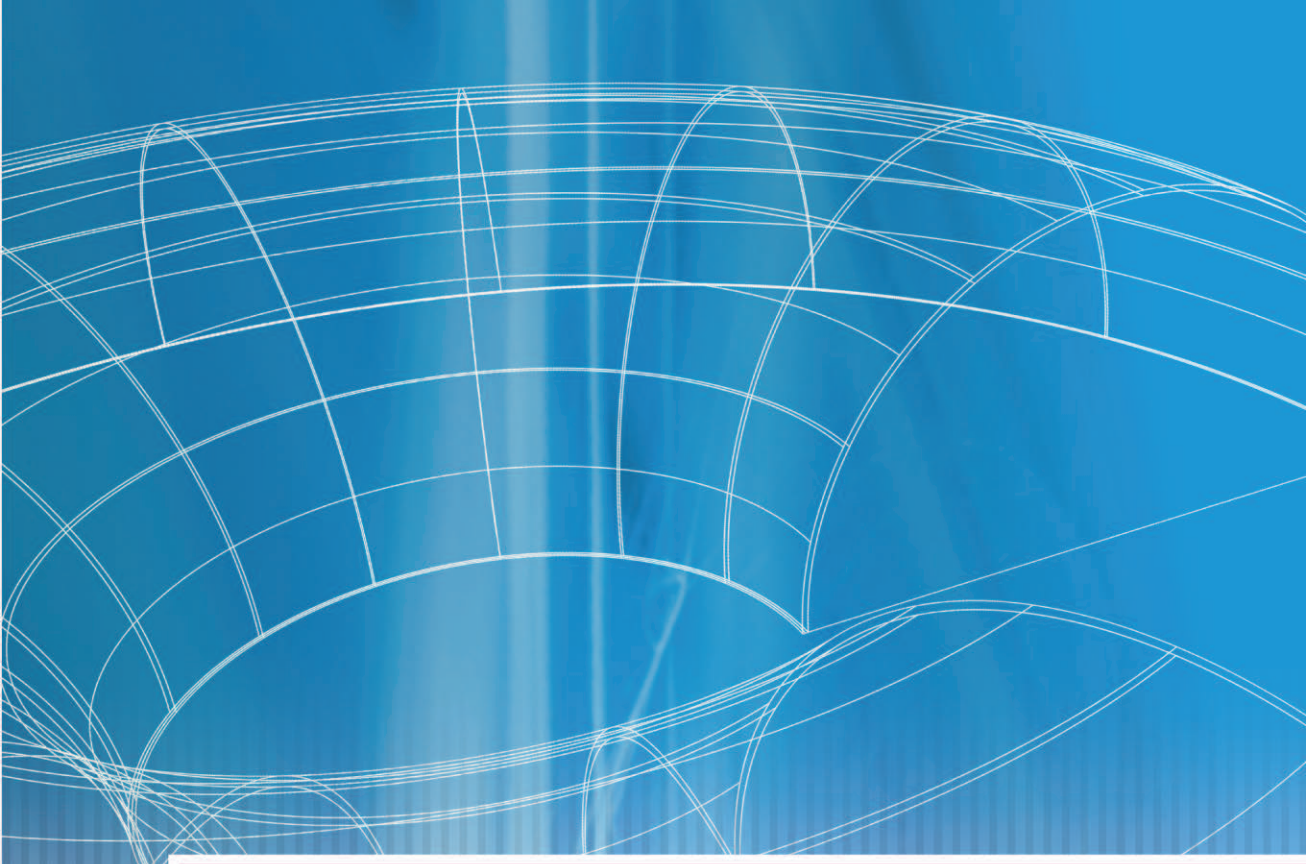

Computernetzwerke

Der Top-Down-Ansatz

6., aktualisierte Auflage

James Kurose
Keith Ross

EXTRAS ONLINE

ALWAYS LEARNING

PEARSON

Inhaltsverzeichnis

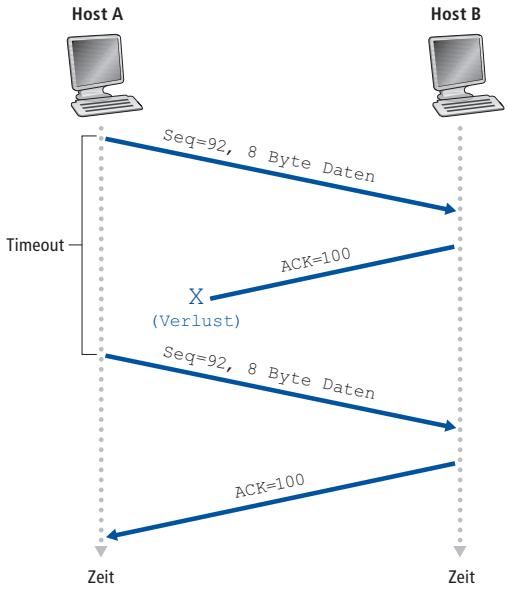

Abbildung 3.34: Erneute Übertragung aufgrund eines verloren gegangenen Acknowledgments.

