Kapitel 8
Sliding Windows

In diesem Kapitel beschäftigen wir uns mit dem Aufbau zuverlässiger Datenübertragungsschichten auf unzuverlässigen tieferliegenden Schichten. Dies gelingt durch eine *Retransmit-on-Timeout*-Richtlinie, d. h., wenn ein Paket übertragen wird und innerhalb des Timeout-Intervalls keine Empfangsbestätigung eintrifft, wird das Paket erneut versendet. Protokolle, bei denen eine Seite Retransmit-on-Timeout implementiert, werden als *ARQ*-Protokolle (für *Automatic Repeat reQuest*) bezeichnet.

Neben der Zuverlässigkeit geht es uns zudem darum, so viele Pakete auf dem Weg zu halten, wie es das Netzwerk zulässt. Die Strategie, mit der dies erreicht werden soll, ist als *Sliding Windows* bekannt. Wir erkennen, dass der Sliding-Windows-Algorithmus auch der Schlüssel zur Bewältigung von Engpässen ist; darauf kommen wir später in Kapitel 19, »TCP Reno und Überlastmanagement«, zurück.

Das *Ende-zu-Ende*-Prinzip (Abschnitt 17.1) verdeutlicht, dass der Versuch fehlschlägt, durch eine zuverlässige tiefere Schicht eine zuverlässige Übertragungsschicht zu bekommen. Der richtige Weg ist es, über die Endpunkte einer Verbindung Zuverlässigkeit zu gewährleisten, so wie hier beschrieben.

8.1 Zuverlässige Datenübertragung: Stop-and-Wait

Retransmit-on-Timeout erfordert in der Regel eine fortlaufende Nummerierung der Pakete. Wenn allerdings gewährleistet ist, dass ein Netzwerkpfad die Paketreihenfolge nicht verändert, können die fortlaufenden Nummern überraschend schnell zurückgesetzt werden (bei Stop-and-Wait genügt eine Nummerierung mit einem einzelnen Bit). Da die Hypothese der unveränderten Reihenfolge jedoch für das Internet im Allgemeinen nicht als gegeben angesehen werden kann, gehen wir von einer konventionellen Nummerierung aus. Data[N] ist das N-te Datenpaket, das mit ACK[N] bestätigt wird.

Bei der *Stop-and-Wait*-Variante von Retransmit-on-Timeout sendet der Absender jeweils nur ein einzelnes ausstehendes Paket. Erfolgt keine Antwort, kann das Paket erneut gesendet werden, aber Data[N+1] verschickt der Sender erst, nachdem er ACK[N] erhalten hat. Natürlich sendet auch die Empfängerseite erst dann ACK[N], wenn sie Data[N] erhalten hat; *beide* Seiten bringen also immer nur ein Paket ins Spiel. Wenn keine Paketverluste auftreten, führt dies zu folgendem Ergebnis:

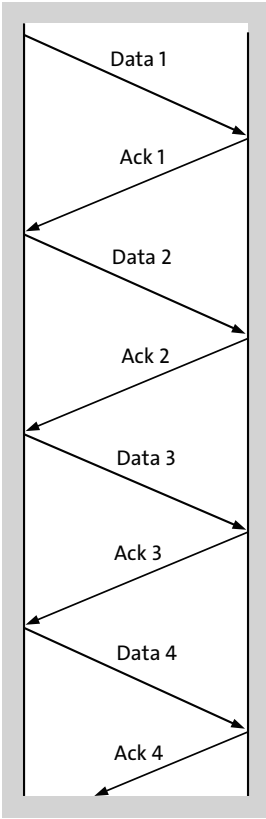


Abbildung 8.1 Stop and Wait

8.1.1 Paketverlust

Verlorene Pakete gehören jedoch zur Realität. Die linke Hälfte von Abbildung 8.2 zeigt ein verlorenes Datenpaket, wobei der Sender der Host ist, der Daten sendet, und der Empfänger der Host, der ACKs sendet. Der Empfänger ist sich des Verlusts nicht bewusst; für ihn sieht es einfach so aus, als ob Data[N] verspätet ankommt.

Die rechte Hälfte des Diagramms veranschaulicht im Vergleich dazu den Fall eines verlorenen ACK. Der Empfänger hat ein *Duplikat* von Data[N] erhalten. Wir gehen hier davon aus, dass der Empfänger eine *Retransmit-on-Duplicate*-Strategie implementiert hat, sodass seine Reaktion auf den Empfang des Duplikats von Data[N] die erneute Übertragung von ACK[N] ist.

Als letztes Beispiel sei in Abbildung 8.3 darauf hingewiesen, dass es möglich ist, dass ACK[N] (bzw. das erste Data[N]) erst nach dem Timeout-Intervall eintrifft. Nicht jedes Paket, das eine Zeitüberschreitung erfährt, ist auch tatsächlich verloren gegangen!

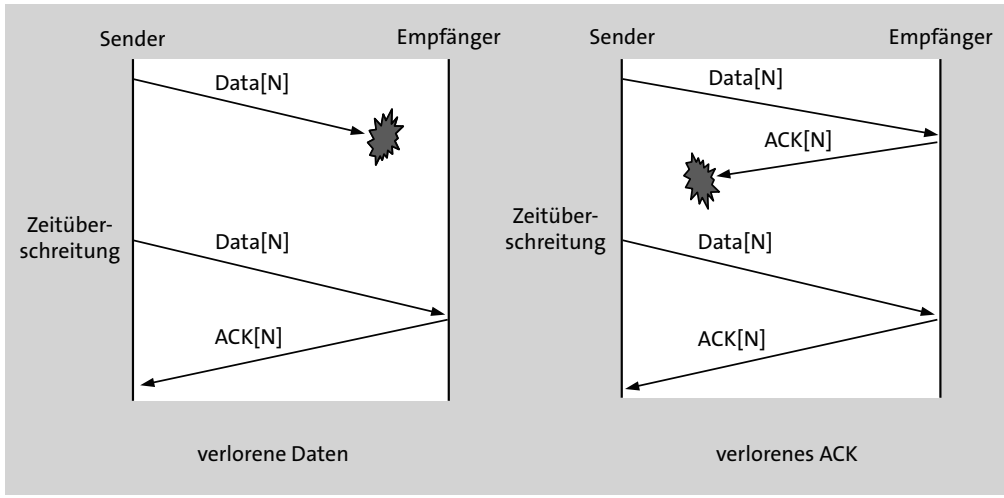


Abbildung 8.2 Paketverlust

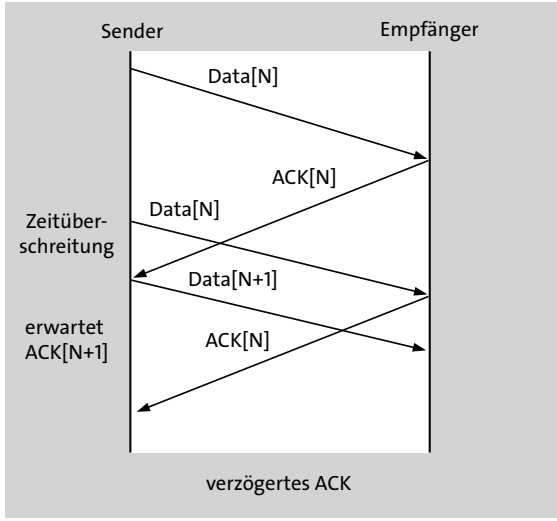


Abbildung 8.3 Verzögertes ACK

In diesem Fall erkennen wir, dass nach dem Versenden von Data[N] der Empfang eines verzögerten ACK[N] (statt des erwarteten ACK[N+1]) *als normales Ereignis angesehen werden muss*.

Im Prinzip kann jede Seite ein Retransmit-On-Timeout durchführen, wenn nichts empfangen wird. Beide Seiten können auch Retransmit-on-Duplicate implementie-

ren; dies hat der Empfänger im zweiten Beispiel oben getan, *nicht* aber der Sender im dritten Beispiel (der Sender hat ein zweites ACK[N] erhalten, aber Data[N+1] nicht erneut übertragen).

Mindestens eine Seite *muss* Retransmit-on-Timeout implementieren; andernfalls führt ein verlorenes Paket zu einem Blockadezustand, bei dem sowohl der Absender als auch der Empfänger ewig warten. Die andere Seite *muss* mindestens eine der Optionen Retransmit-on-Duplicate oder Retransmit-on-Timeout implementieren, normalerweise nur Erstere. Wenn beide Seiten Retransmit-on-Timeout mit unterschiedlichen Timeout-Werten implementieren, funktioniert das Protokoll in der Regel trotzdem noch.

8.1.2 Sorcerer's Apprentice Bug

Eine merkwürdige Sache passiert, wenn eine Seite Retransmit-on-Timeout implementiert, aber *beide* Seiten Retransmit-on-Duplicate verwenden. Dies kann passieren, wenn der Implementierer die naive Ansicht vertritt, dass Retransmit-on-Duplicate »sicherer« sei; die Moral lautet hier ist, dass zu viel Redundanz auch Schaden anrichten kann.

Stellen wir uns vor, dass eine Implementierung diese Strategie verwendet (der Sender wiederholt seine Übertragung bei Timeouts), und dass das anfängliche ACK[3] so lange verzögert wird, dass Data[3] bei Zeitüberschreitung erneut gesendet wird. In Abbildung 8.4 ist das einzige Paket, das aufgrund einer Zeitüberschreitung erneut übertragen wird, das zweite Data[3]; alle weiteren Duplikate sind auf die beidseitige Retransmit-on-Duplicate-Strategie zurückzuführen.

Alle Pakete werden ab Data[3] *doppelt* gesendet. Die Übertragung wird normal abgeschlossen, nimmt aber die doppelte Bandbreite in Anspruch. Die übliche Lösung besteht darin, dass eine Seite (in der Regel der Absender) nur bei Zeitüberschreitung erneut sendet. Bei TCP ist dies der Fall; siehe Abschnitt 18.12, »Zeitüberschreitung und Neuübertragung bei TCP«. Siehe auch Übung 2.

Der Zauberlehrling

Der »Sorcerer's Apprentice Bug« ist nach Goethes Ballade vom Zauberlehrling benannt: Dieser verhext einen Besen, damit er für ihn Wasser schleppen muss. Nachdem alle Eimer voll sind, zerhackt der Lehrling den Besen in zwei Teile, nur um festzustellen, dass nun beide Besenhälften immer weiter Wasser herbeischaffen. Eine Trickfilmversion ist in dem Disney-Film Fantasia zu sehen, unterlegt mit der Musik von Paul Dukas.

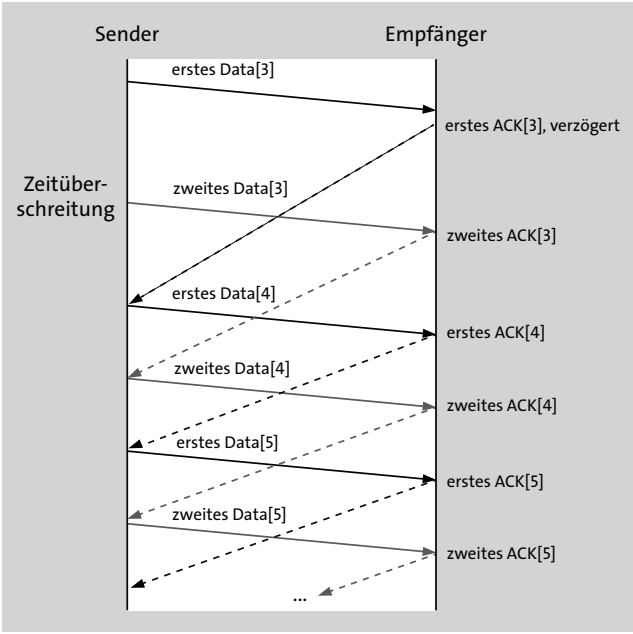


Abbildung 8.4 Der Sorcerer's Apprenctice Bug; erste Übertragungen in Schwarz, zweite Übertragungen in Blau

8.1.3 Datenflusssteuerung

Stop-and-Wait bietet auch eine einfache Form der *Flusssteuerung*, um zu verhindern, dass Daten schneller beim Empfänger ankommen, als sie verarbeitet werden können. Unter der Annahme, dass die zur Verarbeitung eines empfangenen Pakets benötigte Zeit kleiner als eine RTT ist, verhindert der Stop-and-Wait-Mechanismus, dass die Daten zu schnell ankommen. Liegt die Verarbeitungszeit leicht über der RTT, muss der Empfänger mit dem Senden von ACK[N] nur warten, bis Data[N] nicht nur angekommen ist, sondern auch verarbeitet wurde, und der Empfänger bereit für Data[N+1] ist.

Bei geringen Verarbeitungsverzögerungen pro Paket funktioniert dies recht gut, aber bei längeren Verzögerungen ergibt sich ein neues Problem: Data[N] kann eine Zeitüberschreitung aufweisen und erneut gesendet werden, obwohl es erfolgreich empfangen wurde; der Empfänger kann kein ACK senden, bevor er die Verarbeitung abgeschlossen hat. Ein Ansatz besteht darin, *zwei* Arten von ACKs zu haben: $ACK_{WAIT}[N]$ was bedeutet, dass Data[N] angekommen ist, der Empfänger aber noch nicht bereit für Data[N+1] ist, und $ACK_{GO}[N]$ was bedeutet, dass der Absender nun Data[N+1] senden kann. Der Empfänger sendet $ACK_{WAIT}[N]$ wenn Data[N] eintrifft, und $ACK_{GO}[N]$, wenn er mit der Verarbeitung dieses Pakets fertig ist.

Normalerweise wollen wir, dass der Sender nach dem Empfang von ACK_{WAIT}[N] keine Zeitüberschreitung verzeichnet und Data[N] erneut sendet, da eine erneute Übertragung unnötig wäre. Dies wirft ein neues Problem auf: Wenn das nachfolgende ACK_{GO}[N] verloren geht und keine der beiden Seiten ein Timeout verzeichnet, wird die Verbindung blockiert. Der Sender wartet auf das verloren gegangene ACK_{GO}[N] und der Empfänger wartet auf Data[N+1], das der Sender erst dann sendet, wenn das fehlende ACK_{GO}[N] eingetroffen ist. Eine Lösung besteht darin, dass der Empfänger auf ein Timeout-Modell umschaltet, vielleicht bis zum Eingang von Data[N+1].

TCP hat eine Lösung für das Problem der Flusssteuerung, die eine absenderseitige Abfrage beinhaltet; siehe Abschnitt 18.10, »Flusssteuerung bei TCP«.

8.2 Die Sliding-Windows-Strategie

Stop-and-Wait ist zuverlässig, aber nicht sonderlich effizient (es sei denn, der Pfad hat weder Zwischenstationen noch eine signifikante Ausbreitungsverzögerung, d. h. er besteht nur aus einer einzigen LAN-Verbindung). Die meisten Verbindungen entlang eines Multi-Hop-Stop-and-Wait-Pfads sind die meiste Zeit über im Leerlauf. Während einer Dateiübertragung wollen wir idealerweise keinen Leerlauf (zumindest nicht auf der langsamsten Verbindungsstrecke; siehe Abschnitt 8.3, »Lineare Flaschenhälse«).

Wir können den Gesamtdurchsatz verbessern, indem wir dem Absender gestatten, mit der Übertragung fortzufahren und Data[N+1] (und weitere Pakete) zu senden, *ohne* auf ACK[N] zu warten. Dabei darf der Absender den zurückgelieferten ACKs jedoch nicht *zu* weit vorauseilen. Wie wir noch sehen werden, enden zu schnell versendete Pakete schlichtweg in Warteschlangen oder, schlimmer noch, werden aus den Warteschlangen gelöscht. Wenn die Bandbreite der Netzwerkverbindungen groß genug ist, können Pakete außerdem auch auf der Empfängerseite verworfen werden.

Da nun beispielsweise Data[3] und Data[4] gleichzeitig unterwegs sein können, müssen wir uns erneut fragen, was ACK[4] bedeutet: Bedeutet es, dass der Empfänger nur Data[4] erhalten hat, oder bedeutet es, dass sowohl Data[3] als auch Data[4] angekommen sind? Wir nehmen Letzteres an, ACKs sind also *kumulativ*: ACK[N] kann erst gesendet werden, wenn Data[K] für alle K≤N eingetroffen ist. Nach diesem Verständnis gilt: Wenn ACK[3] verloren geht, wird dies durch ein später eintreffendes ACK[4] wettgemacht; andernfalls gilt: Wenn ACK[3] verloren geht, besteht die einzige Wiederherstellung darin, Data[3] erneut zu senden.

Der Absender wählt eine *Fenstergröße*, winsize. Die Grundidee von Sliding Windows ist, dass der Absender diese Anzahl von Paketen senden darf, bevor er auf ein ACK warten muss. Genauer gesagt verwaltet der Absender eine Zustandsvariable *last_ACKed*, die das letzte Paket darstellt, für das er ein ACK von der Gegenseite erhalten

hat; wenn die Datenpakete von 1 beginnend nummeriert werden, dann gilt zu Anfang last_ACKed = 0.

Fenstergröße

In diesem Kapitel gehen wir davon aus, dass sich winsize nicht ändert. TCP variiert die winsize jedoch nach oben und unten, mit dem Ziel, den Wert so groß wie möglich zu machen, ohne Überlastungen zu verursachen; wir werden in Kapitel 19, »TCP Reno und Überlastmanagement«, darauf zurückkommen.

Zu jedem Zeitpunkt kann der Absender Pakete mit den Nummern last_ACKed + 1 bis last_ACKed + winsize versenden; dieser Paketbereich wird als *Fenster* bezeichnet. Wenn die erste Verbindung im Pfad nicht die langsamste ist, wird der Absender in der Regel all diese Pakete gesendet haben.

Wenn ACK[N] mit N > last_ACKed eintrifft (typischerweise ist N = last_ACKed+1), dann *verschiebt* sich das Fenster *nach vorne*; wir setzen last_ACKed = N. Dies erhöht auch die Obergrenze des Fensters und gibt dem Absender die Möglichkeit, mehr Pakete zu senden. Mit winsize = 4 und last_ACKed = 10 ist das Fenster zum Beispiel [11,12,13,14]. Wenn ACK[11] eintrifft, verschiebt sich das Fenster auf [12,13,14,15], sodass der Absender Data[15] senden kann. Wenn stattdessen ACK[13] eintrifft, wird das Fenster nach [14,15,16,17] weitergeschoben (es sei daran erinnert, dass ACKs kumulativ sind), und drei weitere Pakete können gesendet werden. Wenn es keine Änderung der Paketreihenfolge und keine Paketverluste gibt (und jedes Paket einzeln mit einem ACK versehen wird), dann schiebt sich das Fenster in Einheiten von jeweils einem Paket vorwärts; das nächste ankommende ACK wird immer ACK[last_ACKed+1] sein.

Beachten Sie, dass die Rückgaberate der ACKs immer genau der Rate entspricht, mit der die langsamste Verbindungsstrecke Pakete übermittelt. Das heißt, wenn die langsamste Verbindung (die »Flaschenhals«-Verbindung) alle 50 ms ein Paket liefert, dann gehen diese Pakete alle 50 ms beim Empfänger ein und die ACKs kommen alle 50 ms zurück. Somit werden neue Pakete mit einer durchschnittlichen Rate gesendet, die genau der Auslieferungsrate entspricht; dies ist die *Selbsttaktungseigenschaft* beim Sliding-Windows-Verfahren. Die Selbsttaktung verringert die Netzwerklast, indem sie die *Senderate* automatisch reduziert, sobald der verfügbare Bandbreitenanteil im Flaschenhals abnimmt.

Auf Vimeo finden Sie ein Video, das Sliding Windows mit Winsize = 5 in in Aktion zeigt: *https://vimeo.com/150452468*. Die zweite Verbindung, 1–2, hat eine Kapazität von fünf Paketen in beide Richtungen, sodass ein »Flight« (volles Fenster) von fünf Paketen diese Verbindung genau auslasten kann. Die Verbindung 0–1 hat eine Übertragungsrate von einem Paket in beide Richtungen. Das Video wurde mit dem Netzwerk-Animator »nam« erstellt.

Der erste Flight mit fünf Datenpaketen verlässt Node 0 kurz nach T=0 und verlässt Node 1 etwa bei T=1 (Videozeit). Die folgenden Durchläufe finden im Abstand von etwa sieben Sekunden statt. Die winzigen Pakete, die sich von Node 2 nach links zu Node 0 bewegen, stellen ACKs dar; gleich zu Beginn des Videos kann man fünf zurückgesendete ACKs des vorherigen Fensters sehen. Zu jedem Zeitpunkt (außer in den Momenten, in denen gerade Pakete empfangen wurden) sind im Prinzip fünf Pakete unterwegs, die entweder als Datenpakete auf einer Verbindung übertragen werden, als ACK übertragen werden oder in einer Warteschlange stehen (letzteres tritt in diesem Video nicht auf). Aufgrund gelegentlicher Video-Artefakte sind in einigen Frames nicht alle ACK-Pakete sichtbar.

8.2.1 Bandbreite × Verzögerung

Wie bereits in Abschnitt 7.1, »Paketverzögerung«, erwähnt, stellt das Produkt aus Bandbreite und RTT die Datenmenge dar, die gesendet werden kann, bevor die erste Antwort eintrifft. Dies spielt eine große Rolle bei der Analyse von Übertragungsprotokollen. In der Literatur wird das Produkt Bandbreite×Verzögerung (Delay) oft mit BDP abgekürzt.

Das Produkt bandwith × RTT ist im Allgemeinen der optimale Wert für die Fenstergröße. Es gibt jedoch einen Haken: Wählt ein Absender eine größere Fenstergröße als diese, dann wächst die RTT – aufgrund von Warteschlangenverzögerungen – einfach bis zu dem Punkt, an dem bandwith × RTT der gewählten Fenstergröße entspricht. Das heißt, der eigene Verkehr einer Verbindung kann die RTT_{actual} bis weit über die RTT_{noLoad} verlängern; siehe »3. Fall: winsize = 6«, weiter unten für ein Beispiel. Aus diesem Grund interessiert sich ein Absender oft mehr für bandwith × RTT_{noLoad} oder zumindest für die ursprüngliche RTT, bevor die eigenen Pakete des Absenders zur Überlastung beigetragen haben.

Wir werden das Produkt bandwith × RTT_{noLoad} gelegentlich als *Transferkapazität* der Route bezeichnen. Wie weiter unten noch deutlicher wird, bedeutet eine kleinere winsize eine unzureichende Auslastung des Netzes, während eine größere winsize dafür sorgt, dass jedes Paket irgendwo in einer Warteschlange steht.

Nachfolgend sind vereinfachte Diagramme für Sliding Windows mit Fenstergrößen von 1, 4 und 6 dargestellt, jeweils mit einer Pfadbandbreite von 6 Paketen/RTT (also bandwith × RTT = 6 Pakete). Abbildung 8.5 zeigt die anfänglichen Pakete, die in einem Pulk gesendet werden; diese verteilen sich dann auf dem Weg durch den Engpass, sodass nach dem ersten Pulk gleichmäßige Abstände zwischen den Paketen bestehen. (Echte Sliding-Windows-Protokolle wie TCP versuchen im Allgemeinen, solche anfänglichen Bündel zu vermeiden).

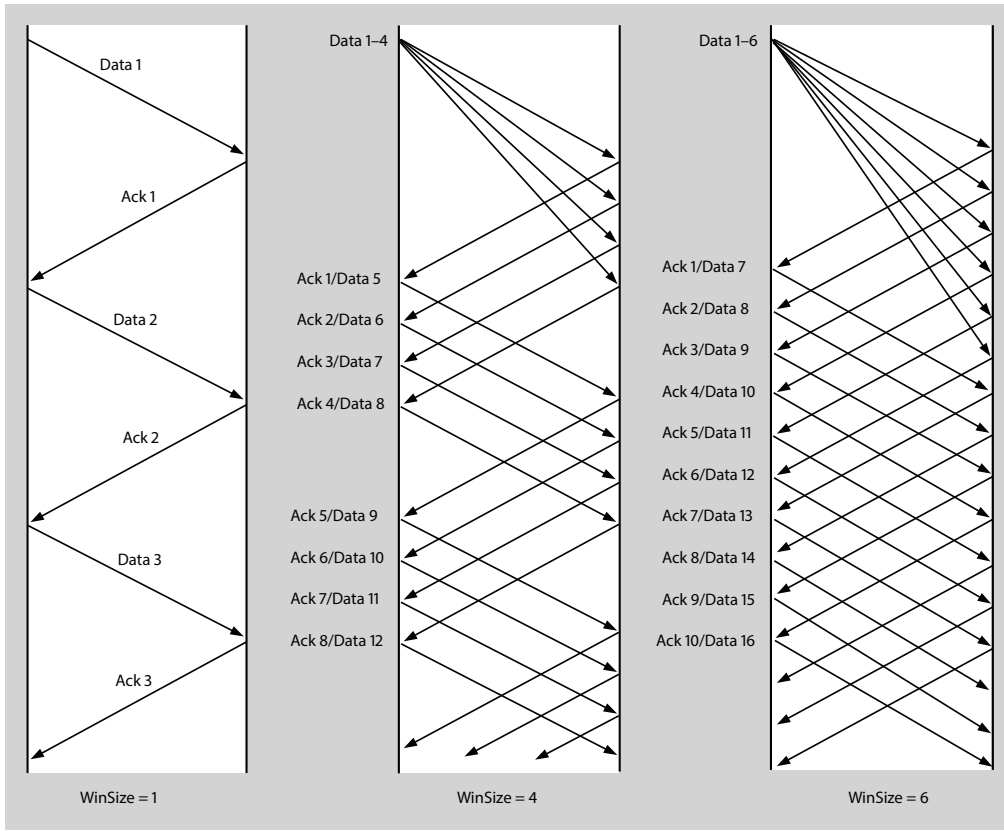


Abbildung 8.5 Sliding Windows, Bandbreite 6 Pakete/RTT

Mit winsize=1 senden wir 1 Paket pro RTT; mit winsize=4 senden wir *im Durchschnitt* immer 4 Pakete pro RTT. Anders ausgedrückt: Die drei Fenstergrößen führen zu einer Flaschenhalsauslastung von $1/6$, $4/6$ bzw. $6/6 = 100\%$.

Es ist zwar verlockend, die Fenstergröße auf bandwith × RTT zu setzen, aber in der Praxis erweist sich dies oft als kompliziert; weder die Bandbreite noch die RTT sind konstant. Die verfügbare Bandbreite kann aufgrund von konkurrierendem Datenverkehr schwanken. Wenn ein Absender winsize zu groß wählt, wird die RTT einfach bis zu dem Punkt aufgebläht, an dem bandwith × RTT mit winsize übereinstimmt; d. h. der eigene Traffic einer Verbindung kann RTT_{actual} bis weit über RTT_{noLoad} ansteigen lassen. Dies geschieht selbst ohne konkurrierenden Datenverkehr.

8.2.2 Die Empfängerseite

Es mag überraschen, dass Sliding Windows ziemlich gut funktionieren, wenn der Empfänger von winsize=1 ausgeht, selbst wenn der Absender in Wirklichkeit einen

viel größeren Wert verwendet. Jeder der Empfänger in den obigen Diagrammen empfängt Data[N] und antwortet mit ACK[N]; der einzige Unterschied mit der größeren Winsize des *Absenders* ist, dass die Data[N]-Pakete schneller ankommen.

