Network Protocols and Architectures - WS11/12
Blatt 08

Sebastian Lohff bis 05.01.2012

Aufgabe 1 - RFC1149

a) IP bietet *Best Effort* Transport von Paketen auf der Vermittlungsschicht an und ist unabhängig von unterliegenden Schichten. Wie ein IP-Paket letztendlich von A nach B kommt, ist IP egal. Latenz oder Paketverlust spielen für IP auch keine Rolle, ein Propagation-Delay von über 30 Minuten stellt für IP selber kein Problem dar.

Und es kann funktionieren: Im Mai 2001 hat eine Gruppe es geschafft mehrere Pings über Brieftauben als Avian Carrier zu senden[1].

b)
- Transmission Delay: Zeit, die man braucht um ein Paket auszudrucken, es an den Carrier zu tapen und auf der anderen Seite wieder einzuscannen.
- Propagation Delay: Zeit, die der Carrier braucht um von A nach B zu fliegen.
- Packet Loss: Tritt laut RFC oft im Frühling auf, falls der Carrier von anderen Carriern abgelenkt wird. Gefährlich ist auch der Windows TSOD[2], Umwelteinflüsse, Jäger oder andere natürlichen Feinde des Carriers[3].

Wenn man Payloads wie TCP hat, kann es Probleme mit dem Delay geben, da ein Paket nicht vor seinem Timeout geackt werden kann. Tut man nichts hiergegen, kann es zu vielen Timeouts und einem sehr kleinen Congestion-Window kommen, in der Regel wird eine Verbindung timeouten, bevor sie überhaupt zustande kommen kann. Entweder man setzt die TCP-Timeout beider Endsysteme hoch (bzw. verändert den Algorithmus, der bestimmt, wann ein Paket neu gesendet weden muss) oder terminiert TCP an beiden Enden des Teilstücks, auf dem Avian Carrier eingesetzt wird, und ersetzt es für den hochlatenten Weg durch ein auf UDP basierendes delay-toleranteres Protokoll.

c) RFC1149 beschreibt ,,IP over Avian Carrier". Wiktionary definiert den Begriff *avian* als *,,Characteristic of or pertaining to, birds, bird-like or flying creatures."*[4]. Im Folgenden der Aufgabe wird von dieser Definition ausgegangen.

Es gibt zwei Ansätze diese Aufgabe zu lösen.

0. Anmerkung zu verwendeter Software

Der Sourcecode zu den folgenden Lösungsansätzen ist in dem unter diesem Absatz angegebenen GIT-Repository verfügbar. Die Skripte basieren auf einer selbstgeschrieben TUN/TAP-Device-Verwaltung und Tunnel-Eventloop-Abstraktion names Ether2Any. Relevanter Sourcecode für die einzelnen Tunnel befindet sich in `ether2any/tunnel/vrfc1149/`, `ether2any/tunnel/rfc1149/` und `ether2any/tunnel/usbip/`. Zum Ausführen von vRFC1149 wird tweepy 1.8[5] und bitarray[6] benötigt. Für usbip wird pyudev[7] benötigt. Weitere Kommentare zur Software erfolgen an entsprechender Stelle.

[1] http://www.blug.net/rfc1149/
[2] Windows ,,Transparent Screen Of Death", tritt häufig in Windows ,,Panorama Edition" auf.
[3] ,,Out of band roak payload-less carrier injection attack" durch ,,Hawk packet encapsulation"
[4] http://en.wiktionary.org/wiki/avian
[5] `http://pypi.python.org/pypi/tweepy/1.8`
[6] `http://pypi.python.org/pypi/bitarray`
[7] `http://pypi.python.org/pypi/pyudev`

```
git clone git://git.someserver.de/git/ether2any/
```

1. vRFC1149 - Virtual RFC1149 aka ,,IP over Twitter“

(a)
Twitter-Logo

(b)
vRFC1149

Twitter, ein Webservice zum Microbloggen, hat als Logo einen Vogel. Das Senden einer Nachricht wird ,,tweeten“ (engl. für zwitschern) bzw. ,,twittern“ genannt. Damit sind die Eigenschaften für ,,bird-like“ erfüllt und man kann Twitter bzw. Tweets als Avian Carrier verwenden. Das Projekt selber wird vRFC1149 genannt, da der Avian Carrier hier virtuell ist.