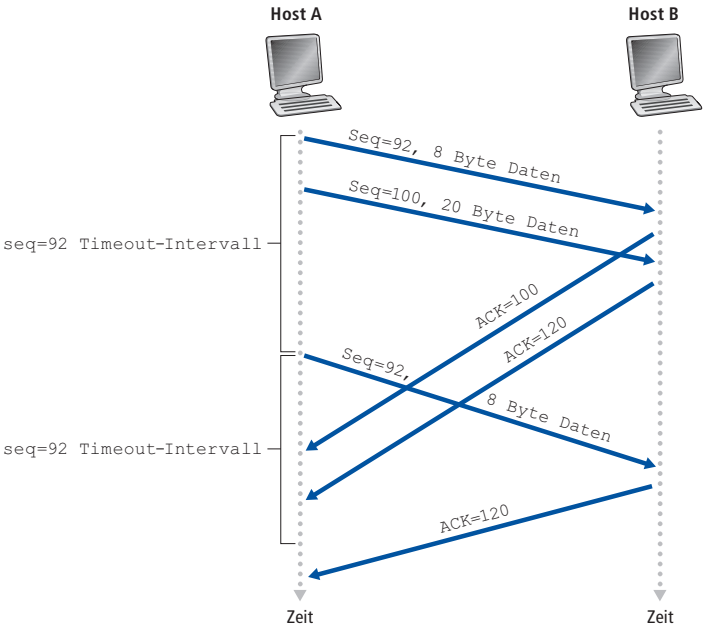

Abbildung 3.35: Segment 100 wird nicht erneut übertragen.

Nehmen Sie in einem dritten und letzten Szenario an, dass Host A die beiden Segmente genau wie im zweiten Szenario sendet. Die Bestätigung des ersten Segments geht im Netz verloren, aber gerade noch rechtzeitig vor dem Timeout-Ereignis erhält Host A eine Bestätigung mit Acknowledgment-Nummer 120. Host A weiß deshalb, dass Host B alles

bis einschließlich Byte 119 erhalten hat. Daher sendet Host A keines der beiden Segmente erneut. Dieses Szenario ist in ▶ Abbildung 3.36 erläutert.

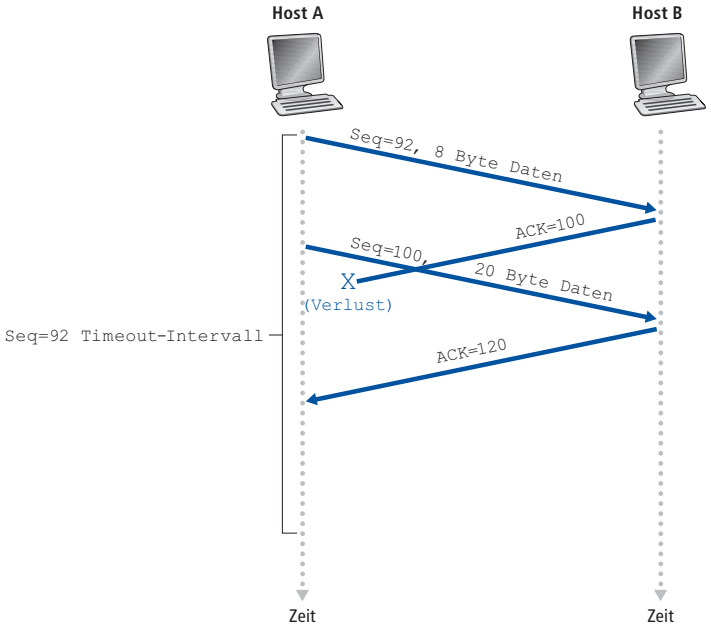

Abbildung 3.36: Das kumulative Acknowledgment verhindert die erneute Übertragung des ersten Segments.

Verdoppeln des Timeout-Intervalls

Wir diskutieren jetzt einige Modifikationen, welche die meisten TCP-Implementierungen verwenden. Die erste betrifft die Länge des Timeout-Intervalls nach dem Ablauf des Timers. Jedes Mal, wenn das Timeout-Ereignis eintrifft, überträgt TCP, wie oben beschrieben, das noch nicht bestätigte Segment mit der kleinsten Sequenznummer erneut. Aber jedes Mal, wenn TCP eine erneute Übertragung durchführt, setzt es das nächste Timeout-Intervall auf das Doppelte des vorherigen Wertes, anstatt es vom letzten *EstimatedRTT* und *DevRTT* abzuleiten (wie in Abschnitt 3.5.3 beschrieben). Nehmen Sie zum Beispiel an, dass das Timeout-Intervall, das zum ältesten noch nicht bestätigten Segment gehört, 0,75 Sekunden beträgt, sobald der Timer zum ersten Mal ausläuft. TCP überträgt dieses Segment nochmals und stellt die neue Ablaufzeit des Timers auf 1,5 Sekunden ein. Wenn der Timer 1,5 Sekunden später wieder ausläuft, überträgt TCP dieses Segment nochmals, stellt nun aber die Ablaufzeit auf 3,0 Sekunden. Auf diese Art wachsen die Intervalle nach jeder Übertragungswiederholung exponentiell an. Wenn jedoch der Timer nach einem der beiden anderen Ereignisse gestartet wird (also wenn Daten von der oberhalb liegenden Anwendung eintreffen oder wenn ein ACK empfangen wird), wird das Timeout-Intervall aufgrund der aktuellen Werte von *EstimatedRTT* und *DevRTT* bestimmt.