Wenn wir den Sliding-Windows-Algorithmus für Einzelverbindungen verwenden, können wir davon ausgehen, dass die Pakete nie umsortiert werden, und eine Empfänger-Winsize von 1 funktioniert recht gut. Sobald jedoch Switches hinzukommen, wird es komplizierter (auch wenn einige Strecken zur Optimierung des Durchsatzes pro Verbindung Sliding-Windows *auf Verbindungsebene* einsetzen können).

Wenn eine Neuordnung der Pakete möglich ist, verwendet der Empfänger in der Regel die gleiche Fenstergröße wie der Sender. Das heisst, der Empfänger muss darauf vorbereitet sein, ein komplettes Fenster voller Pakete zu puffern. Wenn das Fenster z. B. [11,12,13,14,15,16] beträgt und Data[11] sich verspätet, muss der Empfänger möglicherweise Data[12] bis Data[16] puffern.

Wie der Absender verfügt auch der Empfänger über die Zustandsvariable last_ACKed, die jedoch nicht vollständig mit der Version des Absenders synchronisiert ist. Der Empfänger ist zu jedem Zeitpunkt bereit, Data[last_ACKed+1] bis Data[last_ACKed+winsize] zu akzeptieren. Jedes dieser eingehenden Pakete, außer dem ersten, muss der Empfänger puffern. Wenn Data[last_ACKed+1] eintrifft, sollte der Empfänger in seinem Pufferspeicher nachsehen und das größtmögliche kumulative ACK für die empfangenen Daten zurücksenden; wenn z. B. das Fenster [11–16] beträgt und sich Data[12], Data[13] und Data[15] in den Puffern befinden, dann ist bei Eingang von Data[11] die richtige Antwort ACK[13]. Data[11] füllt die »Lücke«, und der Empfänger hat nun alles bis zu Data[13] empfangen. Das neue Empfangsfenster ist [14–19], und sobald das ACK[13] den Absender erreicht, wird dies auch zum neuen Sendefenster.

8.2.3 Verlustwiederherstellung unter Sliding Windows

Angenommen, winsize = 4 und Paket 5 ist verloren gegangen. Es ist durchaus möglich, dass die Pakete 6, 7 und 8 empfangen wurden. Die einzige (kumulative) Bestätigung, die zurückgeschickt werden kann, ist jedoch ACK[4]; der Absender weiß nicht, wie viel des Fensterinhalts durchkam. Wegen der Möglichkeit, dass *nur* Data[5] (oder allgemeiner Data[last_ACKed+1]) verloren gegangen ist, und weil Verluste in der Regel mit einer Netzwerküberlastung einhergehen, wo wir das Netzwerk also gerade *nicht* weiter über Gebühr strapazieren möchten, wird der Absender in der Regel nur das erste verlorene Paket, z. B. Paket 5, erneut senden. Wenn die Pakete 6, 7 und 8 ebenfalls verloren gegangen sind, erhält der Sender nach der erneuten Übertragung von Data[5] ACK[5] und kann davon ausgehen, dass nun Data[6] gesendet werden muss. Wenn jedoch die Pakete 6–8 durchkamen, erhält der Sender nach der erneuten Übertragung ACK[8] zurück und weiß somit, dass 6–8 nicht erneut zu übermitteln sind und dass als nächstes Paket Data[9] gesendet werden kann.

Normalerweise würden Data[6] bis Data[8] kurz nach Data[5] ein Timeout erleiden. Nach der ersten Zeitüberschreitung unterdrücken Protokolle mit Sliding Windows jedoch im Allgemeinen weitere Zeitüberschreitungs-/Wiederübertragungsantworten, bis die Wiederherstellung weitgehend abgeschlossen ist.

Beim Eintritt einer vollständigen Zeitüberschreitung ist in der Regel der Sliding-Windows-Prozess selbst zum Erliegen gekommen, da dann meist auch keine Pakete mehr im Umlauf sind. Dies wird manchmal als *pipeline drain* bezeichnet. Nach der Wiederherstellung muss der Sliding-Windows-Prozess wieder anlaufen. Die meisten TCP-Implementierungen verfügen, wie wir später sehen werden, über einen (»Fast Recovery«)-Mechanismus zur frühzeitigen Erkennung von Paketverlusten, bevor die Pipeline vollständig leergelaufen ist.

8.3 Lineare Flaschenhälse

Betrachten Sie den Abbildung 8.6 dargestellten einfachen Netzwerkpfad. Die jeweiligen Bandbreiten sind in Paketen/ms angegeben. Die Mindestbandbreite, und damit auch die *Pfadbandbreite*, beträgt 3 Pakete/ms.

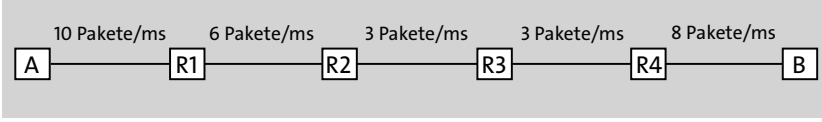


Abbildung 8.6 Ein einfacher Netzwerkpfad

Die langsamen Verbindungen sind R2-R3 und R3-R4. Wir werden die langsamste Verbindung als *Flaschenhals* oder *Bottleneck* bezeichnen; wenn es (wie hier) am unteren Ende mehrere gleich langsame Verbindungen gibt, dann gilt die erste dieser Verbindungen als Flaschenhals. Der Flaschenhals ist die Stelle, an der sich die Warteschlange ausbildet. Wenn der Datenverkehr mit einer Übertragungsrate von 4 Paketen pro Sekunde von A nach B gesendet wird, staut er sich in einer immer länger werdenden Warteschlange bei R2. Bei R3 entsteht *kein* Stau; der Traffic kommt hier mit der gleichen Geschwindigkeit an, mit der er auch wieder abfließt.

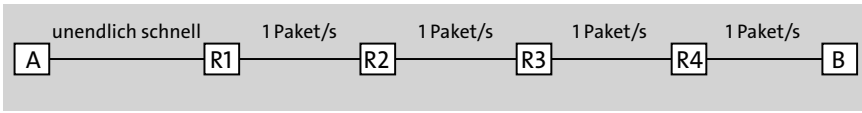
Wenn außerdem Sliding Windows verwendet wird (statt dass der Absender mit einer festen Rate sendet), staut sich der Netzwerkverkehr letztendlich bei keinem anderen Router als bei R2: Die Daten können B nicht schneller als mit der Rate von 3 Paketen/ms erreichen, und daher wird B auch keine ACKs mit einer höheren Rate zurücksenden, und daher wird A letztendlich auch keine Daten mit einer höheren Rate als dieser verschicken. Bei dieser Rate von 3 Paketen pro Sekunde staut sich der Verkehr nicht bei R1 (oder R3 oder R4).

Es bringt einen erheblichen Vorteil mit sich, eher von winsize als von *Übertragungsrate* zu sprechen. Wenn A mit einer Rate von über 3 Paketen/ms an B sendet, ergibt

sich eine instabile Situation, da die Flaschenhals-Warteschlange unbegrenzt anwächst und keine Konvergenz hin zu einem stabilen Zustand besteht. Es tritt aber keine analoge Instabilität auf, wenn A Sliding Windows verwendet, selbst wenn die gewählte winsize recht groß ist (obwohl eine ausreichend große winsize die Warteschlange am Flaschenhals überlaufen lässt). Wenn ein Absender eher eine Sendefenstergröße als eine Rate angibt, konvergiert das Netz in kurzer Zeit zu einem stabilen Zustand; sollte sich eine Warteschlange bilden, wird diese mit der gleichen Rate, mit der die Pakete von dort abgehen, stetig wieder aufgefüllt und hat somit eine feste Größe.

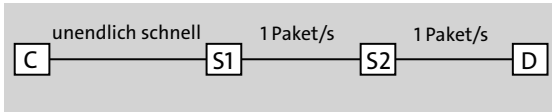
8.3.1 Einfache Analyse für feste Fenstergröße

Wir werden die Auswirkungen der Fenstergröße auf den Gesamtdurchsatz und auf die RTT analysieren. Betrachten wir den folgenden Netzwerkpfad, wobei die Bandbreiten nun in Paketen/*Sekunde* angegeben sind.



Wir nehmen an, dass in der Rückwärtsrichtung B⟶A alle Verbindungen unendlich schnell sind, d. h. mit einer Verzögerung von Null; dies ist oft ein guter Näherungswert, da in dieser Richtung nur ACK-Pakete übertragen werden und diese vernachlässigbar klein sind. In der A⟶B-Richtung nehmen wir an, dass die A⟶R1-Verbindung unendlich schnell ist, die anderen vier aber jeweils eine Bandbreite von 1 Paket/Sekunde (und keine Laufzeitkomponente) haben. Dadurch wird die Verbindung R1⟶R2 zum Flaschenhals; jegliche Warteschlange bildet sich also bei R1. Die »Pfadbandbreite« beträgt 1 Paket/Sekunde, und die RTT liegt bei 4 Sekunden.

Als annähernd gleichwertiges Alternativbeispiel könnten wir Folgendes verwenden:



Die C-S1-Verbindung ist unendlich schnell (null Verzögerung), S1⟶S2 und S2⟶D benötigen jeweils 1,0 Sekunden Bandbreitenverzögerung (zwei Pakete benötigen also 2,0 Sekunden pro Verbindung usw.), und ACKs haben ebenfalls eine Bandbreitenverzögerung von 1,0 Sekunden in umgekehrter Richtung.

Wenn wir in beiden Szenarien ein Paket senden, dauert es in einem unausgelasteten Netz 4,0 Sekunden, bis das ACK zurückkommt. Dies bedeutet, dass die Verzögerung ohne Netzwerklast, RTT_{noLoad}, 4,0 Sekunden beträgt.

(Diese Modelle ändern sich erheblich, wenn wir die Bandbreitenverzögerung von 1 Paket/Sekunde durch eine *Ausbreitungsverzögerung* von 1 Sekunde ersetzen; im ersten Fall benötigen 2 Pakete 2 Sekunden, im zweiten Fall 1 Sekunde. Siehe Übung 6.)

Wir gehen davon aus, dass nur eine einzige Verbindung hergestellt wird, d. h. es gibt keine Konkurrenz. bandwidth × delay beträgt hier 4 Pakete (1 Paket/Sekunde × 4 Sekunden RTT)

1. Fall: winsize = 2

In diesem Fall ist winsize < bandwidth × delay (mit delay = RTT). Die folgende Tabelle zeigt, was in jeder Sekunde von A und von R1-R4 gesendet wird. Jedes Paket wird 4 Sekunden nach dem Senden bestätigt, d. h. RTT_{actual} = 4 s und gleich RTT_{noLoad}; dies bleibt auch bei kleinen Änderungen der winsize (z. B. auf 1 oder 3) gültig. Der Durchsatz ist proportional zu winsize: Wenn winsize = 2 ist, beträgt der Durchsatz 2 Pakete in 4 Sekunden, oder 2/4 = 1/2 Paket/s. Während jeder Sekunde bleiben zwei der Router R1-R4 untätig. Der gesamte Pfad wird zu weniger als 100 % ausgelastet.

Zeit	A	R1	R1	R2	R3	R4	B
T	sendet	puffert	sendet	sendet	sendet	sendet	ACKs
0	1,2	2	1				
1			2	1			
2				2	1		
3					2	1	
4	3		3			2	1
5	4		4	3			2
6				4	3		
7					4	3	
8	5		5			4	3
9	6		6	5			4

Beachten Sie den kurzen Rückstau bei R1 (dem Flaschenhals!) zu Beginn. Im eingefahrenen Zustand gibt es dann aber keine Warteschlangen mehr. Echte Sliding-Windows-Protokolle verfügen in der Regel über eine Möglichkeit, diesen »Initial Pileup« zu minimieren.

2. Fall: winsize = 4

Bei winsize=4 sind in jeder Sekunde alle vier langsamen Verbindungen belegt. Es gibt wieder einen anfänglichen Datenstoß, der zu einem kurzen Ansteigen der Warteschlange führt; RTT_{actual} für Data[4] beträgt 7 Sekunden. Die RTT_{actual} für jedes nachfolgende Paket beträgt jedoch 4 Sekunden, und nach T=2 gibt es keine Verzögerungen in der Warteschlange (und die Warteschlange ist leer). Der stabile Datendurchsatz der Verbindung beträgt 4 Pakete in 4 Sekunden, also 1 Paket/Sekunde. Beachten Sie, dass der Gesamtdurchsatz des Pfads nun der Bandbreite der Flaschenhalsstrecke entspricht, dies ist also der bestmögliche Durchsatz.

T	A sendet	R1 puffert	R1 sendet	R2 sendet	R3 sendet	R4 sendet	B ACKs
0	1,2,3,4	2,3,4	1				
1		3,4	2	1			
2		4	3	2	1		
3			4	3	2	1	
4	5		5	4	3	2	1
5	6		6	5	4	3	2
6	7		7	6	5	4	3
7	8		8	7	6	5	4
8	9		9	8	7	6	5

Bei T=4 hat R1 gerade das Senden von Data[4] abgeschlossen, wenn Data[5] von A eintrifft; R1 kann sofort mit dem Senden von Paket 5 beginnen. Dabei entsteht keine Warteschlange.

Der 2. Fall entspricht dem »congestion knee« (Überlastungsgrenze) von of Chiu und Jain [CJ89], definiert in Abschnitt 1.7, »Überlast«.

3. Fall: winsize = 6

T	A sendet	R1 puffert	R1 sendet	R2 sendet	R3 sendet	R4 sendet	B ACKs
0	1,2,3,4,5,6	2,3,4,5,6	1				
1		3,4,5,6	2	1			
2		4,5,6	3	2	1		
3		5,6	4	3	2	1	

T	A sendet	R1 puffert	R1 sendet	R2 sendet	R3 sendet	R4 sendet	B ACKs
4	7	6,7	5	4	3	2	1
5	8	7,8	6	5	4	3	2
6	9	8,9	7	6	5	4	3
7	10	9,10	8	7	6	5	4
8	11	10,11	9	8	7	6	5
9	12	11,12	10	9	8	7	6
10	13	12,13	11	10	9	8	7

Beachten Sie, dass Paket 7 bei T=4 gesendet und die Bestätigung bei T=10 empfangen wird, was einer RTT von 6,0 Sekunden entspricht. Alle nachfolgenden Pakete haben die gleiche RTT_{actual}. Das heißt, die RTT ist von RTT_{noLoad} = 4 Sekunden auf 6 Sekunden angestiegen. *Beachten Sie, dass wir weiterhin ein komplett gefülltes Fenster pro RTT senden*; d. h. der Durchsatz beträgt immer noch winsize/RTT, aber die RTT liegt jetzt bei 6 Sekunden.

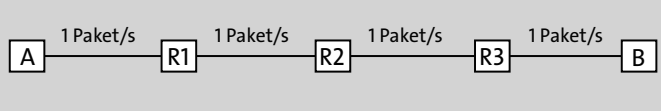
Man könnte zunächst vermuten, dass bei einer winsize größer als dem Produkt aus bandwith×RTT_{noLoad} das gesamte Fenster nicht auf einmal in der Übertragung sein kann. Dies stimmt allerdings nicht; der Absender *hat* meistens das gesamte Fenster gesendet und in Übertragung, aber *die RTT wurde in die Höhe getrieben*, sodass es für den Absender so aussieht, als wäre winsize gleich dem Produkt aus bandwith×RTT.

Wenn winsize > bandwith×RTT_{noLoad} ist, stauen sich die zusätzlichen Pakete im Allgemeinen irgendwo auf dem Weg an einem Router (insbesondere an dem Router vor dem Flaschenhals). RTT_{actual} wird durch die Warteschlangenverzögerung auf winsize/Bandwidth angehoben, wobei die Bandbreite jene des Flaschenhalses ist; das bedeutet winsize = bandwith×RTT_{actual}. Der Gesamtdurchsatz ergibt sich aus dieser Bandbreite.

Von den 6 Sekunden RTT_{actual} in diesem Beispiel verbringt ein Paket 4 Sekunden damit, auf den einzelnen Verbindungen übertragen zu werden, da RTT_{noLoad}=4. Die anderen beiden Sekunden müssen also in einer Warteschlange zugebracht werden; es gibt keinen anderen Ort, an dem Pakete warten könnten. Ein Blick auf die Tabelle zeigt, dass sich tatsächlich in jeder Sekunde zwei Pakete in der Warteschlange bei R1 befinden.

Wenn sich der Flaschenhals in der allerersten Verbindung befindet, kann es sein, dass die Pakete zurückkommen, bevor der Absender das gesamte Fenster abgeschickt hat.

In diesem Fall können wir argumentieren, dass das gesamte Fenster vom Absender zumindest in die Warteschlange gestellt und somit in diesem Sinne »gesendet« wurde. Angenommen, das Netz sei zum Beispiel



wobei, wie zuvor, jede Verbindung 1 Paket/Sekunde von A nach B transportiert und in der umgekehrten Richtung unendlich schnell ist. Wenn A dann winsize = 6 setzt, bildet sich bei A eine Warteschlange von 2 Paketen aus.

8.3.2 RTT-Berechnungen

Wir können einige quantitative Beobachtungen zum Verhalten von Sliding Windows und zur Auslastung der Warteschlangen machen. Erstens stellen wir fest, dass RTT_{noLoad} die physische »Reisedauer« ist (vorbehaltlich der in Abschnitt 7.2, »Schwankungen der Paketverzögerung«, angesprochenen Einschränkungen); jeder Zeitbetrag, der über RTT_{noLoad} hinausgeht, wird irgendwo in einer Warteschlange zugebracht. Daher gilt das Folgende unabhängig von konkurrierendem Datenverkehr und sogar für einzelne Pakete:

$$queue\_time = RTT_{actual} - RTT_{noLoad}$$

Wenn der Flaschenhals gesättigt (d. h. immer ausgelastet) ist, ist die Anzahl der Pakete, die sich tatsächlich auf dem Weg befinden (nicht in der Warteschlange stehen), immer $bandwith \times RTT_{noLoad}$.

Zweitens senden wir pro tatsächlicher RTT immer genau ein Fenster voll, wobei wir davon ausgehen, dass keine Verluste auftreten und jedes Paket einzeln bestätigt wird. Dies ist vielleicht am besten durch einen Blick auf die obigen Diagramme nachzuvollziehen, aber hier ist ein einfaches, nichtvisuelles Argument: Wenn wir Data[N] zum Zeitpunkt T_D senden und ACK[N] zum Zeitpunkt T_A eintrifft, dann gilt per Definition $RTT = T_A - T_D$. Zum Zeitpunkt T_A darf der Absender Data[N+winsize] senden, also muss der Absender während des RTT-Intervalls $T_D \leq T < T_A$ Data[N] bis Data[N+winsize–1] gesendet haben; das heißt, die Anzahl winsize an Paketen in der Zeit RTT. Daher gilt (unabhängig davon, ob es konkurrierenden Verkehr gibt oder nicht) immer

$$throughput = winsize/RTT_{actual}$$

wobei »throughput« die Paketsenderate der Verbindung ist.

Diese Beziehung gilt auch dann, wenn sich die winsize oder die Bandbreite des Flaschenhalses plötzlich ändert, obwohl sich in diesem Fall die RTT_{actual} von einem Pa-

ket zum nächsten ändern kann und der Durchsatz hier als eine über die RTT eines bestimmten Pakets gemittelte Messung betrachtet werden muss. Wenn der Absender seine Winsize verdoppelt, landen diese zusätzlichen Pakete sofort irgendwo in einer Warteschlange (vielleicht eine Warteschlange beim Absender selbst, weshalb es in Beispielen oft klarer ist, wenn die erste Verbindung eine unendliche Bandbreite hat, um dies zu verhindern). Wenn die Bandbreite des Flaschenhalses halbiert wird, ohne die Winsize zu verändern, muss die RTT aufgrund der Warteschlange ansteigen. Siehe Übung 17.

Im *stationären Zustand* von Sliding Windows, in dem Durchsatz und RTT_{actual} einigermaßen konstant sind, entspricht die durchschnittliche Anzahl der Pakete in der Warteschlange einfach throughput×queue_time (wobei der Durchsatz in Paketen/Sek. gemessen wird):

$$\begin{aligned} \text{queue\_usage} &= \text{throughput} \times (RTT_{actual} - RTT_{noLoad}) \\ &= \text{winsize} \times (1 - RTT_{noLoad}/RTT_{actual}) \end{aligned}$$

Um die Mittelwertbildung etwas zu verdeutlichen, verbringt jedes Paket die Zeit $(RTT_{actual} - RTT_{noLoad})$ in der Warteschlange, wie aus der ersten Gleichung oben hervorgeht. Die Gesamtzeit, die ein Fenster voller Pakete aufwendet, beträgt winsize × $(RTT_{actual} - RTT_{noLoad})$, und wenn man dies durch RTT_{actual} dividiert, erhält man die durchschnittliche Anzahl der Pakete in der Warteschlange über das betreffende RTT-Intervall.

Bei vorhandenem konkurrierendem Traffic stellt der oben erwähnte Durchsatz einfach den aktuellen Anteil der Verbindung an der Gesamtbandbreite dar. Diesen Wert erhalten wir, wenn wir die Rate der zurückkommenden ACKs messen. Wenn es keinen konkurrierenden Datenverkehr gibt und winsize unterhalb des Congestion Knee liegt – winsize < bandwidth × RTT_{noLoad} – dann ist winsize der begrenzende Faktor für den Durchsatz. Wenn es schließlich keinen Wettbewerb gibt und winsize ≥ bandwidth × RTT_{noLoad} ist, dann nutzt die Verbindung 100% der Kapazität des Flaschenhalses und der Durchsatz der Gesamtverbindung ist gleich der physischen Bandbreite des Flaschenhals-Links. Mit anderen Worten,

$$\begin{aligned} RTT_{actual} &= \text{winsize}/\text{bottleneck\_bandwidth} \\ \text{queue\_usage} &= \text{winsize} - \text{bandwidth} \times RTT_{noLoad} \end{aligned}$$

Dividiert man die erste Gleichung durch RTT_{noLoad} und stellt fest, dass bandwidth × RTT_{noLoad} = winsize – queue_usage = transit_capacity, erhalten wir

$$\begin{aligned} RTT_{actual}/RTT_{noLoad} &= \text{winsize}/\text{transit\_capacity} \\ &= (\text{transit\_capacity} + \text{queue\_usage}) / \text{transit\_capacity} \end{aligned}$$

Unabhängig vom Wert von winsize sendet der Absender im stationären Zustand niemals schneller als die Flaschenhals-Bandbreite. Das liegt daran, dass die Flaschenhals-

Bandbreite die Rate der an der Gegenseite ankommenden Pakete bestimmt, die wiederum die Rate der beim Sender ankommenden ACKs bestimmt, die wiederum die weitere Senderate bestimmt. Dies veranschaulicht die selbsttaktende Eigenschaft von Sliding Windows.

In Kapitel 20, »TCP-Dynamik«, kommen wir auf die Frage der Bandbreite bei konkurrierendem Traffic zurück. Nehmen wir an, ein Sliding-Windows-Sender verwendet winsize > bandwidth × RTT_{noLoad}, was, so wie oben, zu einer festen Auslastung der Warteschlange führt, und es gibt keinen Wettbewerb. Dann startet eine andere Verbindung und konkurriert um die Flaschenhalsstrecke. Die *effektive* Bandbreite der ersten Verbindung verringert sich dadurch. Das bedeutet also, dass bandwidth × RTT_{noLoad} abnimmt und somit die Auslastung der Warteschlange der Verbindung zunimmt.

8.3.3 Grafiken an der Überlastungsgrenze

Betrachten Sie in Abbildung 8.7 die folgenden Diagramme der Fenstergröße gegenüber

- Durchsatz
- Verzögerung
- Warteschlangenauslastung

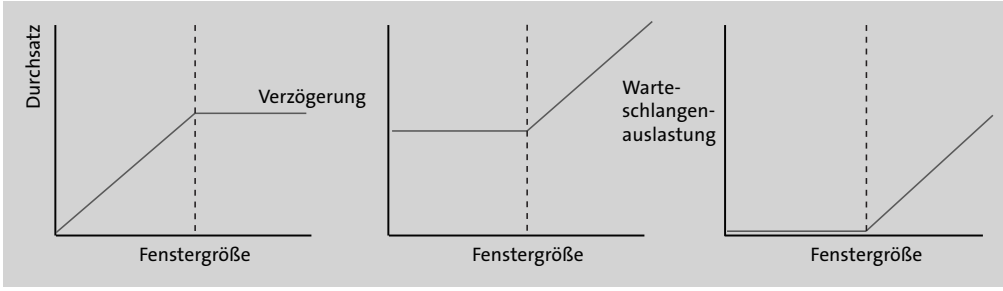


Abbildung 8.7 Graphen der Fenstergröße gegenüber Durchsatz, Verzögerung und Warteschlangenauslastung. Die vertikale gestrichelte Linie steht für winsize = Bandbreite × Verzögerung bei Nulllast

Der kritische Wert für winsize ist gleich bandwidth × RTT_{noLoad}; er wird als *Überlastungsgrenze* (»congestion knee«) bezeichnet. Für winsize unterhalb dieses Wertes gilt:

- der Durchsatz ist proportional zur Fenstergröße
- die Verzögerung ist konstant
- die Warteschlangenauslastung ist im stationären Zustand gleich Null

Für jede winsize oberhalb dieses Werts gilt

- der Durchsatz ist konstant (gleich der Flaschenhalsbandbreite)
- die Verzögerung steigt linear mit winsize an
- die Warteschlangenauslastung steigt linear mit winsize an

Im Idealfall liegt winsize an der kritischen Grenze. Der genaue Wert schwankt jedoch mit der Zeit: Die verfügbare Bandbreite ändert sich durch das Ein- und Aussetzen von konkurrierendem Datenverkehr und die RTT ändert sich aufgrund von Warteschlangen. Standardmäßiges TCP ist bestrebt, die meiste Zeit deutlich *über* der Grenze zu bleiben, vermutlich aufgrund der Überlegung, dass die Maximierung des Durchsatzes wichtiger ist als die Minimierung des Warteschlangenbedarfs.

