IBN Übungsblatt 11 Tutor: Denitsa Ilieva

Lyubomira Dimitrova, Dung Nguyen, Maryna Charniuk

Aufgabe 1

1. Warum braucht der Java-Server mit TCP zwei Sockets?

   Der Eingangssocket empfängt die Verbindungsanfragen der Clients und ordnet (schickt) jedem Client die Portnummer eines Verbindungssockets zu (3-Wege-Handshake). Zwischen Client-Socket und dem zugehörigen Verbindungssocket erfolgt die eigentliche Kommunikation, durch einen zuverlässigen Datenstrom.

2. Wenn der TCP-Server gleichzeitig n Verbindungen unterstützen muss, jede von einem anderen Client-Host, wie viele Sockets würde der TCP-Server dann brauchen?

   $1 + n$. Ein Eingangssocket, und ein Verbindungssocket pro Client.

Aufgabe 2

Nein, Kommunikation mit verschiedenen Hosts passiert über verschiedenen Sockets. Ob die gleiche Portnummer an mehrere Sockets vergeben werden kann, hängt von dem Protokol ab:

- In dem Fall mit dem **UDP Protokol** kann es nicht passieren: wenn eine Portnummer schon vergeben wird, kommt ein Fehler bei dem Aufruf von `DatagramSocket()` aus.
- In dem Fall mit dem **TCP Protokol** sind beide Antworte möglich:
  - Bei dem welcome-Socket des Servers und dem Socket des Clients ist es unmöglich gleiche Portnummer in mehreren Kommunikationen zu bekommen, weil sie eindeutig sein muss um Client und Server richtig zu identifizieren.
  - Bei dem Socket `connectionSocket = welcomeSocket.accept()`, der automatisch generiert wird, kann es passieren, dass die gleiche Portnummer an mehreren Sockets vergeben wird.

Aufgabe 3

TCP- und UDP-Server selbst implementiert, da Port 7 immer 'Connection Refused' erzeugt. Der Server wird in einem Terminal gestartet (z.B. python3 tcp_server.py), und der Client in einem anderen.

Zeit für 1000 Wiederholungen:

- **TCP**: etwa 0.11s
- **UDP**: etwa 0.1s

Aufgabe 4

a) Betrachtet ist die Datei *tcp-ethereal-trace-1*.

b) Frage:

4. (*Segment No.1*)

   (i) Sequenznummer von TCP SYN Segment zur Initialisierung der TCP-Verbindung zwischen Computer des Clients und gaia.cs.umass.edu ist `232129012`.

   (ii) Das `Syn-Flag` wird als `1` markiert und da steht am Ende `Set`. Daher kann es identifiziert werden, dass dieses Segment ein *SYN Segment* ist.

5. (*Segment No.2*)

   (i) Sequenznummer von SYNACK Segment, das vom *gaia.cs.umass.edu* an Computer des Clients als Antwort für SYN gesendet wurde, ist `883061785`.

   (ii) Der Wert von *Acknowledgement field* im SYNACK Segment ist `232129013`.

   (iii) *gaia.cs.umass.edu* hat diesen Wert durch die Inkrementierung von der initialisierten Sequenznummer des SYN Segment vom Computer des Clients um 1 (siehe 4.**(i)**)

   (iv) Analog zu 4.**(ii)**, die beiden `Syn-Flag` und `Acknowledgement-Flag` werden als `1` markiert und am Ende der beiden Zeilen steht `Set`. Von dem kann es identifiziert werden, dass dieses Segment ein *SYNACK Segment* ist.

6. (*Segment No.4*)

   (i) Sequenznummer vom TCP Segment, das den HTTP POST Befehl enthält, ist `232129013`.

8. (*Segmenten No. 4, 5, 7, 8, 10, 11*) Länge der ersten 6 TCP Segmenten:

   – Das erste TCP Segment, das HTTP POST enthält, hat die Länge von `565 Bytes`.

   – Die anderen 5 Segmenten haben die Länge von `1460 Bytes`.

9. (*Segment No.2*)

   (i) Die kleinste verfügbare Puffergröße für das ganze Trace ist im ersten ACK Segment gefunden. Dies hat die Größe von `5840 Bytes` (*Calculated window size*).

   (ii) Diese fehlende Puffergröße kann den Sender nicht einschränken, da der Puffer sich noch weiter vergrößert (bis zur Größe von `62780 Bytes`).

10. Nein, es gibt keine wieder übertragene Segmente im Trace-File. Um das zu überprüfen betrachten wir das Time Sequence Graph (Stevens) im Folgenden.