Diese Änderung beinhaltet eine eingeschränkte Form der Überlastkontrolle. (Umfassendere Mechanismen zur Überlastkontrolle in TCP werden in Abschnitt 3.7 diskutiert.) Der Ablauf des Timers wird am wahrscheinlichsten durch Überlast im Netz verursacht.

Das heißt, es kommen zu viele Pakete an einer (oder mehreren) Router-Warteschlange auf dem Pfad zwischen Quelle und Ziel an und werden verworfen oder übermäßig verzögert. Würden die Quellen während einer Überlastsituation ständig ihre Paketübertragungen wiederholen, könnte sich die Überlast noch verschlimmern. Stattdessen ist TCP eleganter: Jeder Sender wiederholt seine Übertragung in immer größeren Intervallen. Wir werden bei der Untersuchung von CSMA/CD in *Kapitel 5* sehen, dass Ethernet eine ähnliche Idee verwendet.

Schnelle Übertragungswiederholung

Eines der Probleme mit Übertragungswiederholungen, die durch einen Timeout ausgelöst werden, liegt darin, dass die Timeout-Periode relativ lang sein kann. Geht ein Segment verloren, zwingt diese lange Timeout-Periode den Absender dazu, die erneute Übertragung des Segments lange zu verzögern, wodurch die Ende-zu-Ende-Verzögerung größer wird. Glücklicherweise kann der Sender oft Paketverluste lange vor dem Eintreten des Timeout-Ereignisses erkennen, indem er auf sogenannte doppelte ACKs (*duplicate ACK*) achtet. Ein doppeltes ACK ist ein ACK, das ein Segment erneut bestätigt, für das der Absender schon früher eine Bestätigung erhalten hat. Um die Reaktion des Senders auf ein doppeltes ACK zu verstehen, müssen wir untersuchen, warum der Empfänger überhaupt ein doppeltes ACK sendet. ▶ Tabelle 3.2 fasst das Vorgehen des TCP-Empfängers bei der ACK-Erzeugung zusammen [RFC 5681]. Erhält ein TCP-Empfänger ein Segment mit einer Sequenznummer, die größer ist als die nächste innerhalb der Reihenfolge erwartete, dann liegt eine Lücke im Datenstrom vor – d. h. ein fehlendes Segment. Diese Lücke könnte das Ergebnis eines verlorenen Segments oder aber das Resultat von im Innern des Netzes umgeordneten Segmenten sein. Da TCP keine negativen Bestätigungen benutzt, kann der Empfänger kein explizites negatives Acknowledgment an den Sender zurücksenden.

Ereignis	Aktion des TCP-Empfängers
Ankunft des Segments in der richtigen Reihenfolge mit der erwarteten Sequenznummer. Alle Daten bis zur erwarteten Sequenznummer sind bereits bestätigt.	Heutige TCP-Implementierungen nutzen hier in der Regel sogenannte *Delayed ACKs* („verzögerte ACKs"): Wartet bis zu 500 ms auf die Ankunft eines anderen Segments in richtiger Reihenfolge. Wenn das nächste Segment nicht in diesem Zeitintervall eintrifft, wird ein ACK gesendet.
Ankunft eines Segments in der richtigen Reihenfolge mit erwarteter Sequenznummer. Ein anderes Segment in der korrekten Reihenfolge wartet auf die ACK-Übertragung.	Sendet sofort ein einzelnes kumulatives ACK, bestätigt beide in richtiger Reihenfolge eingetroffene Segmente.
Ankunft eines Segments außerhalb der Reihenfolge mit einer Sequenznummer, die größer ist als erwartet. Lücke im Bytestrom aufgetreten.	Sendet sofort ein doppeltes ACK, in dem er die Sequenznummer des nächsten erwarteten Bytes angibt.
Ankunft eines Segments, das die Lücke in den erhaltenen Daten ganz oder teilweise ausfüllt.	Sendet sofort ein ACK, vorausgesetzt, das Segment beginnt mit der Sequenznummer des nächsten erwarteten Bytes. Bestätigt alle nun lückenlos vorliegenden Bytes.

Tabelle 3.2: Empfehlungen für die Erzeugung von TCP-ACKs [RFC 5681].

Stattdessen bestätigt er einfach noch einmal das letzte in richtiger Reihenfolge eingetroffene Datenbyte, das er erhalten hat (das heißt, er generiert ein doppeltes ACK). (Beachten Sie, dass Tabelle 3.2 den Fall berücksichtigt, dass der Empfänger Segmente außerhalb der Reihenfolge nicht verwirft.)