Die *Leistung* einer Verbindung ist definiert als Durchsatz/RTT. Für Sliding Windows unterhalb der kritischen Grenze ist die RTT konstant und die Leistung ist proportional zur Fenstergröße. Für Sliding Windows oberhalb der Schwelle ist der Durchsatz konstant und die Verzögerung proportional zur Fenstergröße; die Leistung ist also proportional zu $1/\text{winsize}$. Abbildung 8.8 zeigt ein ähnliches Diagramm wie das vorhergehende, Fenstergröße gegen Leistung:

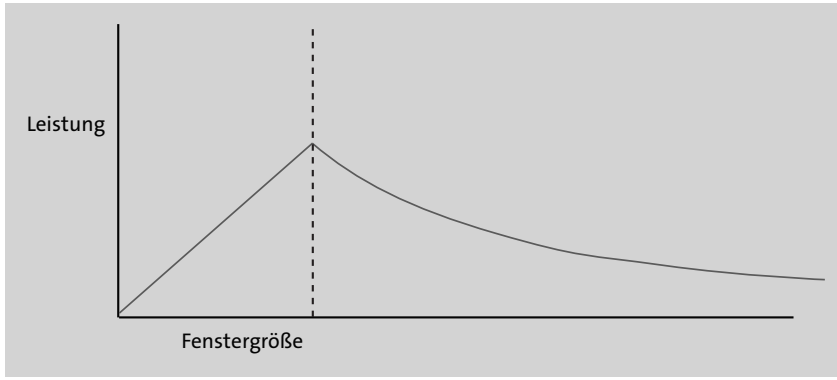


Abbildung 8.8 Das Verhältnis von Fenstergröße zur Leistung

8.3.4 Einfache paketbasierte Sliding-Windows-Implementierung

Hier ist ein Pseudocode für die Empfängerseite einer Sliding-Windows-Implementierung, wobei verlorene Pakete und Timeouts ignoriert werden. Wir nutzen die folgenden Abkürzungen:

```
W: winsize
LA: last_ACKed
```

Das nächste erwartete Paket ist also LA+1 und das Fenster ist [LA+1, ..., LA+W]. Wir haben eine Datenstruktur EarlyArrivals, in der wir Pakete ablegen können, die noch nicht an die Empfängeranwendung geliefert werden können.

Bei Eintreffen von Data[M]:

```
if M≤LA or M>LA+W, ignoriere das Paket
if M>LA+1, lege das Paket in EarlyArrivals ab
if M==LA+1:
   liefere das Paket (also Data[LA+1]) an die Anwendung
   LA = LA+1 (verschiebe das Fenster um 1 nach vorne)
   while (Data[LA+1] is in EarlyArrivals) {
      output Data[LA+1]
      LA = LA+1
   }
   send ACK[LA]
```

Eine mögliche Implementierung von EarlyArrivals ist ein Array von Paketobjekten der Größe W. Wir setzen das Paket Data[M] immer an die Position M % W.

Zu keinem Zeitpunkt zwischen den Paketankünften befindet sich Data[LA+1] in EarlyArrivals, aber einige nachfolgende Pakete können enthalten sein.

Auf der Sendeseite beginnen wir mit dem Senden eines gesamten Fensters voller Pakete Data[1] bis Data[W] und setzen LA=0. Wenn ACK[M] eintrifft, gilt LA<M≤LA+W und das Fenster verschiebt sich von [LA+1...LA+W] weiter zu [M+1...M+W], und wir dürfen nun Data[LA+W+1] bis Data[M+W] senden. Der einfachste Fall ist M=LA+1.

Bei Eintreffen von ACK[M]:

```
if M≤LA or M>LA+W, ignoriere das Paket
sonst:
   set K = LA+W+1, das erste Paket nach dem alten Fenstern
   set LA = M, direkt nach dem Ende es neuen Fensters
   for (i=K; i≤LA+W; i++) send Data[i]
```

Beachten Sie, dass neue ACKs eintreffen können, während wir uns in der Schleife auf der letzten Zeile befinden. Wir gehen hier davon aus, dass der Absender stillschweigend sendet, was er senden kann, und erst danach beginnt, weitere eintreffende ACKs zu verarbeiten. Einige Implementierungen wählen vielleicht einen asynchroneren Ansatz, bei dem ein Thread eintreffende ACKs verarbeitet und LA inkrementiert und ein anderer Thread alles sendet, was er senden darf.

Um auch Zeitüberschreitungen und erneute Übertragungen zu unterstützen, müsste jedes übertragene Paket zusammen mit dem Zeitpunkt seiner Übertragung gespeichert werden. In regelmäßigen Abständen muss diese Sammlung gespeicherter Pake-

te dann nach Paketen durchsucht werden, bei denen `send_time` + `timeout_interval` ≤ `current_time` ist; diese Pakete werden erneut übertragen. Wenn ein Paket Data[N] quittiert wird (vielleicht durch ein ACK[M] für M>N), kann es gelöscht werden.

8.4 Epilog

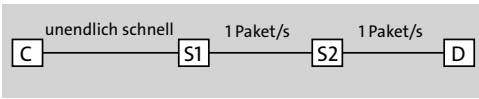
Dies schließt unsere Diskussion über den Sliding-Windows-Algorithmus im abstrakten Rahmen ab. Konkrete Implementierungen behandeln wir in Abschnitt 16.4.1, »TFTP und der Zauberlehrling«, (Stop-and-Wait) und in Abschnitt 18.7, »Sliding Windows bei TCP«; Letzteres zählt zu den wichtigsten Mechanismen im Internet.

8.5 Übungen

1. Skizzieren Sie ein Leiterdiagramm für Stop-and-Wait, wenn Data[3] beim ersten Senden verloren geht. Setzen Sie dabei voraus, dass es beim Absender keine Zeitüberschreitungen gibt (der Absender sendet jedoch bei Duplikaten erneut) und dass beim *Empfänger* eine Zeitüberschreitung von 2 Sekunden gilt. Führen Sie das Diagramm bis zu dem Punkt fort, an dem Data[4] erfolgreich übertragen wird. Nehmen Sie eine RTT von 1 Sekunde an.
2. Zeichnen Sie das Zauberlehrling-Diagramm aus Abschnitt 8.1.2, »Sorcerer's Apprentice Bug«, neu, wobei Sie davon ausgehen, dass der Absender jetzt bei Duplikaten *nicht* mehr erneut sendet, der Empfänger aber immer noch. ACK[3] wird, wie zuvor, so lange verzögert, bis der Absender Data[3] erneut sendet.
3. Angenommen, ein Stop-and-Wait-Empfänger wurde fehlerhaft implementiert. Wenn Data[1] eintrifft, werden ACK[1] und ACK[2] gesendet, getrennt durch ein kurzes Intervall; erst danach sendet der Empfänger beim Eintreffen von Data[N] ACK[N+1] statt des korrekten ACK[N].
   a) Zeichnen Sie ein Diagramm, das mindestens drei RTTs umfasst. Gehen Sie davon aus, dass keine Pakete verloren gehen.
   b) Wie hoch ist der durchschnittliche Durchsatz in Datenpaketen pro RTT? (Bei normalem Stop-and-Wait liegt der mittlere Durchsatz bei 1).
   c) Kann der Absender irgendetwas tun, um dieses Empfängerverhalten vor dem letzten Paket zu erkennen, wenn man davon ausgeht, dass keine Pakete verloren gehen und dass der Absender auf jedes ACK antworten muss, sobald es eintrifft?

   (Wenn ein Datenpaket verloren geht, kann es sein, dass der Empfänger es bereits quittiert hat, woraus sich ein Problem ergibt).

4. ◊ Betrachten Sie das alternative Modell von Abschnitt 8.3.1, »Einfache Analyse für feste Fenstergröße«:



   a) Berechnen Sie anhand der Formeln in Abschnitt 8.3.2, »RTT-Berechnungen«, die stationäre Auslastung der Warteschlange für eine Fenstergröße von 6.

   b) Setzen Sie wiederum eine Fenstergröße von 6 voraus und erstellen Sie eine Tabelle wie in Abschnitt 8.3.1, »Einfache Analyse für feste Fenstergröße«, bis T = 8 Sekunden.

5. Erstellen Sie eine Tabelle wie in Abschnitt 8.3.1, »Einfache Analyse für feste Fenstergröße«, für das ursprüngliche Netzwerk A–––R1–––R2–––R3–––R4–––B mit winsize = 8. Nehmen Sie wie in den Textbeispielen eine Bandbreitenverzögerung von 1 Paket/Sekunde für die Verbindungen R1⟶R2, R2⟶R3, R3⟶R4 und R4⟶B an. Die Verbindung A-R1 und alle gegenläufigen Verbindungen (von B nach A) sind unendlich schnell. Führen Sie die Tabelle für 10 Sekunden fort.

6. Erstellen Sie eine Tabelle wie in Abschnitt 8.3.1, »Einfache Analyse für feste Fenstergröße«, für ein Netzwerk A–––R1–––R2–––B. Die Verbindung A-R1 ist unendlich schnell; die Verbindungen R1-R2 und R2-B haben jeweils eine *Ausbreitungsverzögerung* von 1 Sekunde in jeder Richtung und eine *Bandbreitenverzögerung* von null (d. h. ein Paket braucht 1,0 Sekunden für die Strecke von R1 nach R2; zwei Pakete brauchen ebenfalls 1,0 Sekunden für die Strecke von R1 nach R2). Nehmen Sie winsize=6 an. Führen Sie die Tabelle für 8 Sekunden lang fort. Beachten Sie, dass bei einer Bandbreitenverzögerung von Null mehrere gemeinsam gesendete Pakete bis zum Zielort zusammenbleiben; die Ausbreitungsverzögerung verhält sich ganz anders als die Bandbreitenverzögerung!

7. Nehmen Sie RTT_{noLoad} = 4 Sekunden an, sowie eine Flaschenhalsbandbreite von 1 Paket alle 2 Sekunden.

   a) Welche Fenstergröße ist erforderlich, um gerade noch an der Überlastungsgrenze zu bleiben?

   b) Nehmen Sie winsize=6 an. Welcher Wert ergibt sich schließlich für RTT_{actual}?

   c) Wiederum sei winsize=6, wie viele Pakete befinden sich im Gleichgewichtszustand in der Warteschlange?

8. Erstellen Sie eine Tabelle wie in Abschnitt 8.3.1, »Einfache Analyse für feste Fenstergröße«, für ein Netzwerk A–––R1–––R2–––R3–––B. Die Verbindung A-R1 sei unendlich schnell. Die Verbindungen R1-R2 und R3-B haben eine Bandbreitenverzögerung von 1 Paket/Sekunde ohne zusätzliche Ausbreitungsverzögerung. Die Verbindung R2–R3 hat eine Bandbreitenverzögerung von 1 Paket / 2 Sekunden und

keine Ausbreitungsverzögerung. Die umgekehrte B⟶A-Richtung (für ACKs) ist unendlich schnell. Nehmen Sie winsize = 6 an.

a) Führen Sie die Tabelle für 10 Sekunden fort. Beachten Sie, dass Sie die Warteschlange sowohl für R1 als auch für R2 darstellen müssen.

b) Führen Sie die Tabelle zumindest teilweise bis T=18 fort, und zwar so detailliert, dass Sie überprüfen können, ob RTT_{actual} für Paket 8 dem in der vorherigen Übung berechneten Wert entspricht. Sie benötigen dazu mehr als 10 Pakete, aber weniger als 16; hier bietet sich der Einsatz der Hexadezimalbezeichnungen A, B, C für die Pakete 10, 11, 12 an.

Tipp: Die Spalte für »R2 sendet« (oder, genauer gesagt, »R2 ist gerade dabei zu senden«) sollte wie folgt aussehen:

T	R2 sendet
0	
1	1
2	1
3	2
4	2
5	3
6	3
...	...

9. Erstellen Sie eine Tabelle wie in Abschnitt 8.3.1, »Einfache Analyse für feste Fenstergröße«, für ein Netzwerk A–––R1–––R2–––B. Die Verbindung A–R1 sei unendlich schnell. Die R1-R2-Verbindung hat eine Bandbreitenverzögerung von 1 Paket / 2 Sekunden und die Verbindung R2-B hat eine Bandbreitenverzögerung von 1 Paket/Sekunde, jeweils ohne zusätzliche Ausbreitungsverzögerung. Die umgekehrte Richtung B⟶A (für ACKs) sei unendlich schnell. Nehmen Sie winsize = 4 an. Empfohlene Spalten sind Zeit, »A sendet«, »R1 puffert«, »R1 sendet«, »R2 sendet« und »B Ack«. Tipp: Die Spalte »R1 sendet« sieht aus wie die Spalte »R2 sendet« aus dem Tipp zum vorherigen Problem, außer dass sie bei T=0 und nicht bei T=1 beginnt.

   Beachten Sie, dass bandwidth × RTT_{noLoad} = 1/2 × 3 = 1.5 gilt, und daher gemäß Gleichung 4 aus Abschnitt 8.3.2, »RTT-Berechnungen«, die Warteschlangenauslastung 4 – 1.5 = 2.5 beträgt.

   a) Führen Sie die Tabelle für 8 Sekunden fort.

   b) Hat R1 im statischen Zustand 2,5 Pakete in der Warteschlange? Wenn ja, was ist unter einem halben Paket zu verstehen?

10. Zeigen Sie, dass, wenn A unter Verwendung von Sliding Windows an B sendet und auf dem Weg von A nach B die erste Übertragungsstrecke aus Sicht von A *nicht* die langsamste ist, A schließlich das gesamte Fenster zum Versand ausstehend haben wird (außer direkt in dem Moment, nachdem ein neues ACK eingetroffen ist).
11. ◊ Angenommen, RTT_{noLoad} ist 100 ms und die verfügbare Bandbreite beträgt 1.000 Pakete/s. Es wird mit Sliding Windows gearbeitet.
    a) Wie hoch ist die Übertragungskapazität der Verbindung?
    b) Wenn RTT_{actual} auf 130 ms ansteigt (durch Verwendung einer größeren winsize), wie viele Pakete befinden sich dann gleichzeitig in einer Warteschlange?
    c) Was ist der Wert von RTT_{actual}, wenn winsize um 50 ansteigt?
12. RTT_{noLoad} sei 50 ms und die verfügbare Bandbreite 2000 Pakete/s (2 Pakete/ms). Es wird mit Sliding Windows gearbeitet.
    a) Wie groß muss das Fenster sein, um gerade noch an der Überlastungsgrenze zu bleiben?
    b) Wenn RTT_{actual} auf 60 ms ansteigt (aufgrund einer höheren winsize), wie viele Pakete befinden sich dann gleichzeitig in einer Warteschlange?
    c) Welcher Wert von winsize würde zu einer RTT_{actual} = 60 ms führen?
    d) Welcher Wert von winsize würde RTT_{actual} auf 100 ms ansteigen lassen?
13. Angenommen, es wird Stop-and-Wait eingesetzt (winsize=1), und Pakete könnten zwar verloren gehen, werden *aber niemals in ihrer Reihenfolge verändert* (d. h. wenn zwei Pakete P1 und P2 in dieser Reihenfolge gesendet werden und beide ankommen, dann kommen sie auch in dieser Reihenfolge an). Zeigen Sie, dass an dem Punkt, an dem der Empfänger den Eingang von Data[N] erwartet, nur die beiden Pakete Data[N] und Data[N–1] möglich sind. (Eine Folge davon ist, dass für Stop-and-Wait ohne Umsortierung eine aus einem Bit bestehende Nummerierung der Pakete ausreicht). Tipp: Wenn der Empfänger auf Data[N] wartet, muss er bereits Data[N–1] empfangen und ACK[N–1] gesendet haben. Außerdem wird der Absender, nachdem er Data[N] gesendet hat, niemals ein Data[K] mit K<N senden.
14. ◊ Setzen Sie winsize=4 in einer Sliding-Windows-Verbindung voraus und nehmen Sie zudem an, dass Pakete zwar verloren gehen können, *aber niemals in ihrer Reihenfolge verändert werden* (d. h. wenn zwei Pakete P1 und P2 in dieser Reihenfolge gesendet werden und beide ankommen, dann kommen sie auch in dieser Reihenfolge an). Zeigen Sie, dass, wenn sich Data[8] im Fenster des Empfängers befindet (es wurde also alles bis hin zu Data[4] empfangen und bestätigt), es nicht mehr möglich ist, dass selbst ein verspätetes Data[0] beim Empfänger eintrifft. (Eine Folge dieses allgemeinen Prinzips ist, dass wir – *ohne Reihenfolgenänderung* – die laufende Nummer des Pakets durch (sequence_number) mod (2×winsize+1) ersetzen können, ohne dass es zu Unklarheiten kommt).

15. Setzen Sie winsize=4 in einer Sliding-Windows-Verbindung voraus, und nehmen Sie wie in der vorherigen Übung an, dass Pakete zwar verloren gehen können, aber nie in ihrer Reihenfolge verändert werden. Nennen Sie ein Beispiel, in dem Data[8] im Fenster des Empfängers liegt (der Empfänger hat also vermutlich ACK [4] gesendet) und dennoch Data[1] ordnungsgemäß eintrifft. (Die in der vorangegangenen Übung ermittelte Grenze für verspätete Pakete stellt also das Optimum dar.)
16. Nehmen Sie ein Netzwerk A–––R1–––R2–––B an, wobei die Verbindung A-R1 unendlich schnell ist und die Verbindung R1-R2 eine Bandbreite von 1 Paket/Sekunde in beide Richtungen hat, was eine RTT_{noLoad} von 2 Sekunden ergibt. Nehmen Sie zudem an, dass A zunächst mit winsize = 6 zu senden beginnt. Nach der Analyse in Unterabschnitt »3. Fall: winsize = 6« des Abschnitt 8.3.1, »Einfache Analyse für feste Fenstergröße«, sollte die RTT auf winsize/bandwidth = 6 Sekunden steigen. Geben Sie die RTTs der ersten acht Pakete an. Wie lange dauert es, bis die RTT auf 6 Sekunden ansteigt?
17. In dieser Übung betrachten wir die Beziehung zwischen Flaschenhalsbandbreite und winsize/RTT_{actual} wenn sich erstere plötzlich ändert. Angenommen, das Netzwerk sieht wie folgt aus

    A–––R1–––R2–––R3–––B

    Die Verbindung A––R1 sei unendlich schnell. Die Verbindung R1→R2 hat eine Bandbreitenverzögerung von 1 Paket/Sekunde in Richtung R1→R2. Die übrigen Verbindungen R2→R3 und R3→B haben eine Bandbreitenverzögerung von 1 s in der angegebenen Richtung. Kleine ACK-Pakete werden unverzögert von B zurück nach A übertragen.

    A sendet an B mit winsize = 3. Die drei Pakete P0, P1 und P2 werden jeweils zu den Zeitpunkten T=0, T=1 und T=2 gesendet.

    Bei T=3 kommt P0 bei B an. Zu diesem Zeitpunkt halbiert sich die Bandbreite R1→R2 plötzlich auf 1 Paket / 2 Sekunden; P3 wird bei T=3 gesendet und kommt bei R2 bei T=5 an. Es kommt zum Zeitpunkt T=7 bei B an.

    a) Vervollständigen Sie die folgende Tabelle mit den Paketankunftszeiten

T	A sendet	R1 Warteschlange	R1 sendet	R2 sendet	R3 sendet	B empfängt/ ACK
2	P2		P2	P1	P0	
3	P3		P3	P2	P1	P0
4	P4	P4	P3 weiter		P2	P1
5	P5	P5	P4	P3		P2

T	A sendet	R1 Warte-schlange	R1 sendet	R2 sendet	R3 sendet	B empfängt/ ACK
6		P5	P4 weiter		P3	
7	P6					P3
8						
9	P7					
10						
11	P8					

b) Berechnen Sie für jedes der Pakete P2, P3, P4 und P5 während seines Umlaufs den Durchsatz, der sich aus winsize/RTT ergibt. Ermitteln Sie die RTT jedes Pakets aus der obigen Tabelle.

c) Wie viel Zeit verbringt jedes Paket auf der Netzwerkstrecke, sobald sich ein stabiler Zustand mit $RTT_{actual} = 6$ eingestellt hat? Wie viel Zeit verbringt jedes Paket in der Warteschlange von R1?

Kapitel 9
IP Version 4

Unterhalb der IP-Schicht gibt es mehrere LAN-Protokolle und darüber mehrere Transportprotokolle. IP selbst steht jedoch für sich alleine. Das Internet ist das IP-Internet. Wenn Sie irgendwo ein eigenes LAN-Protokoll oder ein eigenes Transportprotokoll verwenden wollen, funktioniert das Internet-Backbone trotzdem einwandfrei für Sie. Aber sobald Sie die IP-Schicht wechseln wollen, stoßen Sie auf Schwierigkeiten. (Sprechen Sie einfach mit den IPv6-Leuten oder den IP-Multicasting- oder IP-Reservierungsgruppen).

In diesem Kapitel geht es um das ursprüngliche IP-Kernprotokoll – bekannt als Version 4 oder IPv4 mit einer Adressgröße von 32 Bit. Der größte Teil des Internets verwendet heute noch IPv4, obwohl IPv6 auf dem Vormarsch ist. Wir werden sehen, dass die IP-Schicht effizientes, skalierbares Routing ermöglicht. Im folgenden Kapitel besprechen wir einige Begleitprotokolle: ICMP, ARP, DHCP und DNS.

Trotz seiner Allgegenwärtigkeit steht IPv4 vor einer ungewissen Zukunft: Dem Internet gehen die neuen großen IPv4-Adressblöcke aus (Abschnitt 1.10, »IP – Internet Protocol«). Daher wächst der Druck, auf IPv6 mit seiner 128-Bit-Adressengröße umzustellen. Der Fortschritt ist jedoch langsam, und Verzögerungstaktiken wie IPv4-Adressmärkte und NAT (Abschnitt 9.7, »Netzwerkadressübersetzung«) – durch die sich mehrere Hosts eine einzige öffentliche IPv4-Adresse teilen können – haben den Fortbestand von IPv4 ermöglicht. Abgesehen von der grundlegenden Änderung der Adressstruktur gibt es relativ wenige Unterschiede zwischen den Routing-Modellen von IPv4 und IPv6. Wir werden uns in diesem und den folgenden Kapiteln mit IPv4 und in Kapitel 11, »IPv6«, mit IPv6 befassen; wenn der Unterschied zwischen IPv4 und IPv6 keine große Rolle spielt, schreibe ich einfach »IP«.

IPv4 (und IPv6) ist in der Tat ein universelles *Routing- und Adressierungsprotokoll*. Routing und Adressierung werden gemeinsam entwickelt; jeder Knoten hat eine IP-Adresse, und jeder Router weiß, wie er mit IP-Adressen umgehen muss. Ursprünglich wurde IP als Möglichkeit betrachtet, mehrere LANs miteinander zu *verbinden*, doch ist es mittlerweile vielleicht sinnvoller, IP als virtuelles LAN zu betrachten, das alle physischen LANs überlagert.

Ein entscheidender Aspekt von IP ist seine *Skalierbarkeit*. Im Jahr 2019 hatte das Internet über 10^9 Hosts; diese Schätzung ist wahrscheinlich niedrig. Gleichzeitig lag die Größe der größten Weiterleitungstabellen jedoch noch unter 10^6 (Abschnitt 15.5, »BGP-Tabellengröße«). Ethernet skaliert hingegen schlecht, da die Weiterleitungstabellen für jeden aktiven Host einen Eintrag benötigen.

Kapitel 14
IP-Routing im großen Maßstab

Im vorigen Kapitel haben wir zwei Klassen von Aktualisierungsalgorithmen für das Routing betrachtet: Distance-Vector und Link-State. Beide Ansätze setzen voraus, dass sich die teilnehmenden Router zunächst auf ein gemeinsames Protokoll und dann auf einen gemeinsamen Nenner zur Zuweisung der Verbindungskosten geeinigt haben. Wir werden dies im folgenden Kapitel über BGP, Kapitel 15, »Border Gateway Protocol (BGP)«, behandeln; ein grundlegendes Problem besteht darin, dass, wenn ein Standort den Hop-Count-Ansatz bevorzugt, bei dem jeder Verbindung Kosten von 1 zugewiesen werden, während ein anderer Standort die Verbindungskosten lieber proportional zu ihrer Bandbreite zuweist, *sinnvolle* Vergleiche der Pfadkosten zwischen den beiden Standorten einfach nicht möglich sind.

Bevor wir uns jedoch mit BGP befassen, müssen wir den Grundgedanken der IP-Weiterleitung nochmals überdenken. Von Anfang an hatten wir die Vorstellung, dass eine IP-Adresse in einen Netzwerk- und einen Host-Anteil aufgesplittet werden kann; die Aufgabe der meisten Router bestand darin, nur das Netzwerkpräfixe zu untersuchen und die next_hop-Weiterleitungsentscheidung auf dieser Grundlage zu treffen. Die Netz-/Host-Aufteilung für IPv4-Adressen basierte ursprünglich auf dem Class A/B/C-Mechanismus; die meisten IPv6-Adressen waren in jeweils 64 Bit für Netz und Host aufgesplittet. In Abschnitt 9.5, »Der klassenlose IP-Delivery-Algorithmus«, haben wir gesehen, wie man auf IPv4-Netzpräfixe beliebiger Länge routen kann; die gleiche Idee funktioniert auch für IPv6.

In diesem Kapitel weiten wir das aus, indem wir verschiedenen Routern an unterschiedlichen Positionen im Routing-Universum gestatten, ihre Weiterleitungsentscheidungen auf der Grundlage unterschiedlicher Netzwerkpräfixelängen zu treffen, und zwar für dieselbe Zieladresse. Das heißt, ein Backbone-Router könnte eine Weiterleitung an eine bestimmte IPv4-Adresse D nur auf Grundlage der ersten 12 Bits von D vornehmen; ein regionaler Router könnte seine Entscheidung auf die ersten 18 Bits stützen, und ein Standort-Router könnte eine Weiterleitung an das endgültige Subnetz auf Grundlage der ersten 24 Bits vornehmen. Auf diese Weise lassen sich *Routing-Hierarchien* mit mehreren Ebenen erstellen, was die Skalierbarkeit des Routings erheblich verbessern und die Größe der Weiterleitungstabellen verringern kann. Die eigentliche Weiterleitung durch einen Router erfolgt wie in 64, »Der klassenlose IP-Delivery-Algorithmus «.

Der Begriff *Routing-Domain* bezeichnet eine Gruppe von Routern unter gemeinsamer Verwaltung, die eine gemeinsame Zuweisung von Verbindungskosten verwenden; ein anderer Begriff dafür ist *Autonomes System*. Im vorigen Kapitel fand das ge-

samte Routing innerhalb einer Routing-Domain statt; jetzt werden wir uns das Internet als Ganzes vorstellen, das aus einem Flickenteppich unabhängiger Routing-Domains besteht. Auch wenn die Verwendung eines gemeinsamen Routing-Update-Protokolls innerhalb der Routing-Domäne keine absolute Voraussetzung ist – einige Subnetze können beispielsweise intern den Distance-Vector-Algorithmus verwenden, während die »Backbone«-Router des Standorts auf Link-State setzen – können wir davon ausgehen, dass alle Router eine einheitliche Sicht auf die Topologie des Standorts und auf die Kostenmetriken haben.

Beim IP-Routing »im großen Maßstab« geht es im Wesentlichen um die Koordinierung des Routings zwischen mehreren unabhängigen Routing-Domains. Schon in den Anfangszeiten des Internets gab es mehrere Routing-Domains, wenn auch nur deswegen, weil die Messung der Verbindungskosten noch zu unklar war (und es immer noch ist), um in Stein gemeißelt zu werden. Eine weitere Komponente des Routings im großen Maßstab ist jedoch die Unterstützung des hierarchischen Routings über die Ebene der Subnetze hinaus; darauf gehen wir als nächstes ein.

14.1 Classless Internet Domain Routing: CIDR

CIDR ist der Mechanismus zur Unterstützung von hierarchischem Routing im Internet-Backbone. Durch Subnetting wird die Trennlinie zwischen Netz und Host weiter nach rechts verschoben; mit CIDR lässt sie sich auch nach links verschieben. Beim Subnetting ist die geänderte Trennlinie nur innerhalb der Organisation sichtbar, welcher der IP-Netzadressenblock gehört; außerhalb ist das Subnetting nicht erkennbar. CIDR ermöglicht die Aggregation von IP-Adressblöcken in einer Weise, die für den Internet-Backbone sichtbar ist.

