Trends und Neuerungen bei der Protokollentwicklung

There is no one-line answer to the question "How fast can TCP go?"

Florian Westphal
fw@strlen.de

Hagen Paul Pfeifer
hagen.pfeifer@jauu.net

Easterhegg 2010 – München

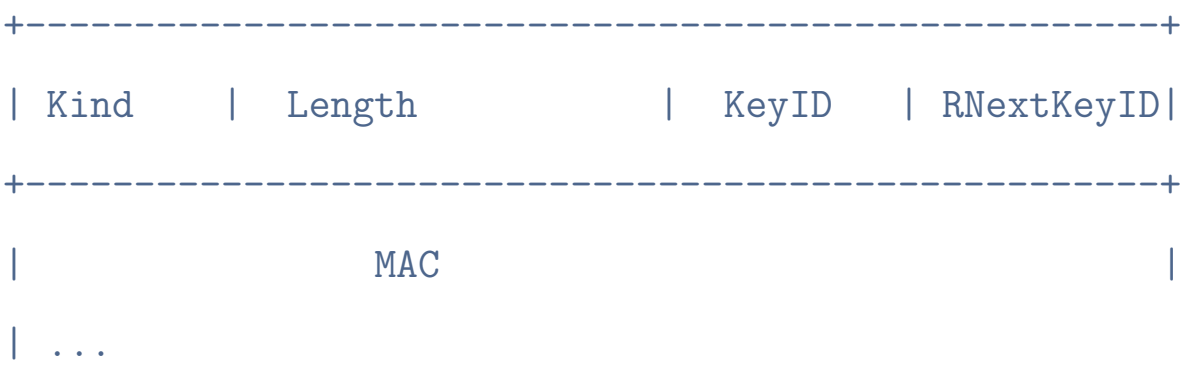
02. April 2010

Von MD5 zu AO

- Problem: Absichern von TCP-Verbindungen gegen "einschleusen" von Paketen
- Betrifft vor allem BGP (da langlebige Verbindungen)
- Bisher: TCP MD5 (RFC 2385)
  - MD5-Checksum über Paket & gemeinsamen Schlüssel
  - Checksumme steht in MD5 TCP-Option
  - Segment verwerfen, wenn md5sum ungültig

Authentication Option (AO)

- Löst TCP-MD5 ab (neue TCP-Option)
- Unterstützt verschiedene Hashfunktionen
- Replayschutz durch Erweiterung d. TCP-Sequenznummern auf 64bit
- Rekeying während laufender Verbindung
- Verbraucht 16 Byte im TCP-Header (spezifizierter default – TCPMD5: 18 Byte)



KeyIDs und SNE

- `KeyID`: Schlüssel-ID welche zur MAC-Erzeugung verwendet wurde (bzw. ID, welche f. Verifikation benutzt werden muss).
- `RNextKeyID`: nächster Schlüssel, der vom Peer verwendet werden kann. Beim Verbindungsaufbau: `KeyID == RNext`
- `SNE`: "Sequence Number Extension". Zähler, der bei jeden Überlauf der TCP-Sequenzummer inkrementiert wird
- SNE wird nicht mitgesendet

MKT – Master Key Tuple

- Eine Verbindung hat mindestens ein MKT.
- MKT kann f. beliebig viele Verbindungen verwendet werden
- MKT kann nicht verändert werden (aber es können neue hinzu bzw. alte enfernt werden)

MKT besteht aus ...

1. Adressen d. Endpunkte (src, dst, IP, sport, dport, Wildcards erlaubt)
2. Secret Key
3. MAC
4. Option flag – regelt, ob TCP-Optionen im MAC berücksichtigt werden

MACs f. TCP-AO in seperatem Dokument (tcp-ao-crypto – sha1-96, AES128-cmac)

MAC Berechnung

- MAC-Berechnung
  1. SNE
  2. IP Pseudoheader
  3. TCP Header (mit oder ohne Optionen, wird vorher festgelegt)
  4. TCP Daten (wenn vorhanden)
- Bei SYN: remote ISN=0.