Über Twitter wird eine Verbindung zwischen zwei Punkten aufgebaut: Man folgt den Tweets eines anderen Accounts, entpackt diese und interpretiert sie als IP-Pakete. Ausgehende Pakete werden von einem selbst getwittert. Über *conf.py* kann man den Endpunkt, die IP-Adressen und die jeweilige Paketkodierung wählen.

Zur Authentifizierung benutzt Twitter OAuth. Um Twitter mit eigenen Accounts zu benutzen, muss man die Anwendung ohne ACCESS_KEY und ACCESS_SECRET starten (default). Bei der zurückgegebenen URL kann man sich mit seinem Twitter Account anmelden und erhält eine PIN. Diese gibt man vRFC1149. Nun darf das Python-Script auch für den jeweiligen User twittern.

Zum Testen wurden die beiden Accounts `tw1tw1tw1`[8] und `tw2tw2tw2`[9] benutzt.

Einpacken und Kodieren

Twitternachrichten erlauben eine Länge von 140 Zeichen. Interessanterweise ist für Twitter entgegen eigener Angabe[10] ein Zeichen ein Unicodezeichen. Mit einem Unicodezeichen kann man theoretisch 2^{20} Bits kodieren, wodurch die maximale Länge eines Tweets bei 350 Byte liegt. Aus Bequemlichkeit wurde entschieden, dass in jedem Unicodezeichen 2 Byte kodiert werden. Es wird ein Header hinzugefügt, der aus einem Fragmentation-Bit, 9 Bit Paketlänge und einer 32 Bit Zufallszahl besteht. Der Header wird auf die verbleibenden höheren 4 Bit der Unicodezeichen verteilt. Die Zufallszahl wird benötigt, damit man bei zwei gleichaussehenden Paketen keine ,,DuplicateTweet“-Exception bekommt.

Problematisch ist, dass Twitter manche Zeichen verändert, verschluckt, ineinander zieht oder zu mehreren macht. Trotzdem bleiben die meisten Zeichen erhalten. So kann man sogar ein Nullbyte (\x00) per Tweet transportieren. Hier eine Übersicht über Probleme, die mit Zeichen aufgetreten sind:

- Mehrere Nullbytes am Ende eines Tweets werden zu einem zusammengefasst

[8] `http://twitter.com/tw1tw1tw1`
[9] `http://twitter.com/tw2tw2tw2`
[10] `http://support.twitter.com/articles/66890-my-direct-message-posted-to-my-public-timeline` “A standard text message is limited to 140 bytes“

- 0xd800 - 0xdfff werden zu einem 12-Byte-String des Formats „\XXX\XXX\XXX“ abgeändert, wodurch auf Twitter 140 Zeichen Input sogar zu 1680 Zeichen Output werden können[11]
- Ein ankommendes Zeichen $\geq$ 0x10000 *kann* für zwei einzelne Unicode-Zeichen stehen

Einige der kaputtgehenden Zeichen sind zwar Non-Printables, aber wenn schon Nullbytes erhalten werden, hätte man erwarten können, dass andere Zeichen auch erhalten bleiben. Für mehr Informationen zu diesem Thema stehen Kommentare im Sourcecode[12].

Als zweite Kodierungsmöglichkeit wurde das Paket mit einem Extra-Header plain zu Twitter geschickt. Die ersten beiden Zeichen werden mit Unicodezeichen befüllt, welche 1 Bit Fragmentation, 8 Bit Paketlänge und 11 Bit Paket-ID (wieder eine Zufallszahl) beinhalten. An diesen Header wird das ausgehende Netzwerkpaket angehängt.

Hierbei treten wiederum andere Merkwürdigkeiten auf: Twitter escapted die Zeichen < / > zu < / > und verschluckt ein \r komplett. Hier muss wieder ein Workaround gefunden werden. Da diese Art des Einpackens nur ein Proof-of-Concept sein soll, wurde HTML-Encoding für diese Zeichen gewählt.

Die dritte Möglichkeit wäre, die Daten base64 encoded hochzuladen. So ist die Wahrscheinlichkeit sehr groß, dass sie gut auf der anderen Seite ankommen. Allerdings verdreifacht sich auch die Paketgröße. Diese Methode ist daher nicht implementiert[13].