Bei der Einführung von CIDR für IPv4 im Jahr 1993 wurden die folgenden Begründungen angeführt, die sich alle auf die zunehmende Größe der Backbone-IP-Weiterleitungstabellen bezogen und nach Maßstäben des damaligen Class A/B/C-Mechanismus ausgedrückt wurden:

- Dem Internet gehen die Class-B-Adressen aus (dies passierte Mitte der 1990er Jahre)
- Es gibt zu viele Class-C-Adressen (sie sind am zahlreichsten), als dass Backbone-Weiterleitungstabellen effizient arbeiten könnten
- Irgendwann werden der IANA (Internet Assigned Numbers Authority) die IP-Adressen ausgehen (dies erfolgte 2011)

Ohne CIDR hätte die Zuweisung mehrerer Klasse-C-Adressen anstelle einer einzigen Klasse-B-Adresse zwar dem ersten Punkt in der obigen Liste Abhilfe verschafft, dabei aber auch den zweiten Punkt verschlechtert.

Ironischerweise wird der sehr angespannte Markt für IPv4-Adressblöcke wahrscheinlich zu immer größeren IPv4-Weiterleitungstabellen der Backbones führen, da die Standorte gezwungen sind, mehrere kleine Adressblöcke anstelle eines großen Blocks zu verwenden.

Im Jahr 2000 hatte CIDR den Class A/B/C-Mechanismus im Backbone-Internet weitgehend abgelöst und die Funktionsweise des Backbone-Routings mehr oder weniger vollständig verändert. Man kaufte einen Adressblock von einem Provider oder einem anderen IP-Adressenzuteiler, und dieser konnte jede gewünschte Größe von /32 bis /15 haben.

CIDR ermöglichte das IP-Routing auf Basis eines beliebig langen Adresspräfixes; der Class A/B/C-Mechanismus verwendet natürlich feste Präfixlängen von 8, 16 und 24 Bit. Darüber hinaus ermöglicht CIDR verschiedenen Routern auf verschiedenen Ebenen des Backbones die Weiterleitung aufgrund von Präfixen unterschiedlicher Länge. Wenn der Organisation P beispielsweise ein /10-Block zugewiesen wurde, könnte P eine *Unterteilung* in /20-Blöcke vornehmen. Auf der obersten Ebene würde das Routing zu P wahrscheinlich auf den ersten 10 Bits basieren, während das Routing innerhalb von P auf den ersten 20 Bits basieren würde.

Bei IPv6 gab es nie Adressklassen, und so wurde CIDR wohl von Anfang an nativ unterstützt. Das Routing zur Adresse 2400:1234:5678:abcd:: könnte auf Grundlage des /32-Präfixes 2400:1234:: oder des /48-Präfixes 2400:1234:5678:: oder des /56-Präfixes 2400:1234:5678:ab:: oder eines Präfixes jeder anderen Länge erfolgen.

CIDR wurde mit RFC 1518 und RFC 1519 offiziell für IPv4 eingeführt. Eine Zeit lang gab es Strategien, um eine Kompatibilität mit nicht CIDR-fähigen Routern zu gewährleisten; diese sind mittlerweile überholt. Insbesondere ist es für große IPv4-Router nicht mehr sinnvoll, bei fehlenden CIDR-Informationen auf den Class A/B/C-Mechanismus zurückzugreifen; wenn diese fehlen, sollte das Routing fehlschlagen.

Die Grundstrategie von CIDR lässt sich als Mechanismus zur Konsolidierung mehrerer Netzwerkblöcke mit demselben Ziel in einem einzigen Eintrag verstehen. Nehmen wir an, ein Router hat vier IPv4-Class-C-Adressen, die alle zum selben Ziel führen:

- 200.7.0.0/24 ⟶ foo
- 200.7.1.0/24 ⟶ foo
- 200.7.2.0/24 ⟶ foo
- 200.7.3.0/24 ⟶ foo

Der Router kann alle diese durch einen einzigen Eintrag ersetzen

- 200.7.0.0/**22** ⟶ foo

Dabei spielt es keine Rolle, ob foo ein einziges Endziel darstellt oder ob es vier Standorte repräsentiert, die zufällig zum selben next_hop geleitet werden.

Es lohnt sich, einen genauen Blick auf die Arithmetik zu werfen, um zu sehen, warum der Einzeleintrag /22 verwendet. Das bedeutet, dass die ersten 22 Bits mit 200.7.0.0 übereinstimmen müssen; dies sind alle ersten und zweiten Bytes und die ersten sechs Bits des dritten Bytes. Sehen wir uns das dritte Byte der obigen Netzwerkadressen in Binärschreibweise an:

- 200.7.000000 00.0/24 ⟶ foo
- 200.7.000000 01.0/24 ⟶ foo
- 200.7.000000 10.0/24 ⟶ foo
- 200.7.000000 11.0/24 ⟶ foo

Das /24 bedeutet, dass die Netzwerkadressen am Ende des dritten Bytes aufhören. Die vier obigen Einträge decken jede mögliche Kombination der letzten beiden Bits des dritten Bytes ab; damit eine Adresse mit einem der obigen Einträge übereinstimmt, reicht es aus, wenn sie mit 200.7 beginnt und die ersten *sechs Bits* des dritten Bytes 0-Bits sind. Dies ist eine andere Art, auszudrücken, dass die Adresse 200.7.0.0/22 entsprechen muss.

Die meisten Implementierungen verwenden eine Bitmaske, z. B. 255.255.252.0, und nicht die Zahl 22. Beachten Sie, dass 252 im Binärformat 1111 1100 ist, mit 6 führenden Einserbits, sodass 255.255.252.0 8+8+6=22 Einserbits gefolgt von 10 Nullerbits hat.

Der IP-Delivery-Algorithmus aus Abschnitt 9.5, »Der klassenlose IP-Delivery-Algorithmus«, funktioniert auch mit CIDR nach wie vor, allerdings mit der Einschränkung, dass die Weiterleitungstabelle des Routers nun eine Netzwerkpräfixe-Länge haben kann, die mit jedem Eintrag verbunden ist. Bei einem Ziel D suchen wir in der Weiterleitungstabelle nach Netzwerkpräfixe-Zielen B/k, bis wir eine Übereinstimmung finden, d. h. Gleichheit der ersten k Bits. Im Hinblick auf Masken wird bei einem Ziel D und einer Liste von Tabelleneinträgen ⟨Präfix,Maske⟩ = ⟨B[i],M[i]⟩ nach einem i gesucht, für das (D & M[i]) = B[i]) gilt.

Aber was ist mit möglichen mehrfachen Übereinstimmungen? Bei Subnetzen lag es in der Verantwortung des Subnetzbetreibers, diese zu vermeiden, aber bei CIDR ist diese Verantwortung viel zu breit gestreut, um durch ein IETF-Mandat für illegal erklärt zu werden. Stattdessen wurde mit CIDR die *Longest-Match-Regel* eingeführt: wenn das Ziel D sowohl mit B_1/k_1 als auch mit B_2/k_2 übereinstimmt, wobei $k_1 < k_2$ ist, dann ist die längere Übereinstimmung B_2/k_2 zu verwenden. (Hinweis: Wenn D mit zwei verschiedenen Einträgen B_1/k_1 und B_2/k_2 übereinstimmt, ist entweder $k_1 < k_2$ oder $k_2 < k_1$).

14.2 Hierarchisches Routing

Streng genommen ist CIDR einfach ein Mechanismus für das Routing zu IP-Adressblöcken beliebiger Präfixlänge, d. h. zum Setzen des Netz-/Host-Teilungspunkts an eine beliebige Stelle innerhalb der 32-Bit-IP-Adresse.

Da dieser Netz-/Host-Unterteilungspunkt jedoch *variabel* ist, unterstützt CIDR das Routing auf Grundlage *unterschiedlicher* Präfixlängen an verschiedenen Stellen der Backbone-Routing-Infrastruktur. So können beispielsweise Top-Level-Router auf Basis von /8- oder /9-Präfixen routen, während zwischengeschaltete Router auf Basis von Präfixen der Länge 14 routen können. Diese Funktion des Routing auf Grundlage einer geringeren Bit-Anzahl an einem Punkt im Internet und mehr Bits an einer anderen Stelle ist genau das, was mit *hierarchischem Routing* gemeint ist.

Wir haben das hierarchische Routing bereits im Zusammenhang mit Subnetzen betrachtet: Der Traffic könnte zunächst zu einem Class-B-Standort 147.126.0.0/16 geleitet werden, und dann innerhalb dieses Standorts zu Subnetzen wie 147.126.1.0/24, 147.126.2.0/24 usw. Mit CIDR ist die Hierarchie jedoch viel flexibler: Die oberste Hierarchieebene kann viel größer sein als die »Kunden«-Ebene, niedrigere Ebenen müssen nicht von den höheren Ebenen administrativ überwacht werden (wie es bei Subnetzen der Fall ist), und kann auch mehr als zwei Ebenen geben.

CIDR ist ein *Mechanismus* zur Zuteilung von Adressblöcken; es sagt nicht direkt etwas über die *Richtlinien* aus, die wir damit umsetzen wollen. Hier sind vier mögliche Anwendungen; die letzten beiden verwenden hierarchisches Routing:

- Anwendung 1 (veraltet): CIDR ermöglicht die Zuweisung mehrerer Klasse-C-Blöcke oder von Fragmenten einer Klasse-A-Adresse an einen einzigen Kunden, sodass für diesen Kunden nur ein Eintrag in der Weiterleitungstabelle erforderlich ist.
- Anwendung 2 (veraltet): CIDR ermöglicht die opportunistische *Aggregation* von Routen: Ein Router, der die vier oben genannten 200.7.x.0/24-Routen in seiner Tabelle sieht, kann sie zu einem einzigen Eintrag zusammenfassen.
- Anwendung 3 (aktuell): CIDR ermöglicht große Providerblöcke mit Unterverteilung durch den Provider. Dies wird als *providerbasiertes* Routing bezeichnet.
- Anwendung 4 (hypothetisch): CIDR ermöglicht große regionale Blöcke mit Unterverteilung innerhalb der Region, ähnlich wie beim ursprünglichen Schema für US-Telefonnummern mit Ortsvorwahlen. Dies wird als *geografisches* Routing bezeichnet.

Jedes dieser Verfahren hat das Potenzial, die Größe der Backbone-Weiterleitungstabellen erheblich zu verringern, was hier wohl das wichtigste Ziel ist. In jedem Fall wird CIDR verwendet, um die Erstellung von Adressblöcken beliebiger Größe zu unterstützen und dann das Routing zu diesen Blöcken als eine einzige Einheit durchzuführen. Der Internet-Backbone wäre zum Beispiel viel besser bedient, wenn alle Rou-

ter nur einen einzigen Eintrag ⟨200.0.0.0/8, R1⟩ pflegen müssten, anstatt 256 Einträge ⟨200.x.0.0/16, R1⟩ für jeden Wert von x. (Wie wir weiter unten sehen werden, ist dies auch dann noch nützlich, wenn einige der x einen anderen next_hop haben). Zu den Sekundärzielen von CIDR gehört es, Ordnung in die Zuweisung von IP-Adressen zu bringen und (für die letzten beiden Punkte in der obigen Liste) eine Routing-Hierarchie zu ermöglichen, die den tatsächlichen Fluss des größten Teils des Datenverkehrs widerspiegelt.

Das hierarchische Routing bringt eine neue Schwierigkeit mit sich: Die gewählten Routen sind aus globaler Sicht möglicherweise nicht mehr optimal, zumindest wenn wir die Algorithmen zur Routing-Aktualisierung auch hierarchisch anwenden. Angenommen, auf der obersten Ebene basiert die Weiterleitung auf den ersten acht Bits der Adresse, und der gesamte Traffic für 200.0.0.0/8 wird an den Router R1 geleitet. Auf der zweiten Ebene leitet R1 dann den Datenverkehr (hierarchisch) über R2 an 200.20.0.0/16 weiter. Ein Paket, das von einem unabhängigen Router R3 an 200.20.1.2 gesendet wird, könnte daher über R1 geleitet werden, *selbst wenn es einen kostengünstigeren Pfad* R3→R4→R2 *gäbe, der R1 umgeht*. Der Top-Level-Weiterleitungseintrag ⟨200.0.0.0/8,R1⟩ könnte mit anderen Worten eine Vereinfachung der realen Situation darstellen. Das Verbot von »Backdoor«-Routen wie R3→R4→R2 ist unpraktisch (und wäre auch nicht gerade hilfreich); Kunden sind unabhängige Entitäten.

Dieses Problem des suboptimalen Routings kann nicht auftreten, wenn sich alle Router auf einen der Mechanismen für den kürzesten Weg aus Kapitel 13, »Routing-Update-Algorithmen«, verständigen; in diesem Fall würde R3 von dem kostengünstigeren Pfad R3→R4→R2 erfahren. Aber dann würden die potenziellen hierarchischen Vorteile durch die Verkleinerung der Weiterleitungstabellen verloren gehen. Schlimmer noch, eine vollständige globale Einigung aller Router auf ein gemeinsames Aktualisierungsprotokoll ist schlichtweg unpraktikabel; tatsächlich besteht eines der Ziele des hierarchischen Routings in der Bereitstellung einer praktikablen Alternative. Wir werden darauf weiter unten in Abschnitt 14.4.3, »Hierarchisches Routing über Provider«, zurückkommen.

14.3 Routing in früherer Zeit

Früher, zu Zeiten des NSFNet, war der Internet-Backbone noch eine einzige Routing-Domain. Die meisten Kunden waren zwar nicht direkt mit dem Backbone verbunden, aber die zwischengeschalteten Anbieter waren aus geografischer Sicht relativ kompakt, also *regional*, und hatten oft einen einzigen primären Routing-Austauschpunkt mit dem Backbone. Die IP-Adressen wurden den Teilnehmern direkt von der IANA zugewiesen, und die Weiterleitungstabellen des Backbones enthielten Einträge für jeden Standort, sogar für jene der Klasse C.

Da das NSFNet-Backbone und die regionalen Anbieter nicht notwendigerweise Kosteninformationen zu ihren einzelnen Links austauschten, waren die Routen selbst zu diesem frühen Zeitpunkt nicht unbedingt global optimal; es wurden Kompromisse und Näherungslösungen gefunden. Im NSFNet-Modell fanden die Router jedoch meist eine vernünftige Näherung für den kürzesten Weg zu jedem Standort, auf den die Backbone-Tabellen verwiesen. Die alte Routing-Domain des Backbones war zwar nicht allumfassend, aber wenn es Unterschiede zwischen zwei Routen gab, waren zumindest die jeweiligen Backbone-Abschnitte – die längsten Komponenten – identisch.

14.4 Providerbasiertes Routing

Beim providerbasierten Routing werden große CIDR-Blöcke an große Provider vergeben. Die verschiedenen Provider wissen jeweils, wie sie zueinander routen können. Die Teilnehmer erhalten ihre IP-Adressen (in der Regel) aus den Blöcken ihrer Provider; daher wird der Datenverkehr von außen zunächst an den Provider und dann *innerhalb* der Routing-Domain des Providers an den Teilnehmer weitergeleitet. Eventuell gibt es sogar eine Hierarchie von Providern, sodass die Pakete zunächst zum großen Provider und schließlich zum lokalen Provider geleitet werden. Es gibt unter Umständen keinen zentralen Backbone mehr; stattdessen können mehrere Anbieter jeweils parallele transkontinentale Netze aufbauen.

Hier ist ein einfacheres Beispiel, bei dem die Anbieter eindeutige Pfade zueinander haben. Nehmen wir an, wir haben die Anbieter P0, P1 und P2 mit folgenden Kunden:

- P0: Kunden A,B,C
- P1: Kunden D,E
- P2: Kunden F,G

Wir gehen außerdem davon aus, dass jeder Anbieter über einen IP-Adressblock wie folgt verfügt:

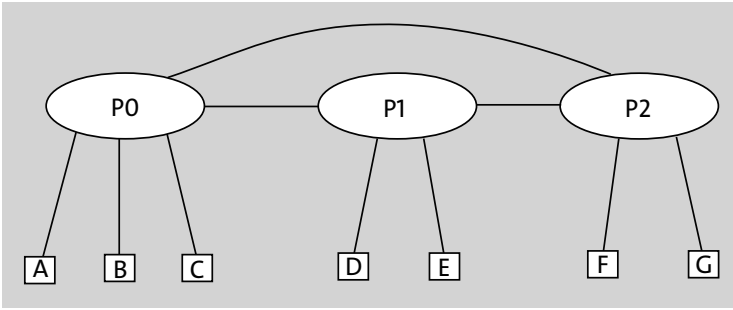
- P0: 200.0.0.0/8
- P1: 201.0.0.0/8
- P2: 202.0.0.0/8

Nun wollen wir den Kunden Adressen zuweisen:

- A: 200.0.0.0/16
- B: 200.1.0.0/16
- C: 200.2.16.0/20 (16 = 0001 0000)
- D: 201.0.0.0/16

- E: 201.1.0.0/16
- F: 202.0.0.0/16
- G: 202.1.0.0/16

Das Routing-Modell sieht vor, dass die Pakete zunächst an den entsprechenden Anbieter und dann an den Kunden weitergeleitet werden. Dieses Modell garantiert zwar im Allgemeinen nicht den kürzesten Ende-zu-Ende-Pfad, in diesem Fall aber schon, da jeder Anbieter nur einen einzigen Verbindungspunkt zu den anderen hat. Hier ist das Netzdiagramm:



Mit diesem Diagramm sieht die Weiterleitungstabelle von P0 etwa so aus:

P0	
destination	**next_hop**
200.0.0.0/16	A
200.1.0.0/16	B
200.2.16.0/20	C
201.0.0.0/8	P1
202.0.0.0/8	P2

Das heißt, die Tabelle von P0 besteht aus

- einem Eintrag für jeden der eigenen Kunden von P0
- einem Eintrag für jeden anderen Anbieter

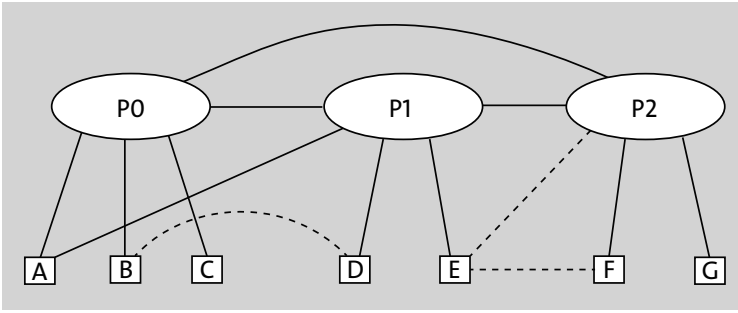
Hätten wir 1.000.000 Kunden, die sich gleichmäßig auf 100 Anbieter verteilen, dann würde die Tabelle jedes Anbieters nur 10.099 Einträge enthalten: 10.000 für seine eigenen Kunden und 99 für die anderen Anbieter. Ohne CIDR hätte die Weiterleitungstabelle eines jeden Anbieters 1.000.000 Einträge.

CIDR ermöglicht hierarchisches Routing, da die Routing-Entscheidung in verschiedenen Kontexten auf der Grundlage unterschiedlicher Präfixlängen getroffen werden kann. Wenn zum Beispiel ein Paket von D nach A gesendet wird, betrachtet P1 die ersten 8 Bits, während P0 die ersten 16 Bits betrachtet. Innerhalb des Kunden A könnte das Routing auf Grundlage der ersten 24 Bits erfolgen.

Selbst wenn wir einige zusätzliche »sekundäre« Verbindungen haben, d. h. zusätzliche Verbindungen, die keine alternativen Pfade zwischen den Anbietern schaffen, bleibt das Routing *relativ* einfach. Hier sind die privaten Kunde-zu-Kunde-Verbindungen C-D und E-F abgebildet; diese werden wahrscheinlich nur von den Kunden genutzt, die sie miteinander verbinden.

Zwei Kunden, A und E haben mehrere Verbindungen zu unterschiedlichen Anbietern: A-P1 und E-P2. Für solche Konstellationen wird häufig der Begriff *Multihoming* verwendet: Hosts mit mehreren Netzwerkschnittstellen in verschiedenen LANs, wozu auch sämtliche Router gehören. Hier meinen wir konkreter, dass es mehrere Netzwerkschnittstellen gibt, die mit verschiedenen Anbietern verbunden sind).

Typischerweise können A und E zwar ihre alternativen Provider-Verbindungen für den *ausgehenden* Traffic nutzen, ihr *eingehender* Traffic läuft jedoch weiterhin über ihren primären Provider P0 bzw. P1.



14.4.1 Internet Exchange Points

Die langen Verbindungen, die die Anbieter in diesen Diagrammen miteinander verbinden, sind etwas irreführend; nicht immer schätzen es Provider, lange gemeinsame Verbindungsstrecken zueinander zu teilen, bei deren Ausfall sich dann die Frage nach der Verantwortrung stellt. Stattdessen verbinden sich die Anbieter oft über *Internet eXchange Points* (IXPs) miteinander; die Verbindung P0––––––P1 könnte also in Wirklichkeit P0–––IXP–––P1 sein, wobei P0 die linke Verbindungsstrecke besitzt und P1 die rechte. IXPs können entweder von Dritten betriebene Standorte sein, die allen Anbietern offen stehen, oder aber es handelt sich dabei um private Austauschpunkte. Der Begriff »Metropolitan Area Exchange« (MAE) taucht in den Namen der IXPs MAE-East, ursprünglich in der Nähe von Washington DC, und MAE-West, ur-

sprünglich in San Jose, Kalifornien, auf; bei beiden handelt es sich jetzt eigentlich um mehrere von IXPs. MAE ist in diesem Zusammenhang inzwischen eine eigene Marke.

14.4.2 CIDR (und wie man nicht ins Gefängnis kommt)

Angenommen, wir wollen den Anbieter wechseln. Eine Möglichkeit dazu besteht darin, einen neuen IP-Adressblock des neuen Anbieters zu akzeptieren und alle unsere IP-Adressen zu ändern. Das Paper *Renumbering: Threat or Menace* [LKCT96] wurde – zumindest in den Anfangstagen von CIDR – häufig als Hinweis darauf zitiert, dass eine solche Neuadressierung zwangsläufig schlecht sei. Deshalb möchten wir prinzipiell zumindest die Möglichkeit einräumen, unsere IP-Adressenzuweisung bei einem Anbieterwechsel *beizubehalten*.

Ein Standard für die Adresszuweisung, der den Wechsel des Anbieters nicht zulässt, könnte sogar einen Verstoß gegen das US-Kartellgesetz darstellen; siehe *American Society of Mechanical Engineers v Hydrolevel Corporation*, 456 US 556 (1982). Die IETF hatte also bei der Ausarbeitung des CIDR-Standards mit Ermöglichung der Portabilität zwischen Anbietern den zusätzlichen Anreiz, nicht ins Gefängnis zu kommen (Kartellrechtsverstöße ziehen in der Regel außerdem auch noch weitere zivilrechtliche Strafen nach sich).

Die CIDR-*Longest-Match*-Regel ist genau das, was wir (und die IETF) brauchen. Nehmen wir in den obigen Diagrammen an, dass Kunde C von P0 nach P1 wechseln möchte, aber keine neuen Adressen wünscht. Welche Routing-Änderungen müssen dann vorgenommen werden? Eine Lösung besteht darin, dass P0 eine Route ⟨200.2.16.0/20, P1⟩ hinzufügt, die den gesamten Traffic von C an P1 weiterleitet; P1 leitet diesen Datenverkehr dann an C weiter. Die Tabelle von P1 sieht wie folgt aus, und P1 verwendet die Longest-Match-Regel, um den Datenverkehr für seinen neuen Kunden C von jenem für P0 zu unterscheiden.

P1	
destination	next_hop
200.0.0.0/8	**P0**
202.0.0.0/8	P2
201.0.0.0/16	D
201.1.0.0/16	E
200.2.16.0/20	**C**

Das funktioniert zwar, aber der gesamte eingehende Traffic von C mit Ausnahme des von P1 ausgehenden Traffics wird nun über den *ehemaligen* Anbieter P0 geroutet, der als solcher vielleicht nicht mehr die besten Beziehungen zu C unterhält: Der Traffic von C aus P2 wird über P2→P0→P1 geroutet, anstatt über den direkteren Weg P2→P1.

Eine bessere Lösung wäre, wenn *alle* Anbieter außer P1 die Route ⟨200.2.16.0/20, P1⟩ hinzufügen würden. Während der Traffic für 200.0.0.0/8 sonst zu P0 geht, wird dieser spezielle Subblock von allen Providern stattdessen zu P1 geroutet. Der wichtige Fall ist hier P2, stellvertretend für alle anderen Anbieter und deren Router: P2 leitet den 200.0.0.0/8-Datenverkehr an P0 weiter, *mit Ausnahme* des Blocks 200.2.16.0/20, der an P1 geht.

Die Tatsache, dass jeder andere Anbieter in der Welt einen Eintrag für C anlegen muss, kann einiges an Geld kosten, und C wird letztlich derjenige sein, der dafür bezahlen muss. Aber zumindest gibt es eine Wahlmöglichkeit: C kann der Neuadressierung zustimmen (was nicht schwer ist, wenn er gewissenhaft DHCP und vielleicht auch NAT verwendet hat), oder er kann dafür bezahlen, dass er seinen alten Adressblock behält.

Was das zweite Diagramm oben mit den verschiedenen privaten Verbindungen (als gestrichelte Linien dargestellt) betrifft, so ist die Longest-Match-Regel hier für deren Funktionieren wahrscheinlich *nicht* erforderlich. Die »private« Verbindung von A zu P1 könnte auch nur bedeuten, dass

- A seinen ausgehenden Traffic über P1 senden kann
- P1 den Datenverkehr von A über die private Verbindung an A weiterleitet

P2 kann also immer noch über P0 nach A weiterleiten. P1 darf seine Route zu A niemandem sonst bekannt geben.

14.4.3 Hierarchisches Routing über Provider

Beim providerbasierten Routing ist die gewählte Route möglicherweise nicht mehr von Ende zu Ende optimal. Wir ersetzen damit das Problem, eine optimale Route von A nach B zu finden, durch die beiden Probleme, eine optimale Route von A zum Anbieter P von B und dann vom Einstiegspunkt von P nach B zu finden. Diese Strategie spiegelt den zweistufigen hierarchischen Routing-Prozess wider, bei dem zunächst die Adressbits zur Identifizierung des Anbieters und dann die Adressbits einschließlich des Teilnehmeranteils geroutet werden.

Diese zweistufige Strategie führt möglicherweise nicht zu demselben Ergebnis wie die Suche nach der global optimalen Route. Das Ergebnis wird gleich sein, wenn die Kunden von B nur über den einzigen Eintrittsrouter RP von P erreicht werden kön-

nen, was die Situation modelliert, dass P und dessen Kunden wie ein einziger Standort aussehen. Dieses Modell kann jedoch durch einen oder beide der folgenden Aspekte gestört werden:

- Es kann mehrere Einstiegsrouter in das Netz des Anbieters P geben, z. B. RP_1, RP_2 und RP_3, mit unterschiedlichen Kosten gegenüber A.
- Der Kunde B von P kann eine alternative Verbindung zur Außenwelt über einen anderen Provider haben, so wie in der zweiten Abbildung in Abschnitt 14.4, »Providerbasiertes Routing«, dargestellt.

