WWW.PEARSON-STUDIUM.DE
CWS
COMPANION WEBSITE

it
informatik

James F. Kurose
Keith W. Ross

Computernetzwerke

Der Top-Down-Ansatz

4., aktualisierte Auflage

ADDISON-WESLEY

PEARSON
Studium

Transportschicht

3

ÜBERBLICK

EINLEITUNG

» Zwischen der Anwendungs- und der Netzwerkschicht gelegen, ist die Transportschicht ein zentrales Element der geschichteten Netzwerkarchitektur. Sie hat die zentrale Rolle, den Anwendungsprozessen auf verschiedenen Hosts Kommunikationsdienste zu erbringen. Der pädagogische Ansatz, den wir in diesem Kapitel benutzen, wechselt zwischen der Diskussion der theoretischen Grundlagen der Transportschicht und Diskussionen, wie diese Grundlagen in vorhandenen Protokollen implementiert sind. Wie üblich widmen wir den Internetprotokollen besondere Aufmerksamkeit, insbesondere den Transportschichtprotokollen TCP und UDP.

Wir beginnen mit der Diskussion der Beziehung zwischen Transport- und Netzwerkschicht. Dies bereitet den Boden, um die erste wichtige Funktion der Transportschicht zu untersuchen – die Erweiterung der Kommunikation zwischen zwei Endsystemen, wie sie die Netzwerkschicht anbietet, hin zur Kommunikation zwischen zwei auf diesen Endsystemen ablaufenden Anwendungsschichtprozessen. Wir werden diese Funktion durch die Betrachtung des verbindungslosen Transportprotokolls UDP illustrieren.

Danach kehren wir zu den Grundlagen zurück und stellen uns einem der fundamentalsten Probleme in Computernetzwerken: Wie können zwei Kommunikationsteilnehmer sicher über ein Medium kommunizieren, das Daten verlieren oder verändern kann? Mittels immer komplizierteren (und realistischeren!) Szenarien erarbeiten wir uns eine Reihe von Techniken, die Transportprotokolle zur Lösung dieses Problems anwenden. Wir zeigen danach, wo wir diese Grundlagen in TCP, dem verbindungsorientierten Transportprotokoll des Internets, wiederfinden.

Anschließend wenden wir uns einem zweiten fundamental wichtigen Problem in Netzwerken zu – der Kontrolle der Übertragungsrate von Instanzen der Transportschicht, um Überlast im Netzwerk zu vermeiden oder abzubauen. Wir erörtern die Ursachen und Konsequenzen von Überlast sowie die gebräuchlichsten Techniken zu ihrer Kontrolle. Nachdem wir so ein grundlegendes Verständnis der Themen gewonnen haben, die hinter der Überlastkontrolle stecken, betrachten wir den von TCP benutzten Ansatz. «

3.1 Einführung und Transportschichtdienste

In den beiden vorangegangenen Kapiteln haben wir die Bedeutung der Transportschicht und der von ihr angebotenen Dienste nur gestreift. Lassen Sie uns kurz wiederholen, was wir bereits über die Transportschicht wissen.

Ein Transportschichtprotokoll ermöglicht die **logische Kommunikation** zwischen Anwendungen, die auf verschiedenen Hosts laufen. Mit *logischer Kommunikation* meinen wir, dass es aus Sicht einer Anwendung erscheint, als wären die Hosts, auf denen die Prozesse ablaufen, direkt miteinander verbunden. Tatsächlich können sich die Hosts auf entgegengesetzten Seiten dieses Planeten befinden und durch zahlreiche Router sowie ein breites Spektrum an verschiedenartigen Verbindungen miteinander verknüpft sein. Anwendungsprozesse benutzen die logische Kommunikation mittels der Transportschicht, um sich gegenseitig Nachrichten zuzusenden, ohne sich um Details der physikalischen Infrastruktur kümmern zu müssen, die diese Nachrichten übertragen. ▶Abbildung 3.1 illustriert den Begriff der logischen Kommunikation.

Wie in Abbildung 3.1 dargestellt, sind die Protokolle der Transportschicht in den Endsystemen implementiert, aber nicht in Netzwerkroutern. Auf der Seite des Senders verpackt die Transportschicht die Nachrichten, die sie von einer sendenden Anwendung empfängt, in Pakete der Transportschicht, die in der Internetterminologie als **Segmente** der Transportschicht bezeichnet werden. Dies erfolgt (wenn nötig), indem die Nachrichten der Anwendungen in kleinere Stücke aufgeteilt werden. Anschließend wird jedem Teil von der Transportschicht ein Header hinzugefügt. Dadurch entsteht das Segment. Die Transportschicht reicht dieses Segment an die Netzwerkschicht des sendenden Systems weiter. Dort wird das Segment in ein Datenpaket der Netzwerkschicht eingepackt (ein sogenanntes Datagramm) und an den Empfänger versandt. Wichtig zu erwähnen ist, dass Netzwerkrouter im Datagramm nur auf die Felder der Netzwerkschicht reagieren, d.h., sie untersuchen nicht die Felder des im Datagramm eingekapselten Transportschichtsegmentes. Auf Empfängerseite extrahiert die Netzwerkschicht das Segment der Transportschicht aus dem Datagramm und reicht das Segment an die Transportschicht weiter. Diese bearbeitet dann das eingegangene Segment und stellt die darin enthaltenen Daten der empfangenden Anwendung zur Verfügung.

Netzwerkanwendungen können mehr als ein Transportschichtprotokoll einsetzen. Das Internet benutzt z.B. zwei Protokolle – TCP und UDP. Jedes dieser Protokolle bietet den aufrufenden Anwendungen einen unterschiedlichen Satz von Transportschichtdiensten.

3.1.1 Beziehung zwischen Transport- und Netzwerkschicht

Erinnern Sie sich daran, dass die Transportschicht im Protokollstapel direkt über der Netzwerkschicht liegt. Während die Transportschicht die logische Kommunikation zwischen *Prozessen* auf unterschiedlichen Hosts ermöglicht, bietet die Netzwerkschicht die logische Kommunikation zwischen den *Hosts*. Der Unterschied mag gering sein, aber er ist wichtig. Untersuchen wir ihn mithilfe einer Analogie aus dem täglichen Leben.

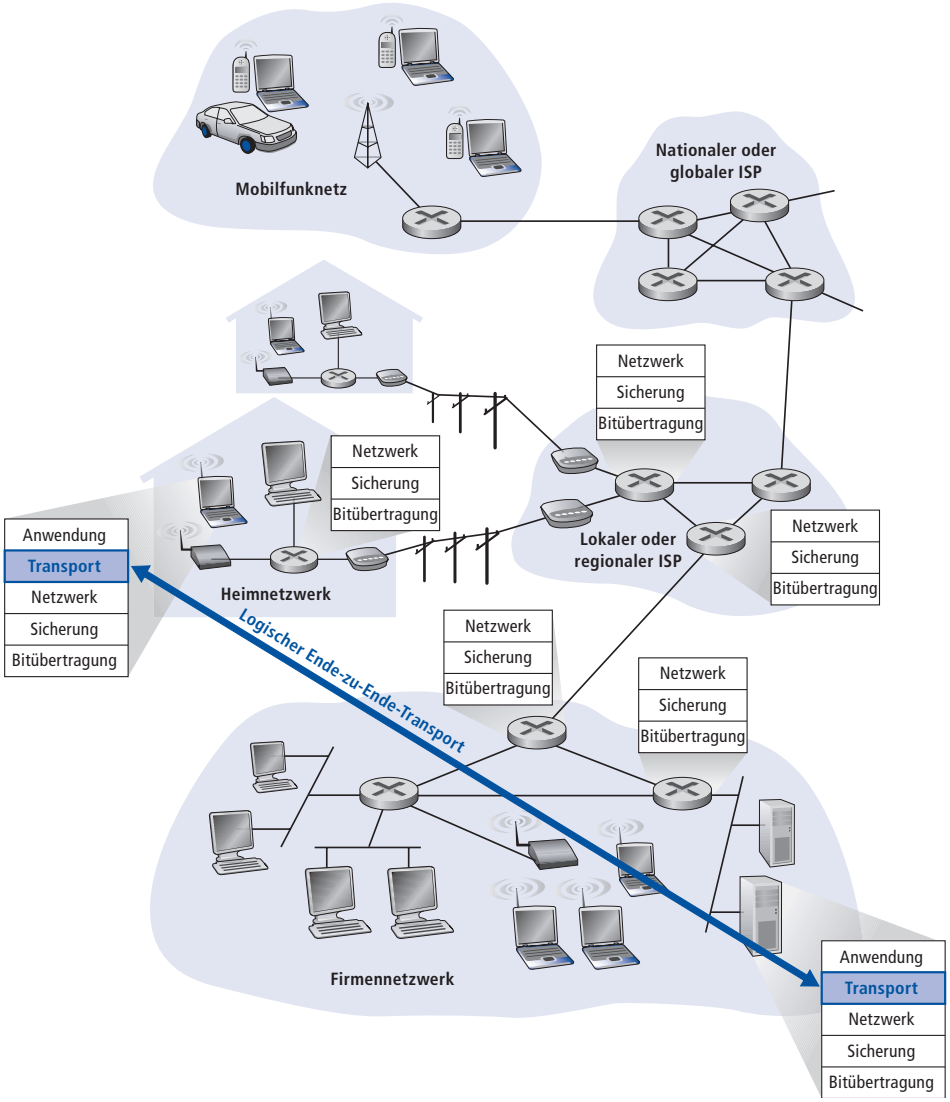


Abbildung 3.1: Die Transportschicht realisiert logische, nicht physikalische Kommunikation zwischen Anwendungsprozessen

Stellen Sie sich zwei Häuser vor, eines an der Nordsee, das andere in den Alpen. In jedem Haus leben ein Dutzend Kinder. Die Kinder an der Nordsee sind Cousins der Kinder, die in den Alpen leben. Die Kinder dieser beiden Haushalte lieben es, einander zu schreiben – jedes Kind schreibt jedem Cousin einmal in der Woche, wobei jeder Brief vom traditionellen Briefdienst der Post in einem eigenen Umschlag transportiert wird. Auf diese Weise sendet jeder Haushalt dem anderen in einer Woche 144 Briefe (die Kinder könnten eine Unmenge Geld sparen, wenn sie E-Mail hätten!). In jedem Haushalt ist ein Kind für das Sammeln und Verteilen der Post zuständig –

Anna macht dies an der Nordsee und Bill in den Alpen. Jede Woche schaut Anna bei all ihren Brüdern und Schwestern rein, sammelt die Post ein und übergibt sie einem Mitarbeiter der Post, der die Häuser täglich besucht. Treffen Briefe im Haus an der Nordsee ein, übernimmt Anna auch die Aufgabe, die Post an ihre Brüder und Schwestern zu verteilen. Bill erledigt die gleichen Arbeiten im anderen Haus.

In diesem Beispiel stellt der Postdienst die logische Kommunikation zwischen den beiden Häusern dar – der Postdienst transportiert Briefe von Haus zu Haus, nicht von Person zu Person. Anna und Bill stellen dagegen die logische Kommunikation zwischen den Cousins her – Anna und Bill sammeln die Post von ihren Geschwistern ein und liefern Post an ihre Geschwister aus. Aus Sicht der Cousins sind Anna und Bill der Postdienst, obwohl Anna und Bill ja nur ein Teil (das letzte Glied) der Transportkette bilden. Dieses Beispiel aus dem Haushalt ist eine nette Analogie, um die Beziehungen zwischen der Transportschicht und der Netzwerkschicht zu erklären:

- Nachrichten der Anwendung = Briefe in Umschlägen
- Prozesse = Cousins
- Hosts (auch Endsysteme genannt) = Häuser
- Transportschichtprotokoll = Anna und Bill
- Netzwerkschichtprotokoll = Postdienst (einschließlich des Zustellers)

Setzen wir diese Analogie fort, dann beachten Sie bitte, dass Anna und Bill ihre Aufgaben nur in ihrem jeweiligen Haushalt durchführen. Die beiden sind keineswegs damit befasst, die Post z.B. in einem Briefzentrum zu sortieren oder von einem Briefzentrum zu einem anderen zu fahren. In gleicher Weise sind Transportschichtprotokolle in den Endsystemen angesiedelt. In einem solchen Endsystem leitet das Transportprotokoll Nachrichten von den Anwendungsprozessen zum Rand des Netzwerkes (*network edge*, also an die Netzwerkschicht) weiter und umgekehrt. Das Transportprotokoll kann aber keinerlei Aussage machen, wie die Nachrichten im Inneren des Netzwerkes *(network core)* übertragen werden. Wie Abbildung 3.1 zeigt, können die dazwischengeschalteten Router Informationen weder erkennen noch reagieren sie auf Informationen, welche die Transportschicht an die Nachrichten der Anwendungen angehängt haben könnte.

Setzen wir unsere Familiensaga fort und nehmen nun für den Fall eines Urlaubs von Anna und Bill an, dass ein anderes Paar von Cousins, nennen wir sie Susanne und Harvey, für sie einspringen und das haushaltsinterne Sammeln und Verteilen von Briefen übernehmen. Unglücklicherweise für die beiden Familien führen Susanne und Harvey das Sammeln und Verteilen nicht in genau derselben Weise durch wie Anna und Bill. Da sie jünger sind, sammeln Susanne und Harvey die Briefe unregelmäßiger und verlieren gelegentlich Briefe (die manchmal vom Familienhund gefressen werden). Dadurch bietet das Paar Susanne-Harvey nicht dieselben Dienste an (genauer gesagt nicht dasselbe Dienstmodell) wie Anna und Bill. In gleicher Weise kann ein Computernetzwerk mehrere Transportprotokolle zur Verfügung stellen, die den Anwendungen jeweils andere Dienstmodelle anbieten.

Der Umfang der Dienstleistungen, die Anna und Bill anbieten, wird klarerweise von den Dienstleistungen eingeschränkt, die der Postdienst anbietet. Wenn dieser beispielsweise keine maximale Zeit sicherstellen kann, in der die Briefe von einem Haus zum anderen transportiert werden (z.B. drei Tage), dann können auch Anna und Bill keine maximale Verzögerungszeit für den Transport der Briefe zwischen den Cousins garantieren. Auf ähnliche Weise werden die Dienste eines Transportprotokolls oft vom Dienstmodell der darunterliegenden Netzwerkschicht eingeschränkt. Falls das Protokoll der Netzwerkschicht keine Verzögerungs- oder Bandbreitengarantien für die zwischen den Hosts ausgetauschten Segmente geben kann, kann auch das Transportschichtprotokoll den zwischen den Prozessen ausgetauschten Nachrichten weder Verzögerung noch Bandbreiten garantieren.

Allerdings *können* bestimmte Dienstleistungen auch dann vom Transportprotokoll angeboten werden, wenn das darunterliegende Netzwerkprotokoll die entsprechenden Dienstleistungen in der Netzwerkschicht nicht zur Verfügung stellt. Wie wir in diesem Kapitel sehen werden, kann z.B. ein Transportprotokoll einer Anwendung auch dann zuverlässige Datentransfers anbieten, wenn die zugrunde liegende Netzwerkschicht unzuverlässig ist, also selbst dann, wenn die Netzwerkschicht Datenpakete verliert, durcheinanderwürfelt oder dupliziert. Als weiteres Beispiel (das wir in Kapitel 8 untersuchen, wenn wir die Sicherheit von Netzwerken diskutieren) könnte ein Transportprotokoll selbst dann mittels Datenverschlüsselung verhindern, dass Eindringlinge die Nachrichten lesen, wenn die Netzwerkschicht die Vertraulichkeit der Transportschichtsegmente nicht gewährleisten kann.

3.1.2 Überblick über die Transportschicht im Internet

Erinnern Sie sich bitte daran, dass das Internet, allgemeiner ein TCP/IP-Netzwerk, der Anwendungsschicht zwei verschiedene Transportschichtprotokolle anbietet. Eines dieser Protokolle ist **UDP** (User Datagram Protocol), das den aufrufenden Anwendungen einen unzuverlässigen, verbindungslosen Dienst zur Verfügung stellt. Das zweite Protokoll ist **TCP** (Transmission Control Protocol), welches den aufrufenden Anwendungen einen zuverlässigen, verbindungsorientierten Dienst anbietet. Wenn Anwendungsentwickler eine Netzwerkanwendung entwerfen, müssen sie sich auf eines dieser beiden Protokolle festlegen. Wie wir in den Abschnitten 2.7 und 2.8 gesehen haben, wählt der Entwickler beim Erstellen der Sockets zwischen UDP und TCP aus.

Um die Terminologie zu vereinfachen, werden wir im Kontext des Internets das Datenpaket der Transportschicht immer als *Segment* bezeichnen. Wir weisen allerdings darauf hin, dass die Internetliteratur (beispielsweise die RFCs) die Transportschichtpakete von TCP als Segmente, die Pakete von UDP dagegen als *Datagramm* bezeichnet. Dieselbe Internetliteratur benutzt den Begriff *Datagramm* auch für die Pakete der Netzwerkschicht! In einer Einführung in Computernetzwerke, wie sie dieses Buch darstellt, ist der Begriff Segment sowohl für TCP- als auch für UDP-Pakete weniger verwirrend und wir verwenden den Begriff *Datagramm* nur für Pakete der Netzwerkschicht.

Bevor wir mit unserer kurzen Einführung in UDP und TCP fortfahren, sind einige Worte über die Netzwerkschicht des Internets nützlich. (Die Netzwerkschicht als solche wird in Kapitel 4 detailliert untersucht.) Das Netzwerkschichtprotokoll des Internets hat einen eigenen Namen: IP für Internet Protocol. IP ermöglicht die logische Kommunikation zwischen Hosts. Das IP-Dienstmodell ist ein sogenannter **Best-Effort-Dienst**. Das bedeutet, IP tut „sein Bestes“, um die Segmente zwischen den kommunizierenden Hosts zu transportieren, aber *es gibt keinerlei Garantien*. Insbesondere garantiert es nicht die Zustellung der Segmente, es garantiert nicht die geordnete Auslieferung der Segmente und es garantiert nicht die Integrität der Daten innerhalb der Segmente. Aus diesen Gründen wird IP als **unzuverlässiger Dienst** bezeichnet. Wir erwähnen an dieser Stelle auch, dass jeder Host mindestens eine Adresse der Netzwerkschicht besitzt, die sogenannte IP-Adresse. Wir werden die Adressierung in IP in Kapitel 4 genauer untersuchen. In diesem Kapitel müssen wir nur im Hinterkopf behalten, dass *jeder Host eine IP-Adresse besitzt*.

Nachdem wir einen kurzen Blick auf das IP-Dienstmodell geworfen haben, wollen wir jetzt die von UDP und TCP angebotenen Dienstmodelle zusammenfassen. Die wichtigste Aufgabe von UDP und TCP ist das Erweitern des IP-Zustelldienstes zwischen zwei Endsystemen auf einen Zustelldienst zwischen zwei Prozessen, die auf den Endsystemen laufen. Die Erweiterung der Host-zu-Host-Zustellung auf Prozess-zu-Prozess-Zustellung wird als **Transportschicht-Multiplexing** und **-Demultiplexing** bezeichnet. Wir werden das Transportschicht-Multiplexing und -Demultiplexing im nächsten Abschnitt diskutieren. UDP und TCP bieten auch Integritätsüberprüfungen, indem sie in die Header der Segmente Felder für die Fehlererkennung einfügen. Diese beiden minimalen Transportschichtdienste – Prozess-zu-Prozess-Kommunikation und -Fehlererkennung – sind die beiden einzigen Dienste, die UDP bietet! Insbesondere ist UDP wie IP ein unzuverlässiger Dienst – es garantiert nicht, dass von einem Prozess verschickte Daten intakt (oder überhaupt!) den Zielprozess erreichen. UDP wird im Detail in Abschnitt 3.3 erörtert.

TCP bietet den Anwendungen andererseits mehrere zusätzliche Dienste an. Zuallererst realisiert es **zuverlässigen Datentransfer**. Mithilfe von Flusskontrolle, Sequenznummern, Acknowledgments und Timern (Techniken, die wir im Detail in diesem Kapitel erkunden werden) stellt TCP sicher, dass die Daten vom sendenden Prozess korrekt und in der richtigen Reihenfolge an den empfangenden Prozess geliefert werden. TCP wandelt auf diese Art den unzuverlässigen Dienst zwischen Endsystemen (der durch IP erbracht wird) in einen zuverlässigen Datentransportdienst zwischen Prozessen um. TCP bietet auch **Überlastkontrolle** an. Überlastkontrolle ist weniger ein Dienst für eine aufrufende Anwendung, als vielmehr ein Dienst für das Internet als Ganzes, ein Dienst für das Allgemeinwohl. Salopp gesagt hindert die TCP-Überlastkontrolle eine TCP-Verbindung daran, die Verbindungen und Router zwischen den kommunizierenden Hosts mit einem unverhältnismäßigen Verkehrsaufkommen zu überschwemmen. TCP ist bestrebt, jeder Verbindung, die einen überlasteten Link durchquert, einen gleich großen Anteil der Verbindungsbandbreite zuzuteilen. Dies erfolgt, indem die Rate, mit der die sendende Seite von TCP-Verbindungen Verkehr

ins Netz senden kann, reguliert wird. Der von UDP verursachte Verkehr ist im Gegensatz dazu ungeregelt. Eine Anwendung, die auf UDP basiert, darf so lange und so schnell senden, wie sie will.