Bandbreite

Twitter hat viele API-Limitierungen[14].

Wichtig für vRFC1149 ist nur das Status-Update-Limit, das bei 1000 Tweets pro Tag und 41 (manchmal auch mehr) Tweets pro Stunde liegt. Benutzt man zur Kodierung der Nachricht die Unicode-Methode, so kann man pro Tweet 280 Byte Transportieren.

$$280\frac{b}{Tweet} * 41\frac{Tweet}{h} = 11480\frac{b}{h} \approx 191.33\frac{b}{min} \approx 3.18\frac{b}{s}$$

Für eine kontinuierliche Datenverbindung muss man sich also auf 3 Byte pro Sekunde beschränken, natürlich unter der Annahme, dass jeder Tweet vollständig mit Daten befüllt ist. Es können die gesamten 11480 Byte auch auf einmal genutzt werden, dann muss aber bis zu eine Stunde für die nächste Datenkommunikation gewartet werden. Diese Beschränkung ist nur für den Upstream. Der Downstream wird durch den Upstream der Gegenseite beschränkt.

Zum Lesen der Tweets wird die Streaming-API benutzt, für die bei diesem Anwendungsfall kein Limit auftritt.

Ergebnisse

Die Software wurde erfolgreich eingesetzt. Getestet wurde mit ICMP-Paketen mit verschiedenen Payloadgrößen, einem HTTP-Request und dem Aufbau einer SSH-Verbindung.

[11] `https://twitter.com/#!/tw1tw1tw1/status/149897161495154688`
[12] phelper.py: UPHelper.reassembleBrokenChars()
[13] Ein Packaginghelper lässt sich leicht mittels der PHelper Basisklasse bauen
[14] `http://support.twitter.com/articles/15364-about-twitter-limits-update-api-dm-and-following`

Die Delays aus b) können hier nicht sehr stark beobachtet werden. Transmission-Delay tritt hauptsächlich durch das En- bzw. Decoding der Pakete, das Delay der Twitter-API beim Statusupdate und dem Herunterladen des Tweets auf. Als Propagation Delay betrachten wir die Zeit, die der Tweet braucht, um von Twitter durch das Netz zum anderen Client, der mit der Streaming-API auf neue Tweets horcht, transportiert zu werden. Im Gegensatz zu herkömmlichen Avian Carrier based Network Solutions haben wir hier aber wesentlich kleinere Delays.

Durch die vielen Encodingprobleme treten häufiger kaputte Pakete auf, die dann TCP-Retransmits auslösen. Einzig die Geschwindigkeit wird hier beeinträchtigt.

Für ICMP ergab sich eine RTT von ungefähr `1500ms`.

Abbildung 1: Teil einer SSH-Verbindung per vRFC1149

2. „IP over Quadrocopter“

RFC1149 kann auch mit anderen Avian Carriers implementiert werden. In diesem Falle wird als Carrier ein Quadrokopter der Dragoncopterreihe benutzt[15]. Im RFC1149 wird von einer MTU von 256mg ausgegangen. Bei dem hier verwendeten Carrier liegt die MTU ungefähr 390 mal höher, nämlich bei 100g. Packetloss bei Carrierverlust ist deutlich teurer als bei einem herkömmlichen Carrier. Der Carrier regeneriert sich nicht selbst, sondern muss nach 15 Minuten Datenverkehr neu aufgeladen werden. Er hat eine Leistungsaufnahme von 70 Watt.

[15] `http://dragoncopter.de`

Erste Implementierung

In einer ersten Implementierung wurde versucht sich strikt an das RFC zu halten. Der Versuch scheiterte jedoch an fehlender Hardware. Nichtsdestrotrotz soll er hier beschrieben werden.