Betrachten wir das folgende Beispiel für die erste (in der Praxis wichtigere) Situation, in der die Anbieter P1 und P2 drei gemeinsame Verbindungspunkte IX1, IX2, IX3 (von Internet eXchange, Abschnitt 14.4.1, »Internet Exchange Points«) haben. Den Verbindungen sind Kosten zugeordnet; wir nehmen an, dass die Kosten von P1 mit denen von P2 vergleichbar sind.

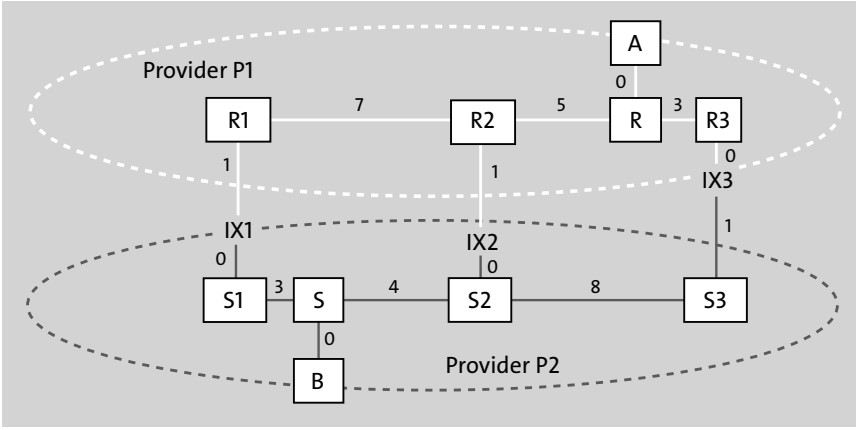


Abbildung 14.1 Verbindung zwischen zwei Providern

Die global kürzeste Route zwischen A und B verläuft über den Verbindungspunkt R2–IX2–S2 mit einer Gesamtlänge von 5 + 1 + 0 + 4 = 10. Der Datenverkehr von A nach B wird jedoch von P1 über den nächstgelegenen Verbindungspunkt zu P2 geleitet, nämlich über die Verbindung R3–IX3–S3. Der Gesamtpfad beträgt 3 + 0 + 1 + 8 + 4 = 16. Der Traffic von B nach A wird von P2 über die Verknüpfung R1–IX1–S1 geroutet, was einer Länge von 3 + 0 + 1 + 7 + 5 = 16 entspricht.

Diese Routing-Strategie wird manchmal auch als *Hot-Potato-Routing* bezeichnet; jeder Anbieter versucht, jeglichen Traffic (die heißen Kartoffeln) so schnell wie möglich loszuwerden, indem er ihn zum nächstgelegenen Austrittspunkt weiterleitet.

Die gewählten Pfade sind nicht nur *ineffizient*, sondern die Routen A⟶B und B⟶A sind auch *asymmetrisch*. Dies kann zum Problem werden, wenn das Timing in beide Richtungen kritisch ist oder wenn einer der beiden Pfade P1 oder P2 deutlich mehr Bandbreite oder weniger Überlast aufweist als der andere. In der Praxis spielt die Asymmetrie der Pfade jedoch nur selten eine Rolle.

Auch die Ineffizienz der Route selbst ist nicht unbedingt ein großes Problem. Routing-Update-Algorithmen konzentrieren sich in erster Linie deshalb auf den kürzesten Pfad, um zu gewährleisten, dass alle berechneten Routen schleifenfrei sind. Solange beide Hälften eines Pfads schleifenfrei sind und sich die Halbstrecken außer an ihrem gemeinsamen Mittelpunkt nicht überschneiden, sind diese Routen auch insgesamt schleifenfrei.

Der BGP-Wert »MED« (Abschnitt 15.6.3, »MULTI_EXIT_DISC«) bietet P1 einen optionalen Mechanismus, um zu vereinbaren, dass Datenverkehr von A⟶B den Verbindungsweg R1–S1 nehmen soll. Dies könnte wünschenswert sein, wenn das Netz von P1 »besser« ist und Kunde A bereit ist, mehr zu bezahlen, um seinen Traffic so lange wie möglich im Netz von P1 zu halten.

14.4.4 IP-Geolokalisierung

Im Prinzip könnten aufeinanderfolgende IP-Adressen durch die providerbasierte Adressierung über einen ganzen Kontinent verstreut sein. In der Praxis verfahren die Anbieter (auch viele Mobilfunkanbieter) jedoch nicht so; ein bestimmter kleiner Adressblock – vielleicht /24 – wird für eine begrenzte geografische Region verwendet. Verschiedene Blöcke gelten für verschiedene Gebiete. Demzufolge ist es prinzipiell möglich, aus einer bestimmten IP-Adresse den entsprechenden ungefähren geografischen Standort zu bestimmen; dies wird als *IP-Geolokalisierung* bezeichnet. Sogar Satelliten-Internetnutzer können geolokalisiert werden, wenn auch manchmal nur bis auf einige hundert Kilometer genau. Verschiedene Unternehmen haben detaillierte Standortkarten erstellt, auf denen viele Orte bis auf die Postleitzahl genau verzeichnet sind und die in der Regel als Online-Dienst zur Verfügung stehen.

Die IP-Geolokalisierung wurde ursprünglich entwickelt, um Werbetreibenden die Möglichkeit zu geben, regional passende Werbung zu schalten. Inzwischen wird sie jedoch für verschiedenste Zwecke verwendet, u. a. zur Ermittlung des nächstgelegenen CDN-Edge-Servers (Abschnitt 1.12.2, »Content-Distribution Networks«), für die Netzwerksicherheit, die Einhaltung nationaler Vorschriften, Tracking von Nutzern auf höherer Ebene und für Einschränkungen beim Streaming von urheberrechtlich geschützten Inhalten.

14.5 Geografisches Routing

Die klassische Alternative zum providerbasierten Routing ist das geografische Routing; als Vorbild dient hier das Telefonvorwahlsystem. Ein Anruf von einem beliebigen Ort in den USA bei der Hauptvermittlungsstelle der Loyola Universität, 773-274-3000, würde traditionell zuerst an die Vorwahl 773 in Chicago weitergeleitet. Von dort wird der Anruf an die 274er-Vermittlungsstelle für das nördliche Stadtgebiet und von dort an den Teilnehmer 3000 weitergeleitet. Eine ähnliche Strategie *kann* auch für das IP-Routing verwendet werden.

Die geografische Adressierung hat einige Vorteile. Die Ermittlung einer guten Route zu einem Ziel ist in der Regel einfach und hinsichtlich der physischen Laufweite nahezu optimal. Beim Anbieterwechsel muss keine Nummernumstellung vorgenommen werden (bei einem Umzug eventuell schon). Und die ungefähre Geolokalisierung der IP-Adressen (Bestimmung des Standorts eines Hosts anhand seiner IP-Adresse) erfolgt automatisch.

Das geografische Routing weist einige kleinere technische Probleme auf. Erstens kann das Routing zwischen unmittelbaren Nachbarn A und B, die zufällig durch eine Grenze zwischen größeren geografischen Zonen getrennt sind, ineffizient sein; Der Pfad könnte von A zum Mittelpunkt der Zone A, zum Mittelpunkt der Zone B und dann zu B führen. Ein weiteres Problem besteht darin, dass beispielsweise viele große Unternehmen selbst geografisch verteilt sind; wenn es um Effizienz geht, bräuchte jedes Büro im Netzwerk einer solchen Organisation einen separaten IP-Adressblock, der seinem physischen Standort entspricht.

Das eigentliche Problem beim geografischen Routing ist aber offensichtlich die Frage, wer den Datenverkehr transportiert. Das providerbasierte Modell hat darauf eine ganz natürliche Antwort: Jede Verbindung gehört einem bestimmten Provider. Beim geografischen IP-Routing könnte mein lokaler Provider anhand des Präfixes sofort wissen, dass ein Paket von mir von Chicago nach San Francisco zugestellt werden soll, aber wer wird es dorthin befördern? Mein Provider muss möglicherweise unterschiedliche Traffic-Verträge für mehrere verschiedene Regionen abschließen. Wenn verschiedene lokale Anbieter unterschiedliche Vereinbarungen für die Zustellung von Paketen auf der Langstrecke treffen, geht die Routing-Effizienz (zumindest in Bezug auf die Tabellengröße) des geografischen Routings wahrscheinlich verloren. Schließlich gibt es keine natürliche Antwort auf die Frage, wem diese langen Verbindungen zwischen den Regionen gehören sollten. Man sollte sich in Erinnerung rufen, dass das gegenwärtige Vorwahlsystem der USA geschaffen wurde, als das amerikanische Telefonnetz in den Händen des Monopolisten AT&T lag und die Frage, wer den Verkehr transportiert, sich überhaupt nicht stellte.

Allerdings repräsentieren die fünf größten regionalen Internet-Registrare geografische Regionen (in der Regel Kontinente), und die anbieterbezogene Adressierung

liegt *unterhalb* dieser Ebene. Das heißt, die IANA hat Adressblöcke an die geografischen RIRs verteilt, und diese Registrare haben dann Adressblöcke an Anbieter vergeben.

Auf interkontinentaler Ebene spielt die Geografie eine Rolle: Einige physische Verbindungswege sind tatsächlich teurer als andere (kürzere). Erdkabel sind viel einfacher zu verlegen als Unterseekabel. Innerhalb eines Kontinents spielt die physische Entfernung jedoch nicht immer so eine große Rolle, wie man vielleicht annehmen könnte. Außerdem kann ein großer Flächenanbieter seine Adressblöcke jederzeit nach Regionen aufteilen, was eine interne geografische Weiterleitung in die richtige Region ermöglicht.

Das Webcomic XKCD hat ein Diagramm der IP-Adressenzuweisung im Jahr 2006 erstellt: *http://xkcd.com/195*

14.6 Epilog

CIDR war eine trügerisch einfache Idee. Auf den ersten Blick handelt es sich um eine einfache Erweiterung des Subnetz-Konzepts, bei der der Netz-/Host-Unterteilungspunkt sowohl nach links als auch nach rechts verschoben wird. Aber es hat zu einem echten hierarchischen Routing geführt, das meist providerbasiert erfolgt. Während CIDR ursprünglich als Lösung einiger frühzeitiger Krisen bei der Zuweisung von IPv4-Adressraum angeboten wurde, ist es inzwischen auch im Kern des IPv6-Routings angekommen.

14.7 Übungen

1. ◊ Betrachten Sie die folgende IPv4-Weiterleitungstabelle, die CIDR verwendet.

destination	next_hop
200.0.0.0/8	A
200.64.0.0/10	B
200.64.0.0/12	C
200.64.0.0/16	D

   Geben Sie für jede der folgenden IP-Adressen an, an welches Ziel sie weitergeleitet wird. 64 entspricht 0x40 bzw. 0100 0000 in Binärform.

   – 200.63.1.1

   – 200.80.1.1

- 200.72.1.1
- 200.64.1.1

2. Betrachten Sie die folgende IPv4-Weiterleitungstabelle, die CIDR verwendet. Die IP-Adressbytes sind hier *hexadezimal*, sodass jede Hexadezimalziffer vier Adressbits entspricht. Dadurch entsprechen Präfixe wie /12 und /20 den Grenzen der Hexadezimalziffern. Zur Erinnerung an die hexadezimale Nummerierung wird »:« als Trennzeichen verwendet und nicht ».«

destination	next_hop
81:30:0:0/12	A
81:3c:0:0/16	B
81:3c:50:0/20	C
81:40:0:0/12	D
81:44:0:0/14	E

Echte Hex-Adressen für IPv4

Das hier verwendete hexadezimale Format für IPv4-Adressen ist in der realen Welt nicht zulässig, aber es gibt ein gültiges Hexadezimalformat, das relativ häufig unterstützt wird, nämlich 0xac.0xd9.0x04.0xce. Dieses Format wurde nie standardisiert, außer der Tatsache, dass es in BSD Unix verwendet wurde, aber seit 2021 wird es (mit vorangestelltem `http://`) von den Browsern Chrome und Firefox erkannt. Weitere Beispiele für archaische IPv4-Adressformate finden Sie in einem Blogbeitrag: *https://blog.dave.tf/post/ip-addr-parsing/*

Geben Sie für jede der folgenden IP-Adressen den next_hop für jeden Eintrag in der obigen Tabelle an, mit dem sie übereinstimmt. Wenn es mehrere Übereinstimmungen gibt, verwenden Sie die Longest-Match-Regel, um festzustellen, wohin das Paket weitergeleitet werden würde.

- 81:3b:15:49
- 81:3c:56:14
- 81:3c:85:2e
- 81:4a:35:29
- 81:47:21:97
- 81:43:01:c0

3. Betrachten Sie die folgende IPv4-Weiterleitungstabelle, die CIDR verwendet. Wie in Übung 1 sind die Bytes der IP-Adressen in *hexadezimaler* Schreibweise angegeben, und zur Erinnerung wird »:« als Trennzeichen verwendet.

destination	next_hop
00:0:0:0/2	A
40:0:0:0/2	B
80:0:0:0/2	C
c0:0:0:0/2	D

   a) Zu welchem next_hop würden die nachfolgenden Adressen jeweils geroutet werden? 63:b1:82:15, 9e:00:15:01, de:ad:be:ef

   b) Erklären Sie, warum jede IP-Adresse irgendwohin weitergeleitet wird, obwohl es keinen Standardeintrag gibt. Tipp: Konvertieren Sie die ersten Bytes ins Binärformat.

4. Geben Sie eine IPv4-Weiterleitungstabelle – unter Verwendung von CIDR – an, die alle Class-A-Adressen (erstes Bit 0) zu next_hop A, alle Class-B-Adressen (erste zwei Bits 10) zu next_hop B und alle Class-C-Adressen (erste drei Bits 110) zu next_hop C leitet.

5. Angenommen, ein IPv4-Router, der CIDR verwendet, hat die folgenden Einträge. Die Adressbytes sind dezimal, mit Ausnahme des dritten Bytes, das *binär* ist.

destination	next_hop
37.149.**00**00 0000.0/18	A
37.149.**01**00 0000.0/18	A
37.149.**10**00 0000.0/18	A
37.149.**11**00 0000.0/18	B

   Wenn der next_hop für den letzten Eintrag ebenfalls A wäre, könnten wir alle vier zu einem einzigen Eintrag 37.149.0.0/**16** → A zusammenfassen. Aber wenn der letzte next_hop B ist, wie lassen sich die vier Einträge dann zu *zwei* Einträgen zusammenfassen? Sie müssen die Longest-Match-Regel anwenden.

6. Angenommen, P, Q und R sind Internetanbieter mit den jeweiligen CIDR-Adressblöcken (mit Bytes in Dezimalzahlen) 51.0.0.0/8, 52.0.0.0/8 und 53.0.0.0/8. P hat dann die Kunden A und B, denen er folgende Adressblöcke zuweist:

A: 51.10.0.0/16

B: 51.23.0.0/16

Q hat die Kunden C und D und weist ihnen folgende Adressblöcke zu:

C: 52.14.0.0/16

D: 52.15.0.0/16

a) ◊ Geben Sie Weiterleitungstabellen für P, Q und R an, unter der Annahme, dass sie untereinander und mit jedem ihrer eigenen Kunden verbunden sind.

b) Nehmen wir nun an, A wechselt von Anbieter P zu Anbieter Q und nimmt seinen Adressblock mit. Geben Sie die Änderungen in den Weiterleitungstabellen von P, Q und R an. Gehen Sie nicht davon aus, dass P bereit ist, Datenverkehr von R weiterzuleiten, der für seinen Ex-Kunden A bestimmt ist. Möglicherweise benötigen Sie die Longest-Match-Regel zur Konfliktlösung.

7. Nehmen wir an, P, Q und R seien die Internetanbieter aus Übung 6. Diesmal nehmen wir an, dass der Kunde C von Provider Q zu Provider R wechselt. R hat nun einen neuen Eintrag 52.14.0.0/16 → C. Geben Sie die Änderungen in den Weiterleitungstabellen von P und Q an. Gehen Sie wiederum davon aus, dass Provider Q den für C bestimmten Verkehr von P nicht weiterleiten möchte.

8. Angenommen, P, Q und R sind Internetanbieter wie in Übung 6. Diesmal sind P und R nicht direkt miteinander verbunden, sondern leiten den Verkehr über Q zueinander weiter. Außerdem ist Kunde B »multihomed« und besitzt eine zusätzliche Sekundärverbindung zu Provider R; Kunde D ist ebenfalls »multihomed« und unterhält eine Sekundärverbindung zu Provider P. R und P verwenden diese Sekundärverbindungen, um an B bzw. D zu senden; diese Sekundärverbindungen werden jedoch nur innerhalb von R bzw. P verwendet. Geben Sie Weiterleitungstabellen für P, Q und R an.

9. Angenommen, das Internet-Routing in den USA verwendet geografisches Routing, und die ersten 12 Bits jeder IP-Adresse stehen für eine geografische Zone von ähnlicher Größe wie ein Telefonvorwahlgebiet. Megacorp erhält das Präfix 12.34.0.0/16, das geografisch in Chicago liegt, und weist seinen Niederlassungen in allen 50 Bundesstaaten Subnetze aus diesem Präfix zu. Megacorp routet seinen gesamten internen Datenverkehr über sein eigenes Netzwerk.

   a) Angenommen, der gesamte Datenverkehr von Megacorp muss in Chicago ein- und ausgehen. Welchen Weg nimmt der Datenverkehr zu und von der Niederlassung in San Diego zu einem Kunden, der ebenfalls in San Diego sitzt?

   b) Nehmen wir nun an, jede Niederlassung hat ihre eigene Verbindung zu einem lokalen Internetanbieter für den ausgehenden Traffic, verwendet aber weiterhin die eigenen 12.34.0.0/16-IP-Adressen. Welchen Weg nimmt nun der Traffic zwischen der Niederlassung in San Diego und dem benachbarten Kunden?

c) Angenommen, Megacorp gibt auf und erhält für jede Niederlassung ein eigenes geografisches Präfix, z. B. 12.35.1.0/24 für San Diego und 12.37.3.0/24 für Boston. Der Traffic von und zu Megacorp wird nun geografisch sinnvolle Wege nehmen. Megacorp möchte nun jedoch sicherstellen, dass der Datenverkehr zwischen den Niederlassungen weiterhin über das interne Netz läuft. Wie muss Megacorp seine internen IP-Weiterleitungstabellen konfigurieren, um dies zu gewährleisten?

Kapitel 21
Weitere TCP-Dynamiken

Ist TCP Reno fair? Bevor wir diese Frage stellen können, müssen wir festlegen, was wir unter Fairness verstehen. Wir schauen uns auch das langfristige Verhalten von TCP Reno (und Reno-ähnlichen) Verbindungen genauer an, da der Wert von `cwnd` entsprechend dem TCP-Sägezahnverlauf ansteigt und abfällt. Insbesondere analysieren wir den durchschnittlichen `cwnd`; es sei daran erinnert, dass der durchschnittliche `cwnd` geteilt durch die RTT den durchschnittlichen Durchsatz der Verbindung darstellt (wir ignorieren hier vorübergehend die Tatsache, dass die RTT nicht konstant ist, aber der dadurch verursachte Fehler ist normalerweise gering).

Nachdem wir eine grundsätzliche Beziehung zwischen TCP Reno `cwnd` und der Paketverlustrate hergestellt haben, sollten wir vielleicht einfach feststellen, dass jedes TCP-Reno-Verfahren »Reno-fair« ist, und eine Regel für »TCP[Reno]-Freundlichkeit« aufstellen.

Der letzte Teil dieses Kapitels behandelt die aktive Warteschlangenverwaltung: die Idee, dass Router einige Annahmen über den TCP-Traffic treffen können, um die sie passierenden Datenströme besser zu verwalten. Es zeigt sich, dass Router das Verhalten von TCP ausnutzen können, um eine bessere Gesamtleistung zu erzielen.

Das Kapitel schließt mit dem TCP-Problem der hohen Bandbreiten und verwandten TCP-Problemen.

21.1 Begriffe der Fairness

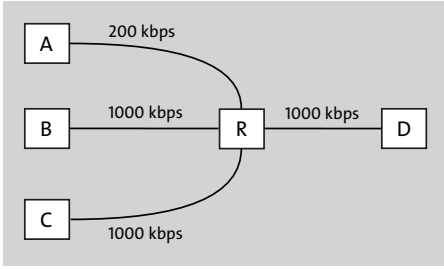
Es gibt mehrere Definitionen für die faire Zuweisung von Bandbreite zwischen Datenströmen, die sich einen Flaschenhalsverbindung teilen. Eine ist die *Fairness der gleichen Anteile*, eine andere die *TCP-Reno-Fairness* (die Bandbreite wird in der Art von TCP Reno aufgeteilt). Es gibt weitere Ansätze, um festzulegen, was eine faire Bandbreitenzuweisung ist.

21.1.1 Max-Min-Fairness

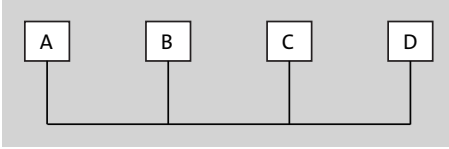
Eine Verallgemeinerung der Fairness der gleichen Anteile für den Fall, dass einige Datenflüsse nach oben gedeckelt werden, ist die Max-Min-Fairness, bei der die Bandbreite eines Datenflusses nicht erhöht werden kann, ohne eine *kleinere* Datenflussrate zu verringern. Alternativ dazu maximieren wir die Bandbreite des Datenflusses mit der kleinsten Kapazität und maximieren dann, wenn dieser Datenfluss festgelegt

ist, den Datenfluss mit der nächstkleineren Bandbreite usw. Intuitivere ist die Erklärung, dass wir die Bandbreite in winzigen Schritten gleichmäßig auf die Datenströme verteilen, bis sie ausgeschöpft ist (was bedeutet, dass wir sie gleichmäßig aufgeteilt haben) oder ein Datenfluss seine von außen festegelegte Bandbreitenobergrenze erreicht. An diesem Punkt setzen wir die Aufteilung unter den verbleibenden Datenströmen fort, bis wir jeweils die äußerste Grenze eines Datenflusses erreichen.

Ein Beispiel: Wir haben die Verbindungen A–D, B–D und C–D, wobei die Verbindung A–R eine Bandbreite von 200 kbit/s und alle anderen Verbindungen eine Bandbreite von 1000 kbit/s haben. Wir beginnen bei Null und erhöhen die Zuweisungen für jede der drei Verbindungen, bis wir bei 200 kbit/s pro Verbindung angelangt sind und die Verbindung A–D die Kapazität der Verbindung A–R ausgeschöpft hat. Die verbleibenden 400 kbit/s werden dann zu gleichen Teilen auf B–D und C–D aufgeteilt, sodass beide am Ende jeweils 400 kbit/s haben.



Ein weiteres Beispiel ist die so genannte *Parkflächentopologie*. Nehmen wir das folgende Netz an:



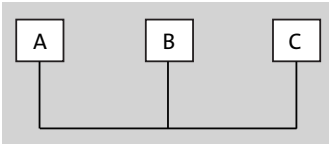
Es gibt vier Verbindungen: eine von A nach D, die alle drei Verbindungen umfasst, und drei Einzelverbindungen A–B, B–C und C–D. Jede Verbindung hat die gleiche Bandbreite. Wenn die Bandbreitenzuweisungen inkrementell auf die vier Verbindungen verteilt werden, ist der erste Punkt, an dem die Bandbreite einer Verbindung ausgeschöpft ist, erreicht, wenn alle vier Verbindungen jeweils 50 % der Verbindungsbandbreite erhalten haben; Max-Min-Fairness bedeutet hier, dass jede Verbindung den gleichen Anteil hat.

21.1.2 Proportionale Fairness

Eine Bandbreitenzuweisung mit den Raten $\langle r_1,r_2,...,r_N\rangle$ für N Verbindungen erfüllt die Bedingung der *proportionalen Fairness*, wenn es sich um eine gültige Bandbreitenzuweisung handelt und für jede andere Zuweisung $\langle s_1,s_2,...,s_N\rangle$ die aggregierte proportionale Änderung folgende Bedingungen erfüllt:

$$(r_1-s_1)/s_1 + (r_2-s_2)/s_2 + ... + (r_N-s_N)/s_N < 0$$

Alternativ bedeutet proportionale Fairness, dass die Summe $\log(r_1) + \log(r_2) + ... + \log(r_N)$ minimiert wird. Wenn sich die Verbindungen nur die Flaschenhalsverbindung teilen, wird die proportionale Fairness mit gleichen Anteilen erreicht. Betrachten wir jedoch das folgende Parkflächennetz:



Angenommen, die Verbindungen A–B und B–C haben eine Bandbreite von 1 Einheit, und wir haben drei Verbindungen A–B, B–C und A–C. Eine proportional faire Lösung besteht dann darin, der Verbindung A–C eine Bandbreite von $\frac{1}{3}$ und jeder der Verbindungen A–B und B–C eine Bandbreite von $\frac{2}{3}$ zuzuweisen (sodass jede Verbindung eine Gesamtbandbreite von 1 hat). Bei jeder Änderung Δb der Bandbreite für die A–C-Verbindung ändern sich die A–B- und B–C-Verbindungen jeweils um $-\Delta b$. Das Gleichgewicht wird an dem Punkt erreicht, an dem eine Abnahme der A-C-Verbindung um 1 % zu zwei Zunahmen um 0,5 % führt, d. h., die Bandbreiten werden im Verhältnis 1:2 geteilt. Wenn x der Durchsatz der A–C-Verbindung ist, minimieren wir mathematisch gesehen $\log(x) + 2\log(1-x)$.

Proportionale Fairness behebt teilweise das Problem, dass TCP Reno lange RTT-Verbindungen benachteiligt; insbesondere ist die Tendenz von TCP hier immer noch nicht proportional fair, aber seine Reaktion kommt einer proportionalen Fairness näher als einer Max-Min-Fairness. Siehe [HBT99].

21.2 TCP-Reno-Verlustrate und cwnd

Wir können die durchschnittliche `cwnd` einer Verbindung als *Paketverlustrate* p ausdrücken, z. B. $p = 10^{-4}$ = ein verlorenes Paket von 10.000. Die Beziehung ergibt sich aus der Annahme, dass alle Paketverluste darauf zurückzuführen sind, dass die Netzobergrenze erreicht wurde. Wir gehen auch davon aus, dass bei Erreichen der Netzobergrenze nur ein Paket verloren geht, obwohl wir auch ein »Cluster« von zusam-

menhängenden Verlusten (innerhalb z. B. einer RTT) als ein einziges *Verlustereignis* zählen könnten.

C steht für die Netzobergrenze, d. h. wenn `cwnd` C erreicht, kommt es zu einem Paketverlust. Während C nur für ein sehr stabiles Netz konstant ist, variiert C normalerweise nicht sehr stark; wir nehmen an, dass es konstant ist. Dann schwankt `cwnd` zwischen C/2 und C, wobei es zu Paketverlusten kommt, wenn `cwnd` = C erreicht wird. Nehmen wir N = C/2 an. Dann werden zwischen zwei aufeinanderfolgenden Paketverlusten, d. h. über einen »Sägezahn« der TCP-Verbindung, insgesamt N+(N+1)+ ... +2N Pakete in N+1 Flights gesendet; diese Summe lässt sich algebraisch als 3/2 N(N+1) ≃ 1,5 N^2 ausdrücken. Die Verlustrate beträgt also ein Paket pro 1,5 N^2, und die Verlustrate p ist 1/(1,5 N^2).