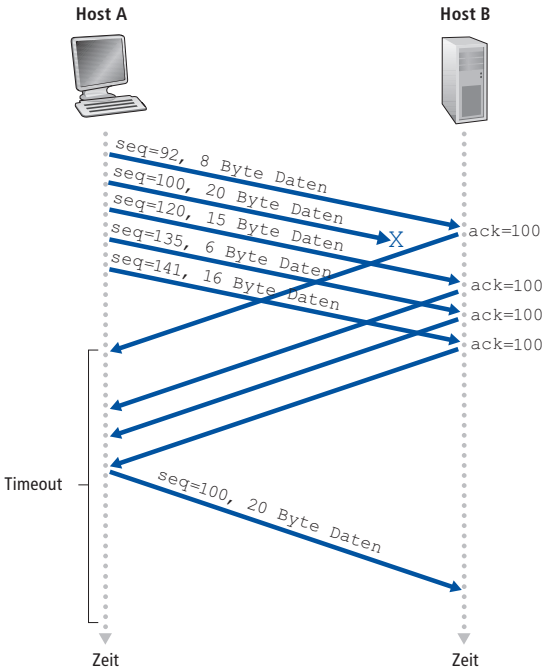

Abbildung 3.37: Fast Retransmit: erneute Übertragung des fehlenden Segments, bevor der Timer des Segments abläuft.

Weil ein Sender oft eine große Anzahl von Segmenten direkt hintereinander sendet, wird es beim Verlust eines Segments wahrscheinlich viele aufeinanderfolgende doppelte ACKs geben. Wenn der TCP-Sender drei doppelte ACKs für dieselben Daten erhält, interpretiert er dies als Zeichen dafür, dass das Segment, das auf das dreimal bestätigte Segment folgt, verloren gegangen ist. (In den Übungsaufgaben untersuchen wir die Frage, warum der Sender auf drei doppelte ACKs anstatt eines einzelnen doppelten ACK wartet.) Falls drei doppelte ACKs empfangen werden, führt der TCP-Sender eine **schnelle Übertragungswiederholung** (*Fast Retransmit*) durch [RFC 5681], wobei er das fehlende Segment nochmals überträgt, bevor der Timer des Segments abläuft. Dies wird in ▶ Abbildung 3.37 gezeigt, in der das zweite Segment verloren gegangen ist und nochmals übertragen wird, bevor sein Timer abläuft. Bei TCP mit schneller Übertragungswiederholung ersetzt der folgende Codeschnipsel das ACK-Empfangsereignis in Abbildung 3.33:

```
event: ACK empfangen, mit Wert des ACK-Felds von y
         if (y > SendBase) {
            SendBase=y
            if (es gibt noch unbestätigte Segmente)
               starte Timer
            }
```

```
else { /* ein doppeltes ACK für bereits bestätigte Segmente */
   erhöhe die Anzahl der doppelten ACKs, die für y empfangen wurden
   if (Zahl der empfangenen doppelten ACK for y==3) {
      /* schnelle Übertragungswiederholung*/
      übertrage Segment mit Sequenznummer y erneut
      }
   break;
```

Wir haben bereits angemerkt, dass viele subtile Fragen auftauchen, wenn ein Timeout-/Übertragungswiederholungsmechanismus in einem tatsächlichen Protokoll wie TCP implementiert wird. Die obigen Mechanismen, die ein Resultat von mehr als 20 Jahren Erfahrung mit TCP-Timern sind, sollten Sie davon überzeugen, dass dies wirklich zutrifft!

Go-Back-N oder Selective Repeat?

Beenden wir unsere Untersuchung der Fehlerbehebungsmechanismen von TCP mit folgender Frage: Ist TCP ein GBN- oder ein SR-Protokoll? Erinnern Sie sich daran, dass TCP-Acknowledgments kumulativ sind und korrekt empfangene, aber außerhalb der Reihenfolge eingetroffene Segmente vom Empfänger nicht individuell bestätigt werden. Folglich muss der TCP-Sender, wie in Abbildung 3.33 (siehe auch Abbildung 3.19) gezeigt, nur die kleinste Sequenznummer eines gesendeten, aber unbestätigten Bytes (*SendBase*) und die Sequenznummer des nächsten zu sendenden Bytes (*NextSeqNum*) verwalten. Aus diesem Blickwinkel erscheint TCP wie ein Protokoll im Stil von GBN. Aber es gibt einige gravierende Unterschiede zwischen TCP und Go-Back-N. Viele TCP-Implementierungen puffern korrekt empfangene Segmente, die nicht in der richtigen Reihenfolge eintreffen [Stevens 1994]. Überlegen Sie also, was geschieht, wenn der Absender eine Folge von Segmenten 1, 2, ..., N sendet und alle Segmente in der richtigen Reihenfolge und fehlerfrei beim Empfänger ankommen. Nehmen Sie weiter an, dass das Acknowledgment für Paket $n < N$ verloren geht, aber die verbleibenden $N - 1$ Acknowledgments vor ihren jeweiligen Timeouts beim Sender eintreffen. In diesem Beispiel würde GBN nicht nur Paket n, sondern auch alle folgenden Pakete $n + 1$, $n + 2$, ..., N erneut übertragen. TCP würde andererseits höchstens ein Segment erneut übertragen, nämlich Segment n. TCP würde darüber hinaus selbst Segment n nicht erneut übertragen, sofern das ACK für Segment $n + 1$ vor dem Timeout von Segment n eintrifft.