Ausgehende Pakete werden in eine hexadezimale Repräsentation gebracht. Vor dem Ausdrucken wird das Paket angezeigt und der User gefragt, ob er es versenden möchte. Dies hat neben gespartem Papier bei fälschlicherweise gesendeten Paketen auch den Vorteil einer durch einen Menschen bedienten Firewall[16].

```
 ?> IP / ICMP 10.10.10.10 > 10.10.10.1 echo-request 0 / Raw --
    hex len 263 checksum 0x40cb0944
 ?? (A)ccept/(D)rop? d
 -- Dropped
 ?> IP / ICMP 10.10.10.10 > 10.10.10.1 echo-request 0 / Raw --
    hex len 263 checksum 0x141c320d
 ?? (A)ccept/(D)rop? a
 >> Moved one packet to printer
45 00 00 54 00 00 40 00 40 01 12 8B 0A 0A 0A 0A 0A 0A 0A 01 08
   00 6A 53 6F 5A 00 01 F6 CC 01 4F 39 32 02 00 08 09 0A 0B 0C 0
   D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21
    22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35
   36 37 14 1C 32 0D
```

Eingehende Pakete werden mit einer Kamera fotografiert („optisch gescannt“). Die SD-Karte der Kamera wird in den Laptop gesteckt und vom Laptop gemountet (in diesem Fall ohne automount sondern mittels eines Udev-Events, welches die Tunnelsoftware bekommt). Das Bild wird durch ImageMagicks `convert` zur Texterkennung vorbereitet und danach mit Tesseract[17] eingelesen. Um OCR-Fehler zu vermeiden wird an das Paket eine crc32 Checksumme angehängt. Stimmt diese nach der Texterkennung nicht, muss das Paket erneut gescannt werden.

Ein ausgehende Paket sollte dann per Duct Tape an einem der vier Beine des Carriers befestigt werden. Leider scheiterte der Versuch daran, dass zur Zeit der Durchführung keine zwei Drucker verfügbar waren. Daher wurde dieser Teil ersetzt und eine alternative Methode verwendet.

USB-Netzwerk

Bei „IP over USB-Stick“ werden ausgehende Pakete gecached, bis ein USB-Stick angesteckt wird. Durch Udev bekommt die Software ein Event und mounted den Stick an einem vorkonfigurierten Ort. Es werden aus einem Verzeichnis auf dem Stick alle Netzwerkpakete (ein Paket pro Datei) gelesen und danach, solange der Stick gemounted ist, neue Pakete auf ihn geschrieben. Auf Tastendruck wird der Stick geunmounted und kann zum Empfänger transportiert werden. So kann mittels eines USB-Sticks IP-Kommunikation zwischen zwei Systemen ermöglicht werden.

```
# ./usbip.py
Starting ip over USB-Stick service...
```

[16] Ideen für durch Menschen bediente Firewalls werden auf Wunsch nachgereicht
[17] http://code.google.com/p/tesseract-ocr/

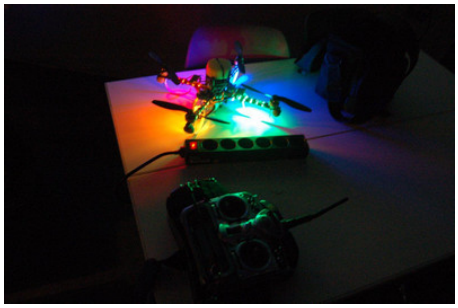
```
++ Mounted stick (/dev/sdb1) at /mnt/
>> Read 1 packet(s)
** Press any key to release usbstick...
>> 6 packet(s) written
** Releasing stick...
** Syncing...
** Unmounted
```

Mit UDP können hier auch ganz leicht sehr große Dateien übertragen werden: Man schickt z.B. die Datei per Netcat an einen anderen Rechner (`hostA# cat bigfile|nc -u 10.10.10.2 7777`), die Pakete werden auf den USB-Stick geschrieben und werden dann vom anderen Rechner gelesen. Dieser muss nur vor dem Lesen der Pakete auf dem endsprechenden UDP-Port lauschen um die Datei zu empfangen (`hostB# nc -u -l -p 7777 > bigfile`).

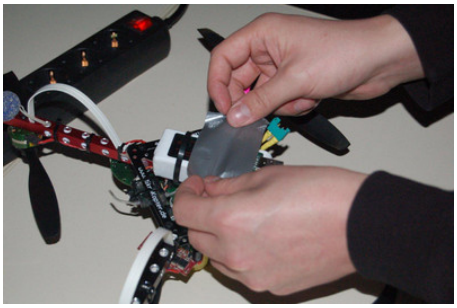
Ausführung des Versuchs

Ein praktischer Versuch von RFC1149 mittels Quadrokoptern wurde auf dem 28. Chaos Communication Congress[18] zwischen Tag 3 und Tag 4 durchgeführt. In mehreren Versuchen wurden etwa 2000 ICMP-Pakete übertragen, wobei meistens mehrere Pakete gleichzeitig übertragen wurden. Die RTT betrug zwischen 70 und 80 Sekunden. Ein Video des Versuchs ist unter `http://vimeo.com/34523140` zu finden.

