Übungen zur Vorlesung
Betriebssysteme I
Studiengang Informatik
WS 2013/14

AG Betriebssysteme FB3
Prof. Dr. Jan Peleska, Florian Lapschies

Abgabe: 06.01.2014

Übungszettel 4

IP-Fragmentierung von UDP-Paketen

In der Vorlesung wurde Euch die Kommunikation über Sockets vorgestellt. Das User Datagram Protocol (UDP) ist Teil der Internetprotokollfamilie und kann zur verbindungslosen Kommunikation mittels Datagrammen genutzt werden, wenn die falsche Reihenfolge oder der Verlust einzelner Datagramme akzeptabel ist. Bei der Übertragung von UDP-Paketen über ein Internet Protokoll Netzwerk werden UDP-Datagramme als Nutzlast von IP-Datagrammen übertragen. Da die Maximalgröße von übertragbaren IP-Datagrammen von dem jeweiligen Übertragungsmedium abhängt, ist es mitunter nötig, UDP-Datagramme auf einzelne IP-Datagramme zu verteilen. Die Zerlegung eines IP-Datagrammes zu mehreren kleineren IP-Datagrammen nennt man IP-Fragmentierung. Diese muss nicht umbedingt direkt beim Versender geschehen, sondern kann auch auf einen der Zwischenstationen wie zum Beispiel einem Router auf dem Weg zum Empfänger erfolgen. Ist ein IP-Datagramm erst einmal fragmentiert, ist es im allgemeinen nicht sichergestellt, dass die einzelnen Fragmente den selben Weg zum Ziel nehmen, in der richtigen Reihenfolge ankommen oder überhaupt ankommen.

Dies erfordert natürlich, dass die IP-Datagrammfragmente auf der Empfängerseite wieder ordentlich zusammengesetzt werden, und genau das ist eure Aufgabe.

Aufgabe 1 – Der Sender

Schreibt ein C-Programm `send`, welches über einen gewöhnlichen UDP/IP-Socket ein einzelnes Datagramm an eine an ihn übergebene IPv4 Adresse und Port sendet. Die zu versendende Nachricht soll dabei über `stdin` eingelesen werden. Verwendet zum Einlesen der Nachricht `read()`, um in einen Puffer einzulesen, den Ihr mit `realloc()` sukzessive vergrößert, bis End-of-File erreicht ist.

Um eine Nachricht an die IP-Adresse `134.102.207.217` und Port `10001` zu senden, ist das Programm wie folgt aufzurufen:

```
send 134.102.207.217:10001
```

Verwendet `inet_pton()`, um die als String übergebene IP in Binärform zu übersetzen.

Als Maximum Transmission Unit (MTU) bezeichnet man die maximale Paketgröße, die ohne Fragmentierung übertragen werden kann. Die einzelnen MTU-Werte können über den Befehl `ifconfig` für die verschiedenen Netzwerkinterfaces erfragt werden. Das Netzwerkinterface zur lokalen Kommunikation ist dabei das Loopbackdevice (`lo`), Ethernetdevices heißen zum Beispiel `eth0` und Wirelessdevices meist `wlan0`. Damit es bei dieser Aufgabe auch wirklich zur Fragmentierung kommt, müssen die UDP-Pakete größer als die jeweilige MTU sein. Standardmäßig werden beim Versenden von UDP Paketen über einen Socket alle Pakete, die größer als die MTU sind, mit einem Fehler zurückgewiesen (siehe `man 7 udp`). Dies kann man jedoch umgehen, indem die sogenannte Path MTU Discovery abgeschaltet wird. Hierzu muss für den Socket mit `setsockopt()` die Option `IP_MTU_DISCOVER` auf `IP_PMTUDISC_DONT` gesetzt werden (siehe `man 7 ip`). Zudem ist die MTU standardmäßig recht groß, so dass Fragmentierung erst bei recht großen Paketen auftritt. Setzt daher die MTU auf 200 Bytes:

```
ethtool -K <DEVICE> ufo off
ifconfig <DEVICE> mtu 200
```

Aufgabe 2 – Receiver

Die von `send` versendeten und möglicherweise fragmentierten UDP-Pakete sollen von einem weiteren C-Programm `receive` empfangen und zusammengesetzt werden. Um `receive` UDP-Pakete empfangen zu lassen, die von der Adresse `134.102.207.130` an die Adresse `134.102.207.217` und dessen Port `10001` geschickt wurden soll das Programm wie folgt aufgerufen werden:

```
receive 134.102.207.130 134.102.207.217:10001
```

Wenn Pakete von einem normalen UDP/IP Socket gelesen werden, läuft das Defragmentieren ganz ohne weiteres zutun ab, so dass man z.B. durch `recvfrom()` bereits defragmentierte Pakete geliefert bekommt.

