client. Je nach Aufruf des Programms, startet es entweder als Server oder als Client. Das Programm arbeitet in 3 Schichten, welche hierarchisch aufgebaut sind und dadurch verschiedene Funktionen und Arbeitszyklen ermöglichen.

Die Anwendungsschicht, als oberste Schicht, stellt das Grundgerüst und den Startpunkt der Anwendung dar. Die darunterliegende FileTransfer-Schicht übernimmt die lokale Dateiarbeit. Die Stop-and-Wait-Schicht, als unterste Schicht, übernimmt den eigentlichen Dateitransfer. Sie stellt das Bindeglied zu den UDP-Sockets dar und verwaltet die Datenübertragung nach dem Stop-And-Wait-Protokoll.

Darüber hinaus gibt es noch einige Nebenfunktionen, welche sich zum Testen und Debuggen der Anwendung eignen (Debug-Modus, Berechnung der Datenrate und Testumgebung des Servers) und welche die Dateiübertragung effizienter gestalten (variables Timeout).

3.1 Server

Der Server hört ununterbrochen auf dem zuvor angegebenen Port auf mögliche Anfragen. Eine Anfrage erreicht den Server in Form eines Startpakets, welches Metadaten über die zu übertragende Datei enthält. Da diese Informationen ausschlaggebend für die weitere Datenübertragung sind, werden diese vom Client mit einer CRC versehen. Der Server überprüft die Richtigkeit des Startpakets anhand dieser CRC. Falls das empfangene Startpaket korrekt ist, schickt der Server eine entsprechende Antwort (in Form eines ACK-Pakets) an den Client und wartet im nächsten Schritt auf Datenpakete.

Erhält der Server ein Datenpaket, überprüft er die Richtigkeit des Paketes anhand der alternierenden ACKs im Stop-And-Wait-Protokoll. Handelt es sich um das nächste erwartete Datenpaket, schreibt der Server die erhaltenen Daten in einen Stream und schickt ein entsprechendes ACK-Paket an den Client zurück.

Anhand der bisher erhaltenen Datenmenge und den Metadaten des Startpakets kann der Server bestimmen, welches Paket das letzte Datenpaket ist. Im letzten Datenpaket befindet sich noch eine CRC über die gesamte gesendete Datei. Der Server überprüft die CRC mit der der erhaltenen Daten und schickt nur bei Übereinstimmung ein letztes ACK-Paket an den Client.

Bekommt der Server ein Datenpaket, welches nicht erwartet wurde, sendet er sein vorheriges ACK-Paket erneut. Wartet der Server während der Übertragung der Datenpakete zu lange auf das nächste Paket vom Client, bricht er die Übertragung ab und wartet auf ein neues Startpaket.