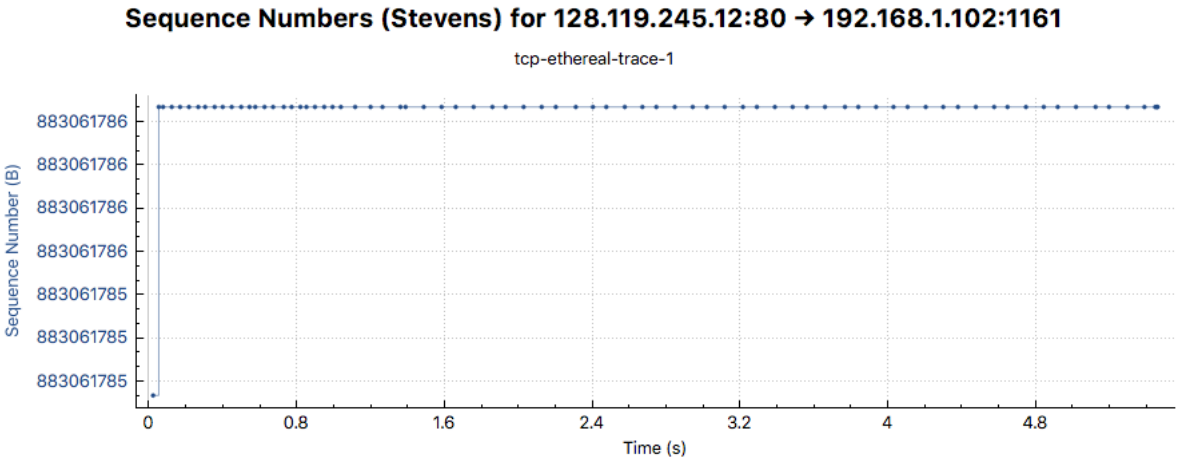

Figure 1: Time Sequence Graph (Stevens) (Screenshot 07.July 2019)

Aus der Figur lässt sich feststellen, dass die Sequenznummer vom Sender (*128.119.245.12*) zum Empfänger (*192.168.1.102*) bezüglich der Zeit sich nur steigen. Gäbe es eine Sendewiederholung des Segment, sollte seine Sequenznummer kleiner als die von den Nachbarsegmenten sein.

11. Betrachten wir immer 2 aufeinanderfolgdende ACK Segmenten, entspricht die Differenz zwischen beiden Sequenznummer dieser Segmenten ($232132498 - 232131038 = 1460$ Bytes) die Daten, die das Server kennt.

```
TCP      1514 1161 → 80 [ACK] Seq=232131038 Ack=883061786 Win=17520 Len=1460
TCP      1514 1161 → 80 [ACK] Seq=232132498 Ack=883061786 Win=17520 Len=1460
```

Figure 2: Example from 2 consecutive ACK segments: No.7 and No.8

Figur 2 hat auch einen Fall aufgewiesen, in dem das Server alle anderen Segmenten bestätigt ("ACKing"), da die 2 Segmenten dieselbe ACK Nummer haben.

12. Der durchschnittliche Durchsatz (Throughput: bytes transferred per unit time) für die TCP-Verbindung ist aus der Division zwischen der gesamten Übertragungsdaten und der gesamten Übertragungszeit errechnet.

    – Die gesamte Übertragungsdaten ist der Abstand zwischen Sequenznummer vom letzten ACK Segment, Segment No.206 und Sequenznummer vom ersten TCP Segment, in dem POST-Befehl enthalten ist:
    $232293103 - 232129013 = 164,090$ Bytes.
    – Die gesamte Übertragungszeit ist der Abstand zwischen der Zeit des letzten ACK Segments, Segment No.206 und der Zeit des ersten TCP Segment, in dem POST-Befehl enthalten ist:
    $5.651141 - 0.026477 = 5.624664$ secs.

– Der durchschnittliche Durchsatz (Throughput) ist:

$$\frac{164,090}{5.624664} \approx 29173.29817\, Bytes/sec \approx 28.4895\, KBytes/sec$$

Aufgabe 5

SYN-Cookies helfen die Information aus der Sequenz-Nummer (Client-IP, Client-Port, Server-IP, Server-Port, Timestamp), die der Client dem Server schickt um die Verbindung herzustellen, nicht zu verfälschen. Dafür wird eine frei wählbare Hash-Funktion benutzt über die Werte: Hash (Client-IP, Client-Port, Server-IP, Server-Port, Timestamp, key), wobei der Key-Wert wird nur dem Server bekannt, für den Client ist die Hash-Funktion transparent. Der Server sendet SYN-ACK-Paket (Hash-Wert, max. Paket-Größe, Timestamp) an dem Client, der Client inkrementiert die SYN-Nummer des SYN-ACK-Pakets um eins und schickt weiter an dem Server. Der Server dekrementiert die erhaltene Nummer um eins und vergleicht sie mit dem gespeicherten Hash-Wert, wenn sie übereinstimmen, dann wird die Verbindung hergestellt, sonst fehlgeschlagen.

Aufgabe 6

Jedes TCP-Segment enthält in seinem Header die 16-Bit RcvWindow Variable. Mit dieser teilt der Sender mit, wie viel Platz in seinem Eingangspuffer für die Antwort übrig wäre. Somit wird der Host A 'wissen', wie viele Bytes Host B bereit ist, zu empfangen.

Aufgabe 7

- Die Auslastung (U_{sender}) bei dem Pipelining-Verfahren lässt sich um Faktor N (Anzahl der Pakete bei jeder Sendung) von der Auslastung bei Stop-and-Wait-Verfahren erhöhen. d.h.: $U_{Pipelining} = N * U_{Stop-and-Wait}$.

$$\Longrightarrow N = \frac{U_{Pipelining}}{U_{Stop-and-Wait}} = \lfloor \frac{95\%}{0.0266} \rfloor \approx 35\, Pakete$$

- Aus der Vorlesung N04 wissen wir, dass beim Stop-and-Wait-Verfahren in jeder 30.0085 ms Zeit ein 1 kByte-Paket übertragen wird, oder ca. 33kBytes/sec wird übertragen. Beim Pipelining-Verfahren (von oben) sehen wir, dass in jeder 30.0085 ms Zeit 35 Pakete der Größe 1kByte übertragen werden, d.h. der Durchsatz ist : $35 * 33 = 1155$ kBytes/sec ≈ 1.128 MB/sec.

Aufgabe 8