TCP-AO und NAT

- NAT Extension in separatem Dokument
- Keine Veränderung des AO-Header
- Endpunkte verwenden NAT-Flag
- Bei `localNAT=1`: lokale IP-Addresse und Port bei Traffic-Key erzeugung 0 setzen
- Analog f. `remoteNAT=1`

SACK Loss Recovery

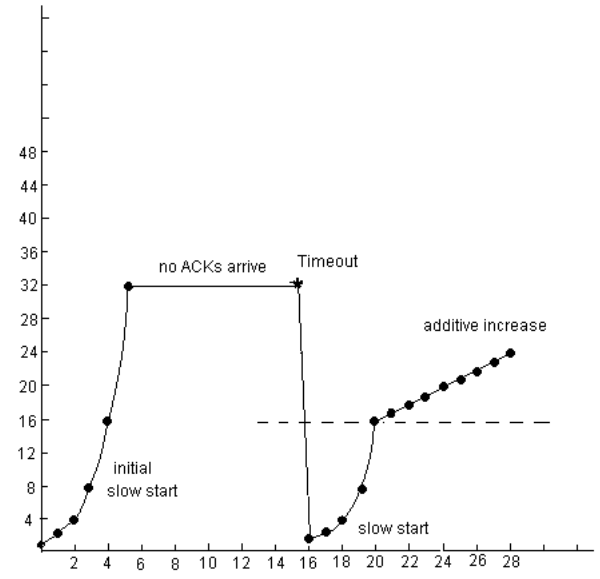
- ”Using SACK Information to Determine DupAcks for Loss Recovery Initiation” (draft-ietf-tcpm-sack-recovery-entry-00.txt)
- TCP hat traditionell zwei Retransmit-Auslöser:
  1. Retransmit-Timer
  2. Duplicate Acks (Schwellwert – z.B. 3)
- Wenn Timer abläuft oder genug Dupacks: Recovery-Phase (erneutes Übertragen noch nicht quittierter Pakete)
- Nutzung von SACK-Information während loss-recovery gut beschrieben (RFC 3782)
- . . . nicht aber für die Erkennung ob Paketverlust vorliegt
- Dupack-Zählung unter manchen Bedingungen problematisch (reordering, Verlust von ACKs, . . .)
- beschriebener Algorithmus im wesentlichen so im Linux TCP Stack implementiert

Algorithmus

- Algorithmus arbeitet während ”normalem” Sendevorgang, nicht während Recovery-Phase
- Soll tatsächliche Situation (out-of-order Empfang) besser erkennen
- Abschätzung, wieviele Pakete noch ”unterwegs” sind
- SACK-Scoreboard: hält Informationen über empfangene SACK-Blöcke
- $\longrightarrow$ kann erkennen,
  - Wenn dieselbe SACK-Information mehrfach empfangen wird
  - Wenn ACKs ”fehlen”
- Normales ACK, welches zusätzlich SACK-Blocke für in-window Daten enthält wird wie dupack behandelt

Staukontrolle

- Ziel ist eine faire Staukontrolle zu bewerkstelligen
  - Vorbeugen vor den Congestion Collapse
  - Fairness: alle Flows teilen sich die verfügbare Bandbreite (siehe Jain's fairness index)
  - Staukontrolle in TCP existiert seit 1988
  - UDP implementiert keine Staukontrolle
    - Anwendungen können UDP Pakete mit max. Linkrate versenden, diese übersteigt in der Regel die verfügbare Pfadkapazität um Längen



Staukontrolle

- Zusammenspiel zwischen Router und Host:
  - Router signalisiert Stau (a) Paket drop, b) ECN - frühzeitig
  - TCP Instanz reduziert Senderate bei erkannter Überlastsituation
  - Funktionsweise:
    - `cwnd < ssthresh`: slow start
    - `cwnd > ssthresh`: congestion avoidance

TCP und Staukontrolle

- Seit 1988 hat sich viel getan, SlowStart und Congestion Avoidance in seiner Urform ist in vielen Umgebungen (großes BDP) inadequate.
- Weit verbreitetste Varianten: NewReno und SACK
- Floyd (2003): HighSpeed TCP for Large Congestion Windows RFC 3649 Aber: HighSpeed TCP hat sich als nicht fair zu bestehenden TCP Variante erwiesen
- Newcomer: BIC, CUBIC, WESTWOOD, ...
- Linux: modulare CC Architektur, Unterstützung von einer Vielzahl von CC Algorithmen

UDP und Staukontrolle

- Kein Rückkanal - keine signalisierung möglich, keine Verbindungssemantik
- Viele Anwendungsprotokolle verlangen aber nach Eigenschaften von UDP: Voice- und Videostreaming, Gamingprotokolle, ...
- Ansatz: Datagram Congestion Control Protocol (DCCP)
  - Keine geordnete Zustellung oder Zuverlässigkeit
  - Unzuverlässiger Versand von Datagramen, mit ACKs
  - Zuverlässiger Verbindungsauf- und abbau
  - Zuverlässiger Aushandlung von Features
  - Mehrere Staukontrollalgorithmen (TCP-friendly, TCP-like congestion control, ...)
  - ECN
  - Path MTU discovery

Staukontrolle – Verschiedenes

- Multicast und Staukontrolle: kein Königsweg
  - `PGMCC: a TCP-friendly single-rate multicast congestion control scheme`
  - `Generalized Multicast Congestion Control (GMCC)`
  - ...