Das durchschnittliche `cwnd` in diesem Szenario ist 3/2 N (d. h. der Durchschnitt von N= $\texttt{cwnd}_{min}$ und 2N= $\texttt{cwnd}_{max}$). Wenn M = 3/2 N der durchschnittliche `cwnd`, $\texttt{cwnd}_{mean}$, ist, können wir die obige Verlustrate in Form von M ausdrücken: die Anzahl der Pakete zwischen den Verlusten beträgt 2/3 M^2, und somit p=3/2 M^{-2}.

Lösen wir dies nun für M=$\texttt{cwnd}_{mean}$ in Bezug auf p; wir erhalten $M^2 = 3/2\ p^{-1}$ und somit

$$M = \texttt{cwnd}_{mean} = 1.225\ p^{-1/2}$$

wobei 1,225 die Quadratwurzel aus $^3/_2$ ist. In dieser Form legt eine gegebene Netzwerkverlustrate die Fenstergröße fest; diese Verlustrate ist letztlich an die Netzwerkkapazität gebunden. Wenn wir uns für den maximalen `cwnd`-Wert statt für den Mittelwert interessieren, multiplizieren wir die obige Zahl mit $^4/_3$.

Daraus ergibt sich die für eine Verbindung verfügbare *Bandbreite* wie folgt (obwohl die RTT nicht konstant sein muss):

$$\text{Bandbreite} = \texttt{cwnd}/RTT = 1{,}225/(RTT \times \sqrt{p})$$

$$\text{bandwidth} = \texttt{cwnd}/RTT = 1.225/(RTT \times \sqrt{p})$$

In [PFTK98] betrachten die Autoren ein TCP-Reno-Modell, das die gemessene Häufigkeit von Coarse-grained Timeouts berücksichtigt (zusätzlich zu schnellen Fast-Recovery-Antworten, die zu einer Halbierung von `cwnd` führen), und entwickeln eine entsprechende Formel.

Mit zunehmender Kapazität der Flaschenhals-Warteschlange steigen sowohl `cwnd` als auch die Anzahl der Pakete zwischen den Verlusten (1/p). Sobald die Warteschlange jedoch so groß ist, dass die Flaschenhalsverbindung zu 100 % ausgelastet ist, nimmt die Bandbreite nicht mehr zu.

Eine andere Annäherung an diese Formel ist, sich daran zu erinnern, dass 1/p die Anzahl der Pakete pro Sägezahn ist; das heißt, 1/p ist die »Fläche« des Sägezahns. Wenn man beide Seiten quadriert, besagt die Formel, dass die TCP Reno-Sägezahnfläche

proportional zum Quadrat der durchschnittlichen Sägezahnhöhe (d. h. zu $\texttt{cwnd}_{\text{mean}}$) ist, wenn die Netzwerkkapazität steigt (d. h. wenn $\texttt{cwnd}_{\text{mean}}$ steigt).

21.2.1 Ungleichmäßige Sägezähne

Im Vorangegangenen haben wir angenommen, dass alle Sägezähne gleich groß sind. Aber wenn das nicht der Fall ist? In [OKM96] wurde dieses Problem unter der Annahme betrachtet, dass jedes Paket die gleiche (kleine) Verlustwahrscheinlichkeit hat (und somit die Intervalle zwischen den Paketverlusten exponentiell verteilt sind). In diesem Modell zeigt sich, dass die obige Formel immer noch gilt, nur die Konstante ändert sich von 1,225 auf 1,309833.

Um zu verstehen, warum unregelmäßige Sägezähne zu einer größeren Konstante führen, stellen Sie sich vor, Sie senden eine große Anzahl K von Paketen, bei denen n Verluste auftreten. Wenn die Verluste in regelmäßigen Abständen auftreten, dann weist der TCP-Graph n gleich große Sägezähne mit jeweils K/n Paketen auf. Wenn die n Verluste jedoch zufällig verteilt sind, werden einige Zähne größer und einige kleiner sein. Die *durchschnittliche* Sägezahnhöhe ist dieselbe wie im Fall der regelmäßigen Verteilung (siehe Übung 7). Die Anzahl der Pakete in einem Sägezahn hängt jedoch normalerweise mit dem *Quadrat* der Höhe dieses Sägezahns zusammen, sodass größere Zähne überproportional viele Pakete enthalten. Daher wird die Zufallsverteilung eine höhere Gesamtzahl an zugestellten Paketen und somit einen höheren mittleren `cwnd`-Wert aufweisen.

Siehe auch Übung 17 für eine einfache Simulation, die eine numerische Schätzung für die Konstante 1,309833 erzeugt.

Beachten Sie, dass Verluste in gleichmäßig verteilten, zufälligen Intervallen auch kein ideales Modell für TCP sind; bei Staus sind die Verlustereignisse weit von statistischer Unabhängigkeit entfernt. Insbesondere ist es unwahrscheinlich, dass unmittelbar nach einem Verlust ein weiterer Verlust auftritt, bis die Warteschlange Zeit hat, sich zu füllen.

21.2.2 Nicht synchronisierte TCP-Verlustereignisse

In Abschnitt 20.3.3, »Beeinflussung der RTT bei TCP Reno«, haben wir ein Modell betrachtet, bei dem alle Verlustereignisse vollständig synchronisiert sind, d. h., immer wenn die Warteschlange sich füllt, kommt es bei *beiden* TCP Reno-Verbindungen zu Paketverlusten. Wenn in diesem Modell $\text{RTT}_2/\text{RTT}_1 = \lambda$, dann sind $\texttt{cwnd}_1/\texttt{cwnd}_2 = \lambda$ und $\text{bandwith}_1/\text{bandwith}_2 = \lambda 2$, wobei $\texttt{cwnd}_1$ und $\texttt{cwnd}_2$ die jeweiligen Durchschnittswerte für `cwnd` sind.

Was passiert, wenn die Verlustereignisse für zwei Verbindungen nicht eine solche eindeutige Eins-zu-Eins-Entsprechung haben? Wir werden das Verhältnis der Verlust-

ereignisse (oder genauer gesagt, der *TCP-Verlustreaktionen*) für Verbindung 1 gegenüber Verbindung 2 anhand der Bandbreiten- und RTT-Verhältnisse ableiten, ohne die Hypothese des synchronisierten Verlusts zu verwenden.

Beachten Sie, dass wir hier die Gesamtzahl der Verlustereignisse (oder Verlustreaktionen) – die Gesamtzahl der TCP-Reno-Sägezähne – über ein großes Zeitintervall vergleichen, und nicht die relativen Verlustwahrscheinlichkeiten *pro Paket*. Eine Verbindung könnte numerisch mehr Verluste haben als eine zweite Verbindung, aber aufgrund einer kleineren RTT mehr Pakete zwischen ihren Verlusten senden als die andere Verbindung und somit *weniger* Verluste pro Paket haben.

losscount_1 und losscount_2 seien die Anzahl der Verlustreaktionen für jede Verbindung über ein langes Zeitintervall T. Für i=1 und i=2 ist die Verlustwahrscheinlichkeit der i-ten Verbindung pro Paket $p_i = \text{losscount}_i/(\text{bandwidth}_i \times T) = (\text{losscount}_i \times \text{RTT}_i)/(\text{cwnd}_i \times T)$. Nach dem Ergebnis von Abschnitt 21.2, »TCP-Reno-Verlustrate und cwnd«, ist $\text{cwnd}_i = k/\sqrt{p_i}$ oder $p_i = k^2/\text{cwnd}_i^2$. Demnach erhalten wir

$$p_i = k^2/\text{cwnd}_i^2 = (\text{losscount}_i \times \text{RTT}_i) / (\text{cwnd}_i \times T)$$

und somit

$$\text{losscount}_i = k^2 T / (\text{cwnd}_i \times \text{RTT}_i)$$

Durch Dividieren und Kürzen erhalten wir

$$\text{losscount}_1/\text{losscount}_2 = (\text{cwnd}_2/\text{cwnd}_1) \times (\text{RTT}_2/\text{RTT}_1)$$

Wir können noch ein wenig weiter gehen: γ sei die obige Verlustquote:

$$\gamma = (\text{cwnd}_2/\text{cwnd}_1) \times (\text{RTT}_2/\text{RTT}_1)$$

Da $\text{RTT}_2/\text{RTT}_1 = \lambda$ ist, müssen wir $\text{cwnd}_2/\text{cwnd}_1 = \gamma/\lambda$ erhalten und somit

$$\text{bandwidth}_1/\text{bandwidth}_2 = (\text{cwnd}_1/\text{cwnd}_2) \times (\text{RTT}_2/\text{RTT}_1) = \lambda^2/\gamma.$$

Beachten Sie: Wenn $\gamma=\lambda$, d. h., wenn die Verbindung mit längerer RTT weniger Verlustereignisse genau umgekehrt proportional zur RTT aufweist, dann gilt: $\text{bandwidth}_1/\text{bandwidth}_2 = \lambda = \text{RTT}_2/\text{RTT}_1$, und auch $\text{cwnd}_1/\text{cwnd}_2 = 1$.

21.3 TCP-Freundlichkeit

Angenommen, wir senden Pakete mit einem Nicht-TCP-Echtzeitprotokoll. Wie sollen wir mit Staus umgehen? Und wie können wir vor allem die Überlast so bewältigen, dass andere Verbindungen – insbesondere TCP-Reno-Verbindungen – fair behandelt werden?

Nehmen wir zum Beispiel an, wir senden interaktive Audiodaten in einer überlasteten Umgebung. Da es sich um Echtzeitdaten handelt, können wir nicht auf die Wie-

derherstellung verlorener Pakete warten und müssen daher UDP statt TCP verwenden. Nehmen wir weiter an, dass wir das Encoding so ändern können, dass die Übertragungsrate bei Bedarf reduziert wird – d. h., dass wir eine *adaptive* Encodierung verwenden –, dass wir es aber vorziehen würden, die Übertragungsrate auf einem hohen Niveau zu halten, wenn keine Staus auftreten. Vielleicht wollen wir auch eine relativ gleichmäßige Übertragungs*rate*; der TCP-Sägezahneffekt führt zu periodischen Schwankungen im Durchsatz, die wir vermeiden wollen.

Unsere Anwendung ist zwar nicht fensterbasiert, aber wir können trotzdem die Anzahl der Pakete überwachen, die zu einem bestimmten Zeitpunkt im Netz unterwegs sind; wenn die Pakete klein sind, können wir stattdessen Bytes zählen. Wir können dies anstelle des TCP-`cwnd` verwenden.

Wir können sagen, dass eine bestimmte Kommunikationsstrategie dann *TCP-freundlich* ist, wenn die Anzahl der Pakete im Netz jederzeit ungefähr gleich dem TCP-Reno-$\texttt{cwnd}_{\text{mean}}$ für die vorherrschende Paketverlustrate p ist. Beachten Sie, dass – unter der Annahme, dass Verluste unabhängige Ereignisse sind, was definitiv nicht ganz, aber oft nahezu richtig ist – in einem ausreichend langen Zeitintervall alle Verbindungen, die einen gemeinsamen Flaschenhals teilen, ungefähr die gleiche Paketverlustrate erfahren.

Der Sinn von TCP Friendliness besteht darin, die Anzahl der ausstehenden Pakete der Nicht-Reno-Verbindung zu regulieren, wenn ein Wettbewerb mit TCP Reno besteht, um ein gewisses Maß an Fairness zu erreichen. Ohne Wettbewerb wird die Anzahl der ausstehenden Pakete jeder Verbindung durch die Transitkapazität plus die Kapazität der Flaschenhals-Warteschlange begrenzt. Einige Nicht-Reno-Protokolle (z. B. TCP Vegas oder Traffic mit konstanter Rate, Abschnitt 21.3.2, »RTP«) können in Abwesenheit von Wettbewerb eine Verlustrate von Null haben, einfach weil sie die Warteschlange nie überfüllen.

Eine andere Möglichkeit, sich der TCP-Freundlichkeit zu nähern, besteht darin, die »Reno-Fairness« als die Bandbreitenzuweisungen zu *definieren*, die TCP Reno angesichts des Wettbewerbs vornimmt. TCP-Freundlichkeit bedeutet dann einfach, dass die gegebene Nicht-Reno-Verbindung ihren Reno-Fairness-Anteil erhält – nicht mehr und nicht weniger.

Wir werden auf TCP-Freundlichkeit im Zusammenhang mit allgemeinem AIMD in Abschnitt 21.4, »Noch einmal AIMD«, zurückkommen.

21.3.1 TFRC

TFRC oder TCP-Friendly Rate Control, RFC 3448, verwendet die erfahrene Verlustrate p und die oben genannten Formeln, um eine Übertragungsrate zu berechnen. Anschließend wird das Senden mit dieser Rate erlaubt, d. h. TFRC ist ratenbasiert und

nicht fensterbasiert. Mit zunehmender Verlustrate wird die Senderate nach unten korrigiert, und so weiter. Die Anpassungen erfolgen jedoch sanfter als bei TCP, sodass die Anwendung eine sich allmählich ändernde Übertragungsrate erhält.

Aus RFC 5348:

> *TFRC ist so konzipiert, dass es im Wettbewerb um Bandbreite mit TCP-Datenflüssen einigermaßen fair ist, wobei wir einen Datenfluss als »einigermaßen fair« bezeichnen, wenn seine Übertragungsrate normalerweise innerhalb eines* ***Faktors von zwei*** *der Übertragungsrate eines TCP-Datenflusses unter den gleichen Bedingungen liegt. (Hervorhebung hinzugefügt; ein Faktor von zwei könnte in manchen Fällen nicht als »nahe genug« angesehen werden).*

Der Nachteil eines gleichmäßigeren Durchsatzes als bei TCP bei gleichzeitigem fairen Wettbewerb um die Bandbreite besteht darin, dass TFRC langsamer als TCP auf Änderungen der verfügbaren Bandbreite reagiert.

TFRC-Sender fügen in jdes Paket eine Sequenznummer, einen Zeitstempel und eine geschätzte RTT ein.

Der TFRC-Empfänger hat die Aufgabe, Rückmeldungspakete zurückzusenden, die als (Teil-)Bestätigungen dienen und auch einen vom Empfänger berechneten Wert für die Verlustrate über die vorherige RTT enthalten. Die Antwortpakete enthalten zudem Informationen über die aktuelle tatsächliche RTT, die der Sender zur Aktualisierung seiner geschätzten RTT verwenden kann. Der TFRC-Empfänger sendet möglicherweise nur ein solches Paket pro RTT zurück.

Das eigentliche Antwortprotokoll besteht aus mehreren Teilen, aber wenn die Verlustrate zunimmt, besteht der primäre Rückkopplungsmechanismus darin, eine neue (niedrigere) Senderate zu *berechnen*, wobei eine Variante der Formel $\texttt{cwnd} = k/\sqrt{p}$ verwendet wird, um dann auf diese neue Rate umzuschalten. Die Rate würde nur dann halbiert, wenn sich die Verlustrate p vervierfacht.

Neuere Versionen von TFRC verfügen über verschiedene Funktionen zur schnelleren Reaktion auf ein ungewöhnlich plötzlich auftretendes Problem, aber im normalen Gebrauch wird die berechnete Senderate die meiste Zeit über verwendet.

21.3.2 RTP

Das *Real-Time Protocol* (RTP) wird manchmal (aber nicht immer) mit TFRC gekoppelt. RTP ist ein UDP-basiertes Protokoll für das Streaming zeitabhängiger Daten.

Einige RTP-Merkmale sind:

- Der Sender legt eine *Rate* (statt einer Fenstergröße) für das Senden von Paketen fest.
- Der Empfänger sendet periodische Zusammenfassungen der Verlustraten.

- ACKs sind relativ selten.
- RTP ist für *Multicast* geeignet; eine sehr begrenzte ACK-Rate ist wichtig, wenn jedes gesendete Paket Hunderte von Empfängern haben kann
- Der Sender passt sein `cwnd`-Äquivalent auf der Grundlage der Verlustrate und der TCP-freundlichen Regel `cwnd`=k/√p nach oben oder unten an.
- Normalerweise wird eine Art »Stabilitätsregel« eingebaut, um plötzliche Änderungen der Rate zu vermeiden.

Als gängiges RTP-Beispiel könnte eine typische VoIP-Verbindung mit einer DS0-Rate (64 kbit/s) alle 20 ms ein Paket senden, das 160 Byte Sprachdaten plus Header enthält.

Damit eine Kombination aus RTP und TFRC Sinn macht, muss die zugrundeliegende Anwendung *ratenadaptiv* sein, sodass die Anwendung auch dann noch funktioniert, wenn die verfügbare Rate reduziert ist. Dies ist bei einfachen VoIP-Kodierungen oft nicht der Fall; siehe Abschnitt 24.11.4, »RTP und VoIP«.

Auf RTP werden wir in Abschnitt 24.11, »Real-time Transport Protocol (RTP)«, zurückkommen.

Das UDP-basierte Transportprotokoll QUIC (Abschnitt 16.1.1, »QUIC«) verwendet einen mit Cubic TCP kompatiblen Stauvermeidungsalgorithmus, was nicht ganz dasselbe ist wie TCP Reno. QUIC hätte jedoch genauso gut TFRC verwenden können, um TCP-Reno-freundlich zu sein.

21.3.3 DCCP-Überlaststeuerung

Wir haben DCCP bereits in Abschnitt 16.1.2, »DCCP«, und Abschnitt 18.15.3, »DCCP«, kennengelernt. DCCP umfasst auch eine Reihe von »Profilen« für das Staumanagement; eine Verbindung kann das Profil wählen, das ihren Anforderungen am besten entspricht. Die beiden Standardprofile sind das TCP-Reno-ähnliche Profil (RFC 4341) und das TFRC-Profil (RFC 4342).

Beim Reno-ähnlichen Profil wird jedes Paket quittiert (obwohl, wie bei TCP, ACKs bei der Ankunft jedes anderen Datenpakets gesendet werden können). Obwohl DCCP-ACKs nicht kumulativ sind, stellt die Verwendung des TCP-SACK-ähnlichen ACK-Vektorformats sicher, dass die Bestätigungen außer in extremen Verlustsituationen zuverlässig empfangen werden.

Der Absender verwaltet `cwnd` ähnlich wie ein TCP-Reno-Sender. Es wird für jede RTT ohne Verlust um eins erhöht und im Falle eines Paketverlustes halbiert. Da Sliding Windows nicht verwendet wird, stellt `cwnd` keine Fenstergröße dar. Stattdessen unterhält der Absender eine Estimated Flight Size (Abschnitt 19.4, »TCP Reno und Fast-Recovery«), die die beste Schätzung des Absenders über die Anzahl der ausstehenden

Pakete darstellt. In RFC 4341 wird dies als *Pipe-Wert* bezeichnet. Der Absender darf dann weitere Pakete senden, solange `pipe` < `cwnd` ist.

Das Reno-ähnliche Profil enthält auch einen Slow-Start-Mechanismus.

Im TFRC-Profil wird mindestens einmal pro RTT ein ACK gesendet. Da ACKs weniger häufig gesendet werden, kann es gelegentlich notwendig sein, dass der Sender ein ACK von ACK sendet.

Wie allgemein bei TFRC ist bei einem DCCP-Sender, der das TFRC-Profil verwendet, nicht die Fenstergröße, sondern die Übertragungsrate begrenzt.

DCCP bietet einen bequemen Programmierrahmen für die Verwendung von TFRC, komplett mit (zumindest in der Linux-Welt) einer traditionellen Socket-Schnittstelle. Der Entwickler bzw. die Entwicklerin muss sich nicht direkt mit den TFRC-Ratenberechnungen befassen.

21.4 Noch einmal AIMD

TCP Tahoe wählt einen Zunahmefaktor von 1, wenn keine Verluste auftraten, und andernfalls einen Abnahmefaktor von 1/2.

Ein anderer Ansatz für TCP-Freundlichkeit ist die Anwendung der AIMD- Strategie von TCP, jedoch mit geänderten Werten. Nehmen wir an, wir bezeichnen mit AIMD(α,β) die Strategie, die Fenstergröße nach einem verlustfreien Fenster um α zu erhöhen und bei einem Verlust die Fenstergröße mit (1–β)<1 zu multiplizieren (β=0,1 bedeutet also, dass das Fenster um 10 % verringert wird). TCP Reno entspricht demnach AIMD(1,0.5).

Jedes AIMD(α,β)-Protokoll folgt ebenfalls einem Sägezahnprofil, wobei die schräge Spitze des Sägezahns die Steigung α hat. Alle Kombinationen von $\alpha > 0$ und $0 < \beta < 1$ sind möglich. Die Abmessungen eines Zahns des Sägezahnprofils sind durch α und β etwas eingeschränkt. h sei die maximale Höhe des Zahns und w die Breite (gemessen in RTTs). Wenn die Verluste in regelmäßigen Abständen auftreten, ist die Höhe des Sägezahns am linken (unteren) Rand (1-β)h und die gesamte vertikale Differenz ist βh. Diese vertikale Differenz muss auch αw sein, und so erhalten wir $\alpha w = \beta h$ oder $h/w = \alpha/\beta$; diese Werte finden Sie auf den Zähnen ganz rechts im Diagramm unten. Diese Gleichungen bedeuten, dass die Proportionen des Sägezahns (h zu w) durch α und β bestimmt werden. Schließlich ist die mittlere Höhe des Sägezahns (1–β/2)h.

Wir sind vor allem an AIMD(α,β)-Fällen interessiert, die TCP-freundlich sind (Abschnitt 21.3, »TCP-Freundlichkeit«). TCP-Freundlichkeit bedeutet, dass eine AIMD(α,β)-Verbindung mit der gleichen Verlustrate wie TCP Reno die gleiche mittlere `cwnd` hat. Jeder Zahn steht für einen Verlust. Die Anzahl der pro Sägezahn gesendeten Pakete ist, unter Verwendung von h und w wie im vorigen Absatz, (1–β/2)hw.

Geometrisch gesehen entspricht die Anzahl der pro Zahn gesendeten Pakete der *Fläche* des Zahns, sodass zwei Verbindungen mit der gleichen Verlustrate pro Paket Zähne mit der gleichen Fläche haben werden. TCP-Freundlichkeit bedeutet, dass zwei Verbindungen den gleichen mittleren `cwnd`-Wert und damit die gleiche durchschnittliche Zahnhöhe haben. Wenn die Zähne zweier Verbindungen die gleiche Fläche und die gleiche durchschnittliche Höhe haben, müssen sie auch die gleiche Breite (in RTTs) haben, und somit müssen die Verlustraten pro *Zeiteinheit* gleich sein, nicht nur die Verlustraten pro Anzahl der Pakete.

Abbildung 21.1 zeigt einen TCP-Reno-Zahn (mitte) zusammen mit einigen unfreundlichen AIMD(α,β)-Zähnen auf der linken Seite und zwei freundlichen Zähnen auf der rechten Seite, wobei der zweite freundliche Zahn dem Reno-Zahn überlagert ist.

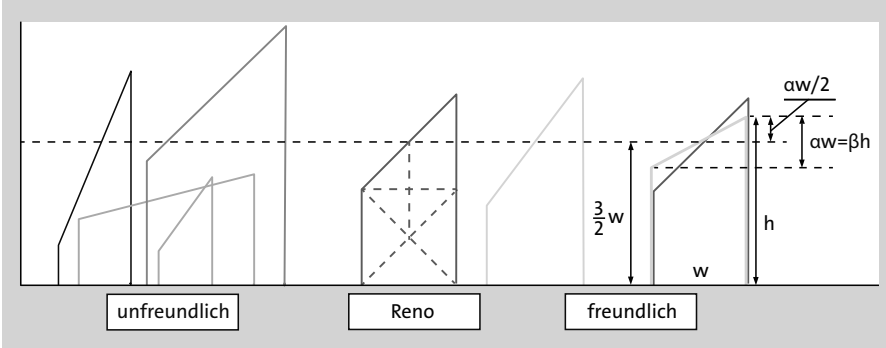


Abbildung 21.1 AIMD-Sägezähne, der Reno-Zahn mit Hilfslinien, die die Mittelpunkte markieren

Die zusätzlichen gestrichelten Linien innerhalb des mittleren Reno-Zahns verdeutlichen die 1×1×2-Proportionen von Reno und zeigen, dass die horizontale gestrichelte Linie, die $\texttt{cwnd}_{\text{mean}}$ darstellt, auf der Höhe $\frac{3}{2}$ w liegt, wobei w wie zuvor die Breite ist.

In dem Zahn ganz rechts, der dem Reno-Zahn überlagert ist, sehen wir, dass $h = (\frac{3}{2}) \times w + (\alpha/2) \times w$. Wir wissen bereits, dass $h = (\alpha/\beta) \times w$ ist; wenn wir diese Ausdrücke gleichsetzen, das w herauskürzen und mit 2 multiplizieren, erhalten wir $(3+\alpha) = 2\alpha/\beta$, oder $\beta = 2\alpha/(3+\alpha)$. Durch Auflösen nach α erhalten wir

$$\alpha = 3\beta/(2-\beta)$$

oder $\alpha \simeq 1{,}5\beta$ für kleine β. Wenn sich der Abnahmefaktor 1-β an 1 annähert, kann das Protokoll durch passende Absenkung von α TCP-freundlich bleiben; z. B. AIMD($\frac{1}{5}$, $\frac{1}{8}$).

Ein kleines β bedeutet, dass eine Verbindung bei Verlusten keinen plötzlichen Bandbreiteneinbrüche unterliegt; dies kann bei Anwendungen wichtig sein, die auf eine regelmäßige Datenübertragungsrate angewiesen sind (z. B. Sprachübertragung). Solche Anwendungen werden manchmal als langsam reagierend bezeichnet, im Gegensatz zur schnellen Reaktion von TCP ($\texttt{cwnd} = \texttt{cwnd}/2$).

21.4.1 AIMD und Konvergenz zur Fairness

TCP-freundliche AIMD(α,β)-Protokolle konvergieren zwar zu Fairness, wenn sie mit TCP Reno konkurrieren (mit gleichen RTTs), aber eine Folge von abnehmendem β ist, dass es länger dauern kann, bis Fairness erreicht wird; hier ist ein Beispiel.

Wir nehmen an, wie oben in Abschnitt 20.3.3, »Beeinflussung der RTT bei TCP Reno«, dass Verlustereignisse für die beiden konkurrierenden Verbindungen synchronisiert sind. Erinnern wir uns daran, dass für zwei TCP-Reno-Verbindungen mit gleicher RTT (d. h. AIMD(α,β) mit $\beta=\frac{1}{2}$) D bei jedem Verlustereignis um die Hälfte reduziert wird, wenn die anfängliche Differenz in den jeweiligen `cwnd`s der Verbindungen D beträgt.