Ein Protokoll, das zuverlässigen Datentransfer und Überlastkontrolle bietet, ist notwendigerweise komplex. Wir werden mehrere Abschnitte brauchen, um die Grundlagen des zuverlässigen Datentransfers und der Überlastkontrolle zu erörtern. Weitere Abschnitte werden wir benötigen, um TCP selbst zu diskutieren. Diesen Themen sind die Abschnitte 3.4 bis 3.8 gewidmet. Der diesem Kapitel zugrunde liegende didaktische Ansatz wechselt zwischen den Grundlagen und dem TCP-Protokoll ab. Zum Beispiel erörtern wir zuerst zuverlässigen Datentransfer in einem allgemeinen Kontext und diskutieren anschließend, wie TCP im Speziellen zuverlässigen Datentransfer anbietet. Auf ähnliche Weise werden wir zuerst Überlastkontrolle allgemein betrachten und danach diskutieren, wie TCP sie konkret durchführt. Aber bevor wir uns diesen Themen zuwenden, wollen wir zunächst das Transportschicht-Multiplexing und -Demultiplexing betrachten.

3.2 Multiplexing und Demultiplexing

In diesem Abschnitt erörtern wir Transportschicht-Multiplexing und -Demultiplexing, also die Erweiterung des von der Netzwerkschicht angebotenen Host-zu-Host-Zustelldienstes zu einem Prozess-zu-Prozess-Zustelldienst für Anwendungen, die auf den Hosts laufen. Um die Diskussion konkret zu halten, erörtern wir diesen grundlegenden Transportschichtdienst im Kontext des Internets. Wir betonen jedoch, dass ein Multiplexing/Demultiplexing-Dienst für alle Computernetzwerke erforderlich ist.

Am Zielhost erhält die Transportschicht Segmente von der direkt darunterliegenden Netzwerkschicht. Die Transportschicht hat die Aufgabe, die Daten in diesen Segmenten an die entsprechenden, im Host laufenden, Anwendungsprozesse zu liefern. Werfen wir einen Blick auf ein Beispiel. Nehmen Sie an, dass Sie vor Ihrem Computer sitzen und Webseiten herunterladen, während Sie eine FTP-Sitzung und zwei Telnet-Sitzungen geöffnet haben. Sie lassen daher vier Anwendungsprozesse laufen – zwei Telnet-Prozesse, einen FTP-Prozess und einen HTTP-Prozess. Sobald die Transportschicht in Ihrem Computer Daten von der darunterliegenden Netzwerkschicht erhält, muss sie die erhaltenen Daten zu einem dieser vier Prozesse lenken. Prüfen wir nun, wie dies geschieht.

Erinnern Sie sich zuerst an die Abschnitte 2.7 und 2.8. Dort haben Sie erfahren, dass ein Prozess (als Teil einer Netzwerkanwendung) einen oder mehrere Sockets haben kann: Türen, durch die Daten vom Netzwerk zum Prozess und umgekehrt vom Prozess zum Netzwerk laufen. Wie in ▶Abbildung 3.2 gezeigt, liefert die Transportschicht im empfangenden Host die Daten nicht direkt an einen Prozess, sondern vielmehr an einen dazwischenliegenden Socket. Weil es zu jedem beliebigen Zeitpunkt mehr als einen Socket im empfangenden Host geben kann, hat jeder Socket eine eindeutige Kennzeichnung. Das Format der Kennzeichnung hängt, wie wir in Kürze diskutieren werden, davon ab, ob der Socket ein UDP- oder ein TCP-Socket ist.

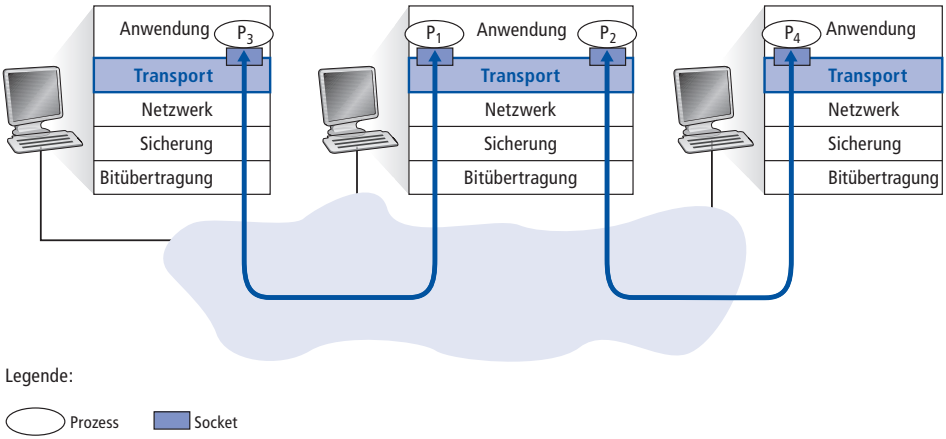


Abbildung 3.2: Transportschicht-Multiplexing/Demultiplexing

Überlegen wir nun, wie ein empfangender Host ein eingehendes Transportschichtsegment an den entsprechenden Socket weiterleitet. Jedes Transportschichtsegment enthält zu diesem Zweck einen Satz von Feldern. Auf der Empfängerseite betrachtet die Transportschicht diese Felder, um den empfangenden Socket zu identifizieren, und leitet dann das Segment an diesen Socket. Diese Aufgabe, das Abliefern der Daten am richtigen Socket durch die Transportschicht, wird **Demultiplexing** genannt. Die Aufgabe, Datenblöcke beim Quellhost von verschiedenen Sockets zu sammeln, jeden Block mit Header-Informationen zu verkapseln (die später beim Demultiplexing benötigt werden), wodurch Segmente entstehen, und das Weiterleiten der Segmente an die Netzwerkschicht wird als **Multiplexing** bezeichnet. Beachten Sie, dass die Transportschicht im mittleren Host in Abbildung 3.2 Segmente demultiplexen muss, die von der darunterliegenden Netzwerkschicht kommen, um sie einem der beiden darüber befindlichen Prozesse P1 oder P2 zukommen zu lassen. Das geschieht, indem die ankommenden Datensegmente zu dem Socket des entsprechenden Prozesses gelenkt werden. Die Transportschicht im mittleren Host muss auch ausgehende Daten von diesen Sockets einsammeln, Transportschichtsegmente formen und diese an die darunterliegende Netzwerkschicht weitergeben. Obwohl wir Multiplexing und Demultiplexing im Kontext der Internet-Transportprotokolle eingeführt haben, muss man sich verdeutlichen, dass sie immer dann benötigt werden, wenn ein einzelnes Protokoll in einer Schicht (der Transportschicht oder einer anderen) von mehreren Protokollen der nächsthöheren Schicht verwendet wird.

Um das Demultiplexing zu verdeutlichen, erinnern wir an die Analogie des Haushaltes im vorherigen Abschnitt. Jedes der Kinder wird durch seinen Namen identifiziert. Wenn Bill einen Stapel Post vom Briefträger erhält, führt er die Operation des Demultiplexing durch, indem er nachsieht, an wen die Briefe adressiert sind, und dann die Briefe per Hand an seine Brüder und Schwestern ausliefert. Anna führt eine Multiplex-Operation durch, indem sie Briefe bei ihren Brüdern und Schwestern abholt und die gesammelte Post an den Briefträger weiterreicht.

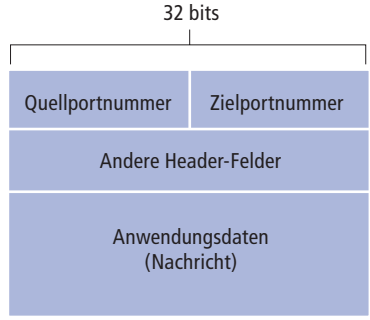


Abbildung 3.3: Quell- und Zielportnummerfelder in einem Transportschichtsegment

Nun, da wir die Rollen des Transportschicht-Multiplexing und -Demultiplexing verstehen, wollen wir überprüfen, wie dies in einem Host tatsächlich durchgeführt wird. Anhand der obigen Diskussion wissen wir, dass das Transportschicht-Multiplexing es erfordert, dass (1) der Socket eine eindeutige Kennzeichnung hat und (2) jedes Segment spezielle Felder besitzt, welche den Socket festlegen, an den das Segment geliefert werden muss. Diese in ▶ Abbildung 3.3 gezeigten speziellen Felder sind das **Portnummerfeld der Quelle** *(source port number field)* und das **Portnummerfeld des Ziels** *(destination port number field)*. (Die UDP- und TCP-Segmente haben zudem noch andere Felder, die in den folgenden Abschnitten dieses Kapitels diskutiert werden.) Jede Portnummer ist eine 16 Bit-Zahl, die zwischen 0 und 65535 liegt. Die Portnummern zwischen 0 und 1023 werden als **wohlbekannte Portnummern** *(well-known port numbers)* bezeichnet und sind festgelegt, d.h., sie sind für die Benutzung durch weit verbreitete Anwendungsprotokolle reserviert, etwa HTTP (das Portnummer 80 benutzt) und FTP (das Portnummer 21 benutzt). Die wohlbekannten Portnummern sind in RFC 1700 aufgelistet und werden unter *http://www.iana.org* aktualisiert [RFC 3232]. Wenn wir eine neue Anwendung entwickeln (wie die der in Abschnitt 2.7 oder 2.8 entwickelten Anwendungen), müssen wir der Anwendung eine Portnummer zuteilen.

Es sollte nun klar sein, wie die Transportschicht den Demultiplexing-Dienst durchführen *könnte*: Jedem Socket im Host könnte eine Portnummer zugeteilt werden. Sobald ein Segment beim Host ankommt, prüft die Transportschicht die Zielportnummer im Segment und leitet das Segment zu dem entsprechenden Socket. Die Daten des Segmentes gehen dann durch den Socket in den dazugehörigen Prozess. Wie wir noch sehen werden, beschreibt dies grundsätzlich die Arbeitsweise von UDP. Wir werden aber auch erkennen, dass Multiplexing/Demultiplexing in TCP noch etwas verzwickter ist.

Verbindungsloses Multiplexing und Demultiplexing

Erinnern Sie sich aus Abschnitt 2.8 daran, dass ein in einem Host ausgeführtes Java-Programm durch die Zeile

```
DatagramSocket mySocket = new DatagramSocket();
```

einen UDP-Socket erzeugen kann. Wird ein UDP-Socket auf diese Weise geschaffen, teilt die Transportschicht dem Socket automatisch eine Portnummer zu. Insbesondere wählt die Transportschicht eine Portnummer aus dem Bereich 1024 bis 65535, die gegenwärtig nicht von irgendeinem anderen UDP-Port des Hosts verwendet wird. Alternativ könnte ein Java-Programm ein Socket mit folgender Zeile erzeugen:

```
DatagramSocket mySocket = new DatagramSocket(19157);
```

In diesem Fall legt die Anwendung einen bestimmten Port für das UDP-Socket fest – nämlich 19157. Wenn der Anwendungsentwickler, der den Code schreibt, die Server-Seite eines „wohlbekannten Protokolls“ entwickelt, dann müsste der Entwickler die entsprechende wohlbekannte Portnummer zuteilen. Die Client-Seite der Anwendung lässt normalerweise zu, dass die Transportschicht die Portnummer automatisch (und transparent) zuteilt, während die Server-Seite der Anwendung eine ganz bestimmte Portnummer festlegt.

Mit Portnummern, die UDP-Sockets zugeteilt sind, können wir jetzt UDP-Multiplexing/Demultiplexing genau beschreiben. Nehmen Sie einen Prozess auf Host A an, mit UDP-Port 19157, der einen Block Anwendungsdaten an einen Prozess mit UDP-Port 46428 auf Host B senden will. Die Transportschicht des Hosts A erzeugt ein Transportschichtsegment, das die Anwendungsdaten, die Portnummer der Quelle (19157), die Portnummer des Ziels (46428) und zwei andere Werte enthält (die wir später kennenlernen werden; für die gegenwärtige Diskussion sind sie unwesentlich). Die Transportschicht reicht dann das entstehende Segment an die Netzwerkschicht weiter. Die Netzwerkschicht kapselt das Segment in einem IP-Datagramm ein und unternimmt einen Best-effort-Versuch, das Segment beim empfangenden Host abzuliefern. Wenn das Segment beim empfangenden Host B ankommt, liest die Transportschicht des empfangenden Hosts die Zielportnummer im Segment (46428) und liefert das Segment an den Socket, der durch Port 46428 identifiziert wird. Beachten Sie, dass auf Host B mehrere Prozesse laufen können, die jeweils ihren eigenen UDP-Socket mit zugehöriger Portnummer besitzen. Wann immer UDP-Segmente aus dem Netz ankommen, leitet (demultiplext) Host B sie zum entsprechenden Socket, indem er die Zielportnummer des Segmentes verwendet.

Es ist wichtig anzumerken, dass ein UDP-Socket vollständig durch das Paar Ziel-IP-Adresse und Zielportnummer identifiziert wird. Daher werden zwei UDP-Segmente, die unterschiedliche Quell-IP-Adressen und/oder unterschiedliche Quellportnummern, aber dieselbe Ziel-IP-Adresse und Zielportnummer aufweisen, durch dasselbe Zielsocket an denselben Zielprozess geleitet.

Sie fragen sich jetzt vielleicht, welchen Zweck die Portnummer der Quelle hat. Wie in ▶Abbildung 3.4 dargestellt, dient die Portnummer der Quelle im Segment, das A an B schickt, als Teil der „Rücksendeadresse“ – falls B ein Segment an A zurücksenden will, erhält dieses als Zielportnummer den Wert aus der Quellportnummer des ursprünglichen Segmentes. (Die vollständige Rücksendeadresse setzt sich aus der IP-Adresse von A und der Quellportnummer zusammen.) Als Beispiel erinnern Sie sich an das UDP-

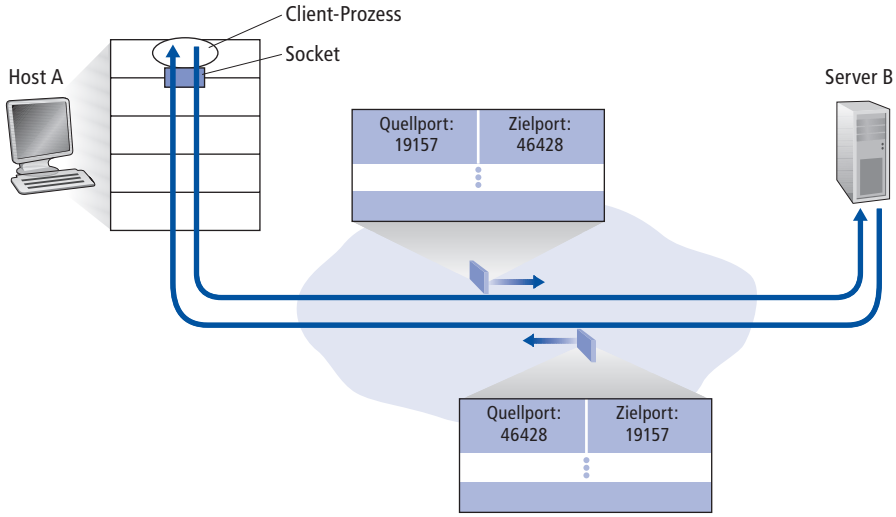


Abbildung 3.4: Die Inversion von Quell- und Zielportnummern

Server-Programm, das wir in Abschnitt 2.8 entwickelt haben. In UDPServer.java verwendet der Server eine Methode, um die Quellportnummer aus dem Segment auszulesen, das er vom Client erhält. Er sendet dem Client dann ein neues Segment zu, in dem die extrahierte Quellportnummer als Zielportnummer des neuen Segmentes dient.

Verbindungsorientiertes Multiplexing und Demultiplexing

Um TCP-Demultiplexing zu verstehen, müssen wir einen genaueren Blick auf TCP-Sockets und den Aufbau von TCP-Verbindungen werfen. Einer der feinen Unterschiede zwischen einem TCP-Socket und einem UDP-Socket besteht darin, dass ein TCP-Socket durch ein Viertupel identifiziert wird: Quell-IP-Adresse, Quellportnummer, Ziel-IP-Adresse, Zielportnummer. Wenn daher ein TCP-Segment aus dem Netz bei einem Host ankommt, nutzt der Host alle vier Werte, um das Segment zum richtigen Socket zu leiten (demultiplexen). Im Gegensatz zu UDP werden insbesondere zwei ankommende TCP-Segmente mit unterschiedlicher Quell-IP-Adresse oder Quellportnummer an zwei verschiedene Sockets weitergeleitet (mit Ausnahme eines TCP-Segmentes, das die ursprüngliche Verbindungsaufbauanfrage enthält). Um noch mehr Einblick zu erhalten, erinnern wir uns an das TCP-Client-Server-Programmierbeispiel aus Abschnitt 2.7:

- Die TCP-Server-Anwendung hat einen Eingangs-Socket, der auf Anfragen zum Verbindungsaufbau von TCP-Clients auf Portnummer 6789 wartet (Abbildung 2.31).
- Der TCP-Client erzeugt ein Segment für den Verbindungsaufbau durch die Zeile `Socket clientSocket = new Socket("serverHostName", 6789);`
- Eine Verbindungsaufbauanfrage ist nichts weiter als ein TCP-Segment mit der Zielportnummer 6789 und einem speziellen Verbindungsaufbau-Bit, das im TCP-Header gesetzt ist (den wir in Abschnitt 3.5 erörtern werden). Das Segment ent-

Fokus Sicherheit

Port-Scanning

Wie wir gesehen haben, wartet ein Server-Prozess geduldig an einem offenen Port auf die Kontaktaufnahme durch einen entfernten Client. Einige Ports werden für wohlbekannte Anwendungen (z.B. FTP, DNS und SMTP-Server) reserviert. Andere Ports werden üblicherweise von populären Anwendungen belegt (z.B. wartet der Microsoft 2000 SQL Server auf Requests auf dem UDP-Port 1434). Wenn wir daher feststellen können, dass ein Port auf einem Host geöffnet ist, sind wir in der Lage, von diesem Port Rückschlüsse auf bestimmte Anwendungen zu ziehen, die auf dem Host laufen. Das ist für Systemadministratoren sehr angenehm, wenn sie wissen möchten, welche Anwendungen auf den Hosts in ihrem Netzwerk laufen. Aber auch Angreifer, die ein Netzwerk ausspionieren, wollen wissen, welche Ports auf einem Zielhost geöffnet sind. Wird ein Host entdeckt, auf dem eine Anwendung mit einem bekannten Sicherheitsproblem läuft, dann ist dieser Host sturmreif (z.B. konnte bei einem SQL-Server, der auf Port 1434 wartete, ein Pufferüberlauf stattfinden, wodurch ein entfernter Benutzer beliebigen Code auf dem verwundbaren Host ausführen konnte – diese Sicherheitslücke wurde vom Slammer-Wurm ausgenutzt [CERT 2003–04]).

Es ist relativ leicht festzustellen, welche Anwendungen auf welche Ports warten. Es gibt in der Tat zahlreiche Public Domain-Programme, sogenannte Port-Scanner, die genau das machen. Das vielleicht am weitesten verbreitete Programm ist nmap, das unter *http://insecure.org/nmap* frei verfügbar und in den meisten Linux-Distributionen enthalten ist. Für TCP scannt nmap sequenziell die Ports und sucht nach Ports, die TCP-Verbindungen akzeptieren. Für UDP scannt nmap ebenfalls nacheinander die Ports ab und sucht nach UDP-Ports, die auf übermittelte UDP-Segmente warten. In beiden Fällen gibt nmap eine Liste der geöffneten, geschlossenen oder unerreichbaren Ports aus. Ein Host, auf dem nmap läuft, kann versuchen, jeden beliebigen Zielhost im gesamten Internet zu scannen. Wir werden nmap in Abschnitt 3.5.6 bei der Verwaltung von TCP-Verbindungen erneut betrachten.

hält auch eine Quellportnummer, die vom Client gewählt wurde. Die obige Zeile erzeugt auch einen TCP-Socket für den Client-Prozess, durch den Daten in den und aus dem Client-Prozess gelangen können.