Eine vorgeschlagene Erweiterung für TCP, sogenannte **selektive Acknowledgments** [RFC 2018], erlaubt es einem TCP-Empfänger, Segmente selektiv zu bestätigen, die nicht in der korrekten Reihenfolge eintrafen, anstatt kumulativ das letzte in der richtigen Reihenfolge eingetroffene Segment zu bestätigen. Wenn TCP mit selektiven Übertragungswiederholungen kombiniert wird – und die erneute Übertragung von Segmenten, die vom Empfänger bereits selektiv bestätigt wurden, übersprungen wird –, dann sieht TCP schon sehr wie unser generisches SR-Protokoll aus. Daher wird der Zuverlässigkeitsmechanismus von TCP wahrscheinlich am besten als Hybride von GBN- und SR-Protokollen eingeordnet.

3.5.5 Flusskontrolle

Wir haben bereits beschrieben, dass die Hosts auf jeder Seite einer TCP-Verbindung einen Eingangspuffer für die Verbindung reservieren. Empfängt die TCP-Verbindung Bytes, die korrekt und in der richtigen Reihenfolge sind, so stellt sie die Daten in den Eingangspuffer. Der zugehörige Anwendungsprozess liest Daten aus diesem Puffer, aber nicht unbedingt in dem Augenblick, in dem die Daten ankommen. In der Tat kann die empfangende Anwendung mit irgendeiner anderen Aufgabe beschäftigt sein und versucht erst lange, nachdem sie angekommen sind, die Daten zu lesen. Ist die Anwendung auch noch relativ langsam beim Lesen, kann der Absender den Eingangspuffer der Verbindung sehr leicht überquellen lassen, indem er zu viele Daten zu schnell sendet.

TCP bietet einen **Flusskontrolldienst** für seine Anwendungen, um die Möglichkeit eines Überlaufens des Eingangspuffers durch den Sender auszuschließen. Flusskontrolle ist daher ein Dienst, um die Geschwindigkeit von Sender und Empfänger einander anzupassen – er vergleicht die Rate, mit der der Sender sendet, mit der Rate, mit der die empfangende Seite den Puffer ausliest. Wie früher erwähnt, kann ein TCP-Sender auch durch Überlast innerhalb des IP-Netzes gedrosselt werden. Diese Form der Geschwindigkeitsregelung beim Sender wird **Überlastkontrolle** genannt, ein Thema, das wir in den Abschnitten 3.6 und 3.7 detailliert erkunden werden. Obwohl die von Fluss- und Überlastkontrolle ergriffenen Maßnahmen ähnlich sind (die Drosselung des Senders), werden sie offensichtlich aus sehr verschiedenen Gründen ergriffen. Unglücklicherweise werfen viele Autoren die Begriffe durcheinander, doch Sie sollten aufpassen, sie klar auseinanderzuhalten. Diskutieren wir nun, wie TCP seinen Flusskontrolldienst erbringt. Damit wir den Wald trotz aller Bäume nicht aus den Augen verlieren, nehmen wir in diesem ganzen Abschnitt an, dass der TCP-Empfänger Segmente außerhalb der Reihenfolge verwirft.

TCP bietet Flusskontrolle, indem der *Sender* eine Variable pflegt, die als **Empfangsfenster** (*receive window*) bezeichnet wird. Einfach ausgedrückt vermittelt das Empfangsfenster dem Sender eine Vorstellung davon, wie viel freier Pufferplatz beim Empfänger verfügbar ist. Weil TCP im Vollduplex-Modus betrieben wird, unterhält jedes der beiden Endsysteme auf seiner Seite der Verbindung ein eigenes Empfangsfenster. Untersuchen wir das Empfangsfenster im Kontext eines Dateitransfers. Nehmen Sie an, dass Host A eine große Datei über die TCP-Verbindung an Host B sendet. Host B alloziert einen Eingangspuffer für diese Verbindung, dessen Größe durch *RcvBuffer* festgelegt wird. Ab und zu liest der Anwendungsprozess in Host B aus dem Puffer. Wir definieren die folgenden Variablen:

- *LastByteRead*: die Nummer des letzten Bytes im Datenstrom, das vom Anwendungsprozess in B aus dem Puffer gelesen wurde.
- *LastByteRcvd*: die Nummer des letzten Bytes im Datenstrom, das aus dem Netzwerk eingetroffen ist und in den Eingangspuffer von B gestellt wurde.

Weil es TCP nicht gestattet ist, den Eingangspuffer überlaufen zu lassen, muss gelten:

$$LastByteRcvd - LastByteRead \leq RcvBuffer$$

Das Empfangsfenster, bezeichnet als *rwnd*, wird im Sender an den im Puffer des Empfängers zur Verfügung stehenden Speicherplatz angepasst:

$$rwnd = RcvBuffer - [LastByteRcvd - LastByteRead]$$

Weil sich der freie Platz im Lauf der Zeit ändert, ist *rwnd* dynamisch. Die Variable *rwnd* wird in ▶ Abbildung 3.38 erläutert.