In dieser Aufgabe sollen nun aber die IP-Datagramme in Rohform von einem speziellen Packet-Socket gelesen werden und dann manuell zu vollständigen UDP-Paketen zusammengesetzt werden. Der Paket-Socket soll wie folgt erstellt werden (siehe `man 7 packet`):

```
sock = socket(AF_PACKET, SOCK_DGRAM, ntohs(ETH_P_IP))
```

Wird nun von diesem Socket gelesen, bekommt man jeweils ein IP-Datagramm mitsamt Header, dass an ein beliebiges Netzwerkinterface und an einen beliebigen Port eures Rechners geschickt wurde, geliefert. Verständlicherweise kann der Socket also nur als `root` erstellt werden.

Am Anfang der gelesenen IP-Pakets steht stets der IP-Header. Die zugehörige C-Struktur ist `struct ip` in `netinet/ip.h`[1]. Danach folgt die Nutzlast, also die Fragmente des UDP-Paket. Am Anfang eines UDP-Pakets steht stets ein UDP-Header. Die zugehörige C-Struktur ist `struct udphdr` in `netinet/udp.h`. Direkt anach folgt die Nutzlast des UDP-Pakets, also die eigentlich Nachricht.

Folgende IP-Pakete können sofort aussortiert werden:

- Pakete, deren IP-Header Version nicht die erwartete ist.
- Pakete, die nicht zum UDP Protokoll gehören.
- Pakete, deren Quell-IP nicht die erwartete ist.
- Pakete, deren Empfäger-IP nicht die erwartete ist.

Beim Versenden von UDP-Paketen über IP bekommen alle IP-Pakete mit Fragmenten des selben UDP-Datagramms die selbe eindeutige ID. So lassen sich Fragmente den UDP-Datagrammen zuordnen. Über ein Offset-Feld lässt sich bestimmen, um welchen Teil des UDP-Datagramms es sich handelt. Ein More-Fragments-Bit gibt Aufschluss darüber, ob es sich bei einem Fragment um das letzte, hinterste Fragment handelt.

Für jedes zusammenzusetzendes UDP-Datagramm benötigt Ihr mindestens folgende Verwaltungsinformationen:

- Die ID des zusammenzusetzenden UDP-Datagramms.
- Ein Puffer, in den die einzelnen Fragmente hineinkopiert werden und welcher nach dem Empfang aller Fragmente das komplette UDP-Datagramm enthalten soll. Da die Gesamtgröße des Puffers nach dem Empfang des ersten Fragments noch nicht feststehen muss, muss die Größe dieses Puffers gegebenenfalls mit `realloc()` vergrößert werden.
- Die aktuelle Größe des Puffers in Byte.
- Da Fragmente in falscher Reihenfolge, doppelt oder gar nicht ankommen können, wird eine Bitliste benötigt, in der bereits empfangene Teile durch ein gesetztes Bit markiert sind. Da der Offset von Fragmenten immer in Vielfachen von 8 Byte angegeben wird, entspricht jedem Bit genau immer 8 Byte. Auch diese Bitliste muss u.U. durch `realloc()` vergrößert werden.

[1] Informationen zum Aufbau des IP-Headers findet Ihr unter
`http://www.informatik.uni-bremen.de/agbs/lehre/ws1314/bs1/hintergrund/ip.htm`,
`http://www.informatik.uni-bremen.de/agbs/lehre/ws1314/bs1/hintergrund/ipfrag.html` und
`http://en.wikipedia.org/wiki/IPv4`

- Ein Flag, dass angibt, ob bereits das Paket mit nicht gesetztem More-Fragment-Bit empfangen wurde.
- Der Zeitpunkt zu dem das erste Fragment dieses UDP-Datagramms empfangen wurde.