- Sobald das Host-Betriebssystem des Computers, auf dem der Server-Prozess läuft, das ankommende Segment mit der Verbindungsaufbauanfrage für Zielportnummer 6789 erhält, macht es den Server-Prozess auf Portnummer 6789 ausfindig, der auf eine Verbindung wartet. Der Server-Prozess erzeugt dann einen neuen Socket durch die Zeile `Socket connectionSocket = welcomeSocket.accept();`.

Die Transportschicht auf dem Server merkt sich die folgenden vier Werte aus dem Verbindungsaufbausegment: (1) die Quellportnummer, (2) die IP-Adresse des Quellsystems, (3) die Zielportnummer und (4) ihre eigene IP-Adresse. Der neu erzeugte Ver-

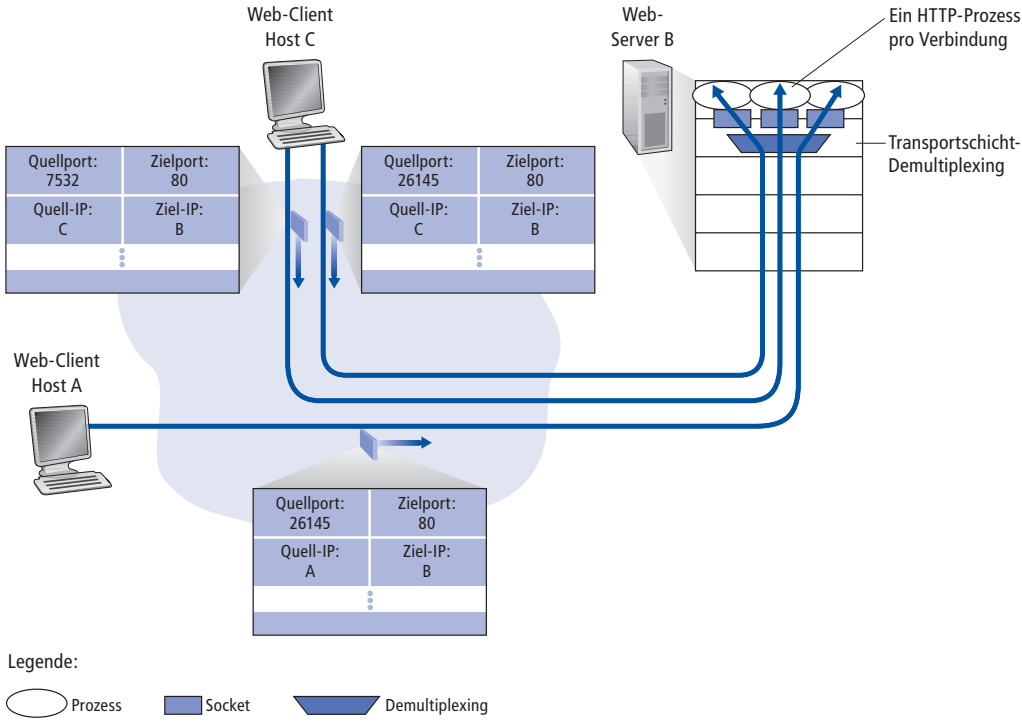


Abbildung 3.5: Zwei Clients, welche dieselbe Zielportnummer (80) benutzen, um mit denselben Server-Anwendungen zu kommunizieren

bindungs-Socket wird durch diese vier Werte identifiziert. Alle später eintreffenden Segmente, deren Quellport, Quell-IP-Adresse, Zielport und Ziel-IP-Adresse mit diesen vier Werten übereinstimmen, werden auf diesen Socket geleitet. Über die nun bestehende TCP-Verbindung können Client und Server Daten austauschen.

Der Serverhost kann gleichzeitig viele TCP-Sockets verwalten, wobei jeder Socket einem Prozess zugeordnet ist und jeder durch sein eigenes Viertupel identifiziert wird. Wenn ein TCP-Segment beim Host ankommt, werden alle vier Felder (Quell-IP-Adresse, Quellport, Ziel-IP-Adresse, Zielport) verwendet, um das Segment zum entsprechenden Socket zu leiten (zu demultiplexen).

Dies zeigt auch ▶Abbildung 3.5, in der Host C zwei HTTP-Sitzungen mit Server B aufbaut, während Host A eine HTTP-Sitzung mit B hat. Die Hosts A und C und der Server B haben jeweils ihre eigenen, eindeutigen IP-Adressen – A, C und B. Host C weist seinen beiden HTTP-Verbindungen zwei verschiedene Quellportnummern zu (26145 und 7532). Weil Host A seine Quellportnummern unabhängig von C wählt, könnte er seiner HTTP-Verbindung auch den Quellport 26145 zuweisen. Das stellt kein Problem dar, denn Server B ist immer noch in der Lage, die beiden Verbindungen korrekt zu demultiplexen, weil sie zwar dieselbe Quellportnummer besitzen, aber die beiden Verbindungen verschiedene Quell-IP-Adressen haben.

Webserver und TCP

Bevor wir diese Diskussion abschließen, sind einige zusätzliche Worte über Webserver und wie sie Portnummern verwenden lehrreich. Stellen Sie sich einen Host vor, auf dem ein Webserver, etwa ein Apache-Webserver, auf Port 80 läuft. Senden Clients (zum Beispiel Browser) Segmente an den Server, dann haben *alle* Segmente den Zielport 80. Insbesondere haben sowohl die anfänglichen Verbindungsaufbausegmente als auch die Segmente, die HTTP-Request-Nachrichten enthalten, den Zielport 80. Wie wir gerade gesehen haben, unterscheidet der Server die Segmente der verschiedenen Clients anhand der Quell-IP-Adressen und Quellportnummern.

Abbildung 3.5 stellt einen Webserver dar, der einen neuen Prozess für jede Verbindung erzeugt. Wie Abbildung 3.5 zeigt, hat jeder dieser Prozesse seinen eigenen Verbindungs-Socket, durch den die HTTP-Requests ankommen und die HTTP-Replys abgesandt werden. Wir wollen nicht unerwähnt lassen, dass nicht immer eine Eineindeutigkeit zwischen Verbindungs-Sockets und Prozessen besteht. Tatsächlich verwenden heutige Hochleistungs-Webserver nur einen Prozess und erstellen für jede neue Verbindung zu einem Client einen neuen Thread. (Einen Thread kann man sich als leichtgewichtigen Teilprozess vorstellen.) Wenn Sie die erste Programmieraufgabe in Kapitel 2 bearbeitet haben, dann haben Sie einen Webserver erstellt, der genau das macht. Ein solcher Server kann zu jeder beliebigen Zeit viele Verbindungs-Sockets haben, die alle mit demselben Prozess verbunden sind.

Wenn Client und Server persistentes HTTP verwenden, dann tauschen Client und Server über denselben Server-Socket HTTP-Nachrichten aus, solange die persistente Verbindung besteht. Verwenden jedoch Client und Server nichtpersistentes HTTP, dann wird für jedes Request/Response-Paar eine neue TCP-Verbindung geöffnet und geschlossen. Gleichermaßen wird für jedes Request/Response-Paar ein Socket geöffnet und später geschlossen. Dieses häufige Erzeugen und Schließen von Sockets kann die Leistung eines stark belasteten Webservers deutlich beeinflussen (obwohl Betriebssysteme eine Reihe von Tricks verwenden können, um mit diesem Problem besser umzugehen). Lesern, die sich für den Themenbereich Betriebssysteme und persistentes bzw. nichtpersistentes HTTP interessieren, empfehlen wir [Nielsen 1997; Nahum 2002].

Nun, da wir Transportschicht-Multiplexing und Demultiplexing erörtert haben, wollen wir mit einem der Transportprotokolle des Internets fortfahren, mit UDP. Im nächsten Abschnitt werden wir sehen, dass UDP der Netzwerkschicht kaum mehr hinzufügt als einen Multiplexing/Demultiplexing-Dienst.

3.3 Verbindungslose Kommunikation: UDP

In diesem Abschnitt werfen wir einen genauen Blick auf UDP, was es bietet und wie es arbeitet. Wir möchten Sie dazu ermuntern, sich mit Abschnitt 2.1 zu befassen, der einen Überblick über das UDP-Dienstmodell enthält, und mit Abschnitt 2.8, der Socket-Programmierung mithilfe von UDP diskutiert.

Um unserer Diskussion über UDP eine Motivation zu geben, nehmen Sie an, dass Sie daran interessiert sind, ein schlichtes, grundlegendes Transportprotokoll zu entwerfen. Wie könnten Sie das anpacken? Vielleicht ziehen Sie zuerst ein leeres Transportprotokoll in Betrachtung. Auf der sendenden Seite könnten Sie einfach die Nachrichten vom Anwendungsprozess annehmen und sie direkt an die Netzwerkschicht weiterreichen. Auf der Empfangsseite könnten Sie die in der Netzwerkschicht ankommenden Nachrichten annehmen und sie direkt an den Anwendungsprozess weiterleiten. Aber, wie wir im vorherigen Abschnitt gelernt haben, müssen wir schon etwas mehr tun als das! Die Transportschicht muss mindestens einen Multiplexing/Demultiplexing-Dienst anbieten, um Daten zwischen der Netzwerkschicht und dem richtigen Prozess der Anwendungsschicht auszutauschen.

UDP, definiert in [RFC 768], macht nur das, was ein Transportprotokoll unbedingt tun muss. Abgesehen von der Multiplexing/Demultiplexing-Funktion und einer einfachen Fehlerprüfung fügt es IP nichts hinzu. Wählt ein Anwendungsentwickler UDP statt TCP, dann kommuniziert die Anwendung in der Tat fast direkt mit IP. UDP nimmt Nachrichten vom Anwendungsprozess entgegen, fügt Quell- und Zielportnummerfelder für den Multiplexing/Demultiplexing-Dienst ein, fügt zwei andere kleine Felder hinzu und leitet das entstehende Segment an die Netzwerkschicht weiter. Diese Netzwerkschicht verkapselt das Transportschichtsegment in ein IP-Datagramm und macht dann einen Best-effort-Versuch, das Segment an den empfangenden Host auszuliefern. Sofern das Segment diesen erreicht, verwendet UDP die Zielportnummer, um die Daten des Segmentes an den richtigen Anwendungsprozess zu liefern. Beachten Sie, dass es bei UDP keinen Handshake zwischen sendenden und empfangenden Instanzen der Transportschicht gibt, bevor das Segment gesendet wird. Aus diesem Grund wird UDP als *verbindungslos* bezeichnet.

DNS ist ein Beispiel eines Anwendungsschichtprotokolls, das typischerweise UDP verwendet. Wenn die DNS-Anwendung in einem Host eine Anfrage stellen will, erzeugt sie eine DNS-Anfrage-Nachricht und reicht diese an UDP weiter. Ohne mit der auf dem Zielendsystem laufenden UDP-Instanz irgendein Handshake auszuführen, fügt UDP auf der Host-Seite Header-Felder an die Nachricht an und leitet das entstehende Segment an die Netzwerkschicht weiter. Die Netzwerkschicht verkapselt das UDP-Segment in ein Datagramm und sendet es an einen Name-Server. Die DNS-Anwendung auf dem anfragenden Host wartet dann auf die Beantwortung der Anfrage. Erhält sie keine Antwort (vielleicht weil das zugrunde liegende Netz die Frage oder die Antwort verloren hat), versucht sie entweder, die Frage an einen anderen Name-Server zu senden, oder sie informiert die anfragende Anwendung, dass sie keine Antwort bekommt.

Sie fragen sich vielleicht, warum ein Anwendungsentwickler jemals UDP verwenden sollte. Wäre TCP nicht immer vorzuziehen, da TCP doch einen zuverlässigen Datentransferdienst erbringt? Die Antwort ist nein; viele Anwendungen eignen sich aus den folgenden Gründen besser für UDP:

- *Bessere, in der Anwendungsschicht angesiedelte Kontrolle über die gesendeten Daten.* Sobald ein Anwendungsprozess Daten an UDP weiterreicht, verpackt dieses

die Daten in einem UDP-Segment und reicht das Segment sofort an die Netzwerkschicht weiter. TCP hat andererseits einen Überlastkontrollmechanismus, der die TCP-Sender drosselt, sobald eine oder mehrere der zwischen Quelle und Zielhost liegenden Verbindungen überlastet ist. TCP sendet ein Segment auch immer wieder, bis dessen Empfang am Zielort bestätigt worden ist, egal wie lange die zuverlässige Übertragung dauert. Da Echtzeitanwendungen oft eine minimale Übertragungsrate erfordern, ihre Segmente nicht übermäßig verzögert sein sollten und sie einen geringen Datenverlust verkraften können, passt das Dienstmodell von TCP nicht wirklich gut zu den Anforderungen dieser Anwendungen. Wie wir weiter unten besprechen werden, können diese Anwendungen UDP verwenden und implementieren, als Teil der Anwendung, jedwede zusätzliche Funktionalität, die über den von UDP gelieferten einfachen Segmentzustelldienst hinaus geht.

- *Kein Verbindungsaufbau*. Wie wir später diskutieren werden, verwendet TCP einen Drei-Wege-Handshake, bevor es mit der Datenübertragung beginnt. UDP legt ohne formale Vorbereitungen einfach los. Auf diese Art führt UDP beim Herstellen einer Verbindung nicht zu Verzögerungen. Das ist wahrscheinlich der Hauptgrund, warum DNS über UDP läuft, statt über TCP – DNS wäre viel langsamer, wenn es TCP benutzen würde. HTTP verwendet TCP statt UDP, da Zuverlässigkeit für Webseiten mit Text entscheidend ist. Wie wir in Abschnitt 2.2 angesprochen hatten, ist aber die durch TCP beim Verbindungsaufbau entstehende Verzögerung in HTTP ein wesentlicher Teil der Verzögerungen, die beim Herunterladen von Webdokumenten entstehen.

- *Kein Verbindungszustand*. TCP merkt sich den Verbindungszustand in den Endsystemen. Dieser Zustand beinhaltet Empfangs- und Sendepuffer, Überlastkontrollparameter und Acknowledgment-Nummer. Wir werden in Abschnitt 3.5 sehen, dass diese Zustandsinformation benötigt wird, um den zuverlässigen Datentransferdienst von TCP und seine Überlastkontrolle zu implementieren. UDP merkt sich dagegen keinen Verbindungsstatus und keinen dieser Parameter. Daher kann ein einzelner Server normalerweise viel mehr aktive Clients unterstützen, wenn die Anwendung über UDP statt TCP läuft.

- *Geringe Header-Größe*. TCP fügt 20 Byte zusätzliche Header-Information zu jedem Segment hinzu, während UDP mit nur 8 Byte auskommt.

▶Abbildung 3.6 listet populäre Internetanwendungen und die von ihnen benutzten Transportprotokolle auf. Wie zu erwarten, laufen E-Mail, Remote Login, das Web und auch der Dateitransfer mit FTP über TCP – all diese Anwendungen brauchen den zuverlässigen Datentransferdienst von TCP. Dennoch laufen viele wichtige Anwendungen über UDP statt über TCP. UDP wird für das Aktualisieren von RIP-Routing-Tabellen eingesetzt (Abschnitt 4.6.1). Da RIP-Aktualisierungen periodisch (normalerweise alle fünf Minuten) gesandt werden, ersetzen neuere Updates ältere oder verlorene, wodurch diese veralteten Aktualisierungen nutzlos werden. UDP wird auch verwendet, um Daten für das Netzwerkmanagement zu übertragen (SNMP, siehe Kapitel 9). UDP wird in diesem Fall oft gegenüber TCP bevorzugt, da Anwendungen

Anwendung	Anwendungsschicht-protokoll	Zugrunde liegendes Transportprotokoll
E-Mail	SMTP	TCP
Remote Login	Telnet	TCP
Web	HTTP	TCP
Dateitransfer	FTP	TCP
Verteilte Dateisysteme	NFS	Normalerweise UDP
Multimedia-Streaming	Normalerweise proprietär	UDP oder TCP
Internettelefonie	Normalerweise proprietär	UDP oder TCP
Netzwerk-Management	SNMP	Normalerweise UDP
Routing-Protokoll	RIP	Normalerweise UDP
Namensauflösung	DNS	Normalerweise UDP

Abbildung 3.6: Populäre Internetanwendungen und ihre zugrunde liegenden Transportprotokolle

zum Netzwerkmanagement auch dann verlässlich funktionieren müssen, wenn die Lage im Netz angespannt ist – eben genau dann, wenn zuverlässiger Datentransfer mit Überlastkontrolle nur schwierig durchzuführen ist. Außerdem, wie wir bereits früher erwähnt haben, läuft DNS über UDP, wodurch die von TCP verursachten Verzögerungen beim Verbindungsaufbau vermieden werden.

Wie Abbildung 3.6 zeigt, werden sowohl UDP als auch TCP heute von Multimedia-Anwendungen wie Internettelefonie, Echtzeitvideokonferenzen und Streaming von Audio und Video verwendet. Wir werfen in Kapitel 7 einen genaueren Blick auf diese Anwendungen. Hier erwähnen wir nur so viel, dass alle diese Anwendungen ein gewisses Maß an Paketverlusten tolerieren können, so dass zuverlässiger Datentransfer für das Funktionieren der Anwendung nicht unbedingt notwendig ist. Weiterhin können Echtzeitanwendungen wie Internettelefonie und Videokonferenzen nicht gut mit der Überlastkontrolle von TCP umgehen. Aus diesen Gründen entscheiden sich Entwickler von Multimedia-Anwendungen oft dafür, ihre Anwendungen über UDP statt TCP auszuführen. Dennoch wird TCP in zunehmendem Maße für die Übertragung von Medieninhalten eingesetzt. So fand z.B. [Sripanidkulchai 2004] heraus, dass fast 75 % der On-Demand- und Live-Streams TCP einsetzen. Solange die Paketverlustrate gering ist und gerade auch weil eine Reihe von Organisationen UDP-Verkehr aus Sicherheitsgründen blockieren (siehe Kapitel 8), wird TCP ein immer attraktiveres Protokoll für die Übertragung von Multimedia-Daten.

Obwohl heutzutage durchaus üblich, sind Multimedia-Anwendungen über UDP umstritten. Wie wir bereits erwähnt haben, hat UDP keine Überlastkontrolle. Diese wird aber gebraucht, um zu verhindern, dass das Netz in einen überlasteten Zustand

gerät, in dem nur wenig nützliche Arbeit möglich ist. Wenn alle Videoströme ohne irgendeine Art von Überlastkontrolle mit hoher Bitrate starten würden, würde an den Routern eine riesige Flut von Paketen eintreffen, so dass nur sehr wenige UDP-Pakete den Weg von Quelle zu Zielort erfolgreich durchqueren könnten. Außerdem würden die hohen Verlustraten, welche die unkontrollierten UDP-Sender verursachen, dazu führen, dass die TCP-Sender (welche, wie wir sehen werden, ihre Senderaten im Fall von Überlast *tatsächlich* senken) ihre Raten dramatisch verringern. Auf diese Weise kann der Mangel an Überlastkontrolle in UDP zu hohen Verlustraten zwischen UDP-Sender und -Empfänger führen und außerdem zu einem Verdrängen von TCP-Sitzungen – ein möglicherweise ernstes Problem [Floyd 1999]. Viele Forscher haben neue Mechanismen vorgeschlagen, um alle Quellen, einschließlich UDP-Quellen, zu einer adaptiven Überlastkontrolle zu zwingen [Mahdavi 1997; Floyd 2000; Kohler 2006; RFC 4340].

Vor der Diskussion der UDP-Segmentstruktur erwähnen wir, dass es einer Anwendung möglich ist, trotz Verwendung von UDP einen zuverlässigen Datentransfer durchzuführen. Das ist machbar, wenn die Zuverlässigkeitsmechanismen in die Anwendung selbst eingebaut sind (zum Beispiel durch Hinzufügen von Bestätigungspaketen und Mechanismen zum Wiederholen von Übertragungen, wie jene, die wir im nächsten Abschnitt untersuchen werden). Aber das ist keine triviale Aufgabe und sie wird einen Anwendungsentwickler viel Zeit für die Fehlersuche kosten. Dennoch würde das direkte Implementieren von Zuverlässigkeit in der Anwendung es dieser ermöglichen, auf zwei Hochzeiten gleichzeitig zu tanzen: Die Anwendungsprozesse könnten zuverlässig miteinander kommunizieren, ohne den Beschränkungen der Übertragungsrate unterworfen zu sein, die durch die Überlastkontrollmechanismen von TCP erzwungen werden.