Nehmen wir nun an, wir haben zwei AIMD(α,β)-Verbindungen mit einem anderen Wert von β und wiederum mit einer Differenz D in ihren `Cwnd`-Werten. Die beiden Verbindungen erhöhen `cwnd` jeweils um α pro RTT, sodass D konstant bleibt, wenn keine Verluste auftreten. Bei Verlustereignissen wird D um den Faktor 1–β verringert. Wenn $\beta=\frac{1}{4}$ ist, was $\alpha=\frac{3}{7}$ entspricht, dann wird D bei jedem Verlustereignis nur auf $\frac{3}{4}$ D reduziert, und die »Halbwertszeit« von D ist fast doppelt so groß. Die beiden Verbindungen werden bei D→0 immer noch zu Fairness konvergieren, aber es wird doppelt so lange dauern.

21.5 Aktives Warteschlangenmanagement

Aktives Warteschlangenmanagement (AQM) bedeutet, dass Router aktiv in die Verwaltung ihrer Warteschlangen eingreifen. Das primäre Ziel von AQM ist die Verringerung übermäßiger Warteschlangenverzögerungen; siehe Abschnitt 21.5.1, »Bufferbloat«. Ein sekundäres Ziel ist die Verbesserung der Leistung von TCP-Verbindungen – die den größten Teil des Internet-Traffic ausmachen – durch den Router. Indem den TCP-Verbindungen signalisiert wird, dass sie `cwnd` reduzieren sollten, werden auch die Gesamt-Warteschlangenverzögerungen verringert.

Normalerweise verwalten Router ihre Warteschlangen, indem sie entweder Pakete *markieren* oder *verwerfen*. Alle Router verwerfen Pakete, wenn kein Platz mehr für neu ankommende Pakete ist, aber es handelt sich um eine aktive Verwaltung, wenn Pakete verworfen werden, bevor der Platz in der Warteschlange vollständig erschöpft ist. Die Warteschlangenverwaltung kann am Überlast-»Knie« erfolgen, wenn sich die Warteschlangen gerade aufbauen (und eine Markierung sinnvoller ist), oder wenn die Warteschlange sich zu füllen beginnt und sich der »Klippe« nähert.

Im Großen und Ganzen könnten Prioritäts-Warteschlangen und Random-Drop-Mechanismen (Abschnitt 20.1, »Ein erster Blick auf das Queuing«) als Formen des AQM betrachtet werden, zumindest wenn das Ziel darin besteht, die Gesamtgröße der Warteschlangen zu verwalten. Gleiches gilt für Fair Queuing und Hierarchical

Queuing (Kapitel 22, »Queuing und Scheduling«). Die Mechanismen, die am häufigsten mit der AQM-Kategorie in Verbindung gebracht werden, sind jedoch RED und seine Nachfolger, insbesondere CoDel. Für eine Diskussion der potenziellen Vorteile von Fair Queuing für die Warteschlangenverwaltung siehe Abschnitt 22.6.1, »Fair Queuing und Bufferbloat«.

21.5.1 Bufferbloat

Wie wir in Abschnitt 19.7, »TCP und Auslastung der Flaschenhalsverbindung«, gesehen haben, sollte bei TCP-Reno-Verbindungen im Idealfall die Kapazität der Warteschlange am Flaschenhals-Router die Transitkapazität der Bandbreite × Verzögerung übersteigen. Die Berechnungen in Abschnitt 19.7.1, »TCP-Warteschlangengrößen«, legen nahe, dass die optimale Puffergröße für TCP Reno am Backbone-Router Hunderte von Megabyte betragen könnte. Da Arbeitsspeicher billig ist, liegen die Warteschlangengrößen in der Praxis oft am oberen Ende der Skala. Übermäßige Verzögerungen aufgrund einer zu großen Warteschlangenkapazität werden als *Bufferbloat* bezeichnet. Natürlich ist »übermäßig« eine Frage der Perspektive; wenn der einzige Traffic, an dem Sie interessiert sind, große TCP-Datenströme sind, sind große Warteschlangen gut. Wenn Sie jedoch an Echtzeit-Datenverkehr wie Sprache und interaktivem Video oder auch nur an schnellen Webseiten-Ladevorgängen interessiert sind, wird Bufferbloat zu einem Problem. Große Warteschlangen können auch zu Verzögerungsschwankungen (*Jitter*) führen.

Backbone-Router sind hier eine, aber nicht die einzige Kategorie von Übeltätern. Viele Router in Privathaushalten haben eine Warteschlangenkapazität, die ein Vielfaches des durchschnittlichen Produkts aus Bandbreite und Verzögerung beträgt, was bedeutet, dass die Warteschlangenverzögerung potenziell viel größer als die Ausbreitungsverzögerung ist. Auch Endsysteme haben oft große Warteschlangen; auf Linux-Systemen kann die Standardgröße der Warteschlange mehrere hundert Pakete betragen.

All diese verzögerungsbedingten Probleme lassen sich nicht gut mit interaktivem Traffic oder Echtzeit-Traffic vereinbaren (RFC 7567). Daher gibt es Vorschläge, Router mit viel kleineren Warteschlangen zu betreiben; siehe z. B. [WM05] und [EGMR05]. Dadurch kann sich die Auslastung der Flaschenhalsverbindung eines *einzelnen* TCP-Flows auf 75 % reduzieren. Bei *mehreren* TCP-Datenströmen mit unsynchronisierten Verlusten ist die Situation jedoch oft viel besser.

Dennoch kann die Entscheidung über die Kapazität einer Warteschlange für Router-Manager ein lästiges Problem darstellen. Der CoDel-Algorithmus ist sehr vielversprechend, aber wir beginnen mit einigen früheren Strategien.

21.5.2 DECbit

Bei der in [RJ90] vorgeschlagenen Technik zur Stauvermeidung *markieren* Router, die erste Anzeichen einer Überlast erkennen, die von ihnen weitergeleiteten Pakete; die Absender nutzen diese Markierungen, um ihre Fenstergröße anzupassen. Das System wurde in Anlehnung an den Arbeitgeber der Autoren als DECbit bekannt und in DECnet implementiert (eng verwandt mit der OSI-Protokollsuite), obwohl es offenbar nie eine TCP/IP-Implementierung gab. Die Idee hinter DECbit wurde schließlich in Form von ECN in TCP/IP übernommen. Während jedoch ECN – wie auch die anderen Überlastreaktionen von TCP – im unmittelbaren Bereich der Überlastklippe greift, schlug DECbit vor, die Begrenzung einzuleiten, wenn die Überlast noch minimal ist, also knapp über dem Überlastknie. DECbit war nie eine Lösung für Bufferbloat; in der DECbit-Ära war Speicher teuer und es gab nur selten große Warteschlangenkapazität.

Der DECbit-Mechanismus ermöglichte es den Routern, ein bestimmtes »Überlastbit« zu setzen. Dieses wurde in dem weiterzuleitenden Datenpaket gesetzt, der Status dieses Bits würde jedoch in der entsprechenden ACK zurückgesendet (andernfalls würde der Absender nie von dem Stau Kenntnis erhalten).

DECbit-*Router* definierten »Überlast« als durchschnittliche Größe der Warteschlange von mehr als 1,0; das heißt, Überlast bedeutete, dass die Verbindung gerade über das »Knie« ging. Die Router setzten das Überlastbit immer dann, wenn diese Bedingung für die durchschnittliche Warteschlangengröße erfüllt war.

Das Ziel für DECbit-*Sender* ist dann, dass 50% der Pakete als »Überlast« markiert werden. Sind weniger als 50 % der Pakete markiert, wird `cwnd` um 1 erhöht; bei mehr als 50 % word `cwnd` um den Faktor 0,875 verringert. Beachten Sie, dass sich dies stark von der TCP-Methode unterscheidet, da DECbit mit der Markierung von Paketen am Überlast-»Knie« beginnt, während TCP Reno nur auf Paketverluste reagiert, die an der »Klippe« auftreten.

Eine Folge dieses kniebasierten Mechanismus ist, dass DECbit im Gegensatz zu TCP Reno eine sehr begrenzte Auslastung der Warteschlange anstrebt. Bei einem überlasteten Router würde eine DECbit-Verbindung versuchen, etwa 1,0 Pakete in der Warteschlange des Routers zu behalten, während eine TCP-Reno-Verbindung den Rest der Warteschlange füllen könnte. Daher schneidet DECbit im Prinzip schlecht gegenüber jeder Verbindung ab, bei der der Absender die markierten Pakete ignoriert und einfach versucht, `cwnd` so groß wie möglich zu halten. TCP Vegas strebt auch eine begrenzte Auslastung der Warteschlange an.

21.5.3 Explicit Congestion Notification (ECN)

ECN ist das TCP/IP-Äquivalent zu DECbit, obwohl die tatsächliche Funktionsweise recht unterschiedlich ist. Die aktuelle Version ist in RFC 3168 spezifiziert und modifiziert eine frühere Version in RFC 2481. Der IP-Header enthält ein Zwei-Bit-ECN-Feld, bestehend aus dem ECN-Capable Transport (ECT)-Bit und dem Congestion Experienced (CE)-Bit; das ECN-Feld ist in Abschnitt 9.1, »Der IPv4-Header«, dargestellt. Das ECT-Bit wird von einem Absender gesetzt, um den Routern anzuzeigen, dass er den ECN-Mechanismus nutzen kann. (Dies sind eigentlich die älteren RFC 2481-Bezeichnungen für die Bits, aber für unsere Zwecke sind sie gut geeignet). Der TCP-Header enthält zwei weitere Bits: das ECN-Echo-Bit (ECE) und das Congestion Window Reduced-Bit (CWR); diese sind in der vierten Zeile in Abschnitt 17.2, »TCP-Header«, dargestellt.

Das ursprüngliche Ziel von ECN war es, den TCP-Durchsatz zu verbessern, indem die meisten Paketverluste und die daraus resultierenden Timeouts vermieden werden. Bufferbloat stand wiederum nicht zur Debatte.

ECN und Middleboxen

In den Anfängen von ECN reagierten einige nicht-ECN-fähige Firewalls auf Pakete mit gesetzten CWR- oder ECE-Bits, indem sie diese verwarfen und ein manipuliertes RST-Paket an den Absender zurückschickten. Siehe RFC 3360 für Details und eine Diskussion über diese nicht standardmäßige Verwendung von RST. Dies ist ein gutes Beispiel dafür, dass »Middleboxen« manchmal ein Hindernis für die Entwicklung von Protokollen darstellen; siehe Abschnitt 9.7.2 »Middleboxen«.

Router setzen das CE-Bit im IP-Header, wenn sie das Paket andernfalls verwerfen könnten (oder möglicherweise, wenn die Warteschlange mindestens halb voll ist, oder anstelle eines RED-Drops). Wie bei DECbit senden die Empfänger den CE-Status im ECE-Bit der nächsten ACK an den Sender zurück; der Grund für die Verwendung des ECE-Bits ist, dass dieses Bit zum TCP-Header gehört und somit die TCP-Schicht die Kontrolle darüber hat.

TCP-Sender behandeln ACKs mit gesetztem ECE-Bit genauso, als ob ein Verlust aufgetreten wäre: `cwnd` wird halbiert. Da kein tatsächlicher Verlust vorliegt, können die eintreffenden ACKs weiterhin mit Sliding Windows fortfahren. Der Fast-Recovery-Mechanismus wird nicht benötigt.

Wenn der TCP-Sender auf ein ECE-Bit geantwortet hat (durch Halbierung von `cwnd`), setzt er das CWR-Bit. Sobald der Empfänger ein Paket mit gesetztem CE-Bit in der IP-Schicht erhalten hat, setzt er das ECE-Bit in allen nachfolgenden ACKs, bis er ein Datenpaket mit gesetztem CWR-Bit erhält. Dies sorgt für eine zuverlässige Übermittlung der Überlastinformationen und hilft dem Absender, nur einmal auf mehrere Paketverluste innerhalb eines einzigen Fensters zu reagieren.

Beachten Sie, dass die anfängliche Paketmarkierung auf der IP-Schicht erfolgt, die Erzeugung des markierten ACK und die Antwort des Absenders auf markierte Pakete jedoch auf der TCP-Schicht (dasselbe gilt für DECbit, obwohl die Schichten unterschiedliche Namen haben).

Nur bei Paketen, die andernfalls verworfen worden wären, wird das CE-Bit gesetzt; der Router markiert *nicht* alle wartenden Pakete, sobald seine Warteschlange einen bestimmten Schwellenwert erreicht. Jedes markierte Paket muss wie üblich in der Warteschlange warten, bis es an der Reihe ist, weitergeleitet zu werden. Der Absender erfährt von der Überlast nach einer vollen RTT, während es bei Fast-Retransmit eine volle RTT plus vier Paketübertragungszeiten sind. Eine sehr viel frühere »Legacy«-Strategie bestand darin, dass Router nach dem Verwerfen eines Pakets sofort ein ICMP Source Quench-Paket an den Absender zurückschicken mussten. Dies ist eine schnellere Möglichkeit (die schnellstmögliche), um einen Absender über einen Verlust zu informieren. Sie wurde jedoch nie auf breiter Basis implementiert und mit RFC 6633 offiziell abgelehnt.

Da ECN-Überlasten genauso behandelt werden wie Paketverluste, steht ECN im Wettbewerb mit TCP Reno.

RFC 3540 ist ein Vorschlag (Stand 2016 noch nicht offiziell), den oben beschriebenen Mechanismus leicht abzuändern, um die Erkennung von Empfängern zu unterstützen, die versuchen, Anzeichen von Überlast zu verbergen. Ein Empfänger würde dazu das ECE-Bit in der ACK nicht setzen, wenn ein als überlastet gekennzeichnetes Datenpaket eintrifft. Ein solcher skrupelloser (oder falsch implementierter) Empfänger kann dann einen größeren Anteil an der Bandbreite erhalten, weil der `cwnd` des Senders größer ist, als er sein sollte. Durch die Änderung wird auch die Löschung des ECE-Bits (oder anderer ECN-Bits) durch Middleboxen erkannt.

Die neue Strategie, bekannt als *ECN nonce*, behandelt die ECN-Bits ECT und CE als eine Einheit. Der Wert 00 wird von nicht ECN-fähigen Absendern verwendet, und der Wert 11 wird von Routern als Überlastmarkierung verwendet. ECN-fähige Absender markieren Datenpakete, indem sie *zufällig* 10 (bekannt als ECT(0) oder 01 (bekannt als ECT(1) wählen. Diese Wahl kodiert das *Nonce-Bit*, wobei ECT(0) für ein Nonce-Bit von 0 und ECT(1) für 1 steht; das Nonce-Bit kann auch als der Wert des zweiten ECN-Bits angesehen werden.

Der Empfänger soll nun nicht nur das ECE-Bit setzen, sondern auch die laufende Ein-Bit-Summe der Nonce-Bits in einem neuen TCP-Header-Bit zurückgeben, dem Nonce-Sum-Bit (NS), das dem CRW-Bit unmittelbar vorausgeht. Diese Summe bezieht sich auf alle Datenpakete, die seit dem letzten Paketverlust oder dem letzten Paket, bei dem ein Stau auftrat, empfangen wurden. Versucht der Empfänger, eine Überlastung zu verbergen, indem er das ECE-Bit auf Null setzt, kann er das NS-Bit nicht richtig setzen, da ihm nicht bekannt ist, ob ECT(0) oder ECT(1) verwendet wurde. Bei jedem

Paket, das als Überlast gekennzeichnet ist, hat der Empfänger eine 50%ige Chance, richtig zu raten, aber mit der Zeit wird ein Rateerfolg immer unwahrscheinlicher. Wird eine ECN-Nichtkonformität erkannt, muss der Absender nun die Nutzung von ECN einstellen und kann als Vorsichtsmaßnahme ein kleineres `cwnd` wählen.

Obwohl wir ECN als einen von Routern implementierten Mechanismus beschrieben haben, kann er genauso gut von Switches implementiert werden und ist in vielen kommerziellen Switches verfügbar.

21.5.4 RED

»Herkömmliche« Router verwerfen Pakete erst dann, wenn die Warteschlange voll ist; die Absender erhalten vorher keinen offensichtlichen Hinweis darauf, dass eine Klippe droht. Durch ECN wird dies verbessert, indem TCP-Verbindungen über die bevorstehende Klippe informiert werden, sodass `cwnd` reduziert werden kann, ohne dass tatsächlich Pakete verloren gehen. Hinter *Random Early Detection* (RED)-Routern, die in [FJ93] vorgestellt wurden, steht die Idee, dass der Router gelegentlich ein Paket viel früher fallen lassen darf, etwa wenn die Warteschlange weniger als halb voll ist. Diese frühen Paketverluste sind ein Signal an die Absender, dass sie langsamer senden sollten; wir nennen sie *Signalverluste*. Zwar gehen tatsächlich Pakete verloren, doch werden sie so verworfen, dass normalerweise nur ein Paket pro Fenster (pro Verbindung) verloren geht. Das klassische TCP Reno kommt insbesondere mit mehreren Verlusten pro Fenster nicht gut zurecht, und RED kann solche Mehrfachverluste verhindern. Das Hauptziel von RED war, wie bei ECN, die Verbesserung der TCP-Leistung. Man beachte, dass RED sechs Jahre vor ECN entwickelt wurde.

RED ist streng genommen ein *Warteschlangenverfahren* im Sinne von Abschnitt 22.4, »Warteschlangenverfahren«; ein weiteres ist FIFO. Es ist jedoch oft hilfreicher, RED als Technik zu betrachten, die ein ansonsten FIFO-fähiger Router einsetzen kann, um die Leistung des TCP-Traffic zu verbessern.

Die Entwicklung eines Early-Drop-Algorithmus ist nicht trivial. Ein Vorgänger von RED, bekannt als Early Random Drop (ERD) Gateways, führte einfach eine kleine einheitliche Drop-Wahrscheinlichkeit p ein, z. B. p=0,01, sobald die Warteschlange einen bestimmten Schwellenwert erreicht hatte. Dies löst das TCP-Reno-Problem recht gut, außer dass das Verwerfen mit einer gleichmäßigen Wahrscheinlichkeit p zu einer überraschend hohen Rate von Mehrfachdrops in einem Cluster oder von langen Abschnitten ohne verworfene Pakete führt. Es musste mehr Gleichmäßigkeit erzielt werden, aber Drops in festen Abständen sind wiederum zu gleichmäßig.

Der eigentliche RED-Algorithmus leistet zweierlei: Erstens steigt die grundsätzliche Drop-Wahrscheinlichkeit– p_{base} – von einem minimalen Warteschlangenschwellenwert q_{min} bis zu einem maximalen Warteschlangenschwellenwert q_{max} stetig an (dies könnten 40 % bzw. 80 % der absoluten Warteschlangenkapazität sein); am ma-

ximalen Schwellenwert ist die Abwurfwahrscheinlichkeit noch recht gering. Die Basiswahrscheinlichkeit p_{base} steigt in diesem Bereich linear gemäß der folgenden Formel an, wobei p_{max} die maximale RED-Drop-Wahrscheinlichkeit ist; der in [FJ93] vorgeschlagene Wert für p_{max} war 0,02.

$$p_{base} = p_{max} \times (avg\_queuesize - q_{min})/(q_{max} - q_{min})$$

Zweitens: Je mehr Zeit nach einem RED-Drop vergeht, desto höher ist die tatsächliche Drop-Wahrscheinlichkeit p_{actual}, wie die folgende Formel zeigt:

$$p_{actual} = p_{base} / (1 - count \times p_{base})$$

Hier ist count die Anzahl der Pakete, die seit dem letzten RED-Drop gesendet wurden. Bei count=0 ist $p_{actual} = p_{base}$, aber p_{actual} steigt von da an mit einem garantierten RED-Drop innerhalb der nächsten $1/p_{base}$ Pakete. Auf diese Weise wird ein Mechanismus geschaffen, bei dem RED-Drops so gleichmäßig verteilt sind, dass es unwahrscheinlich ist, dass zwei RED-Drops im selben Fenster derselben Verbindung auftreten, und doch so zufällig, dass es unwahrscheinlich ist, dass die RED-Drops mit einer einzigen Verbindung synchronisiert bleiben und diese somit unfair beeinflussen.

Ein erheblicher Nachteil von RED ist, dass die Wahl der verschiedenen Parameter entschieden *ad hoc* erfolgt. Es ist nicht eindeutig, wie sie so eingestellt werden können, dass TCP-Verbindungen mit kleinen und großen Produkten aus Bandbreite und Verzögerung korrekt gehandhabt werden, oder wie sie überhaupt für eine bestimmte Ausgangsbandbreite eingestellt werden können. Die Wahrscheinlichkeit p_{base} sollte z. B. ungefähr 1/winsize betragen, aber winsize für TCP-Verbindungen kann um mehrere Größenordnungen variieren. RFC 2309 aus dem Jahr 1998 empfahl RED, aber der Nachfolger RFC 7567 aus dem Jahr 2015 ist davon abgerückt und empfiehlt stattdessen, eine geeignete AQM-Strategie zu implementieren, deren Details aber dem Router-Manager überlassen werden sollten.

In Abschnitt 24.8, »RED with In and Out«, sehen wir uns eine Anwendung von RED auf Quality-of-Service-Garantien an.

21.5.5 ADT

In [SKS06] wird der adaptive Drop-Tail-Algorithmus vorgeschlagen, bei dem die maximale Warteschlangenkapazität in Intervallen (etwa 5 Minuten) angepasst wird, um ein bestimmtes gewünschtes Ziel für die Verbindungsauslastung (etwa 95 %) zu erreichen. Am Ende jedes Intervalls wird die verfügbare Warteschlangenkapazität erhöht oder verringert (etwa um 5 %), je nachdem, ob die Auslastung der Verbindung unter oder über dem Zielwert lag. ADT lässt Pakete sonst nicht selektiv fallen; wenn innerhalb der aktuellen Warteschlangenkapazität Platz für ein ankommendes Paket ist, wird es akzeptiert.

ADT passt die Gesamtkapazität der Warteschlange gut an sich langsam ändernde Umstände an, z. B. an die Größe des Benutzerpools. ADT reagiert jedoch nicht auf kurzfristige Schwankungen. Insbesondere versucht es nicht, auf Schwankungen zu reagieren, die innerhalb eines einzelnen TCP-Reno-Sägezahns auftreten. ADT hält auch keinen zusätzlichen Warteschlangenplatz für vorübergehende Paket-Bursts vor.

21.5.6 CoDel

Der CoDel-Warteschlangenverwaltungsalgorithmus (ausgesprochen »coddle«) versucht wie RED, Signalisierungsverluste zu nutzen, um Verbindungen dazu zu bringen, ihre Warteschlangenauslastung zu reduzieren. Auf diese Weise kann CoDel eine große Gesamtkapazität der Warteschlange aufrechterhalten, die zur Aufnahme von Bursts zur Verfügung steht, während gleichzeitig die tatsächliche Auslastung der Warteschlange im Durchschnitt wesentlich geringer ist. Um dies zu erreichen, ist CoDel in der Lage, zwischen vorübergehenden Warteschlangenspitzen und der Auslastung einer »stehenden Warteschlange« zu unterscheiden, die über mehrere RTTs hinweg anhält; das klassische Beispiel für Letzteres ist der Aufbau der Warteschlange von TCP Reno am rechten Rand eines jeden Sägezahns. Im Gegensatz zu RED hat CoDel im Wesentlichen keine einstellbaren Parameter und passt sich an ein breites Spektrum von Bandbreiten und Trafficarten an. Siehe [NJ12] und den Internet-Draft *draft-ietf-aqm-codel-06* (*https://tools.ietf.org/html/draft-ietf-aqm-codel-06*). Ein ausdrückliches Ziel von CoDel ist die Verringerung von Bufferbloat.

CoDel misst den Mindestwert der Warteschlangenauslastung über einen bestimmten kurzen Zeitraum, der als *Intervall* bezeichnet wird. Das Intervall soll etwas größer sein als die meisten RTT_{noLoad}-Werte für Verbindungen; es beträgt normalerweise 100 ms. Anhand dieser Statistik über die minimale Auslastung kann CoDel leicht die Phase des Warteschlangenaufbaus einer TCP-Reno-Verbindung erkennen, die – außer bei Verbindungen mit kurzer RTT – viele Intervalle lang dauert.

CoDel misst die Auslastung der Warteschlange anhand der Verweildauer des Pakets in der Warteschlange und nicht anhand der Größe der Warteschlange in Bytes. Während diese beiden Größen auf jedem Router proportional sind, bedeutet die Verwendung von Zeit anstelle von Raum, dass der CoDel-Algorithmus unabhängig von der abgehenden Bandbreite ist und daher nicht für diese Bandbreite konfiguriert werden muss.

Die Zielvorgabe von CoDel für die Mindestauslastung der Warteschlange beträgt in der Regel 5 % des Intervalls oder 5 ms, obwohl auch 10 % angemessen sind. Ist die Mindestauslastung kleiner, werden keine Maßnahmen ergriffen. Wenn die Mindestauslastung größer wird, geht CoDel in den »Abwurfmodus« über, verwirft ein Paket und beginnt, weitere Paketabwürfe zu planen. Dies dauert so lange, bis die Min-

destauslastung wieder unter dem Zielwert liegt und CoDel in seinen »Normalmodus« zurückkehrt.

Sobald der Abwurfmodus beginnt, wird der zweite Drop für ein Intervall nach dem ersten geplant, wobei der zweite Drop möglicherweise nicht erfolgt, wenn CoDel in den Normalmodus zurückkehren kann. Während CoDel im Abwurfmodus verbleibt, werden weitere Paketabwürfe nach Intervall/$\sqrt{2}$, Intervall/$\sqrt{3}$ usw. geplant, d. h. die Abwurfrate beschleunigt sich proportional zu $\sqrt{n}$, bis die Mindestzeit, die Pakete in der Warteschlange verbringen, wieder klein genug ist, dass CoDel in den Normalmodus zurückkehren kann.

Wenn der Traffic aus einer einzigen TCP-Reno-Verbindung besteht, wird CoDel eines seiner Pakete fallen lassen, sobald die Auslastung der Warteschlange 5 % erreicht. Dies führt zu einer Halbierung von `cwnd`, was höchstwahrscheinlich sogar einen zweiten Paketabwurf unnötig macht. Wenn die RTT der Verbindung ungefähr gleich dem Intervall ist, dann ist die Auslastung der Verbindung die gleiche, wie wenn die Kapazität der Warteschlange auf 5 % der Transitkapazität oder 79 % festgelegt wäre (Abschnitt 19.12, Übung 13). Bei einer geringen Anzahl von unsynchronisierten TCP-Verbindungen steigt die Auslastung der Verbindung jedoch auf über 90 % ([NJ12], Abb. 5 und 8).

Wenn der Traffic aus mehreren TCP-Reno-Verbindungen besteht, sollten einige wenige Drops genügen, damit die meisten Verbindungen ihre `cwnd`s halbieren und damit ihre kollektive Warteschlangenauslastung stark reduzieren. Aber auch wenn der Traffic aus einer einzigen UDP-Verbindung mit *fester Rate* besteht, deren Rate für den Flaschenhals zu hoch ist, funktioniert CoDel noch. In diesem Fall werden so viele Pakete wie nötig verworfen, um die Auslastung der Warteschlange zu verringern, und dieser Zyklus wird bei Bedarf wiederholt. Eine weitere Besonderheit von CoDel besteht darin, dass die Abwurfrate bei einem schnellen Wiedereintritt in den Abwurfmodus dort fortgesetzt wird, wo sie unterbrochen wurde.

Ein Anwendungsbeispiel für CoDel finden Sie in Abschnitt 23.6, »Verzögerung begrenzen«.