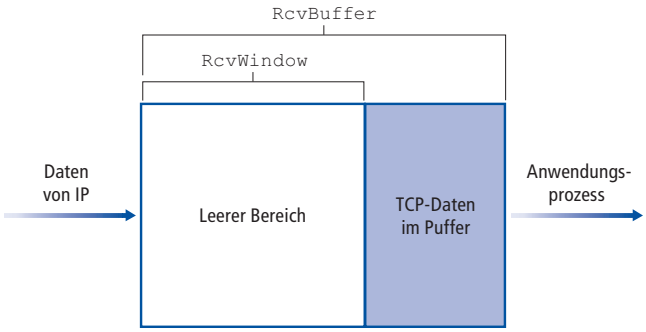

Abbildung 3.38: Das Empfangsfenster (*rwnd*) und der Eingangspuffer (*RcvBuffer*).

Wie verwendet die Verbindung die Variable *rwnd* für die Flusskontrolle? Host B sagt Host A, wie viel freien Speicherplatz er im Verbindungspuffer besitzt, indem er den aktuellen Wert von *rwnd* in das Empfangsfensterfeld jedes Segments schreibt, das er an Host A sendet. Zu Beginn setzt Host B *rwnd* = *RcvBuffer*. Beachten Sie, dass Host B, um dies zu erreichen, mehrere verbindungsspezifische Parameter im Auge behalten muss.

Host A verfolgt seinerseits zwei Variablen: *LastByteSent* und *LastByteAcked*, die offensichtliche Bedeutungen haben. Beachten Sie, dass die Differenz zwischen diesen zwei Variablen, *LastByteSent* − *LastByteAcked*, die Menge an unbestätigten Daten ist, die A in die Verbindung gesandt hat. Indem er die Menge dieser unbestätigten Daten geringer hält als den Wert von *rwnd*, kann sich Host A sicher sein, dass er nicht zu viele Daten an B sendet. Deshalb stellt Host A während der gesamten Lebensdauer der Verbindung sicher, dass

$$LastByteSent - LastByteAcked \leq rwnd$$

Dieses Vorgehen zieht ein kleines technisches Problem nach sich. Um es zu erkennen, nehmen Sie an, dass der Eingangspuffer von Host B vollläuft, sodass *rwnd* = 0. Nachdem er Host A mitgeteilt hat, dass *rwnd* = 0 ist, nehmen wir weiter an, dass B nichts an A zu senden hat. Überlegen wir nun, was geschieht. Während der Anwendungsprozess in B den Puffer leert, sendet das dortige TCP keine neuen Segmente mit neuen *rwnd*-Werten an Host A; tatsächlich sendet TCP nur dann ein Segment an Host A, wenn es entweder Daten oder ein Acknowledgment zu senden hat. Deshalb wird Host A nie darüber informiert, dass im Eingangspuffer von Host B neuer Platz zur Verfügung steht – Host A ist blockiert und kann keine weiteren Daten senden! Um dieses Problem zu lösen, verlangt die TCP-Spezifikation, dass Host A weiterhin Segmente mit einem Datenbyte sendet, wenn das Eingangsfenster von Host B null ist. Diese Segmente werden vom Empfänger bestätigt. Irgendwann beginnt der Puffer, sich zu leeren, und die Acknowledgments enthalten einen Wert für *rwnd*, der nicht null ist.

Die Website dieses Buchs bietet ein interaktives Java-Applet, das das Funktionieren des TCP-Eingangsfensters erläutert.

Nachdem wir den TCP-Flusskontrolldienst beschrieben haben, erwähnen wir hier nur kurz, dass UDP keine Flusskontrolle bietet. Um den Unterschied zu verstehen, betrachten Sie den Versand einer Serie von UDP-Segmenten von einem Prozess auf Host A an einen Prozess auf Host B. Bei einer typischen UDP-Implementierung legt UDP die Segmente in einen Puffer begrenzter Größe, der vor dem entsprechenden Socket (also der Tür zum Prozess) liegt. Der Prozess liest ein ganzes Segment auf einmal aus dem Puffer. Sollte der Prozess die Segmente nicht schnell genug aus dem Puffer auslesen, dann läuft der Puffer über und Segmente werden verworfen.

3.5.6 TCP-Verbindungsverwaltung

In diesem Unterabschnitt werfen wir einen näheren Blick darauf, wie eine TCP-Verbindung hergestellt und abgebaut wird. Obwohl dieses Thema nicht unbedingt spannend scheint, ist es wichtig, weil der Aufbau einer TCP-Verbindung signifikant zu wahrgenommenen Verzögerungen beitragen kann (zum Beispiel wenn wir im Internet surfen). Weiterhin nutzen viele der häufigsten Netzwerkangriffe – darunter die bekannten SYN-Flood-Angriffe – verwundbare Stellen im TCP-Verbindungsmanagement aus. Betrachten wir zuerst, wie eine TCP-Verbindung aufgebaut wird. Stellen Sie sich einen Prozess auf einem Host (Client) vor, der eine Verbindung mit einem anderen Prozess auf einem anderen Host (Server) initiieren will. Der Client-Anwendungsprozess informiert zuerst die Client-Instanz von TCP darüber, dass er eine Verbindung zu einem Prozess auf dem Server aufbauen will. TCP wird diese Verbindung dann auf folgende Art und Weise herstellen:

- *Schritt 1:* TCP auf der Clientseite sendet zuerst ein spezielles TCP-Segment zur Serverseite. Dieses spezielle Segment enthält keine Anwendungsschichtdaten. Im Header dieses speziellen Segments ist jedoch eines der Flag-Bits (siehe Abbildung 3.29), das SYN-Bit, auf 1 gesetzt. Deshalb wird dieses spezielle Segment als SYN-Segment bezeichnet. Außerdem wählt der Client eine zufällige initiale Sequenznummer (*client_isn*) und schreibt diese ins Sequenznummernfeld des TCP-SYN-Segments. Dieses Segment wird in ein IP-Datagramm gepackt und an den Server gesandt. Es gab beträchtliche Anstrengungen, die Wahl von *client_isn* korrekt zu randomisieren, um bestimmte Sicherheitslücken zu vermeiden [CERT 2001-09].
- *Schritt 2:* Sobald das IP-Datagramm mit dem anfänglichen TCP-SYN-Segment beim Server-Host ankommt (vorausgesetzt, dass es ankommt!), holt der Server das TCP-SYN-Segment aus dem Datagramm heraus, alloziert TCP-Puffer und Variablen für die Verbindung und sendet dem Client-TCP ein Antwortsegment zu. (Wir werden in *Kapitel 8* sehen, dass die Zuweisung dieser Puffer und Variablen vor dem Beenden des dritten Schrittes des Drei-Wege-Handshakes TCP gegen einen Denial-of-Service-Angriff verwundbar macht, der als SYN-Flooding bekannt ist.) Dieses Antwortsegment enthält ebenfalls keine Anwendungsschichtdaten. Jedoch enthält sein Segment-Header drei wichtige Informationen. Erstens ist das SYN-Bit auf 1 gesetzt. Außerdem wird das Acknowledgment-Feld des TCP-Segment-Headers auf *client_isn* + 1

gesetzt. Und zu guter Letzt wählt der Server seine eigene initiale Sequenznummer (*server_isn*) und schreibt diesen Wert ins Sequenznummernfeld des TCP-Headers. Dieses Antwortsegment besagt im Prinzip: „Ich habe dein SYN-Paket erhalten, das eine Verbindung mit deiner Anfangssequenznummer, *client_isn*, aufbauen soll. Ich stimme dem Herstellen dieser Verbindung zu. Meine eigene initiale Sequenznummer lautet *server_isn*.“ Das Antwortsegment wird als ein **SYNACK-Segment** bezeichnet.

- *Schritt 3:* Beim Erhalt des SYNACK-Segments alloziert der Client ebenfalls Puffer und Variablen für die Verbindung. Der Client-Host schickt dem Server dann ein weiteres Segment. Dieses letzte Segment bestätigt das SYNACK-Segment des Servers (der Client tut dies, indem er den Wert *server_isn* + 1 ins Acknowledgment-Feld des TCP-Segment-Headers einträgt). Das SYN-Bit wird auf null gesetzt, da die Verbindung nun hergestellt ist. Diese dritte Stufe des Drei-Wege-Handshake kann bereits Anwendungsdaten enthalten.

Sobald diese drei Schritte durchgeführt worden sind, können die Client- und Server-Hosts einander Segmente senden, die Daten enthalten. In jedem dieser künftigen Segmente wird das SYN-Bit auf null gesetzt. Beachten Sie, dass für den Verbindungsaufbau drei Pakete zwischen beiden Hosts ausgetauscht werden, wie ▶ Abbildung 3.39 zeigt. Deshalb wird dieses Verfahren für den Verbindungsaufbau oft als **Drei-Wege-Handshake** bezeichnet. Verschiedene Aspekte des TCP-Drei-Wege-Handshakes werden in den Übungsaufgaben untersucht. (Warum sind initiale Sequenznummern erforderlich? Warum braucht man einen Drei-Wege-Handshake und nicht nur einen Zwei-Wege-Handshake?) Es ist interessant anzumerken, dass auch Bergsteiger und ihre Sicherungsleute (diese befinden sich unterhalb des Kletterers und haben die Aufgabe, das Sicherungsseil des Bergsteigers zu halten) ein Kommunikationsprotokoll mit einem Drei-Wege-Handshake verwenden, der dem von TCP sehr ähnlich ist. Sie stellen auf diese Weise sicher, dass beide Seiten bereit sind, bevor der Bergsteiger mit dem Aufstieg beginnt.

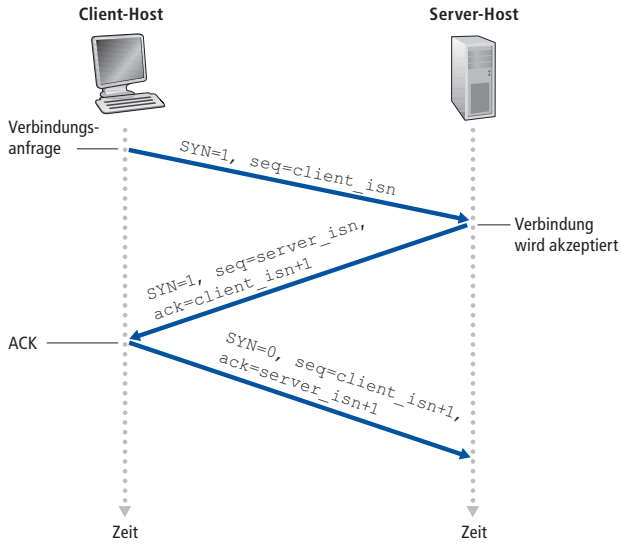

Abbildung 3.39: Segmentaustausch beim Drei-Wege-Handshake.