3.3.1 UDP-Segmentstruktur

Die in ▶Abbildung 3.7 gezeigte UDP-Segmentstruktur wird in RFC 768 definiert. Die Anwendungsdaten füllen das Datenfeld des UDP-Segmentes. So enthält z.B. im Fall eines DNS-Paketes das Datenfeld entweder eine Anfrage-Nachricht oder eine Antwort-Nachricht. Bei Anwendungen für Audioströme enthält das Datenfeld die Audiodaten. Der UDP-Header hat nur vier Felder, die jeweils aus zwei Byte bestehen. Wie im vorherigen Abschnitt besprochen, ermöglichen die Portnummern es dem Zielhost, die Anwendungsdaten an den richtigen Prozess weiterzuleiten (d.h., er kann das Demultiplexing durchführen). Die Prüfsumme *(Checksum)* wird vom empfangenden Host dazu verwendet, die Segmente auf Fehler zu untersuchen. Tatsächlich werden zusätzlich zum UDP-Segment auch einige Felder des IP-Headers in die Berechnung der Prüfsumme einbezogen. Wir ignorieren dieses Detail aber, um den Überblick nicht zu verlieren, und diskutieren die Prüfsummenberechnung unten. Grundprinzipien der Fehlererkennung werden in Abschnitt 5.2 beschrieben. Das Längenfeld enthält die Länge des UDP-Segmentes einschließlich des Headers in Byte.

3.3.2 UDP-Prüfsumme

Die UDP-Prüfsumme ermöglicht eine Fehlererkennung. Die Prüfsumme wird also verwendet, um festzustellen, ob Bits innerhalb des UDP-Segmentes verändert worden sind, während sie sich von der Quelle zum Zielort bewegten (z.B. aufgrund von Störungen in den Verbindungen oder während es in einem Router zwischengespeichert wurde). UDP auf der Absenderseite berechnet das 1er-Komplement der Summe aller 16 Bit-Worte im Segment, wobei ein möglicher Übertrag beim Addieren in die niedrigstwertige Stelle zurückübertragen wird. Dieses Ergebnis wird in das Prüfsummenfeld des UDP-Segmentes eingetragen. Wir zeigen hier ein einfaches Beispiel für die Prüfsummenberechnung. Details effektiver Implementierungen der in RFC 1071 definierten Berechnung und ihrer Geschwindigkeit auf realen Daten sind in [Stone 1998; Stone 2000] zu finden.

Nehmen Sie z.B. an, dass uns die folgenden drei 16 Bit-Worte vorliegen:

0110011001100000
0101010101010101
1000111100001100

Die Binärsumme der ersten beiden dieser 16 Bit-Worte ist:

0110011001100000
0101010101010101
1011101110110101

Addieren des dritten Wortes zur obigen Summe führt zu:

1011101110110101
1000111100001100
0100101011000010

Beachten Sie, dass bei dieser letzten Addition ein Überlauf auftrat, der zurück ans rechte Ende übertragen wurde, deswegen lauten die letzten beiden Stellen des Ergebnisses 10 und nicht 01. Das 1er-Komplement entsteht, indem jede 0 in eine 1 umgewandelt wird und umgekehrt. Das 1er-Komplement der Summe 0100101011000010 lautet

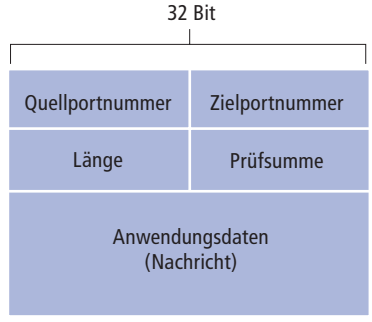


Abbildung 3.7: UDP-Segmentstruktur

daher 1011010100111101, was als Prüfsumme verwendet wird. Beim Empfänger werden alle vier 16 Bit-Worte, einschließlich der Prüfsumme, addiert. Haben sich keine Fehler in das Paket eingeschlichen, dann lautet die Summe beim Empfänger 1111111111111111. Ist ein Bit eine 0, dann wissen wir, dass bei der Übertragung Fehler aufgetreten sind.

Sie können sich fragen, warum UDP überhaupt eine Prüfsumme verwendet, wo doch so viele Sicherungsschichtprotokolle (einschließlich des weitverbreiteten Ethernet-Protokolls) ebenfalls eine Fehlerprüfung anbieten. Der Grund liegt darin, dass es keinerlei Garantie gibt, dass alle Leitungen zwischen Quelle und Zielort eine Fehlerprüfung durchführen; das bedeutet, dass irgendeine der Leitungen ein Sicherungsschichtprotokoll verwenden kann, das keine Fehlerprüfung beinhaltet. Darüber hinaus können Fehler auch beim Speichern der Segmente im Router auftreten, also selbst dann, wenn die Segmente korrekt über jede einzelne Leitung übertragen wurden. Da eine korrekte Übertragung auf jeder einzelnen Leitung und eine Fehlererkennung im Speicher eines jeden Routers nicht garantiert werden kann, muss UDP die Fehlererkennung *auf einer Ende-zu-Ende-Basis* in der Transportschicht durchführen, sofern der Ende-zu-Ende-Übertragungsdienst Fehlererkennung anbieten soll. Dies ist ein Beispiel für das berühmte Ende-zu-Ende-Prinzip des Systemdesigns [Saltzer 1984], welches besagt, dass bestimmte Funktionalität (in diesem Fall Fehlererkennung) auf einer Ende-zu-Ende-Basis durchgeführt werden muss: „In niedrigeren Schichten angesiedelte Funktionen können redundant oder von geringem Mehrwert im Verhältnis zu den Kosten sein, die ihre Gewährleistung auf einer höheren Schicht verursacht."

Weil IP praktisch jedes beliebige Schicht-2-Protokoll nutzen können soll, ist eine Fehlerprüfung in der Transportschicht als Sicherheitsmaßnahme sinnvoll. Obwohl UDP eine Fehlerprüfung durchführt, ist es außerstande, einen Fehler zu korrigieren. Einige Implementierungen von UDP verwerfen das beschädigte Segment einfach; andere reichen es mit einer Warnung an die Anwendung weiter.

Das beendet unsere Diskussion von UDP. Wir werden bald sehen, dass TCP seinen Anwendungen zuverlässigen Datentransfer garantiert, sowie einige andere Dienste, die UDP nicht bietet. Natürlich ist TCP auch komplexer als UDP. Bevor wir jedoch TCP diskutieren, sollten wir uns zurücklehnen und zuerst die zugrunde liegenden Prinzipien des zuverlässigen Datentransfers erörtern.

3.4 Grundlagen des zuverlässigen Datentransfers

In diesem Abschnitt erörtern wir das Problem des zuverlässigen Datentransfers im Allgemeinen. Dies ist sinnvoll, weil uns das Implementieren eines zuverlässigen Datentransfers nicht nur in der Transportschicht, sondern auch noch in der Sicherungsschicht und der Anwendungsschicht begegnen wird. Das allgemeine Problem ist daher von zentraler Bedeutung für die Arbeit in Netzwerken. Müsste jemand eine Liste der zehn wichtigsten Netzwerkprobleme verfassen, wäre dies sicher ein Kandidat für die Spitzenposition. Im nächsten Abschnitt werden wir TCP untersuchen und

insbesondere zeigen, dass in TCP viele der hier beschriebenen Grundlagen zur Anwendung kommen.

▶Abbildung 3.8 illustriert den Rahmen unserer Untersuchung eines zuverlässigen Datentransfers. Der abstrakte Dienst, der den Instanzen der höheren Schichten zur Verfügung gestellt wird, entspricht einem zuverlässigen Kanal zur Übertragung von Daten. Auf einem zuverlässigen Kanal werden die übertragenen Bits nicht verändert (sie können also nicht von 0 auf 1 bzw. von 1 auf 0 springen), sie können nicht verloren gehen und werden alle in der Reihenfolge zugestellt, in der sie abgesandt wurden. Das ist genau das Dienstmodell, das TCP den aufrufenden Internetanwendungen bietet.

Es liegt in der Verantwortung eines **zuverlässigen Datentransferprotokolls**, diesen abstrakten Dienst zu implementieren. Diese Aufgabe wird durch die Tatsache erschwert, dass die Schicht *unterhalb* des zuverlässigen Datentransferprotokolls unzuverlässig sein kann. So ist beispielsweise TCP ein zuverlässiges Datentransferprotokoll, das oberhalb der unzuverlässigen Ende-zu-Ende-Netzwerkschicht (IP) implementiert ist. Allgemein betrachtet könnte die Schicht zwischen den beiden zuverlässigen Endpunkten der Kommunikation aus einer einzigen physikalischen Verbindung (wie im Fall eines Datentransferprotokolls auf der Sicherungsschicht) oder aus einem globalen Verbund von Netzwerken wie dem Internet bestehen (wie im Fall eines Protokolls auf der Transportschicht). Für unsere Zwecke genügt es jedoch, diese tieferliegende Schicht als einen unzuverlässigen Punkt-zu-Punkt-Kanal aufzufassen.

In diesem Abschnitt werden wir immer detaillierter die sendenden und empfangenden Seiten eines zuverlässigen Datentransferprotokolls entwickeln, wobei wir immer komplexere Modelle des zugrunde liegenden Kanals betrachten. Abbildung 3.8 (b) zeigt die Schnittstellen unseres Datentransferprotokolls. Die sendende Seite des Datentransferprotokolls wird von oben durch einen Anruf an `rdt_send()` aktiviert. Mit diesem Aufruf wird ein Dienst angefordert, der die Daten zuverlässig an die empfangende Seite weiterleitet. (Hier bedeutet `rdt` *zuverlässiger Datentransfer (reliable data transfer)* und `_send` deutet darauf hin, dass die sendende Seite von `rdt` aufgerufen wird. Der erste Schritt bei der Entwicklung jedes Protokolls liegt in der Wahl eines guten Namens!) Auf der empfangenden Seite wird `rdt_rcv()` aufgerufen, sobald Pakete bei der empfangenden Seite des Kanals ankommen. Will das `rdt`-Protokoll Daten an die darüberliegende Schicht weitergeben, ruft es `deliver_data()` auf. Im Folgenden benutzen wir den Begriff „Paket“ anstelle des in der Transportschicht üblichen Begriffes „Segment“. Da die in diesem Abschnitt entwickelte Theorie ganz allgemein auf Computernetzwerke anwendbar ist und nicht nur für das Internet gilt, ist der generische Begriff „Paket“ besser geeignet.

Wir werden in diesem Abschnitt nur den Fall eines **unidirektionalen Datentransfers** berücksichtigen, also die Datenübertragung von der sendenden zur empfangenden Seite. Der Fall des verlässlichen **bidirektionalen Datentransfers** (also Vollduplex) ist eigentlich nicht schwieriger, müsste aber äußerst weitschweifig erklärt werden. Obwohl wir nur unidirektionalen Datentransfer betrachten, dürfen wir nicht vergessen, dass die sendende sowie die empfangende Seite unseres Protokolls trotzdem Pakete in beide Richtungen übertragen müssen, was auch in Abbildung 3.8 skizziert

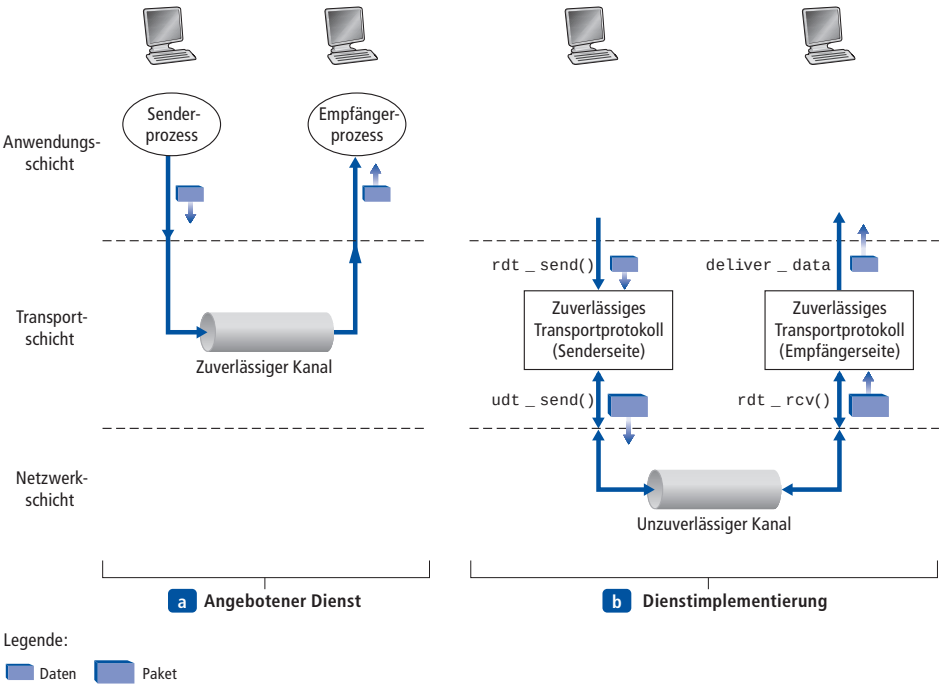


Abbildung 3.8: Verlässlicher Datentransfer: Dienstmodell und Dienstimplementierung

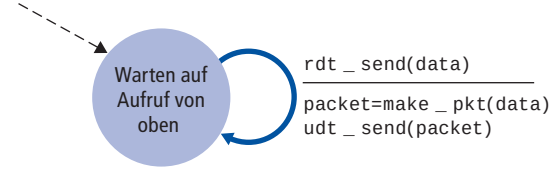
wird. Wir werden in Kürze sehen, dass zusätzlich zum Austausch von Paketen, in denen die zu übertragenden Daten enthalten sind, die sendenden und empfangenden Seiten von `rdt` Kontrollpakete austauschen müssen. Diese beiden Seiten von `rdt` senden die Pakete mittels eines Aufrufes von `udt_send()` zur anderen Seite (wobei `udt` für *unzuverlässigen Datentransfer (unreliable data transfer)* steht).

3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

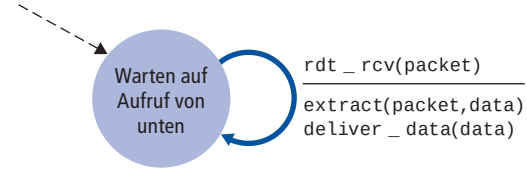
Wir gehen nun eine Reihe von immer komplexer werdenden Protokollen durch, mit dem Ziel eines einwandfreien und zuverlässigen Datentransferprotokolls.

Zuverlässiger Datentransfer über einen perfekt zuverlässigen Kanal: rdt1.0

Wir gehen zunächst vom einfachsten Fall aus, in dem der zugrunde liegende Kanal vollkommen zuverlässig ist. Das Protokoll, das wir `rdt1.0` nennen, ist daher trivial. Die Definitionen der **endlichen Automaten** (*finite state machine*, **FSM**) für den Sender und Empfänger von `rdt1.0` sind in ▶Abbildung 3.9 dargestellt. Die FSM in Abbildung 3.9 (a) legt die Operationen des Senders fest, während die FSM in Abbildung 3.9 (b) die Operationen des Empfängers definiert. Zu beachten ist, dass es für Sender und Empfänger voneinander *unabhängige* FSMs gibt. Sender- und Empfänger-FSM in ▶Abbildung 3.9 besitzen jeweils nur einen Zustand. Die Pfeile in der Beschreibung der FSM zeigen den Übergang des Protokolls von einem in einen ande-



a rdt1.0-Sender



b rdt1.0-Empfänger

Abbildung 3.9: `rdt1.0` – ein Protokoll für einen zuverlässigen Kanal

ren Zustand. (Da jede FSM in Abbildung 3.9 nur einen Zustand besitzt, ist ein Übergang von diesem einen Zustand auf sich selbst notwendig. In Kürze werden wir kompliziertere Zustandsdiagramme sehen.) Das Ereignis, das diesen Übergang hervorruft, ist oberhalb der horizontalen Linie zu sehen, die den Übergang kennzeichnet und die Aktionen, die nach Eintreten des Ereignisses ausgeführt werden, sind unterhalb dieser horizontalen Linie dargestellt. Löst ein Ereignis keine Aktion aus, oder tritt kein Ereignis ein und es wird trotzdem eine Aktion ausgeführt, benutzen wir das Symbol Λ oberhalb beziehungsweise unterhalb dieser Linie, um explizit das Fehlen einer Aktion oder eines Ereignisses zu kennzeichnen. Der Ausgangszustand der FSM wird durch den gestrichelten Pfeil dargestellt. Die FSMs in Abbildung 3.9 können nur einen einzigen Zustand einnehmen. Die FSMs, denen wir in Kürze begegnen, können jedoch mehrere Zustände besitzen. Daher ist es wichtig, den Ausgangszustand jeder FSM zu kennzeichnen.

Die sendende Seite von `rdt` akzeptiert einfach Daten der darüberliegenden Schicht mittels des Ereignisses `rdt_send(data)`, erzeugt ein Paket, welches diese Daten enthält (mithilfe der Aktion `make_pkt(data)`) und sendet das Paket in den Kanal. In der Praxis würde das `rdt_send(data)`-Ereignis durch einen Funktionsaufruf (z.B. an `rdt_send()`) der Anwendung ausgelöst werden.

Auf der empfangenden Seite erhält `rdt` von dem darunterliegenden Kanal ein Paket durch das Ereignis `rdt_rcv(paket)`, es entfernt die Daten aus dem Paket (mithilfe der Aktion `extract(paket, data)`) und reicht die Daten an die darüberliegende Schicht weiter (durch die Aktion `deliver_data(data)`). In der Praxis würde das Ereignis `rdt_rcv(paket)` von einem Funktionsaufruf des Protokolls der tieferliegenden Schicht ausgelöst.

In diesem einfachen Protokoll gibt es keinen Unterschied zwischen einer Dateneinheit und einem Paket. Darüber hinaus geht der komplette Paketfluss vom Sender zum Empfänger. Da wir einen perfekt zuverlässigen Kanal annehmen, besteht keinerlei Notwendigkeit seitens des Empfängers, irgendeine Rückmeldung an den Sender zu geben, denn nichts kann schief gehen! Beachten Sie, dass wir auch vorausgesetzt haben, dass der Empfänger die Daten genauso schnell annehmen kann, wie sie der Sender liefert. Daher besteht keine Notwendigkeit für den Empfänger, den Sender um eine langsamere Übertragung zu bitten.

Zuverlässiger Transfer über einen Kanal mit Bitfehlern: rdt2.0

In einem realistischeren Modell des zugrunde liegenden Kanals können die Bits innerhalb eines Paketes verändert worden sein. Solche Bitfehler treten üblicherweise in einer physikalischen Komponente des Netzwerkes auf, während ein Paket übertragen, weitergeleitet oder gepuffert wird. Wir gehen zunächst weiter davon aus, dass alle übertragenen Pakete in derselben Reihenfolge empfangen werden, in der sie gesendet wurden (allerdings könnten ihre Bits verändert worden sein).

Bevor wir ein Protokoll für die zuverlässige Kommunikation über einen derartigen Kanal entwickeln, lassen Sie uns erst darüber nachdenken, wie Menschen mit so einer Situation umgehen. Stellen Sie sich vor, dass Sie selbst eine lange Nachricht über das Telefon diktieren. In einem typischen Szenario würde der Empfänger der Nachricht nach jedem Satz, den er gehört, verstanden und gespeichert hat, mit „OK“ antworten. Hört der Empfänger einen verstümmelten Satz, werden Sie gebeten, den Satz zu wiederholen. Dieses Nachrichten-Diktat-Modell benutzt sowohl **positive Rückmeldungen** („OK“) als auch **negative Rückmeldungen** („Bitte wiederholen Sie das“). Mit diesen Kontrollnachrichten kann der Empfänger dem Sender mitteilen, was er richtig verstanden hat und was fehlerhaft empfangen wurde und daher wiederholt werden muss. Im Kontext eines Computernetzwerkes werden zuverlässige Datentransferprotokolle auf der Basis solcher Rückmeldungen als **ARQ-Protokolle** (**Automatic Repeat reQuest**, *automatische Wiederholungsanfrage*) bezeichnet.