Quick Start TCP

- Initiales CWND: 1 bis 4 der TCP-MSS
- Kurzlebige Verbindungen konvergieren schlecht (Exponentielles Slow-start) – die zu Verfügung stehende Bandbreite wird nicht ausgenutzt
- Grosse RTTs verstärken dies
- Bei langlebigen Verbindungen ist slow-start schnell genug `[TM]` (Bulk data, z.B. `FTP`, ...)
- Verletzt nicht die Grundsätze: „be conservative in what you do, . . . “
- Slow-Start
  - Initial
  - Lange idle Perioden
  - Retransmission Timeouts
- Quick Start for TCP and IP (`RFC 4782`)
- Idee:
  - Erlaubt slow-start mit einem CWND größer von 4

- Konservativ: alle Netzelemente auf dem Pfad müssen zustimmen

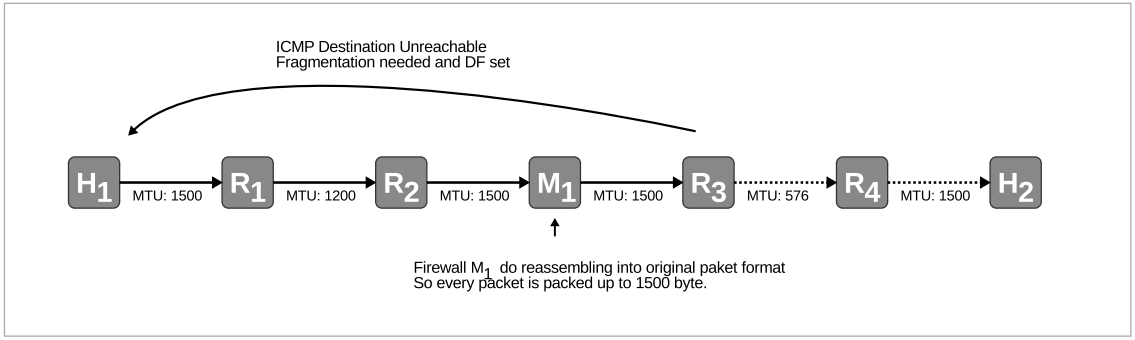
- Geschwindigkeitsgewinn für Interaktive Applikationen wenn die RTT is groß
- Implementierungsdetails:
  - Alle Router müssen zustimmen
  - Router müssen die Verfügbare Bandbreite bestimmen (Schichtübergreifend: IP-Link Layer)

Paket Fragmentierung

- Notwendig wenn nächster Link eine kleinere MTU aufweist als die größe des Paketes
- Die Frage ist wer die Fragmentierung vornimmt?
  - Router
  - Sender
- In den Anfangsjahren der Standardisierung hatte man die Auffassung vertreten das diese Funktionalität auch durch Router abgedeckt werden soll.
- Fragmentation is like classful addressing – an interesting early architectural error that shows how much experimentation was going on while IP was being designed. – Paul Vixie

Paket Fragmentierung soll vermieden werden

- Fragmentierung und Defragmentierung benötigt zusätzliche Bearbeitungszeit und führt zu höherer Latenz
- Fragmentierung verschlechtert das Kontroll/Nutzlast Verhältnis: jedes Fragment bekommt einen eigenen IP Header (dies wirkt sich negativ auf die PER aus).
- Fragmentierung bereitet Firewalls, Analyser, ... Probleme.
  - Im Zweifel werden Pakete verworfen wenn die Firewall nicht reassemblieren kann
  - Wenn die Firewall fragmentiert kann es zu Blackhole Problemen kommen (siehe Beispiel)
- Probleme im Zusammenspiel mit Tunneln (z.B. IPIP, Teredo, OpenVPN)



PMTUD

- Beantwortet die Frage was die kleinste MTU zwischen Sender und Empfänger ist
- Funktionsweise:
  - Sender setzt das Don't Fragment Bit im IP Header (bei IPv6 gibt es dieses Flag nicht - es ist immer gesetzt)
  - Generiert ICMP Paket: "Destination Unreachable-Fragmentation Needed and DF Bit Set"
  - Verhalten bei IPv6 (implizit gesetzt)

PMTUD und TCP

- Maximum Segment Size
- Beim Drei-Wege-Verbindungsaufbau wird die MSS übertragen
  - Initial ist die MSS die MTU des ausgehenden Links
  - die kleinste MSS beider Verbindungen wird verwendet
- Standardiserung/Implementationsverwirrung: Wie wird die MSS bestimmt?
  - RFC 879: MSS = MTU - sizeof(TCPHDR) - sizeof(IPHDR)
    `[..] TCP header is minimum size (20 octets), because there are no TCP header options currently defined [..]`
  - RFC 1122: "`Eff.snd.MSS = min(SendMSS+20, MMS_S) - TCPhdrsize - IPoptionsize`"
  - `draft-ietf-tcpm-tcpmss-03`:
    "`The MSS value to [..] should be equal to the effective MTU minus the fixed IP and TCP headers`"