Alles muss irgendwann zu Ende gehen, das gilt auch für eine TCP-Verbindung. Jeder der beiden Prozesse, die an einer TCP-Verbindung beteiligt sind, kann die Verbindung beenden. Wenn eine Verbindung endet, werden die „Ressourcen" (das heißt die Puffer und Variablen) in den Hosts freigegeben. Nehmen Sie zum Beispiel an, dass der Client beschließt, die Verbindung zu beenden, wie in ▶ Abbildung 3.40 gezeigt. Der Client-Anwendungsprozess gibt einen entsprechenden Befehl. Daraufhin sendet die TCP-Instanz auf dem Client ein spezielles TCP-Segment an den Server-Prozess. Im Kopf dieses speziellen Segments ist ein spezielles Flag-Bit, das FIN-Bit, auf eins gesetzt (siehe Abbildung 3.29). Sobald der Server dieses Segment erhält, sendet er dem Client im Gegenzug ein Acknowledgment-Segment zu. Der Server sendet daraufhin sein eigenes Abschlusssegment, welches das FIN-Bit auf 1 setzt. Schließlich bestätigt der Client das Abschlusssegment des Servers. Zu diesem Zeitpunkt werden alle Ressourcen in den beiden Hosts freigegeben.

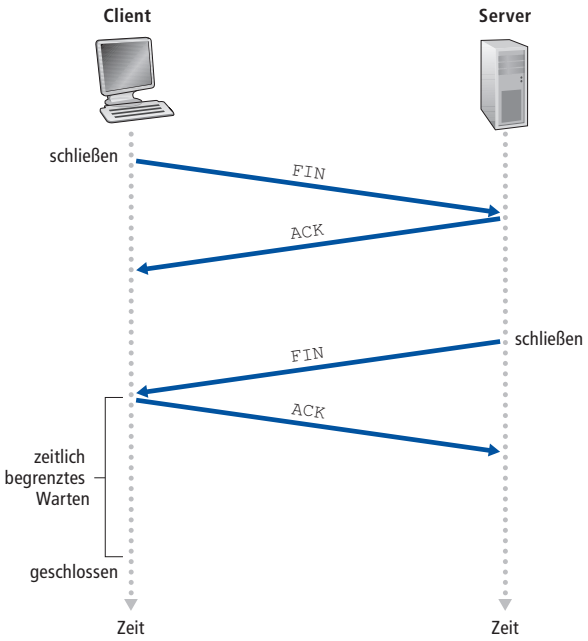

Abbildung 3.40: Schließen einer TCP-Verbindung.

Während der Lebensdauer einer TCP-Verbindung durchläuft das TCP-Protokoll, das in jedem Host läuft, verschiedene **TCP-Zustände**. ▶ Abbildung 3.41 zeigt eine typische Sequenz von TCP-Zuständen, die vom clientseitigen TCP durchlaufen werden. Es beginnt im Zustand CLOSED (geschlossen). Die Anwendung auf der Clientseite initiiert eine neue TCP-Verbindung (durch Erzeugen eines Socket-Objekts in unseren Python-Beispielen von *Kapitel 2*.) Dadurch sendet der Client ein SYN-Segment an den Server. Nachdem das SYN-Segment gesandt wurde, beginnt für das clientseitige TCP der Zustand SYN_SENT (SYN gesendet). Im SYN-SENT-Zustand wartet TCP auf ein Segment vom serverseitigen TCP, welches eine Bestätigung für das vorherige Segment des Clients beinhaltet und das SYN-Bit auf 1 setzt. Nachdem es solch ein Segment

Copyright

Daten, Texte, Design und Grafiken dieses eBooks, sowie die eventuell angebotenen eBook-Zusatzdaten sind urheberrechtlich geschützt. Dieses eBook stellen wir lediglich als **persönliche Einzelplatz-Lizenz** zur Verfügung!

Jede andere Verwendung dieses eBooks oder zugehöriger Materialien und Informationen, einschließlich

- der Reproduktion,
- der Weitergabe,
- des Weitervertriebs,
- der Platzierung im Internet, in Intranets, in Extranets,
- der Veränderung,
- des Weiterverkaufs und
- der Veröffentlichung

bedarf der **schriftlichen Genehmigung** des Verlags. Insbesondere ist die Entfernung oder Änderung des vom Verlag vergebenen Passwortschutzes ausdrücklich untersagt!
Bei Fragen zu diesem Thema wenden Sie sich bitte an: info@pearson.de

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf unseren Websites ist eine freiwillige Leistung des Verlags. **Der Rechtsweg ist ausgeschlossen.**

Hinweis

Dieses und viele weitere eBooks können Sie rund um die Uhr und legal auf unserer Website herunterladen:

http://ebooks.pearson.de