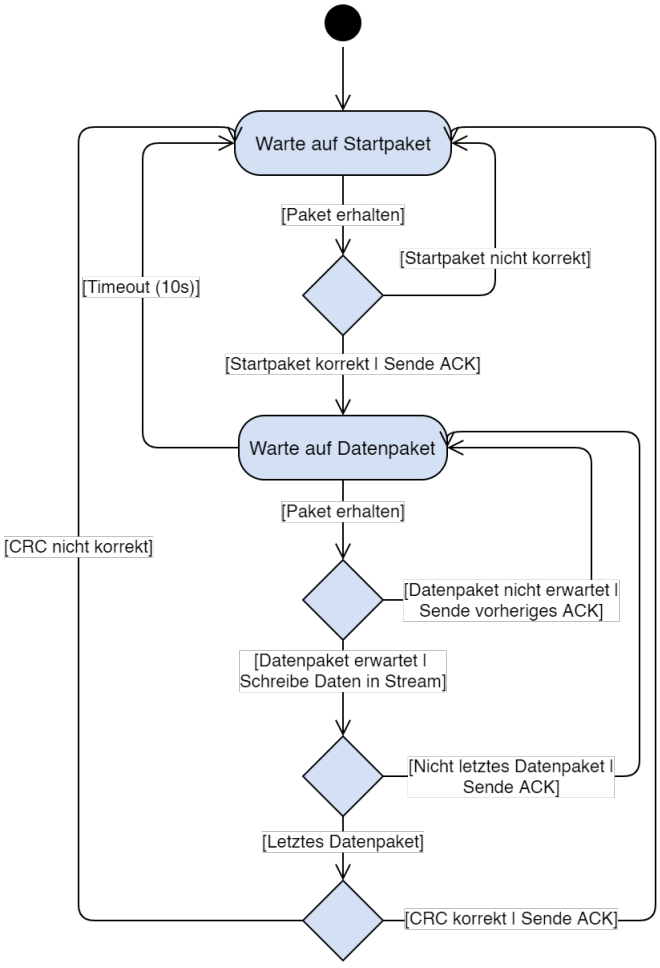

Abbildung 1: Zustandsdiagramm Server

3.1.1 Testumgebung

Der Server kann über die optionalen Parameter `loss_rate` und `avg_delay` eine Ausfallwahrscheinlichkeit und eine mittlere Verzögerung der erhaltenen Datenpakete und der zu sendenden ACK-Pakete simulieren.

Beispielhaft wird der Server mit 10% Paketverlust und 10 ms Verzögerung gestartet: `./filetransfer server 1234 0.1 10`

3.1.2 Ermittlung des Durchsatzes

Theoretisch kann der maximal erzielbare Durchsatz beim Stop-And-Wait-Protokoll folgendermaßen berechnet werden:

$$\eta_{sw} = \frac{T_p}{T_p + T_w}(1 - P_{de})(1 - P_{rü})R$$

- **Übertragungsverzögerung** T_p

  Die Übertragungsverzögerung berechnet sich aus der Paketlänge L und der Übertragungsrate r_b. In diesem Fall ist $L = 1400$ Byte und r_b wird mit 50 MBit/s (entspricht 6.25 MB/s) angenommen:

  $T_p = L/r_b = \frac{1.4\,\mathrm{KB}}{6250\,\mathrm{KB/s}} = 224\,\mu\mathrm{s}$

- **Warteverzögerung** T_w

  Die Warteverzögerung beschreibt die Verzögerung zwischen dem Senden von Datenpaketen. Sie wird berechnet mit $T_w = 2T_a + T_ACK$, wobei die Ausbreitungsverzögerung T_a hier mit 10 ms festgelegt ist. T_ACK beschreibt die Übertragungsverzögerung des ACK-Paketes (mit $L = 3$ Byte) und kann hier vernachlässigt werden, da $0.003\,\mathrm{KB}/6250\,\mathrm{KB/s} \approx 0$. Daher ergibt sich für T_w:

  $T_w \approx 2T_a = 20\,\mathrm{ms}$

- **Fehlerwahrscheinlichkeiten** P_{de}**,** $P_{rü}$

  Die Fehlerwahrscheinlichkeiten P_{de}, $P_{rü}$ beschreiben die Wahrscheinlichkeit eines Paketverlustes auf dem Hin- bzw. Rückkanal. Die Fehlerwahrscheinlichkeit wird hier vom Server mit 10% simuliert und gilt gleichermaßen für Hin- und Rückkanal:

  $P_{de} = P_{rü} = 0.1$

- **Coderate** R

  Die Coderate eines Datenpakets gibt dessen Informationsanteil an. In diesem Fall besteht ein Datenpacket aus 1400 Byte, von denen 1397 Byte verwertbare Daten beinhalten. Die Coderate ist daher:

  $R = 1397/1400 = 0.9979$

Damit ergibt sich ein theoretisch maximal erzielbarer Durchsatz von:

$$\eta_{sw} = \frac{0.224\,\text{ms}}{0.224\,\text{ms} + 20\,\text{ms}}(1-0.1)(1-0.1)\cdot 0.9979 = 0.00895 \approx 1\%$$

Normierter Durchsatz:

$$r_{d_t} = \eta_{sw} \cdot r_b = 0.00895 \cdot 6250\,\text{KB/s} = \underline{55.94\,\text{KB/s}}$$

Im praktischen Versuch mit den eingestellten Parametern verändert sich bei der Bestimmung des Durchsatzes nur T_w. Nach 5 Versuchen liegt der praktische Durchsatz im Mittel bei 25.8 KB/s. Da es sich um dieselbe Konfiguration handelt, wie theoretisch angenommen, können anhand des Durchsatzes Rückschlüsse auf das praktische T_w gezogen werden:

$$T_w = \frac{T_p \cdot r_b \cdot R \cdot P_{rü} \cdot P_{de}}{r_{d_p}} - T_p = 44\,\text{ms}$$

Dieses deutlich größere T_w kommt zustande, da zu der simulierten Ausbreitungsverzögerung des Servers auch die Timeout-Dauer des Clients bei Ausfall eines Pakets hinzukommt, in der die Übertragung stillsteht. In der theoretischen Betrachtung fällt dieser Faktor weg. Theoretisch wird lediglich einbezogen, ob ein Paket ausfällt, aber nicht wie lange der Client benötigt, um diesen Ausfall zu behandeln.