Grundsätzlich müssen ARQ-Protokolle drei zusätzliche Fähigkeiten beinhalten, um die Anwesenheit von Bitfehlern behandeln zu können:

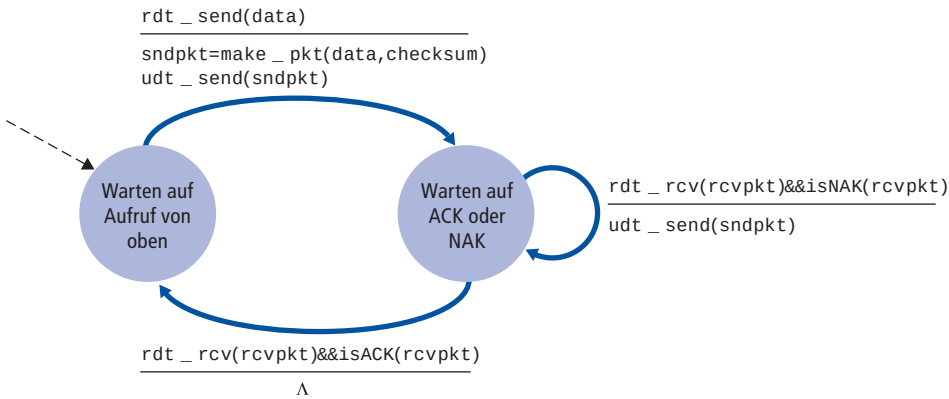
- *Fehlererkennung*: Zuerst wird ein Mechanismus benötigt, der es dem Empfänger ermöglicht, aufgetretene Bitfehler zu erkennen. Erinnern Sie sich an den vorangegangenen Abschnitt und daran, dass UDP ein Prüfsummenfeld für genau diesen Zweck benutzt. In Kapitel 5 werden wir Techniken zur Fehlererkennung und Fehlerkorrektur detaillierter betrachten. Solche Techniken erlauben einem Empfänger das Erkennen und manchmal auch das Korrigieren von Bitfehlern in Paketen. Zum jetzigen Zeitpunkt müssen wir nur wissen, dass diese Techniken die Übertragung zusätzlicher Bits (über die Bits der ursprünglich übertragenen Originaldaten hinaus) vom Sender zum Empfänger erfordern. Diese Bits befinden sich im Feld der Paketprüfsumme des `rdt2.0`-Paketes.

- *Rückmeldung des Empfängers*: Da Sender und Empfänger üblicherweise auf unterschiedlichen Endsystemen laufen, die möglicherweise Tausende Kilometer voneinander entfernt sind, besteht die einzige Möglichkeit für den Sender, etwas über den Zustand in der Welt des Empfängers zu erfahren (genau gesagt zu erfahren, ob ein Paket korrekt angekommen ist), darin, dass der Empfänger auf jeden Fall eine Rückmeldung an den Sender übermittelt. Die positiven (ACK – für ACKnowledgment) und negativen (NAK – für Negative AcKnowledgment) Rückmeldungen im Nachrichten-Diktat-Szenario sind Beispiele einer solchen Rückantwort. Unser `rdt2.0`-Protokoll sendet analog dazu ACK- und NAK-Pakete vom Empfänger an den Sender zurück. Im Prinzip müssen diese Pakete nur ein Bit lang sein. Beispielsweise könnte der Wert 0 das NAK und der Wert 1 das ACK bedeuten.
- *Wiederholte Übertragung*: Ein Paket, das beim Empfänger fehlerhaft ankommt, wird vom Sender erneut übertragen.

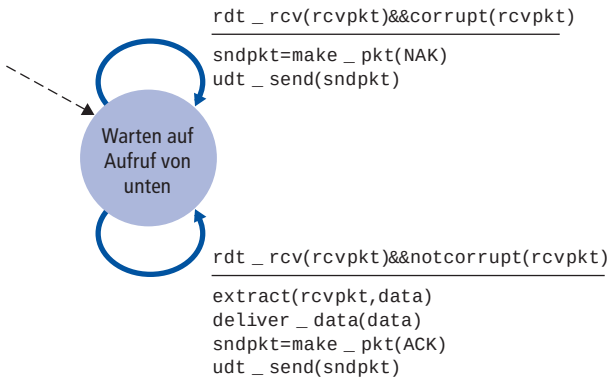
▶Abbildung 3.10 zeigt die FSM-Darstellung von `rdt2.0`, einem Datentransferprotokoll, das Fehlererkennung beherrscht und positive sowie negative Rückmeldungen gibt. Die sendende Seite von `rdt2.0` besitzt zwei Zustände. Im linken Zustand wartet das senderseitige Protokoll auf Daten, die von der oberen Schicht heruntergereicht werden. Sobald das Ereignis `rdt_send(data)` eintritt, erzeugt der Sender ein Paket (`sndpkt`), welches die zu übertragenden Daten sowie eine Paketprüfsumme enthält (z.B. diejenige, die wir im Abschnitt 3.2 für den Fall eines UDP-Segmentes diskutiert haben) und schickt danach das Paket mittels der Operation `udt_send(sndpkt)` ab. Im rechten Zustand wartet das Senderprotokoll auf ein ACK- oder NAK-Paket vom Empfänger. Trifft ein ACK-Paket ein (die Notation `rdt_rcv(rcvpkt) && isACK(rcvpkt)` in Abbildung 3.10 entspricht diesem Ereignis), weiß der Sender, dass das zuletzt übertragene Paket korrekt angekommen ist. Daher kehrt das Protokoll in den Zustand zurück, in den es auf Daten von der oberen Schicht wartet. Trifft ein NAK ein, wiederholt das Protokoll das zuletzt gesendete Paket und wartet wieder auf ein ACK oder NAK, welches vom Empfänger als Antwort auf die erneute Datenübertragung zurückgesandt wird. Dabei ist es wichtig, dass der Sender keinerlei Daten von der oberen Schicht empfangen *kann*, solange er sich im Zustand des Wartens auf ACK oder NAK befindet. Das bedeutet, das Ereignis `rdt_send()` kann nicht eintreten. Dieses Ereignis kann erst dann wieder auftreten, wenn der Sender ein ACK erhält und diesen Zustand verlässt. Daher überträgt der Sender keine neuen Daten, bis er sicher ist, dass der Empfänger das aktuelle Paket richtig erhalten hat. Aufgrund dieses Verhaltens werden Protokolle wie `rdt2.0` als **Stop-and-Wait-Protokolle** *(Anhalten-und-Warten)* bezeichnet.

Die Empfänger-FSM für `rdt2.0` hat immer noch nur einen einzigen Zustand. Bei der Ankunft eines Paketes antwortet der Empfänger entweder mit einem ACK oder einem NAK, je nachdem, ob das erhaltene Paket beschädigt war oder nicht. In Abbildung 3.10 entspricht die Notation `rdt_rcv(rcvpkt) && corrupt(rcvpkt)` dem Ereignis eines empfangenen Paketes, in dem Fehler gefunden wurden.

Unser Protokoll `rdt2.0` erweckt zwar den Anschein, als ob es funktionieren würde, aber es hat einen fatalen Fehler. Wir haben noch nicht die Möglichkeit berücksichtigt,



a rdt2.0-Sender



b rdt2.0-Empfänger

Abbildung 3.10: `rdt2.0` – ein Protokoll mit Behandlung von Bitfehlern

dass die ACK- oder NAK-Pakete selbst fehlerhaft sein könnten! (Bevor Sie weiterlesen, sollten Sie darüber nachdenken, wie dieses Problem gelöst werden könnte.) Unglücklicherweise ist unsere Nachlässigkeit nicht so harmlos, wie es scheint. Zunächst müssen wir auf jeden Fall Prüfsummenbits an ACK-/NAK-Pakete hinzufügen, um solche Fehler zu entdecken. Die schwierigere Frage ist aber, wie das Protokoll sich von Fehlern in ACK- oder NAK-Paketen erholen sollte. Die Schwierigkeit besteht darin, dass im Fall eines korrumpierten ACK oder NAK der Absender nicht erfahren kann, ob der Empfänger den letzten Teil der gesendeten Daten richtig erhalten hat.

Betrachten Sie folgende drei Möglichkeiten, wie Sie mit korrupten ACKs oder NAKs umgehen könnten:

- Überlegen Sie zuerst, was ein Mensch im Nachrichten-Diktat-Szenario tun könnte. Wenn der Sprecher die Antworten „OK“ oder das „Bitte wiederholen Sie das“ des Empfängers nicht versteht, würde der Sprecher wahrscheinlich rückfragen, „Was

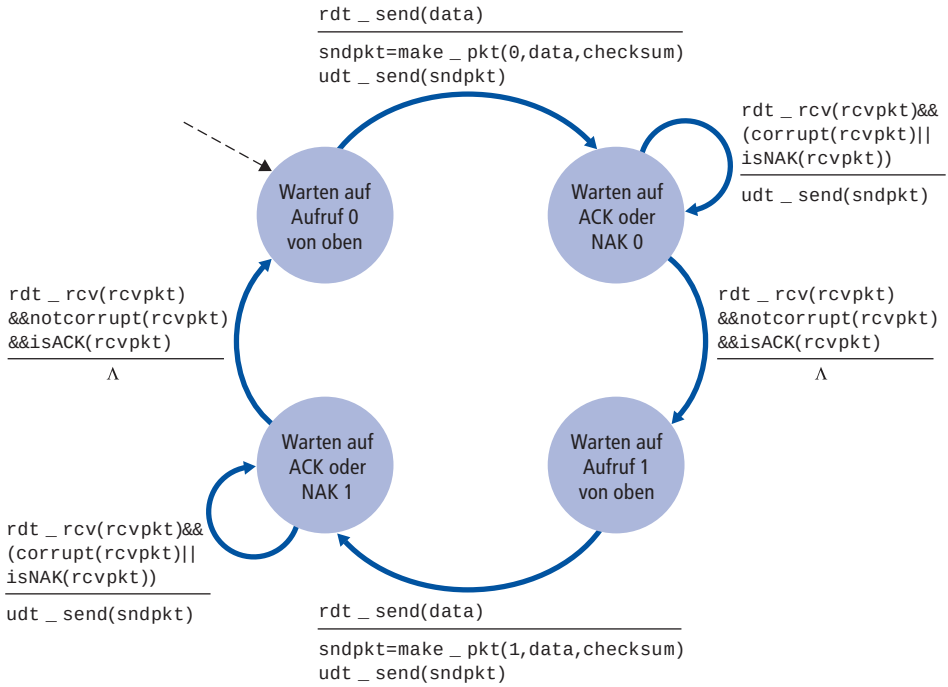


Abbildung 3.11: `rdt2.1`-Sender

sagten Sie?“ (und auf diese Weise unserem Protokoll ein neues Sender-zu-Empfänger-Paket hinzufügen). Der Sprecher würde dann die Antwort wiederholen. Aber was, wenn die Rückfrage des Sprechers ebenfalls korrumpiert ist? Der Empfänger, der keine Ahnung hat, ob der wirre Satz Teil des Diktats war oder eine Bitte, die letzte Antwort zu wiederholen, würde wahrscheinlich mit „Was sagten *Sie*?“ antworten. Natürlich könnte auch diese Antwort undeutlich sein. Wir bewegen uns eindeutig auf einem schmalen Grat.

- Die zweite Alternative besteht darin, so viele Prüfsummenbits hinzuzufügen, dass der Absender Bitfehler nicht nur wahrnimmt, sondern sie auch korrigieren kann. Dies löst das unmittelbare Problem für einen Kanal, der Pakete zwar verändern, sie aber nicht verlieren kann.
- Ein dritter Ansatz besteht darin, dass der Sender einfach das gegenwärtige Datenpaket erneut abschickt, wenn er ein verfälschtes ACK- oder NAK-Paket erhält. Dieser Ansatz fügt jedoch **Paketduplikate** in den Kanal vom Sender zum Empfänger ein. Die grundlegende Schwierigkeit bei Paketduplikaten besteht darin, dass der Empfänger ja nicht weiß, ob das zuletzt gesandte ACK oder NAK vom Absender richtig empfangen wurde. Auf diese Art kann der Empfänger *a priori* gar nicht wissen, ob ein ankommendes Paket neue Daten enthält oder die Wiederholung einer Übertragung ist!

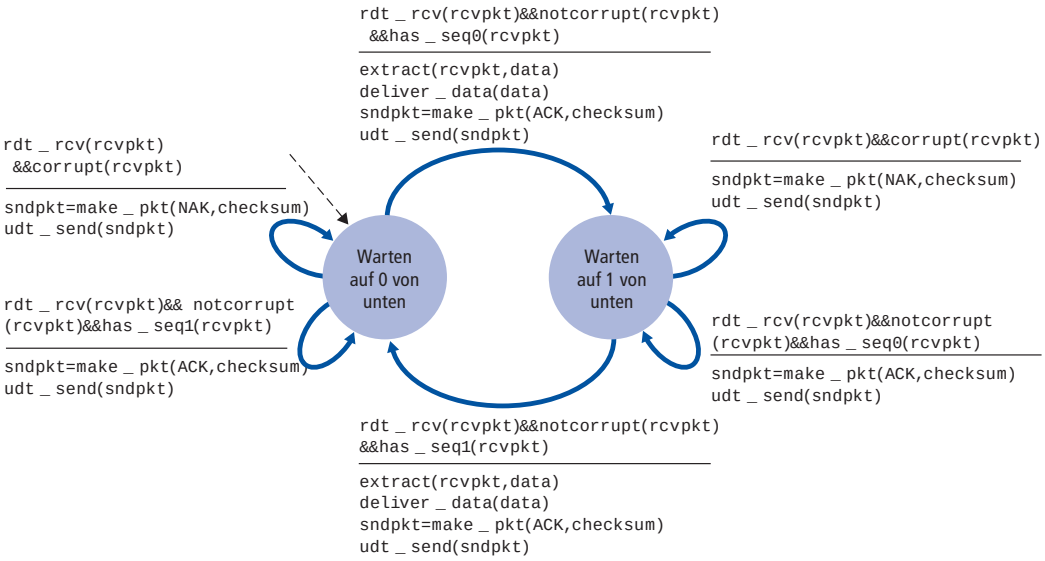


Abbildung 3.12: `rdt2.1`-Empfänger

Eine einfache Lösung dieses neuen Problems (eine, die von fast allen heute existierenden Datentransferprotokollen, einschließlich TCP, übernommen wurde) besteht darin, dem Datenpaket ein neues Feld hinzuzufügen und den Absender seine Datenpakete nummerieren zu lassen, indem er in dieses Feld eine fortlaufende **Sequenznummer** *(sequence number)* einträgt. Der Empfänger muss dann nur diese Zahl überprüfen, um zu bestimmen, ob das erhaltene Paket die Wiederholung einer früheren Übertragung ist. Für den einfachen Fall eines Stop-and-Wait-Protokolls genügt sogar eine 1-Bit-Zahl, da sie dem Empfänger sagt, ob der Absender das zuvor gesendete Paket erneut sendet (dann ist die Sequenznummer des eingetroffenen Paketes identisch mit der des zuletzt erhaltenen Paketes) oder ob es sich um ein neues Paket handelt (die Sequenznummer ändert sich und wird in einer Modulo-2-Arithmetik hochgezählt). Da wir momentan von einem Kanal ausgehen, der keine Pakete verliert, müssen ACK- und NAK-Pakete die Sequenznummer der Pakete, die sie gerade bestätigen, nicht mit angeben. Der Absender weiß, dass ein erhaltenes ACK- oder NAK-Paket (ob verfälscht oder nicht) als Antwort auf sein zuletzt gesendetes Datenpaket erzeugt wurde.

▶Abbildung 3.11 und ▶Abbildung 3.12 zeigen die FSM-Beschreibung für `rdt2.1`, unsere korrigierte Version von `rdt2.0`. Die FSMs des `rdt2.1`-Senders und -Empfängers haben jetzt doppelt so viele Zustände wie zuvor, weil der Protokollzustand wiedergeben muss, ob das gerade gesendete Paket (das vom Sender stammt) oder das (beim Empfänger) erwartete Paket die Sequenznummer 0 oder 1 haben sollte. Beachten Sie, dass die Aktionen in jenen Zuständen, in denen ein mit einer 0 gekennzeichnetes Paket versandt oder erwartet wird, die Spiegelbilder jener Aktionen sind, in denen ein mit 1 nummeriertes Paket versandt bzw. erwartet wird. Der einzige Unterschied besteht in der Behandlung der Sequenznummern.

Das Protokoll `rdt2.1` benutzt sowohl positive als auch negative Acknowledgments vom Empfänger zum Sender. Trifft ein Paket außerhalb der Reihenfolge ein, sendet der Empfänger ein positives Acknowledgment für das davor erhaltene Paket. Wird ein korrumpiertes Paket empfangen, sendet der Empfänger ein negatives Acknowledgment. Wir erzielen dasselbe Ergebnis wie ein NAK, indem wir statt eines NAK, ein ACK für das letzte *richtig* erhaltene Paket senden. Ein Absender, der zwei ACKs für dasselbe Paket erhält (das heißt, er erhält **doppelte ACKs** *(duplicate ACKs)*), weiß, dass der Empfänger das Paket nicht richtig erhalten hat, das auf das zweimal bestätigte Paket folgte. Unser NAK-freies zuverlässiges Datenübertragungsprotokoll für einen Kanal mit Bitfehlern, `rdt2.2`, ist in ▶Abbildung 3.13 und ▶Abbildung 3.14 zu sehen. Eine kleine Änderung zwischen `rtdt2.1` und `rdt2.2` besteht darin, dass der Empfänger nun die Sequenznummer des Paketes, das durch die ACK-Nachricht bestätigt wird, angeben muss (indem das Argument ACK,0 oder ACK,1 an `make_pkt()` in der Empfänger-FSM übergeben wird). Außerdem muss der Absender nun die Sequenznummer des von einer eingegangenen ACK-Nachricht bestätigten Paketes überprüfen (dies erfolgt durch Übergabe der Argumente 0 oder 1 an `isACK()` in der Sender-FSM).

Zuverlässiger Datentransfer über einen verlustbehafteten Kanal mit Bitfehlern: rdt3.0

Nehmen Sie nun an, dass außer dem Verfälschen von Bits auch Pakete auf dem zugrunde liegenden Kanal verloren gehen können – ein nicht ungewöhnliches Ereig-

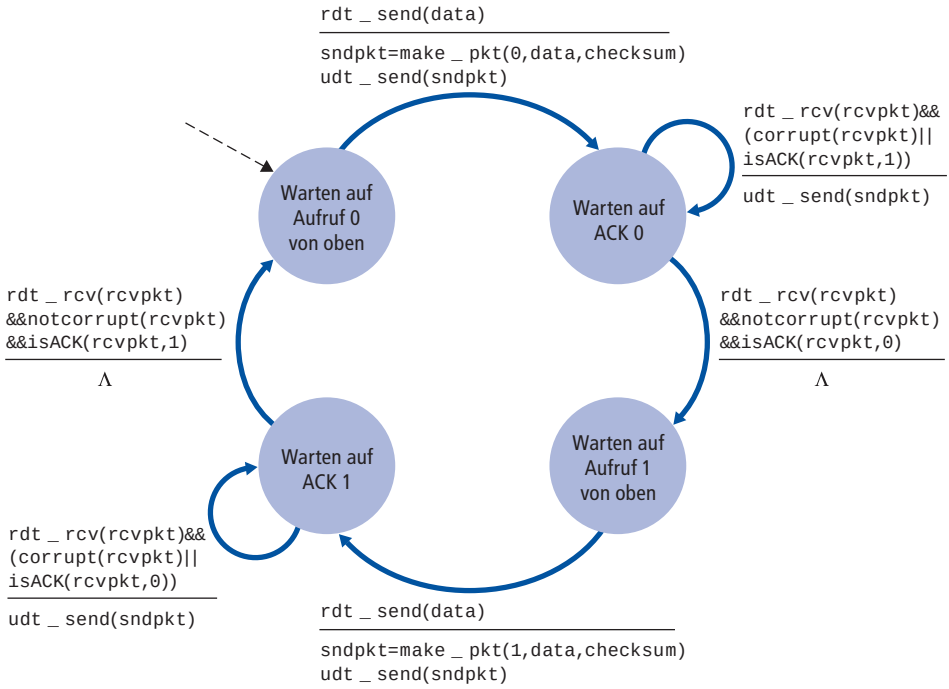


Abbildung 3.13: `rdt2.2`-Sender

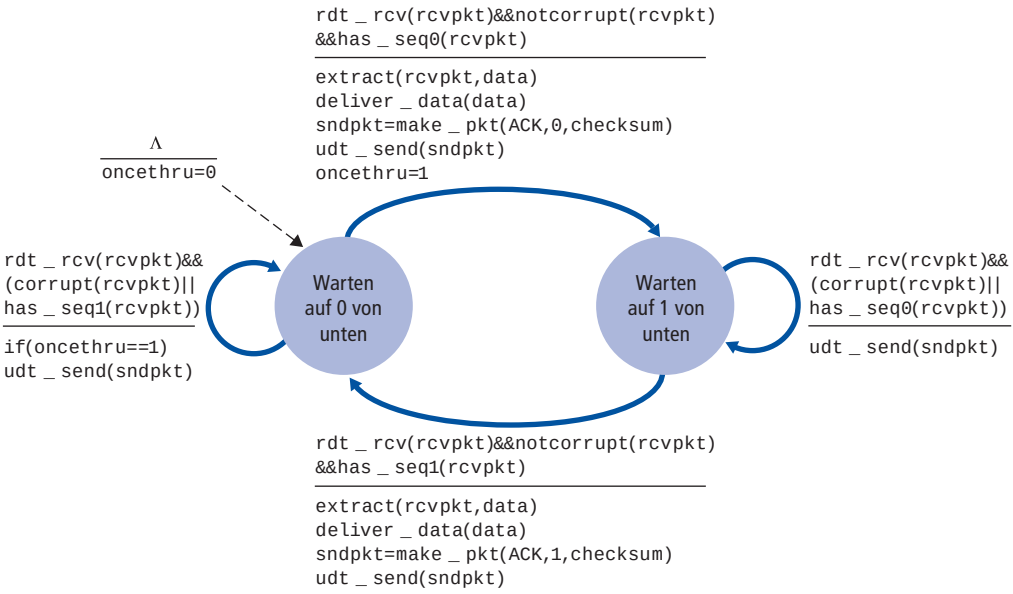


Abbildung 3.14: `rdt2.2`-Empfänger

nis in den heutigen Computernetzwerken (einschließlich des Internets). Zwei zusätzliche Fragen müssen jetzt durch das Protokoll angegangen werden: Wie kann Paketverlust entdeckt werden und was soll geschehen, wenn er eintritt? Das Berechnen von Prüfsummen, Sequenznummern, ACK-Pakete und wiederholte Übertragungen – Techniken, die bereits in `rdt2.2` entwickelt wurden – ermöglichen es uns, letztere Frage anzugehen. Für die erste Frage wird ein neuer Protokollmechanismus notwendig.