Ein UDP-Paket ist dann komplett, wenn das Fragment mit nicht gesetztem More-Fragment-Bit empfangen wurde und zusätzlich alle Bits der zugehörigen Bitliste gesetzt sind. In diesem Fall wird das fertig zusammengesetzte Pakete auf `stdout` ausgegeben und danach alle Verwaltungsinformationen zu dem Paket gelöscht.

Damit zu einer UDP-ID auch effizient die Verwaltungsinformationen gefunden werden, sollen sich diese in in einer Hash-Map befinden, wobei die ID der Schlüssel ist. Implementiert Hashing mit Kollisionsauflösung durch Verkettung. Die Größe der Hashtabelle soll dabei konstant sein, sollte aber eine Primzahl (z.B.: 4621) sein, um statistisch gesehen die Anzahl der Kollisionen gering zu halten.

Da einzelne Fragmente verloren gehen können, können einzelne UDP-Datagramme u.U. nicht zusammengesetzt werden. Damit sich über die Laufzeit des Programms nicht immer mehr Verwaltungsinformationen ansammeln und sich später sogar die IDs wiederholen, soll ein Timeout implementiert werden, nachdem UDP-Pakete verworfen werden. Die Verwaltungsinformationen eines zusammenzusetzenden UDP-Pakets befinden sich hierzu zusätzlich in einer Queue, die nach der Zeit sortiert ist. Wenn ein Fragment eines komplett neuen UDP-Pakets empfangen wird, wird dessen Verwaltungsinformationen hinten in die Queue eingefügt und die aktuelle Zeit vermerkt. Um die aktuelle Zeit zu erfahren benutzt Ihr `gettimeofday()`. Jedes Mal, nachdem ein IP-Paket verarbeitet wurde, werden alle UDP-Pakete vom Anfang der Queue entnommen, für die der Timeout abgelaufen ist, und die zugehörigen Verwaltungsinformationen gelöscht. Setzt den Timeout auf 5 Sekunden.

Zusatzinformationen über aussortierte IP-Pakete, abgelaufenen Timeouts usw. sollen auf `stderr` ausgegeben werden.

Aufgabe 3 – Test

Sendet mit dem `send` Datagramme unterschiedlicher Größe in Form von Text über verschiedene Netzwerkdevices, so dass es auch zur Fragmentierung kommt, und weist nach, dass diese von `receive` korrekt empfangen werden. Weist anhand Eurer Ausgaben nach, dass Pakete korrekt aussortiert werden.

Hinweise

- Alle in dieser Aufgabe geforderten Programme sollen mit einem Makefile übersetzt werden können. In diesem Makefile sollen Abhängigkeiten des Build-Prozesses richtig erfasst sein. Des weiteren soll das Makefile auch eine Regel `clean` enthalten, die alle Komplilate wieder löscht.
- Achtet darauf, dass Ihr die Rückgaben aller Systemaufrufe auf mögliche Fehler überprüft. Im ungewünschten Fehlerfall soll eine erklärende Ausgabe auf `stderr` erfolgen und das Programm dann mit `exit(EXIT_FAILURE)` (siehe `man 3 exit`) beendet werden. Viele Funktionen setzen im Fehlerfall die Variable `errno` (siehe `man 3 errno`), über die der genaue Fehler identifiziert werden kann. Hilfreich für Fehlerausgaben ist die Funktion `strerror()` (siehe `man 3 strerror`) zur Formatierung von `errno`. Studiert auf jeden Fall die entsprechenden man-Pages, um zu erfahren welche Rückgabewerte Fehler kennzeichnen und welche Fehler auftreten können.
- Achtet darauf, dass Ihr allen Speicher, den Ihr mit `malloc()` alloziert habt, auch zum Programmende wieder mit `free()` freigegeben habt. Bei der Behandlung von Fehlerfällen, bei denen das Programm mit einem Fehlercode beendet wird, darf darauf verzichtet werden. Hilfreich zum Aufdecken von Speicherlecks und Speicherzugriffsfehlern ist das Tool `valgrind`.
- Die Abgabe erfolgt als Ausdruck am Ende der Vorlesung und zusätzlich elektronisch über das GIT Repository.
- Die Dokumentation der Aufgabenlösung ist in LaTeX anzufertigen.