(a) Transmitter

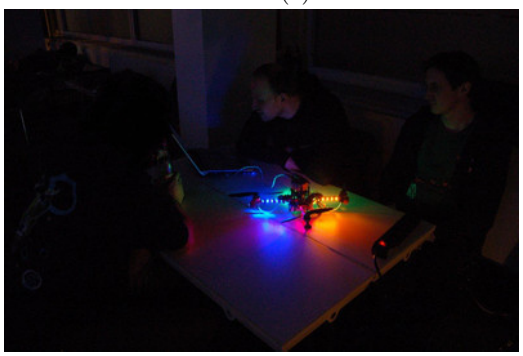


(b) Carrier



(c) Mountversuch

(d) Ping



(e) Pong

Beim Datentransfer konnten die verschiedenen Delays aus Aufgabenteil b) deutlich beobachtet werden.

[18] `http://events.ccc.de/`

Da statt Druckern und Scannern USB-Sticks verwendet wurden, ist das Transmission Delay etwas kleiner, aber trotzdem noch gut messbar. Es müssen Pakete geschrieben, der USB-Stick muss vom Betriebssystem geunmountet und gesynct, danach die USB-Verlängerung vom Stick abgezogen werden. Auf der anderen Seite muss der USB-Stick wieder mit dem USB-Verlängerungskabel verbunden[19] und vorhandene Pakete gelesen werden. Das Transmission Delay beträgt 10-20 Sekunden (je nach Betriebssystem, gecachedtem Datenvolumen[20] und Geschick des Nutzers mit USB-Geräten). Durch das USB-Verlängerungskabel muss der USB-Stick nicht jedesmal erneut am Carrier fixiert werden und es ensteht kein zusätzliches „Taping-Delay“.

Das Propagation Delay hängt von Abstand der Transmitter, Art der Hindernisse, Wetter und dem Geschick des Piloten ab. Der Pilot muss aufpassen, dass er bei der Landung nicht gegen den Laptop fliegt, aber gleichzeitig in Reichweite der USB-Verlängerung landet - somit ist das Landing Delay ein integraler Bestandteil des Propagation Delays. In der Flugzeit durch den 3D-Ether ist der Carrier auch den in Aufgabenteil b) beschriebenen Gefahren ausgesetzt, wobei der Jäger sich eher als Angreifer auf den Pilot-Carrier-Kommunikationskanal manifestiert. Herkömmliche Jäger bleiben trotzdem eine Gefahr.

Die Reichweite ist bei einem Piloten mit Funkfernbedienung auf die Funkreichweite, solange Sichtkontakt mittels einer Kamera gegeben ist, beschränkt. Versuche mit autonom fliegenden Carriern wurden nicht unternommen, sind aber mit der richtigen Software und einem GPS-Empfänger (beim Dragoncopter bereits vorhanden) vorstellbar. Ist dies gegeben, wird die Reichweite von der Leistung der Energiequelle des Carriers beschränkt.

Eine TCP-Verbindung war bei der hohen Latenz von über `70000ms` nicht möglich.

Bei den vorliegenden Tunneln wurden keine Tier- oder Kopterschutzrichtlinien verletzt. Kein Carrier kam zu schaden[21].

d) Bei Quadrokoptern könnte Bulk-Retrival via Get-Net sehr teuer werden. Kopter würden gegenseitig ihre Hardware schreddern. Dies entspricht nicht den Kopterschutzrichtlinien.

[19] http://chzmemebase.files.wordpress.com/2011/04/memes-always.jpg

[20] Wiederum abhängig von der Schreib- und Lesegeschwindigkeit des USB-Sticks

[21] Es gab fast einen Loss, als der stickbeladene Kopter nicht ganz seinen Landeplatz traf, aber sowohl Stick als auch Kopter blieben unversehrt. Es musste lediglich die Steuerleitung zu einem der vier Brushless-Motoren neu festgelötet werden.