Es gibt viele mögliche Ansätze für den Umgang mit Paketverlusten (einige erkunden wir in den Übungen am Ende des Kapitels). Im Moment bürden wir dem Sender die Verantwortung auf, Paketverluste zu entdecken und zu korrigieren. Nehmen Sie an, dass der Absender ein Datenpaket sendet, und entweder dieses Paket oder dessen ACK durch den Empfänger geht verloren. Auf jeden Fall trifft beim Absender keine Antwort des Empfängers ein. Sofern der Absender bereit ist, so lange zu warten, dass er *sicher* sein kann, dass ein Paket verloren gegangen ist, kann er das Datenpaket einfach noch einmal übertragen. Sie sollten sich davon überzeugen, dass dieses Protokoll wirklich funktioniert.

Aber wie lange muss der Sender warten, bis er sicher sein darf, dass etwas verloren gegangen ist? Klarerweise muss der Sender mindestens eine Rundlaufzeit zwischen Absender und Empfänger warten (in der ja auch das Puffern in dazwischenliegenden Routern steckt) plus der Zeitdauer, die das Paket für die Verarbeitung beim Empfänger benötigt. In vielen Netzwerken ist es schon sehr schwierig, diese maximale Verzögerung auch nur abzuschätzen, geschweige denn, sie sicher zu kennen. Außerdem sollte das Protokoll idealerweise so schnell wie möglich den Paketverlust beheben. Die Worst-Case-Verzögerung abzuwarten, könnte bedeuten, dass bis zur Fehlerbehebung eine lange Wartezeit verstreicht. Der in der Praxis benutzte Ansatz besteht darin,

einen Zeitraum geschickt so festzulegen, dass Paketverlust wahrscheinlich ist, auch wenn er nicht sicher eingetreten ist. Wird innerhalb dieser Zeit kein ACK empfangen, wird das Paket nochmals übertragen. Beachten Sie, dass ein Paket, das eine besonders lange Verzögerung erfährt, auch erneut übertragen werden kann, obwohl weder das Datenpaket noch sein ACK verloren gegangen sind. Dadurch besteht die Möglichkeit **doppelter Datenpakete** auf dem Kanal zwischen Sender und Empfänger. Glücklicherweise hat das `rdt2.2`-Protokoll bereits genügend Funktionalität (in Form von Sequenznummern), um mit doppelten Paketen umgehen zu können.

Aus Sicht des Absenders ist die Übertragungswiederholung ein Allheilmittel. Der Sender weiß nicht, ob ein Datenpaket bzw. ein ACK verloren ging oder ob Datenpaket bzw. ACK nur übermäßig verzögert sind. In allen Fällen bleibt die daraus resultierende Handlung dieselbe: erneute Übertragung. Das Implementieren eines zeitbasierten Übertragungswiederholungsmechanismus erfordert einen **Countdown-Timer**, der den Absender benachrichtigen kann, nachdem eine bestimmte Zeit verstrichen ist. Der Absender muss daher in der Lage sein, (1) jedes Mal, wenn ein Paket (entweder zum ersten Mal oder als Wiederholung) verschickt wird, den Timer neu zu starten, (2) auf einen auslaufenden Timer zu reagieren (und die entsprechenden Maßnahmen zu ergreifen) und (3) den Timer anzuhalten.

▶ Abbildung 3.15 zeigt die Sender-FSM für `rdt3.0`, ein Protokoll, das zuverlässig Daten über einen Kanal überträgt, der Pakete verändern oder verlieren kann.

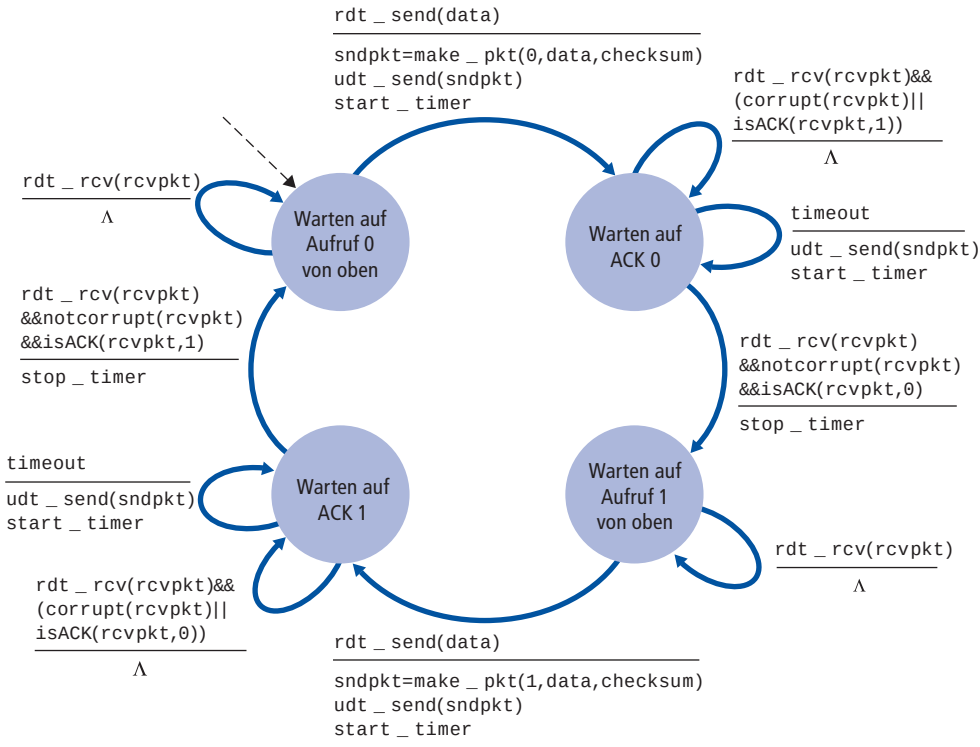


Abbildung 3.15: `rdt3.0`-Sender

In den Übungsaufgaben werden Sie sich damit beschäftigen, die Empfänger-FSM für `rdt3.0` zu konstruieren. ▶Abbildung 3.16 zeigt, wie das Protokoll verlustfrei bzw. ohne verzögerte Pakete abläuft und wie es mit dem Verlust von Datenpaketen fertig wird. In Abbildung 3.16 verläuft die Zeit von oben nach unten im Diagramm. Ein solches Diagramm nennt man auch Zeit-Ablauf-Diagramm. Beachten Sie, dass der Empfangszeitpunkt eines Paketes notwendigerweise später liegen muss als der zugehörige Sendezeitpunkt, eine Folge der Übertragungs- und Ausbreitungsverzögerungen. In den Abbildungen 3.16 (b)–(d) geben die senderseitigen Klammern die Zeitpunkte an,

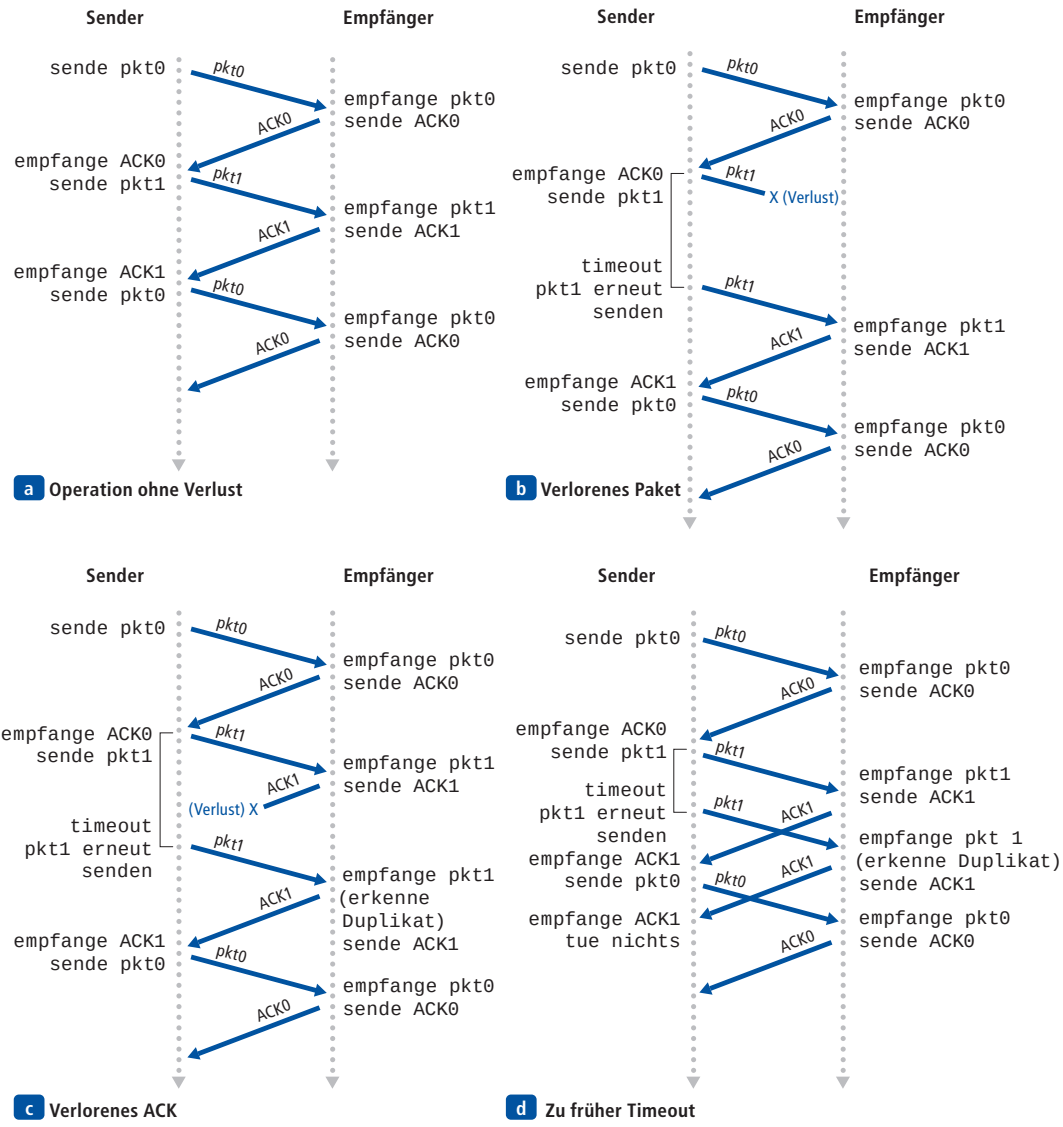


Abbildung 3.16: Arbeitsweise von `rdt3.0` (Alternierendes-Bit-Protokoll)

zu denen der Timer gesetzt wurde und später abläuft. Mehr Feinheiten dieses Protokolls werden Sie in den Übungen am Ende dieses Kapitels erkunden. Weil die Sequenznummern der Pakete immer zwischen 0 und 1 wechseln, wird das Protokoll `rdt3.0` manchmal als **Alternierendes-Bit-Protokoll** *(alternating-bit protocol)* bezeichnet.

Wir haben nun die Schlüsselelemente eines zuverlässigen Datentransferprotokolls zusammengetragen. Prüfsummen, Sequenznummern, Timer sowie positive und negative Acknowledgment-Pakete spielen jeweils eine entscheidende Rolle im Ablauf des Protokolls. Wir haben jetzt ein lauffähiges zuverlässiges Datentransferprotokoll!

3.4.2 Zuverlässige Datentransferprotokolle mit Pipelining

`rdt3.0` ist zwar ein korrekt funktionierendes Protokoll, aber es ist unwahrscheinlich, dass irgendjemand mit seiner Leistung zufrieden wäre, insbesondere in den heutigen Hochgeschwindigkeitsnetzen. Der Kern des Leistungsproblems von `rdt3.0` liegt darin, dass es ein Stop-and-Wait-Protokoll ist.

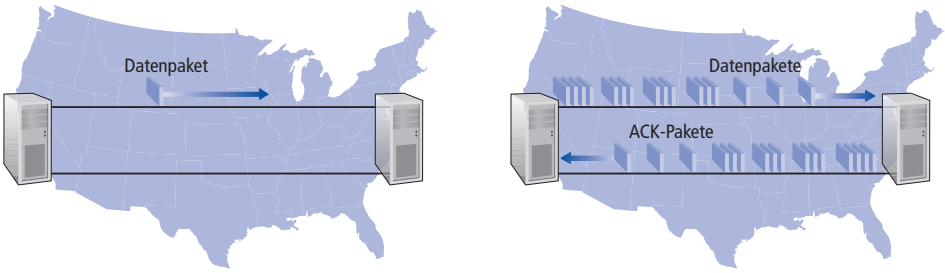


Abbildung 3.17: Stop-and-Wait im Vergleich zu Pipelining

Um den Einfluss dieses Stop-and-Wait-Verfahrens auf die Leistung abzuschätzen, betrachten Sie den idealisierten Fall zweier Hosts, von denen einer an der amerikanischen Westküste, der andere an der Ostküste steht, wie in ▶Abbildung 3.17 gezeigt. Die bei einer Signalausbreitung mit Lichtgeschwindigkeit auftretende Rundlauf-Ausbreitungsverzögerung zwischen diesen beiden Endsystemen, die *RTT*, beträgt etwa 30 Millisekunden. Nehmen Sie an, dass Sie durch einen Kanal verbunden sind, dessen Übertragungsgeschwindigkeit R 1 Gbps (10^9 Bit pro Sekunde) beträgt. Bei einer Paketgröße L von 1.000 Byte ergibt sich

$$d_{\text{trans}} = \frac{L}{R} = \frac{8000 \text{ Bits/Paket}}{10^9 \text{ Bits/s}} = 8 \text{ Mikrosekunden/Paket}$$

▶Abbildung 3.18 (a) zeigt, was bei unserem Stop-and-Wait-Protokoll geschieht: Überträgt der Sender das Paket ab dem Zeitpunkt $t = 0$, dann tritt bei $t = L/R =$ 8 Mikrosekunden das letzte Bit auf der Absenderseite in den Kanal ein. Das Paket reist dann in 15 ms quer durch das Land, wobei das letzte Bit zum Zeitpunkt

$t = RTT/2 + L/R = 15{,}008$ ms auf der Empfängerseite aus dem Kanal austritt. Der Einfachheit halber nehmen wir an, dass ACK-Pakete äußerst klein sind (so dass wir ihre Übertragungszeit ignorieren können) und dass der Empfänger ein ACK sendet, sobald das letzte Bit eines Datenpaketes eingetroffen ist. Dann taucht das ACK zum Zeitpunkt $t = RTT + L/R = 30{,}008$ ms beim Empfänger auf. Zu diesem Zeitpunkt kann der Sender die nächste Nachricht senden. Während der Gesamtdauer von 30,008 ms hat der Sender also nur 0,008 ms lang Daten übertragen. Definieren wir die **Auslastung** des Absenders (oder des Kanals) als den Bruchteil der Zeit, in der der Absender tatsächlich damit beschäftigt ist, Bits in den Kanal einzuspeisen, so zeigt die Analyse in Abbildung 3.18 (a), dass das Stop-and-Wait-Protokoll nur eine ziemlich schwache Auslastung U_{Sender} von

$$U_{\text{Sender}} = \frac{L / R}{\text{RTT} + L / R} = \frac{0{,}008}{30{,}008} = 0{,}00027$$

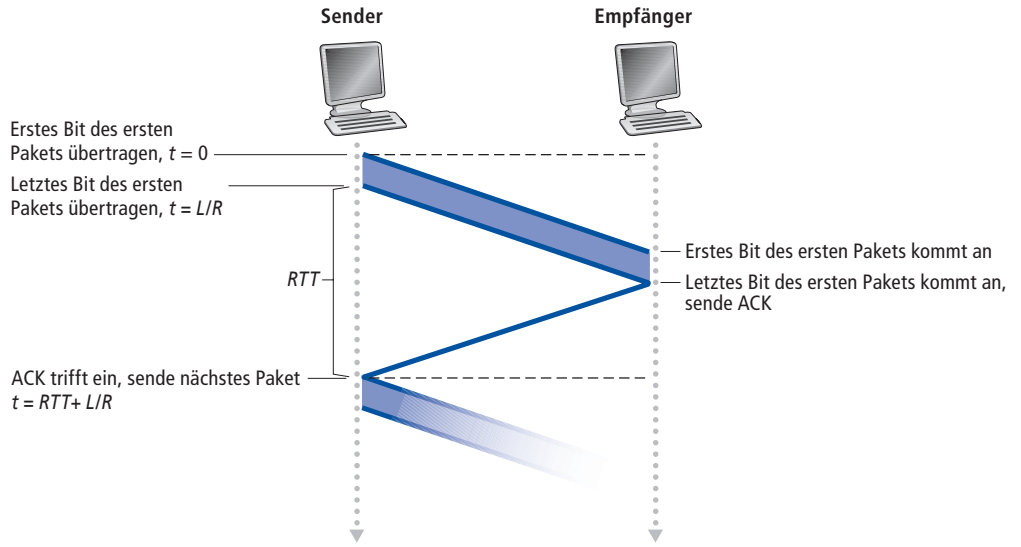
erreicht: Der Sender war nur während weniger als drei Hundertstel eines Prozents der gesamten Zeitspanne aktiv! Aus einem anderen Blickwinkel betrachtet hat der Sender während der 30,008 Millisekunden nur 1.000 Byte versendet, was einem effektiven Durchsatz von nur 267 Kbps entspricht – obwohl eine 1 Gbps-Leitung zur Verfügung steht! Stellen Sie sich den unglücklichen Netzwerkadministrator vor, der gerade ein kleines Vermögen für eine Gigabit-Leitung gezahlt hat und dann nur einen Durchsatz von 267 Kbit pro Sekunde erreicht!

Dies ist ein deutliches Beispiel dafür, wie Netzprotokolle das Ausnutzen der zugrunde liegenden Netzwerk-Hardware einschränken können. Zudem haben wir die Verarbeitungszeiten von Prozessen in den niedrigeren Schichten von Sender und Empfänger vernachlässigt, ebenso wie die Verarbeitungs- und die Warteschlangenverzögerungen, die in den zwischen Sender und Empfänger liegenden Routern auftreten würden. Das Berücksichtigen dieser Effekte würde die Verzögerung weiter vergrößern und die Leistung noch mehr verschlechtern.