21.6 Das TCP-Problem der hohen Bandbreiten

Der TCP-Reno-Algorithmus hat eine schwerwiegende Konsequenz für Verbindungen mit hoher Bandbreite: Das benötigte `cwnd` impliziert eine sehr kleine – unrealistisch kleine – Paketverlustrate p. »Rausch«-Verluste (Verluste, die nicht auf Überlast zurückzuführen sind) sind zwar nicht häufig, aber nicht mehr vernachlässigbar; diese halten das Fenster deutlich kleiner, als es sein sollte. Die folgende Tabelle aus RFC 3649 basiert auf einer RTT von 0,1 Sekunden und einer Paketgröße von 1500 Byte für verschiedene Durchsatzraten. Die `cwnd`-Werte stellen die Produkte aus Bandbreite × RTT dar.

TCP-Durchsatz (Mbit/s)	RTTs zwischen Verlusten	cwnd	Paketverlustrate P
1	5.5	8.3	0.02
10	55	83	0.0002
100	555	833	2×10^{-6}
1000	5555	8333	2×10^{-8}
10,000	55555	83333	2×10^{-10}

Man beachte den sehr kleinen Wert der Verlustwahrscheinlichkeit, der zur Unterstützung von 10 Gbit/s erforderlich ist; dies entspricht einer Bitfehlerrate von weniger als 2×10^{-14}. Bei Glasfaser-Datenverbindungen wird leider oft eine physische Bitfehlerrate von 10^{-13} als akzeptabel angesehen; es gibt also keine Möglichkeit, die Fenstergröße der letzten Zeile der obigen Tabelle zu unterstützen. (Die Verwendung von fehlerkorrigierenden Codes auf OTN-Verbindungen, Abschnitt 6.2.3, »Optical Transport Network«, kann die Bitfehlerrate auf weniger als 10^{-15} reduzieren). Eine weitere Quelle für »Rausch«-Verluste sind Überläufe von Warteschlangen in Ethernet-Switches; Switches haben in der Regel viel kürzere Warteschlangen als Router. Bei 10 Gbit/s leitet ein Switch ein Paket pro Mikrosekunde weiter; bei dieser Rate muss ein Burst nicht lange dauern, um die Warteschlange des Switches zu überlasten.

Hier eine ähnliche Tabelle, die `cwnd` in Form der Paketverlustrate ausdrückt:

Paketverlustrate P	cwnd	RTTs zwischen Verlusten
10^{-2}	12	8
10^{-3}	38	25
10^{-4}	120	80
10^{-5}	379	252
10^{-6}	1,200	800
10^{-7}	3,795	2,530
10^{-8}	12,000	8,000
10^{-9}	37,948	25,298
10^{-10}	120,000	80,000

Die beiden obigen Tabellen zeigen, dass große Fenstergrößen extrem kleine Abwurfraten erfordern. Dies ist das *TCP-Problem bei hoher Bandbreite*: Wie können wir ein

großes Fenster aufrechterhalten, wenn ein Pfad ein großes Produkt aus Bandbreite und Verzögerung aufweist? Das Hauptproblem besteht darin, dass nicht-kongestive Paketverluste (Rauschen) die Fenstergröße möglicherweise weit unter den möglichen Wert senken. Ein sekundäres Problem besteht darin, dass selbst wenn solche zufälligen Verluste nicht signifikant sind, der Anstieg von `cwnd` auf ein vernünftiges Niveau recht langsam sein kann. Wenn die Netzobergrenze bei etwa 2.000 Paketen liegt, dann würde die normale sägezahnförmige Rückkehr zur Obergrenze nach einem Verlust 1.000 RTTs dauern. Das ist zwar langsam, aber der Absender hätte immer noch einen durchschnittlichen Durchsatz von 75 %, wie wir in Abschnitt 19.7, »TCP und Auslastung der Flaschenhalsverbindung«, gesehen haben. Noch gravierender wäre es, wenn sich die Netzobergrenze aufgrund der Abnahme des konkurrierenden Traffic auf 4.000 Pakete verdoppeln würde. Dann würde der Absender weitere 2.000 RTTs benötigen, um den Punkt zu erreichen, an dem die Verbindung gesättigt ist.

In Abbildung 21.2 sind die Netzobergrenze und der ideale TCP-Sägezahn oben dargestellt. Der ideale TCP-Sägezahn sollte zwischen 50 % und 100 % der Obergrenze liegen. Im Diagramm treten bei den X »Rauschen« oder nicht-kongestive Verluste auf, die den Durchsatz auf ein viel niedrigeres Durchschnittsniveau senken.

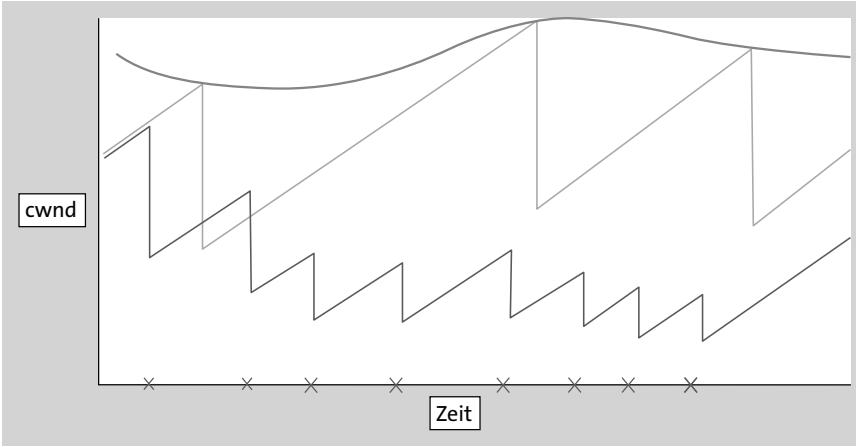


Abbildung 21.2 TCP ohne (oben) und mit (unten) zufälligen Verlusten; in diesem Diagramm treten zufällige Verluste 3–4 mal so häufig auf wie Engpassverluste.

21.7 Das Problem der verlustbehafteten Verbindungen

Eng verwandt mit dem Problem der hohen Bandbreite ist das Problem der verlustbehafteten Verbindungen, bei denen eine Verbindung auf dem Pfad eine relativ hohe *nicht-kongestive Verlustrate* aufweist; das klassische Beispiel für eine solche Verbindung ist Wi-Fi. Wenn TCP auf einem Pfad mit einer Verlustrate von 1,0 % verwendet

wird, dann zeigt Abschnitt 21.2, »TCP-Reno-Verlustrate und cwnd«, dass der Absender einen durchschnittlichen `cwnd` von nur etwa 12 erwarten kann, egal wie hoch das Produkt aus Bandbreite und Verzögerung ist.

Der einzige Unterschied zwischen dem Problem der verlustbehafteten Verbindungen und dem Problem der hohen Bandbreite ist der Umfang; das Problem der verlustbehafteten Verbindungen betrifft ungewöhnlich große Werte von p, während das Problem der hohen Bandbreite Umstände betrifft, in denen p zwar recht niedrig, *aber nicht niedrig genug* ist. Wenn bei einer gegebenen nicht-kongestiven Verlustrate p das Produkt aus Bandbreite und Verzögerung weit über $1{,}22/\sqrt{p}$ liegt, ist der Sender nicht in der Lage, einen `cwnd`-Wert nahe der Netzobergrenze zu erreichen.

21.8 Das Problem der Satelliten-TCP-Verbindungen

Ein drittes TCP-Problem, das nur teilweise mit den beiden vorhergehenden zusammenhängt, ist das der TCP-Benutzer mit sehr langen RTTs. Das dramatischste Beispiel hierfür sind Satelliten-Internetverbindungen (Abschnitt 4.4.2, »Satelliteninternet«). Bei der Kommunikation wird das Signal über einen Satelliten in einer geostationären Umlaufbahn geleitet; ein Umlauf umfasst vier Auf- oder Abwärtsreisen von jeweils ~36.000 km und hat somit eine Ausbreitungsverzögerung von etwa 500 ms. Wenn wir von einer Bandbreite von 1 Mbit/s pro Nutzer ausgehen (Satelliten-ISPs bieten in der Regel eine recht begrenzte Bandbreite an, obwohl die Spitzenbandbreiten höher sein können), dann beträgt das Produkt aus Bandbreite und Verzögerung etwa 40 Pakete. Das ist nicht besonders hoch, selbst wenn man die typischen Warteschlangenverzögerungen von weiteren ~500ms mit einbezieht, aber die Tatsache, dass es viele Sekunden dauert, um auch nur eine moderate `cwnd` zu erreichen, ist für viele Anwendungen ein Ärgernis. Die meisten ISPs bieten einen »Beschleunigungs«-Mechanismus an, wenn sie eine TCP-Verbindung als Dateidownload identifizieren können; dabei wird die Datei in der Regel mithilfe eines proprietären Protokolls über den Satellitenabschnitt des Pfades übertragen. Dies ist jedoch nicht sehr nützlich bei TCP-Verbindungen mit mehreren bidirektionalen Austauschvorgängen, z. B. bei der Verwendung von VPN-Verbindungen.

21.9 Epilog

Der zentrale Überlastalgorithmus von TCP Reno basiert auf Algorithmen aus dem inzwischen fünfundzwanzig Jahre alten Papier von Jacobson und Karel aus dem Jahr 1988 [JK88]. Es gibt sowohl Bedenken, dass TCP Reno zu viel Bandbreite verwendet (das Problem der Gier) als auch, dass es nicht genug verwendet (das Problem der hohen Bandbreite bei TCP).

Im nächsten Kapitel betrachten wir alternative Versionen von TCP, die versuchen, einige der oben genannten Probleme im Zusammenhang mit TCP Reno zu lösen.

21.10 Übungen

1. Finden Sie für jeden Wert α oder β unten den anderen Wert, sodass AIMD(α,β) TCP-freundlich ist.

   a) $\beta = 1/5$

   b) $\beta = 2/9$

   c) $\alpha = 1/5$

   Wählen Sie dann das Paar mit dem kleinsten α aus und zeichnen Sie ein Sägezahnziagramm, das annähernd proportional ist: α sollte die Steigung des linearen Anstiegs sein und β sollte der Anteil des Rückgangs am Ende jedes Zahns sein.

2. Angenommen, zwei TCP-Datenströme konkurrieren miteinander. Die Datenströme haben die gleiche RTT. Der erste Datenfluss verwendet AIMD(α1,β1) und der zweite AIMD(α2,β2); keiner der beiden Datenströme ist notwendigerweise TCP-Reno-fähig. Die beiden Verbindungen konkurrieren jedoch fair miteinander, d. h. sie haben die gleichen durchschnittlichen Paketverlustraten. Zeigen Sie, dass

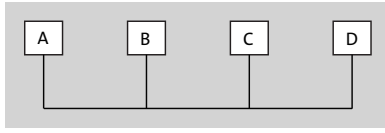
   $\alpha_1/\beta_1 = (2-\beta_2)/(2-\beta_1) \times \alpha_2/\beta_2$.

   Nehmen Sie regelmäßige Verluste an und verwenden Sie die Methoden aus Abschnitt 21.4, »Noch einmal AIMD«. Tipp: Zeigen Sie zunächst mithilfe des dortigen Arguments, dass die Zähne der beiden Datenflüsse die gleiche Breite w und die gleiche durchschnittliche Höhe haben müssen. Die durchschnittliche Höhe ist nicht mehr 3w/2, kann aber immer noch durch w, α und β ausgedrückt werden. Zeigen Sie anhand eines Diagramms, dass für jeden Zahn die durchschnittliche Höhe = h×(1–β/2) ist, wobei h die Höhe der rechten Kante des Zahns ist. Setzen Sie anschließend die beiden Durchschnittshöhen der Zähne h_1/β_1 und h_2/β_2 gleich. Nutzen Sie schließlich die Beziehungen $\alpha_i w = \beta_i h_i$, um h_1 und h_2 zu eliminieren.

3. Zeigen Sie anhand des Ergebnisses der vorigen Übung, dass AIMD(α_1,β) (im Sinne eines fairen Wettbewerbs) äquivalent ist zu AIMD(α_2,0.5), mit $\alpha_2 = \alpha 1 \times (2-\beta)/3\beta$.

4. Angenommen, zwei 1-kB-Pakete werden als Teil einer Paket-Paar-Prüfung gesendet, und die gemessene Mindestzeit zwischen dem Eintreffen beträgt 5 ms. Wie groß ist die geschätzte Flaschenhalsbandbreite?

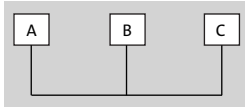
5. Betrachten Sie erneut das Netz aus Abschnitt 21.1.1, »Max-Min-Fairness«:



a) Angenommen, wir haben *zwei* Ende-zu-Ende-Verbindungen zusätzlich zu einer Einzelverbindung für jede Verbindung. Finden Sie die max-min-faire Zuteilung.

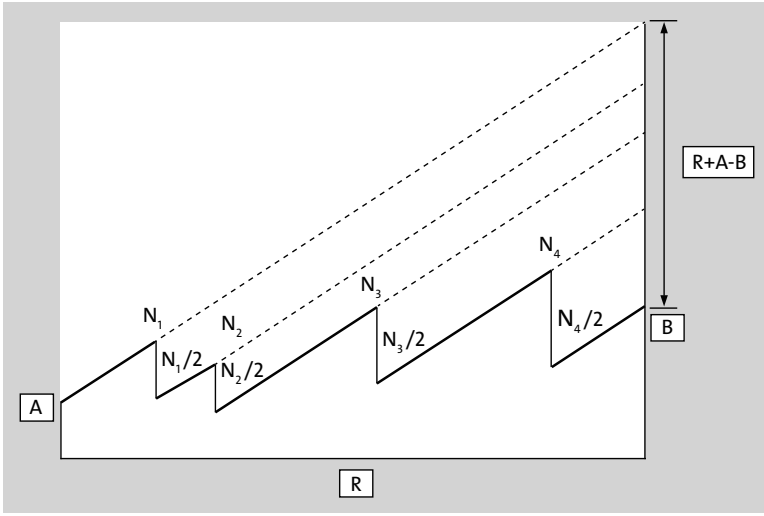
b) Angenommen, wir haben eine einzige Ende-zu-Ende-Verbindung und eine B–C- und C–D-Verbindung, aber zwei A-B-Verbindungen. Finden Sie die max-min-faire Zuteilung.

6. Betrachten Sie das Netzwerk mit zwei Verbindungen:



Angenommen, es gibt zwei A–C-Verbindungen, eine A–B-Verbindung und eine A–C-Verbindung. Finden Sie eine Aufteilung, die proportional fair ist.

7. Angenommen, wir verwenden TCP Reno, um K Pakete über R RTT-Intervalle zu senden. Die Übertragung erfährt n nicht notwendigerweise einheitliche Verlustereignisse; der TCP-`cwnd`-Graph hat daher n Sägezahnspitzen der Höhe N_1 bis N_n. Am Anfang des Graphen ist `cwnd` = A, und am Ende des Graphen ist `cwnd` = B. Zeigen Sie, dass die Summe $N_1 + ... + N_n$ 2(R+A–B) ist, und dass insbesondere die durchschnittliche Zahnhöhe unabhängig von der Verteilung der Verlustereignisse ist.



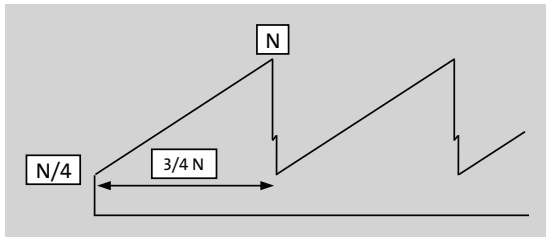
8. Angenommen, das Produkt aus Bandbreite und Verzögerung für einen Netzwerkpfad beträgt 1000 Pakete. Der einzige Traffic auf dem Pfad kommt von einer einzigen TCP-Reno-Verbindung. Ermitteln Sie für jeden der folgenden Fälle den durchschnittlichen `cwnd` und die ungefähre Anzahl der Pakete zwischen den Verlusten (den reziproken Wert der Verlustrate). Ihre Antworten sollten insgesamt die Formel in Abschnitt 21.2, »TCP-Reno-Verlustrate und cwnd«, widerspiegeln.

a) ◊ Die Kapazität der Flaschenhals-Warteschlange ist nahe null.

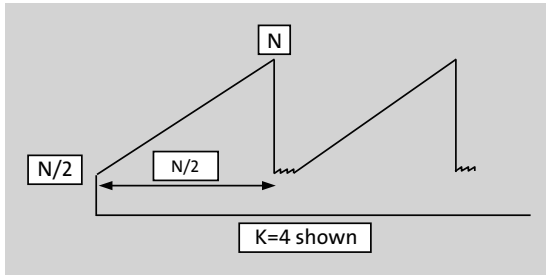
b) Die Kapazität der Flaschenhals-Warteschlange beträgt 1000 Pakete.

c) Die Kapazität der Flaschenhals-Warteschlange beträgt 3000 Pakete.

9. Angenommen, TCP Reno hat regelmäßig beabstandete Sägezahnspitzen gleicher Höhe, die Paketverluste treten jedoch paarweise auf, mit gerade so viel Abstand, dass beide Verluste in einem Paar getrennt gezählt werden. N ist groß genug, dass der Abstand zwischen den beiden Verlusten vernachlässigbar ist. Im Endeffekt reicht jeder große Zahn von N/4 bis N. In Abschnitt 21.2, »TCP-Reno-Verlustrate und cwnd«, ist $\texttt{cwnd}_{\text{mean}} = K/\sqrt{p}$ für eine Konstante K. Ermitteln Sie die Konstante. Hinweis: Aus den gegebenen Informationen kann man sowohl $\texttt{cwnd}_{\text{mean}}$ als auch die Anzahl der in einem Zahn gesendeten Pakete bestimmen. Die Verlustrate ist p = 2/(Anzahl der in einem Zahn gesendeten Pakete).

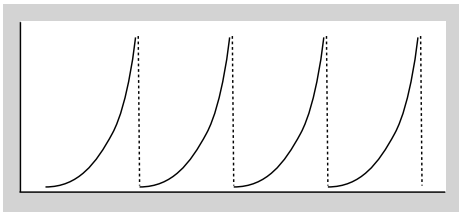


10. Wie in der vorherigen Übung nehmen wir an, dass eine TCP-Übertragung große Sägezähne der Höhe N hat. Zwischen jedem Paar aufeinander folgender großer Zähne gibt es jedoch K–1 zusätzliche Verluste, die zu K–1 zusätzlichen kleinen Zähnen führen; N ist groß genug, dass diese kleinen Zähne ignoriert werden können. Es wird eine Nicht-Reno-Variante von TCP verwendet, sodass zwischen diesen winzigen Zähnen angenommen wird, dass `cwnd` nicht halbiert wird; während des Verlaufs dieser winzigen Zähne ändert sich `cwnd` überhaupt nicht. Der große Zahn hat eine Breite N/2 und eine Höhe von N/2 bis N, und es gibt K Verluste pro großem Zahn. Ermitteln Sie das Verhältnis $\texttt{cwnd}/(1/\sqrt{p})$ in Abhängigkeit von K. Wenn K=1 ist, sollte Ihre Antwort auf die in Abschnitt 21.2, »TCP-Reno-Verlustrate und cwnd«, abgeleitete Lösung hinauslaufen.



11. Angenommen, ein TCP-Reno-Zahn beginnt mit cwnd = c und enthält N Pakete. w sei die Breite des Zahns, wie üblich in RTTs. Zeigen Sie, dass $w = (c^2 + 2N)^{1/2} - c$. Hinweis: Die maximale Höhe des Zahns wird c+w und die durchschnittliche Höhe c + w/2 sein. Stellen Sie eine Gleichung auf, die c, w und N in Beziehung setzt, und lösen Sie w mit Hilfe der Quadratformel.

12. Angenommen, wir haben eine Nicht-Reno-Implementierung von TCP, bei der die Formel, die cwnd c mit der Zeit t, gemessen in RTTs seit dem letzten Verlustereignis, in Beziehung setzt, c = t2 lautet (im Gegensatz zu TCP Renos c = c0 + t). Der Sägezahn sieht dann wie folgt aus:



Die Anzahl der in einem Zahn der Breite T RTTs gesendeten Pakete entspricht dem Kehrwert der Verlustrate p und beträgt ungefähr T3/3 (dies kann man entweder als gegeben hinnehmen oder durch Bildung eines Integrals über t2 von 0 bis T oder mithilfe der Formel für 12 + 22 + ... + T2 nachweisen. Der durchschnittliche cwnd-Wert ist demnach T3/3T = T2/3.

Leiten Sie eine Formel her, die die durchschnittliche cwnd in Abhängigkeit von der Verlustrate p ausdrückt. Hinweis: Der Exponent für p sollte $-\frac{2}{3}$ sein, anders als die $-\frac{1}{2}$ in der Formel in Abschnitt 21.2, »TCP-Reno-Verlustrate und cwnd«.

13. Unter den Annahmen von Übung 12 wird cwnd um etwa 2t pro RTT inkrementiert. Zeigen Sie, dass die Regel für die Erhöhung von cwnd wie folgt ausgedrückt werden kann:

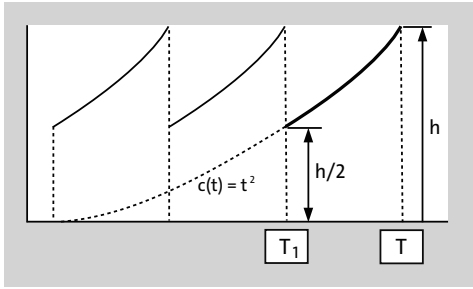
    $\text{cwnd} \mathrel{+}= \alpha \text{cwnd}^{1/2}$

    und finden Sie den Wert von α.

14. Zeigen Sie unter Verwendung derselben TCP-Annahmen wie in Übung 12, dass cwnd immer noch proportional zu $p^{-2/3}$ ist, wobei p die Verlustrate ist, und nehmen Sie Folgendes an:

    - Die obere Grenze jedes Zahns folgt der Kurve cwnd = c(t) = t^2, wie zuvor.
    - Jeder Zahn hat eine rechte Grenze bei t=T und eine linke Grenze bei t=T_1, wobei $c(T_1) = 0{,}5 \times c(T)$.

    (In der vorherigen Übung haben wir angenommen, dass T_1 = 0 ist und dass cwnd nach jedem Verlustereignis auf 0 fällt; hier nehmen wir an, dass eine multiplikative Abnahme mit β=1/2 wirksam ist). Die Anzahl der in einem Sägezahn gesendeten Pakete ist nun $k \times (T^3 - T_1{}^3)$, und der mittlere cwnd-Wert ist dies geteilt durch $T - T_1$.



Beachten Sie, dass die Sägezähne hier mit zunehmender Höhe proportional schmaler werden. Tipp: Zeigen Sie, dass $T_1 = (0{,}5)^{0{,}5} \times T$ ist, und entfernen Sie dann T_1 aus den obigen Gleichungen.

15. Nehmen wir an, dass bei einem TCP-Reno-Lauf jedes Paket mit gleicher Wahrscheinlichkeit verloren geht; die Anzahl der Pakete N in jedem Zahn ist daher exponentiell verteilt. Das heißt, $N = -k \log(X)$, wobei X eine gleichmäßig verteilte Zufallszahl im Bereich $0<X<1$ ist (k, das hier keine Rolle spielt, ist das mittlere Intervall zwischen Verlusten). Schreiben Sie ein einfaches Programm, das einen solchen TCP-Reno-Lauf simuliert. Geben Sie am Ende der Simulation einen Schätzwert für die Konstante C in der Formel $\texttt{cwnd}_{mean} = C/\sqrt{p}$ aus. Sie sollten einen Wert von etwa 1,31 erhalten, wie in der Formel in Abschnitt 21.2.1, »Ungleichmäßige Sägezähne«.

    Tipp: Es ist nicht nötig, Paketübertragungen zu simulieren; wir erstellen einfach eine Reihe von Zähnen mit zufälliger Größe und führen laufende Summen über die Anzahl der gesendeten Pakete, die Anzahl der dafür benötigten RTT-Intervalle und die Anzahl der Verlustereignisse (d. h. Zähne). Nach jedem Verlustereignis (jedem Zahn) aktualisieren wir:

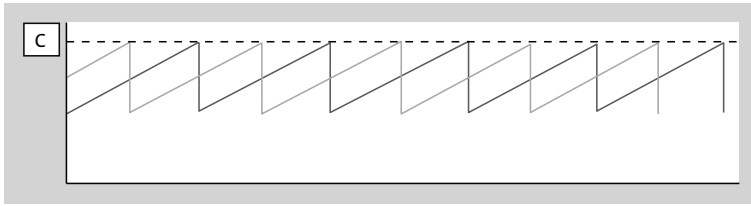
    – total_packets += in diesem Zahn gesendete Pakete
    – RTT_intervals += RTT-Intervalle in diesem Zahn
    – loss_events += 1 (ein Zahn = ein Verlustereignis)

    Wenn ein Verlustereignis, das das Ende eines Zahns markiert, bei einem bestimmten Wert von `cwnd` auftritt, beginnt der nächste Zahn bei der Höhe $c = \texttt{cwnd}/2$. Wenn N der Zufallswert für die Anzahl der Pakete in diesem Zahn ist, dann ist nach der vorherigen Übung die Zahnbreite in RTTs $w = (c^2 + 2N)^{1/2} - c$; die nächste Spitze (d. h. Verlustereignis) tritt also auf, wenn $\texttt{cwnd} = c + w$. Aktualisieren Sie die Summen wie oben und fahren Sie mit dem nächsten Zahn fort. Es sollte möglich sein, diese Simulation für 1 Million Zähne in recht kurzer Zeit durchzuführen.

16. Angenommen, zwei TCP-Verbindungen haben die gleiche RTT und teilen sich eine Flaschenhalsverbindung, um die es keine andere Konkurrenz gibt. Die Größe der Flaschenhals-Warteschlange ist im Vergleich zum Produkt Bandbreite × RTT_{noLoad} vernachlässigbar. Verlustereignisse treten in regelmäßigen Abständen auf.

In Übung 12 des vorigen Kapitels sollten Sie zeigen, dass bei synchronisierten Verlusten die beiden Verbindungen zusammen 75 % der Gesamtkapazität des Flaschenhalses ausnutzen

Nehmen wir nun an, dass die beiden TCP-Verbindungen keine gemeinsamen Verluste haben, sondern sich vielmehr in regelmäßigen Abständen *abwechseln*, wie im folgenden Diagramm dargestellt.



Beide Verbindungen haben ein maximales `cwnd` von C. Wenn bei Verbindung 1 ein Verlust auftritt, hat Verbindung 2 `cwnd` = 75% von C und umgekehrt.

a) Wie hoch ist die kombinierte Transitkapazität der Pfade, ausgedrückt in C? (Da die Größe der Warteschlange vernachlässigbar ist, ist die Transitkapazität ungefähr die Summe der `cwnd` am Verlustpunkt).

b) Ermitteln Sie die Auslastung der Flaschenhalsverbindung. Hinweis: Da die Größe der Warteschlange vernachlässigbar ist, entspricht dies ungefähr dem Verhältnis der durchschnittlichen Gesamt-`cwnd` zur Transitkapazität von Teil (a). Sie sollte mindestens 85% betragen.