3.2 Client

Der Client verschickt zunächst ein Startpaket an den angegebenen Server, in dem Metadaten der zu übertragenden Datei und die CRC des Pakets enthalten sind. Der Server antwortet nach Erhalt des Startpakets mit einem entsprechenden ACK-Paket.

Jetzt sendet der Client stückweise Datenpakete, die die zu übertragende Datei ergeben. Erst nachdem der Server ein Datenpaket entsprechend bestätigt hat, wird das nächste versendet. Im letzten Datenpaket fügt der Client noch eine CRC über die gesamte gesendete Datei an den Server. Wird auch dieses letzte Paket bestätigt, war die Übertragung erfolgreich, der Client wird beendet.

Empfängt der Client ein falsches ACK-Paket, schickt er das zuvor gesendete Datenpaket noch einmal. Antwortet der Server nicht in einer bestimmten Zeit auf ein Datenpaket, versendet der Client maximal 10 Mal das entsprechende Datenpaket erneut. Nach dem 10. Versuch bricht der Client den Vorgang ab.

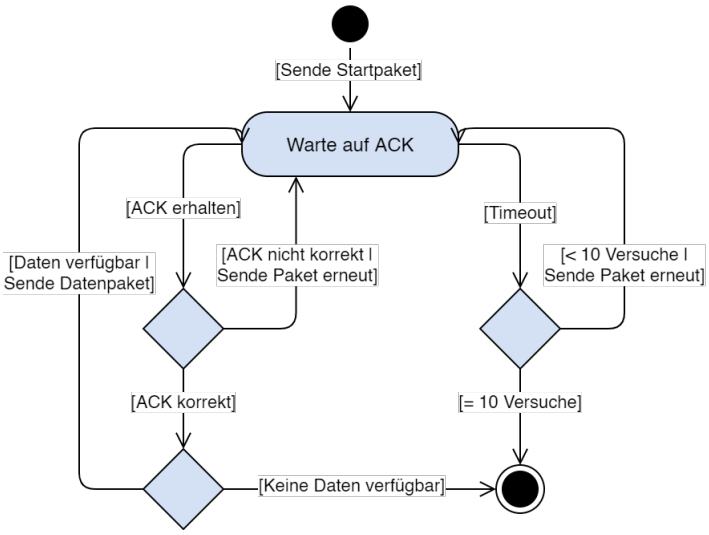

Abbildung 2: Zustandsdiagramm Client

3.3 Stop-And-Wait Protokoll

Die Anwendung sichert einen verlustfreien Datenverkehr durch das Stop-And-Wait-Protokoll ab. Dabei muss ein Datenpaket immer erst bestätigt werden, bevor ein nächstes Paket versendet werden kann. Dadurch werden keine großen Puffer im Programm benötigt, da immer nur ein Paket bearbeitet wird.

Um die Paketreihenfolge einzuhalten, wird mit alternierenden Paketnummern (0 oder 1) gearbeitet, durch die ein neues von einem alten Paket unterschieden werden kann. Das erste Paket, das Startpaket, erhält in dieser Anwendung immer die Paketnummer 0.

Gehen Pakete verloren und es kommt zum Timeout, verschickt der Client diese Pakete erneut.

3.4 Schichten

Da die Anforderung der Dateiübertragung von einer einfachen Konsoleneingabe über Dateiverwaltung, Stop-And-Wait-Protokoll und UDP-Sockets relativ komplex wirkt, ist es sinnvoll die einzelnen Aufgaben auf Schichten zu verteilen. Sowohl Server als auch Client arbeiten auf diesen Schichten und leiten ihre Anfragen und Antworten jeweils an die darüber- bzw. darunterliegende Schicht weiter:

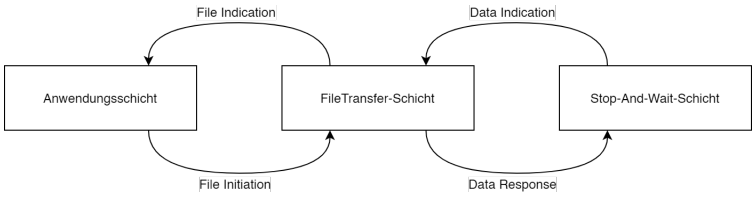

Abbildung 3: Schichten des Servers

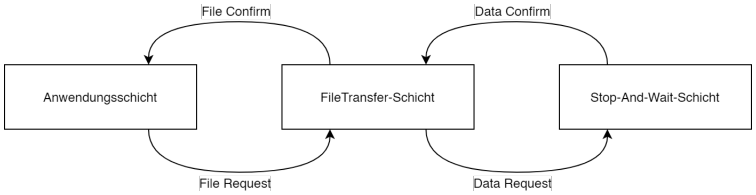

Abbildung 4: Schichten des Clients

3.4.1 Anwendungsschicht

Die Anwendungsschicht nimmt die Eingaben des Nutzers auf, überprüft diese auf Korrektheit und gibt die Informationen formatiert an die FileTransfer-Schicht weiter.

Beim Server hält die Anwendungsschicht das Programm nach erfolgreicher Datenübertragung in einer Endlosschleife.

3.4.2 FileTransfer-Schicht

Die FileTransfer-Schicht kümmert sich um die Dateiverwaltung, sowohl um das stückweise Verpacken beim Client, als auch das stückweise Zusammenfügen und Speichern beim Server. Außerdem werden in dieser Schicht die Pakete durch CRC auf Korrektheit geprüft.

Der Client erzeugt hier das Startpaket und die entsprechenden CRCs für Startpaket und letztes Paket.

3.4.3 Stop-And-Wait-Schicht

Die Stop-And-Wait-Schicht sorgt dafür, dass bei der Datenübertragung über UDP das Stop-And-Wait-Protokoll eingehalten wird. Hier werden die Pakete letztendlich an die UDP-Sockets weitergegeben oder von ihnen empfangen. Kommt es auf dieser Schicht zu einem Paketausfall, wird die erneute Sendung und Antwort des Pakets komplett innerhalb der Schicht geregelt. Die oberen Schichten bekommen in der Regel erst eine Rückmeldung, wenn verwertbare Daten vorhanden sind.

Der Client führt auf der Stop-And-Wait-Schicht Messungen durch, um die Datenrate zu bestimmen und den Timeout anzupassen. Der Server simuliert auf dieser Schicht den fehlerbehafteten Kanal.

3.5 Weitere Funktionen

3.5.1 Berechnung der Datenrate

Der Client berechnet periodisch (jede Sekunde) anhand der versendeten Daten, die Datenrate und gibt sie auf der Konsole aus. Am Ende der Übertragung wird zusammenfassend die mittlere, die minimale und die maximale Datenrate ausgegeben.

3.5.2 Variables Timeout

Der Client passt anhand der Antwortzeit des Servers die Dauer bis zum Timeout mit jedem Paket neu an. Der Startwert liegt bei einer Sekunde. Durch Anpassung des Timeouts, ähnlich wie bei TCP, wird die Datenübertragung effizienter und der Durchsatz wird erhöht.

4 Limitierungen

Das Bestätigen jedes einzelnen Datenpakets führt dazu, dass in einem relativ großen Zeitraum der Übertragung keine neuen Daten der zu übertragenden Datei verschickt werden. Da die Übertragung im Normalfall fehlerfrei verläuft, verbringt das Programm somit die meiste Zeit mit Warten. Das ist vor allem am Durchsatz erkennbar (siehe oben). Um die Übertragungsdauer zu verringern und den Mehraufwand der ACK-Pakete zu reduzieren, wäre es sinnvoll eher eine Implementierung in Richtung Go-Back-N-Protokoll oder Selective-Repeat-Protokoll anzustreben und mehrere Datenpakete auf einmal zu bestätigen.

Eine andere Möglichkeit wäre, die Paketgröße der Datenpakete an den Übertragungskanal anzupassen (ähnlich wie die Fluss-Kontrolle bei TCP). Das würde die Übertragung ebenfalls beschleunigen.

Mit der jetzigen Implementierung des Stop-And-Wait-Protokolls lassen sich nur vergleichsweise kleine Dateien in vertretbarer Zeit übertragen.