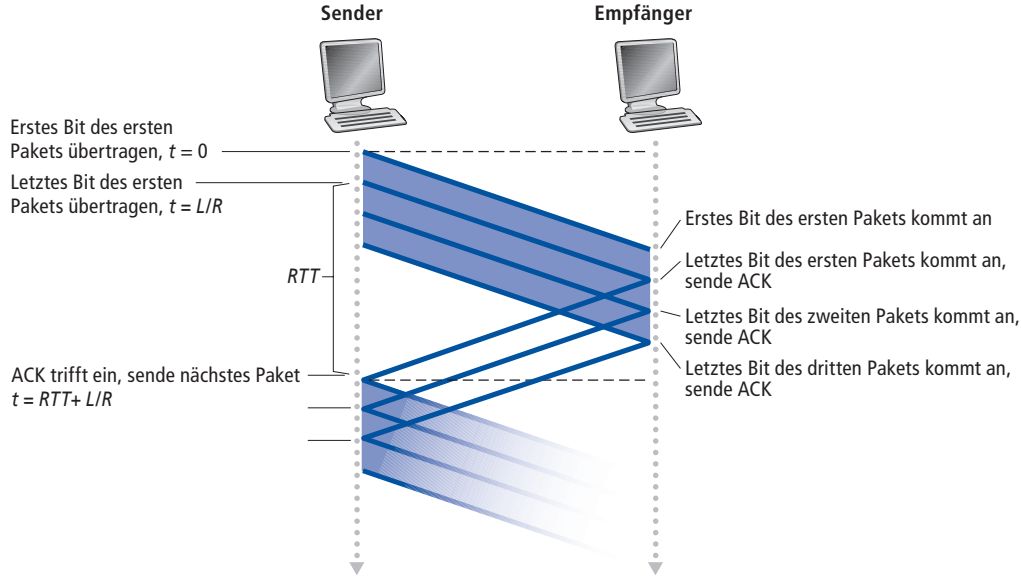
Die Lösung für dieses spezielle Leistungsproblem ist einfach: Anstatt ein Stop-and-Wait-Verfahren zu benutzen, darf der Absender, wie in Abbildung 3.17 (b) gezeigt, mehrere Pakete senden, ohne zwischenzeitlich auf eine Bestätigung warten zu müssen. Wie Abbildung 3.18 (b) zeigt, wird die Auslastung des Senders praktisch verdreifacht, wenn er drei Pakete absenden darf, bevor er auf Bestätigungen warten muss. Weil man sich die vielen Pakete im Transit zwischen Sender und Empfänger wie das Befüllen einer Pipeline vorstellen kann, wird diese Technik als **Pipelining** bezeichnet. Pipelining hat die folgenden Konsequenzen für zuverlässige Datentransferprotokolle:

- Der Wertebereich der Sequenznummern muss vergrößert werden, da jedes Paket im Transit (ohne die Übertragungswiederholungen zu berücksichtigen) eine eindeutige Sequenznummer haben muss und es mehrere unbestätigte Pakete im Transit geben kann.
- Sender- und Empfängerseite des Protokolls müssen unter Umständen mehr als ein Paket zwischenspeichern. Zumindest muss der Absender die Pakete puffern,

die gesendet, aber noch nicht bestätigt worden sind. Auch das Zwischenspeichern korrekt empfangener Pakete beim Empfänger könnte, wie weiter unten noch diskutiert wird, notwendig werden.



a Ablauf bei Stop-and-Wait



b Ablauf bei Pipelining

Abbildung 3.18: Datenübertragung mit Stop-and-Wait sowie Pipelining

- Der Bereich der Sequenznummern und die Anforderungen an das Zwischenspeichern hängen davon ab, wie ein Datentransferprotokoll auf verlorene, verfälschte und übermäßig verzögerte Pakete reagiert. Es gibt zwei grundlegende Möglichkeiten für die Fehlerbehebung in Protokollen mit Pipelining: **Go-Back-N** *(N-Schritte zurück)* und **Selective Repeat** *(Selektive Sendewiederholung)*.

3.4.3 Go-Back-N (GBN)

Bei einem **Go-Back-N-Protokoll** (GBN) darf der Sender mehrere Pakete senden (sofern er welche hat), ohne auf eine Bestätigung zu warten. Er darf aber nicht mehr als eine maximal zulässige Zahl, *N*, von unbestätigten Paketen in der Pipeline haben. Wir werden das GBN-Protokoll in diesem Abschnitt detailliert beschreiben. Aber bevor Sie weiterlesen, möchten wir Sie dazu ermuntern, mit dem GBN-Applet (einem wirklich beeindruckenden Applet!) zu spielen, das Sie auf der Website unseres Buches finden.

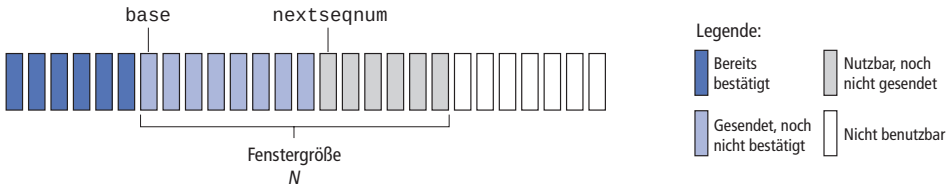


Abbildung 3.19: Sequenznummern aus Sicht des Senders bei Go-Back-N

▶Abbildung 3.19 zeigt die Sichtweise des Senders auf den Sequenznummernbereich in einem GBN-Protokoll. Definieren wir *base* als die Sequenznummer des ältesten unbestätigten Paketes und *nextseqnum* als die kleinste ungenutzte Sequenznummer (also die Sequenznummer des nächsten zu sendenden Paketes), dann können wir in den Sequenznummern vier Bereiche erkennen. Sequenznummern im Intervall [0, *base*–1] entsprechen Paketen, die schon gesendet und bestätigt worden sind. Das Intervall [*base*, *nextseqnum*–1] umfasst Pakete, die zwar gesendet wurden, aber noch nicht bestätigt worden sind. Sequenznummern im Intervall [*nextseqnum*, *base*+*N*–1] können für Pakete genutzt werden, die sofort abgesandt werden dürfen, sollten Daten von der darüberliegenden Schicht eintreffen. Schließlich können Sequenznummern größer oder gleich *base*+*N* nicht benutzt werden, bis ein noch unbestätigtes, derzeit in der Pipeline befindliches Paket (genauer gesagt, das Paket mit der Sequenznummer *base*) bestätigt wurde.

Wie Abbildung 3.19 deutlich macht, kann der Bereich der erlaubten Sequenznummern für gesendete, aber noch nicht bestätigte Pakete als Fenster der Größe *N* über dem Bereich der Sequenznummern betrachtet werden. Während das Protokoll abläuft, schiebt sich dieses Fenster über den Bereich der Sequenznummern vorwärts. Deshalb wird *N* oft als **Fenstergröße** und das GBN-Protokoll selbst als ein **Protokoll mit Schiebefenster** *(sliding window protocol)* bezeichnet. Sie fragen sich vielleicht, warum wir überhaupt die Anzahl von offenen, unbestätigten Paketen auf den Wert von *N* begrenzen. Warum nicht eine unbegrenzte Anzahl solcher Pakete erlauben? Wir

werden in Abschnitt 3.5 sehen, dass ein Grund, eine solche Beschränkung für den Sender festzulegen, die Flusskontrolle ist. Wir werden in Abschnitt 3.7, wenn wir die TCP-Überlastkontrolle studieren, noch einen anderen Grund dafür kennenlernen.

In der Praxis wird die Sequenznummer eines Paketes in einem Feld fester Länge im Paket-Header übertragen. Ist k die Anzahl der Bits im Feld der Sequenznummern des Paketes, dann ist der Bereich der Sequenznummern $[0, 2^k-1]$. Mit einem begrenzten Bereich von Sequenznummern müssen alle arithmetischen Operationen, die Sequenznummern betreffen, modulo-2^k ausgeführt werden. (Das heißt, den Raum der Sequenznummern kann man sich als Ring der Größe 2^k denken, bei dem auf die Sequenznummer 2^k-1 wieder die Sequenznummer 0 folgt.) Erinnern Sie sich daran, dass `rdt3.0` eine 1 Bit-Sequenznummer und einen Bereich von Sequenznummern von [0,1] hatte. Mehrere Übungsaufgaben am Ende dieses Kapitels untersuchen die Folgen eines begrenzten Bereiches von Sequenznummern. Wir werden in Abschnitt 3.5 sehen, dass TCP ein Sequenznummernfeld mit einer Größe von 32 Bit benutzt, wobei es Bytes im übertragenen Datenstrom statt Paketen nummeriert.

▶Abbildung 3.20 und ▶Abbildung 3.21 geben erweiterte FSM-Beschreibungen der Sender- und Empfängerseite eines ACK-basierten, NAK-freien GBN-Protokolls wieder. Wir nennen diese FSM-Beschreibung *erweiterte FSMs*, weil wir Variablen für *base* und *nextseqnum* hinzugefügt haben (die den Variablen in Programmiersprachen entsprechen), dazu auch Operationen auf diesen Variablen und bedingte Aktionen, an denen diese Variablen beteiligt sind. Beachten Sie, dass die erweiterte FSM-Spezifikation gewisse Ähnlichkeiten mit Anweisungen in einer Programmier-

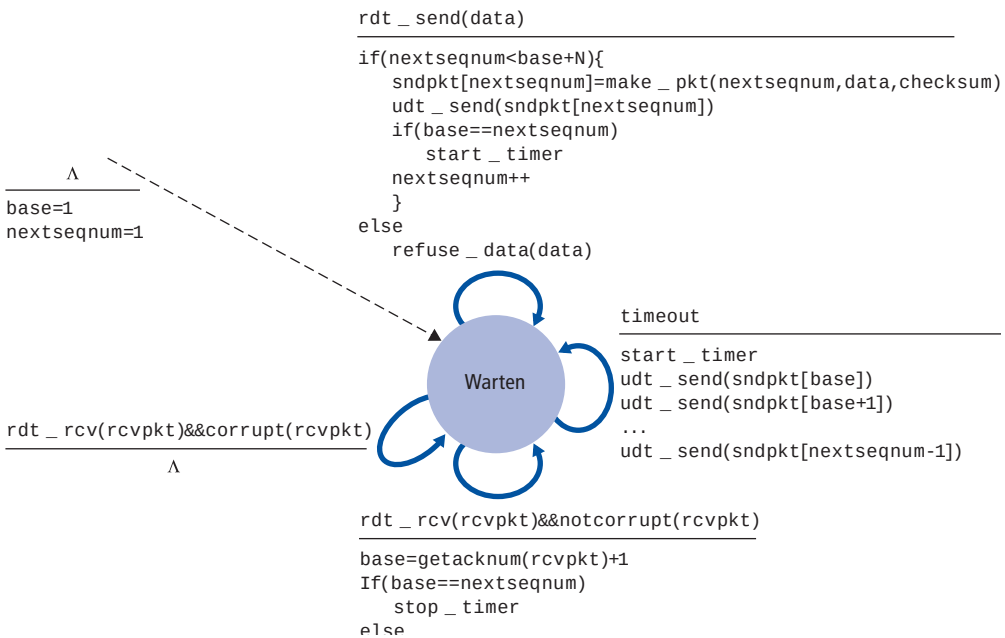


Abbildung 3.20: Erweiterte FSM-Beschreibung des GBN-Senders

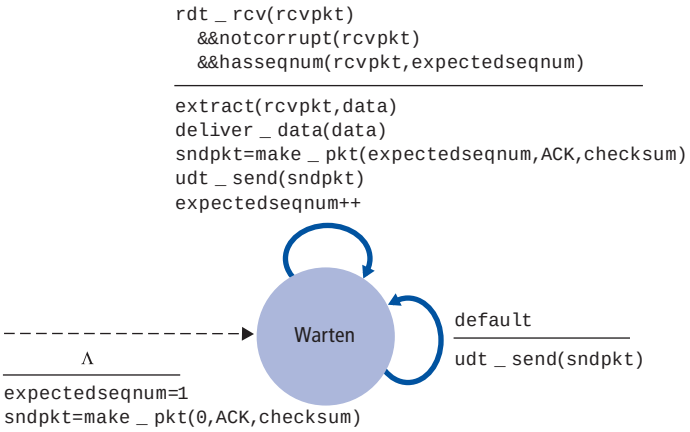


Abbildung 3.21: Erweiterte FSM-Beschreibung des GBN-Empfängers

sprache aufweist. [Bochman 1984] liefert eine exzellente Übersicht über zusätzliche Erweiterungen von FSMs sowie weiterer programmiersprachenbasierter Methoden, um Protokolle zu spezifizieren.

Der GBN-Absender muss auf drei Arten von Ereignissen antworten:

- *Aufrufe von oben.* Wird `rdt_send()` von höheren Schichten aufgerufen, prüft der Sender zuerst, ob das Fenster voll ist, d.h., ob es N unbestätigte Pakete gibt. Ist das Fenster nicht voll, wird ein Paket erzeugt und verschickt und die Variablen werden entsprechend aktualisiert. Ist das Fenster aber bereits gefüllt, gibt der Sender die Daten einfach an die höher liegende Schicht zurück, wodurch implizit mitgeteilt wird, dass das Fenster voll ist. Die obere Schicht wird es dann später nochmals versuchen müssen. In einer echten Implementierung wäre es wahrscheinlicher, dass der Sender die Daten entweder zwischenspeichert (aber nicht sofort sendet) oder einen Mechanismus für die Synchronisation besitzt (zum Beispiel einen Signalgeber oder ein Flag), so dass die obere Schicht `rdt_send()` nur dann aufruft, wenn das Fenster nicht gefüllt ist.
- *Erhalt eines ACK.* In unserem GBN-Protokoll wird eine Bestätigung für ein Paket mit Sequenznummer *n* als **kumulative Bestätigung** *(cumulative acknowledgment)* benutzt, mit der deutlich gemacht wird, dass alle Paketsequenznummern bis einschließlich *n* korrekt beim Empfänger angekommen sind. Wir werden in Kürze auf dieses Thema zurückkommen, wenn wir die Empfängerseite von GBN betrachten.
- *Ein Timeout-Ereignis.* Das Protokoll namens „Go-Back-N“ wird vom Verhalten des Senders beim Auftreten verloren gegangener oder übermäßig verzögerter Pakete abgeleitet. Wie beim Stop-and-Wait-Protokoll wird wieder ein Timer verwendet, um verlorene Daten oder Acknowledgment-Pakete zu reparieren. Tritt ein Timeout auf, sendet der Absender *alle* bereits zuvor abgesandten, aber noch nicht bestätigten Pakete erneut. Unser Sender in Abbildung 3.20 verwendet nur einen einzelnen Timer, den Sie sich als Timer für das älteste übertragene, aber bislang unbestätigte

Paket vorstellen können. Wenn ein ACK empfangen wird, aber es immer noch übertragene, unbestätigte Pakete gibt, wird der Timer neu gestartet. Wenn es keine offenen unbestätigten Pakete mehr gibt, dann wird der Timer gestoppt.

Der Empfänger in GBN ist ebenfalls einfach strukturiert. Wenn ein Paket mit Sequenznummer n korrekt empfangen wird und auch die Reihenfolge der Pakete stimmt (d.h., die zuletzt an die obere Schicht gelieferten Daten kamen von einem Paket mit Sequenznummer $n-1$), dann sendet der Empfänger ein ACK für Paket n und gibt den Datenteil des Paketes an die obere Schicht weiter. In allen anderen Fällen verwirft der Empfänger das Paket und sendet ein ACK für das zuletzt erhaltene und korrekte Paket. Beachten Sie, dass Pakete einzeln an die obere Schicht geliefert werden und daher alle Pakete mit einer Sequenznummer niedriger als k richtig ausgeliefert wurden, sofern Paket k empfangen und nach oben weitergegeben worden ist. Daher ist die Verwendung von kumulativen Bestätigungen eine natürliche Wahl für GBN.

In unserem GBN-Protokoll verwirft der Empfänger Pakete, die in der falschen Reihenfolge eintreffen. Obwohl es albern und verschwenderisch scheint, wenn ein richtig (aber in der falschen Reihenfolge) eingetroffenes Paket verworfen wird, gibt es einen Grund für dieses Vorgehen. Erinnern Sie sich daran, dass der Empfänger die Daten in der richtigen Reihenfolge an die obere Schicht liefern muss. Nehmen Sie an, Paket n wird erwartet, aber Paket $n+1$ kommt an. Weil die Daten in der richtigen Reihenfolge abgeliefert werden müssen, könnte der Empfänger Paket $n+1$ zwischenspeichern und es später, nachdem er auch Paket n empfangen hat, an die höherliegende Schicht weitergeben.

Geht das Paket n verloren, wird wegen der Arbeitsweise der Übertragungswiederholung in GBN jedoch vermutlich auch Paket $n+1$ nochmals vom Sender übertragen. Daher kann der Empfänger auch einfach Paket $n+1$ verwerfen. Der Vorteil dieses Ansatzes ist die Einfachheit des Pufferns beim Empfänger – der Empfänger muss kein Paket zwischenspeichern, das außerhalb der korrekten Reihenfolge eintrifft. Während der Sender also die obere und untere Grenze seines Fensters und die Position von *nextseqnum* innerhalb dieses Fensters speichern muss, ist die einzige Information, die der Empfänger sich merken muss, die Sequenznummer des nächsten Paketes in der Reihenfolge. Dieser Wert wird, wie in ▶Abbildung 3.21 gezeigt, in der Variablen *expectedseqnum* der Empfänger-FSM gespeichert. Natürlich liegt der Nachteil der Vorgehensweise, ein eigentlich korrekt erhaltenes Paket wegzuwerfen, darin, dass es bei der anschließenden Übertragungswiederholung verloren gehen oder verfälscht werden könnte und auf diese Art noch mehr Übertragungswiederholungen erforderlich werden.

▶Abbildung 3.22 zeigt die Arbeitsweise des GBN-Protokolls für den Fall einer Fenstergröße von vier Paketen. Wegen dieser Beschränkung der Fenstergröße sendet der Absender die Pakete 0 bis 3, muss dann aber warten, bis ein oder mehrere dieser Pakete bestätigt wurden, bevor er weitersenden kann. Während die einzelnen aufeinanderfolgenden ACKs (zum Beispiel ACK0 und ACK1) empfangen werden, wird das Fenster weitergeschoben und der Sender darf ein neues Paket (Paket 4 beziehungs-

weise Paket 5) senden. Auf dem Weg zum Empfänger geht aber Paket 2 verloren und daher sind die Pakete 3, 4 und 5 nicht mehr in der richtigen Reihenfolge und werden verworfen.

Bevor wir unsere Diskussion von GBN abschließen, sollten wir anmerken, dass eine Implementierung dieses Protokolls in einem Protokollstapel sehr wahrscheinlich eine Struktur hätte, die der erweiterten FSM in Abbildung 3.20 ähneln würde. Die Implementierung würde wohl auch die Form verschiedener Prozeduren haben, die als Antwort auf die verschiedenen möglichen Ereignisse die entsprechenden Aktionen durchführen. Bei dieser **ereignisbasierten Programmierung** werden die verschiedenen Funktionen entweder von anderen Funktionen im Protokollstapel oder infolge eines Interrupts *(Unterbrechung)* aufgerufen. Beim Absender wären diese Ereignisse (1) ein Aufruf aus der höheren Schicht, um `rdt_send()` zu starten, (2) ein Timer-Interrupt und (3) ein Aufruf von `rdt_rcv()` durch die niedrigere Schicht, wenn ein Paket ankommt. Die Programmierübungen am Ende dieses Kapitels geben Ihnen Gelegenheit, diese Routinen tatsächlich in einer simulierten, aber realistischen Netzwerkumgebung zu implementieren.

Wir möchten hier anmerken, dass das GBN-Protokoll schon fast alle Techniken beinhaltet, denen wir begegnen werden, wenn wir die zuverlässigen Datentransferkomponenten von TCP in Abschnitt 3.5 studieren. Diese Techniken beinhalten die Verwendung von Sequenznummern, kumulative Bestätigungen, Prüfsummen und eine Operation für Timeouts und erneutes Übertragen.

3.4.4 Selective Repeat (SR)

Das GBN-Protokoll ermöglicht es dem Absender in Abbildung 3.17, die Pipeline mit Paketen zu füllen, um auf diese Art die Kanalauslastungsprobleme zu vermeiden, denen wir bei Stop-and-Wait begegnet sind. Es gibt jedoch Szenarien, in denen GBN selbst Leistungsprobleme aufweist. Insbesondere, wenn sowohl die Fenstergröße als auch das Produkt von Bandbreite und Verzögerung groß sind, können viele Pakete in der Pipeline sein. Ein einzelner Paketfehler kann daher bewirken, dass GBN eine sehr große Anzahl von Paketen erneut überträgt, viele davon unnötigerweise. Steigt die Wahrscheinlichkeit von Übertragungsfehlern, kann die Pipeline mit diesen unnötigen Übertragungswiederholungen gefüllt sein. Stellen Sie sich in unserem Nachrichten-Diktat-Szenario vor, dass jedes Mal, wenn ein Wort fehlerhaft wäre, die umliegenden 1.000 Wörter (z.B. bei einer Fenstergröße von 1.000 Wörtern) wiederholt werden müssten. Das Diktat würde wegen all der wiederholten Wörter äußerst langsam voranschreiten.