Blackhole Detection

- Middleboxes sind oft nicht Standardkonform, sind fehlerhaft implementiert.
- Arten von Blackholes:
  - Router welche eine niedere MTU haben verwerfen Pakete einfach, ohne eine ICMP Nachricht zu verschicken (bei einigen Routern kann man sogar expliziet einstellen das keine ICMP Nachrichten generiert werden soll. Diese Option wurde wiederum eingeführt um ältere BSD Stacks zu schützen, welche bei jeglichen "destination unreachableÏCMP Nachrichten alle Verbindungen zu einen Host abbrechen)
  - ICMP Blocker
  - Window Scaling
  - MSS/SACK TCP reordering SOHO router bug

TCP Cookie Transactions: Motivation

- DNSSEC: DNS-Antworten sind größer
- Müssen häufig fragmentiert werden
- Nicht alle Middleboxes kommen damit zurecht
- Ggf. wiederholt Client Anfrage über TCP
- Nach signieren von .org: Anstieg der Anfragen über TCP um 600%

Zwei Probleme bei Nutzung von TCP:

1. Nach schliessen einer Verbindung: `TIME_WAIT` → Resourcenverbrauch
2. Höherer Protokolloverhead (3 WHS, Verbindungsabbau)

TCPCT sollen die Nachteile vermeiden

TCP Cookie Transactions

Vermengung vieler früherer Ideen:

- T/TCP (RFC 1644, 1994)
- TCP Cookies zur Vermeidung von Server-State während 3whs (Karn 1994)
- Long Options bzw. SL-Option (`draft-eddy-tcp-loo`, 2004)
- TIMEWAIT-Vermeidung (`draft-faber-time-wait-avoidance`, 1997)
- Randomisierte TCP Timestamps

Realität:

- T/TCP benutzt keiner (auch z.B. wegen Sicherheitsbedenken)
- Long Options Draft nie Formal veröffentlicht (hautpsächlich bedenken bez. Kompatibilität)
- Selbes gilt f. timewait-avoidance

TCP/CT "Lösung": tcpm-wg umgehen

TCP CT: Protokollübersicht Terminologie:

- Initiator – Client
- Responder – Server

Hauptmerkmale:

- Beide Parteien senden ”cookie” an den anderen
- Empfänger eines SYN mit cookie-Option bleibt stateless (wenn TCPCT unterstützt werden)
- TCB wird erst erstellt wennn ACK empfangen wird (nach cookie-Überprüfung)
- Cookies werden auch beim Verbindungsabbau genutzt → kein TIMEWAIT bei responder

TCP CT: Protokollübersicht

Terminologie:

- Initiator – Client
- Responder – Server

Hauptmerkmale:

- Beide Parteien senden ”cookie” an den anderen
- Empfänger eines SYN mit cookie-Option bleibt stateless (wenn TCPCT unterstützt werden)
- TCB wird erst erstellt wennn ACK empfangen wird (nach cookie-Überprüfung)
- Cookies werden auch beim Verbindungsabbau genutzt → kein TIMEWAIT bei responder

draft-simpson-tcpct-00.txt

- Zwingende Voraussetzungen: TCP Timestamps (RFC 1323), zufällige ISN (RFC 1948)
- Definiert 2 neue TCP Optionen:
  1. TCP Cookie Option (31)
  2. TCP Timestamps Extended Option (32)
- Option 1 wird via 'Length' Multiplexed:
  1. Länge 10-18: TCP Cookie Option (SYN, SYN/ACK)
  2. Länge 18-34: TCP Cookie Standard Option
  3. Länge 4: TCP Cookie-Pair Extended Option (nur bei RFC1323 Timestamps)
  4. Länge 2: TCP Cookie-less Extension Option
- TCP Cookie Option folgt immer direkt nach Option 32 (oder nach 32bit Timestamp)
- Patches und draft (fast) simultan veröffentlicht → kaum Diskussion `8-(`

Sonstiges ...

Viele Änderungen am TCP-Protokoll:

- Bisherige SYN-only Optionen – z.B. TCPMSS – auch nach 3whs erlaubt
- 'plain RST' auf existierenden Verbindungen werden ignoriert
- Serverseitig kein `TIME_WAIT` mehr ("rapid close")
- An TCPCT gekoppelte 64bit Timestamps

Implementation: partieller Support in Linux 2.6.33 (Option 253, keine 64bit TS, ...)

→ momentan unklar wie es weitergeht