Wie der Name schon andeutet, vermeiden Selective-Repeat-Protokolle unnötige Übertragungswiederholungen, indem sie nur jene Pakete nochmals vom Sender übertragen lassen, von denen sie vermuten, dass sie Fehler hatten (d.h., sie sind verlorengegangen oder wurden korrumpiert). Diese individuell wiederholten Übertragungen erfolgen nur, wenn sie notwendig werden. Dies erfordert, dass der Empfänger die richtig erhaltenen Pakete einzeln bestätigt. Eine Fenstergröße von N wird auch hier wieder verwendet, um die Anzahl offener, unbestätigter Pakete in der Pipeline zu beschränken.

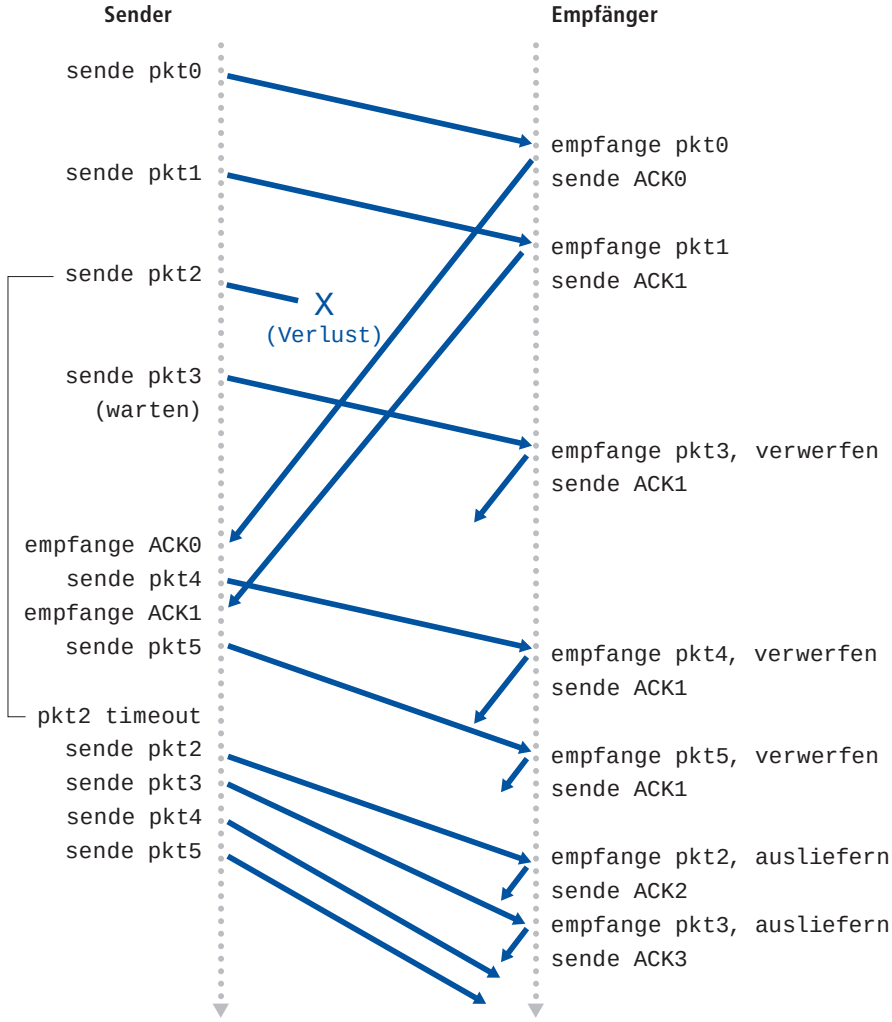


Abbildung 3.22: Arbeitsweise von Go-Back-N

Jedoch kann der Sender im Gegensatz zu GBN schon ACKs für einige der Pakete im Fenster erhalten haben. ▶Abbildung 3.23 zeigt die Sicht des Selective-Repeat-Senders auf den Bereich der Sequenznummern. ▶Abbildung 3.24 zeigt detailliert die verschiedenen Aktionen des SR-Senders.

Der SR-Empfänger bestätigt ein richtig erhaltenes Paket, ungeachtet dessen, ob es in der richtigen Reihenfolge eingetroffen ist. Pakete außerhalb der Reihe werden zwischengespeichert, bis die fehlenden Pakete (das heißt, Pakete mit niedrigeren Sequenznummern) empfangen werden, wonach ein Schub von Paketen – in der richtigen Reihenfolge – an die obere Schicht geliefert werden kann. ▶Abbildung 3.25 verdeutlicht die vom SR-Empfänger ergriffenen Maßnahmen. ▶Abbildung 3.26 zeigt ein Beispiel der SR-Operation im Falle von verlorenen Paketen. Beachten Sie, dass in

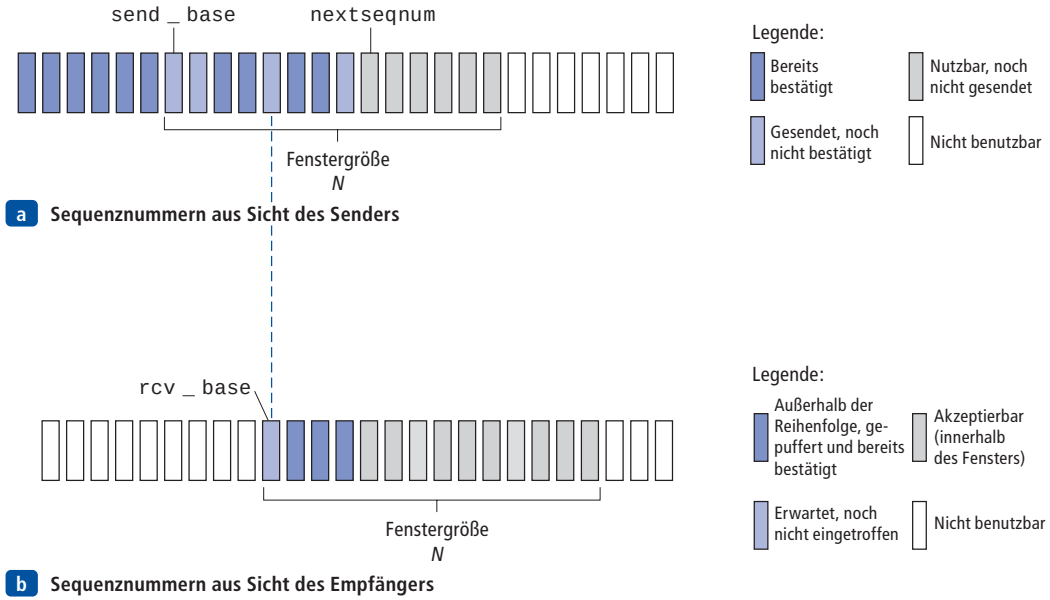


Abbildung 3.23: Sender- und Empfängersicht des Bereiches der Sequenznummern bei Selective Repeat (SR)

Abbildung 3.26 der Empfänger anfangs die Pakete 3, 4 und 5 puffert und sie, als Paket 2 schließlich empfangen wurde, zusammen mit Paket 2 an die obere Schicht weiterleitet.

1. *Daten von oben erhalten.* Wenn Daten von oben empfangen werden, bestimmt der SR-Sender die nächste verfügbare Sequenznummer für das Paket. Wenn die Sequenznummer innerhalb des Fensters des Senders ist, werden die Daten in ein Paket verpackt und gesendet; ansonsten werden sie entweder gepuffert oder, wie in GBN, an die obere Schicht zurückgegeben.
2. *Timeout.* Timer werden auch hier gegen verloren gegangene Pakete eingesetzt. Jedoch muss jedes Paket nun seinen eigenen logischen Timer haben, weil nur ein einzelnes Paket beim Timeout gesendet wird. Ein einzelner Hardware-Timer kann verwendet werden, um die Operation von mehreren logischen Timern zu simulieren [Varghese 1997].
3. *Eintreffen eines ACK.* Wird ein ACK empfangen, kennzeichnet der SR-Sender dieses Paket als bestätigt, vorausgesetzt, es befindet sich im Fenster. Ist die Sequenznummer des Paketes gleich send_base, wird send_base zu dem unbestätigten Paket weitergeschoben, das die kleinste Sequenznummer besitzt. Bewegt sich das Fenster und es gibt noch nicht übertragene Pakete mit Sequenznummern, die jetzt innerhalb des Fensters liegen, werden diese Pakete gesendet.

Abbildung 3.24: Ereignisse und Aktionen im SR-Sender

1. *Paket mit Sequenznummer im Intervall* [*rcv_base*, *rcv_base* + *N* – 1] *wird fehlerfrei empfangen.* In diesem Fall fällt das eingetroffene Paket in das Fenster des Empfängers und ein selektives ACK-Paket wird an den Absender zurückgesandt. Wurde das Datenpaket zuvor noch nicht empfangen, wird es gepuffert. Hat das Paket eine Sequenznummer gleich der Basis des Empfangsfensters (*rcv_base* in ▶Abbildung 3.22), dann wird dieses Paket die höhere Schicht weitergegeben. Danach wird geprüft, ob zuvor gepufferte Pakete jetzt an die höhere Schicht ausgeliefert werden können. Das Empfangsfenster wird dann um die Anzahl der an die obere Schicht ausgelieferten Pakete weiterbewegt. Als Beispiel betrachten Sie ▶Abbildung 3.26. Wenn ein Paket mit einer Sequenznummer von *rcv_base* = 2 empfangen wird, kann es, zusammen mit den Paketen 3, 4 und 5 der oberen Schicht zugestellt werden.
2. *Paket mit Sequenznummer im Intervall* [*rcv_base* – *N*, *rcv_base* – 1] *wird fehlerfrei empfangen.* In diesem Fall muss ein ACK erzeugt werden, obwohl dies ein Paket ist, das der Empfänger bereits zuvor bestätigt hat.
3. *Sonst.* Das Paket wird ignoriert.

Abbildung 3.25: Ereignisse und Aktionen im SR-Empfänger

Es ist wichtig anzumerken, dass beim zweiten Schritt in Abbildung 3.25 der Empfänger schon erhaltene Pakete mit bestimmten Sequenznummern unterhalb des aktuellen Fensters erneut bestätigt (anstatt sie stillschweigend zu ignorieren). Sie sollten sich klarmachen, dass diese erneute Bestätigung wirklich erforderlich ist. Wird z.B. bei den Sequenznummerbereichen für Sender und Empfänger aus Abbildung 3.23 kein ACK für das Paket *send_base*, das sich auf dem Weg vom Sender zum Empfänger befindet, geschickt, dann wird der Sender das Paket *send_base* schließlich erneut senden, obwohl klar ist (für uns, nicht für den Sender!), dass der Empfänger das Paket schon erhalten hat.

Würde der Empfänger dieses Paket nicht bestätigen, würde sich das Fenster des Absenders nie vorwärts bewegen! Dieses Beispiel verdeutlicht einen wichtigen Aspekt von SR-Protokollen (und auch vielen anderen Protokollen). Sender und Empfänger werden nicht immer den gleichen Blick auf das haben, was korrekt empfangen worden ist und was nicht. Für SR-Protokolle bedeutet dies, dass die Sender- und Empfängerfenster nicht immer zusammenfallen.

Der Mangel an Synchronisierung zwischen Sender- und Empfängerfenstern hat wichtige Folgen, wenn wir mit der Realität eines begrenzten Bereiches von Sequenznummern konfrontiert sind. Überlegen Sie, was z.B. bei einem endlichen Bereich von vier Paketsequenznummern – 0, 1, 2, 3 – und einer Fenstergröße von drei geschehen würde. Nehmen Sie an, dass die Pakete 0 bis 2 übertragen wurden, richtig beim Empfänger eingetroffen sind und bestätigt wurden. An dieser Stelle ist das Fenster des Empfängers über dem vierten, fünften und sechsten Paket, welche die Sequenznummern 3, 0 und 1 tragen. Jetzt stellen Sie sich zwei Szenarien vor. Im ersten, dargestellt

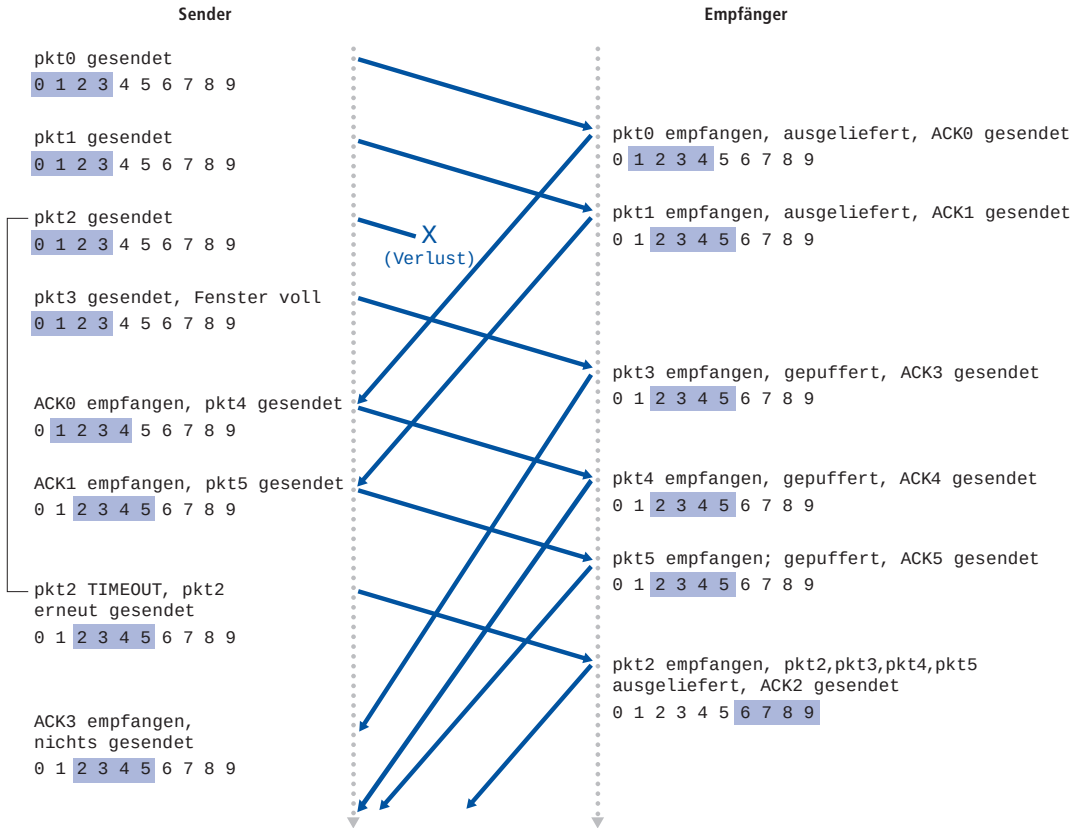


Abbildung 3.26: SR-Arbeitsweise

in ▶Abbildung 3.27 (a), gehen die ACKs der ersten drei Pakete verloren und der Absender überträgt diese Pakete erneut. Der Empfänger erhält auf diese Art ein Paket mit Sequenznummer 0 – eine Kopie des ersten übertragenen Paketes.

Im zweiten Szenario, dargestellt in Abbildung 3.27 (b), werden die ACKs der ersten drei Pakete alle richtig abgeliefert. Der Absender bewegt daher sein Fenster vorwärts und sendet das vierte, fünfte und sechste Paket mit den Sequenznummern 3, 0 und 1. Das Paket mit Sequenznummer 3 geht verloren, aber das Paket mit Sequenznummer 0 kommt an – ein Paket, das *neue* Daten enthält.

Jetzt nehmen Sie den Standpunkt des Empfängers in Abbildung 3.27 ein. Ein sprichwörtlicher Vorhang zwischen Sender und Empfänger verhindert, dass der Empfänger sehen kann, welche Maßnahmen vom Sender ergriffen wurden. Alles, was der Empfänger beobachtet, ist die Abfolge der Nachrichten, die er vom Kanal erhält und in den Kanal sendet. Soweit er betroffen ist, sind die beiden Szenarien in Abbildung 3.27 *identisch*. Es gibt keinen Weg, die wiederholte Übertragung des ersten Paketes von einer Originalübertragung des fünften Paketes zu unterscheiden. Offensichtlich funktioniert also eine Fenstergröße, die um eins geringer ist als der Bereich der Sequenznummern,

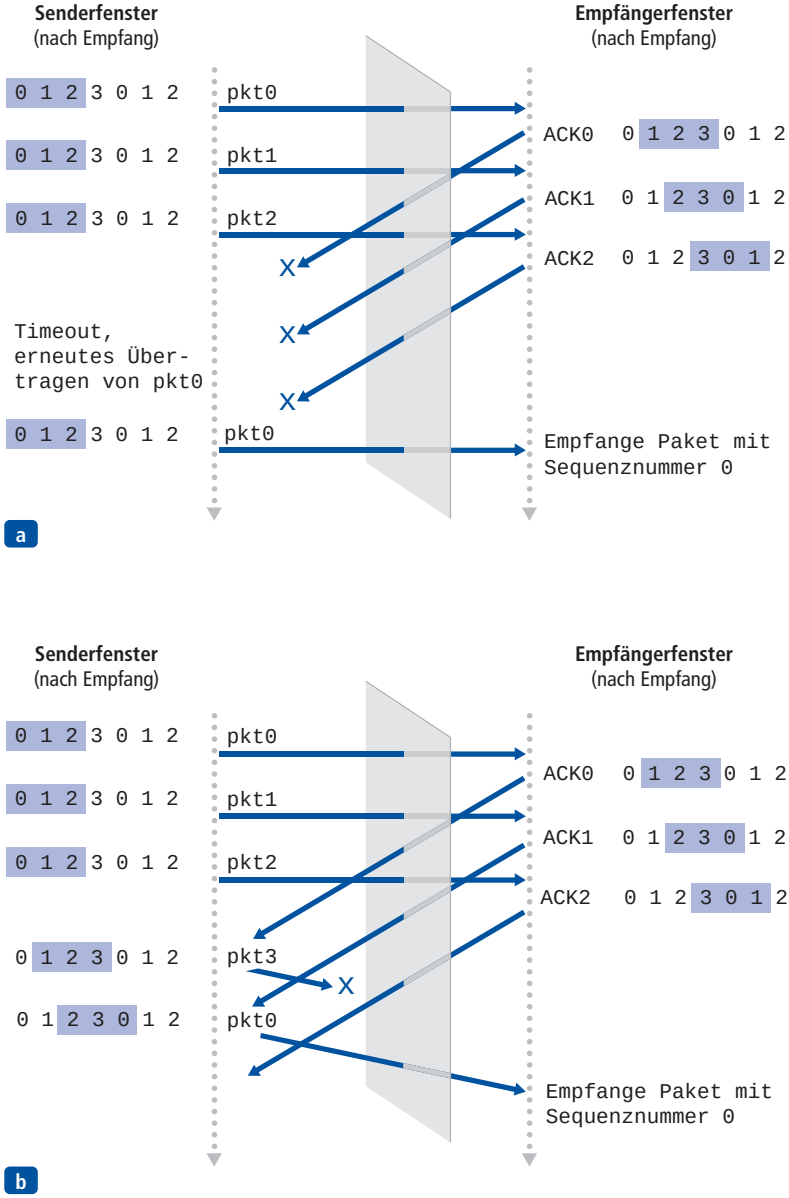


Abbildung 3.27: Dilemma des SR -Empfängers bei zu großen Fenstern: neues Paket oder Übertragungswiederholung?

nicht. Aber wie groß muss das Fenster sein? In einer Übungsaufgabe am Ende des Kapitels sollen Sie zeigen, dass bei SR-Protokollen die Fenstergröße kleiner oder gleich der Hälfte des Sequenznummernbereiches sein muss.

Auf der Webseite zum Buch finden Sie ein Applet, das die Operation des SR-Protokolls anschaulich macht. Versuchen Sie, dieselben Versuche durchzuführen, die Sie

mit dem GBN-Applet gemacht haben. Stimmen die Ergebnisse mit dem überein, was Sie erwarten?

Dies beendet unsere Diskussion zuverlässiger Datenübertragungsprotokolle. Wir haben viele Themenbereiche betrachtet und zahlreiche Mechanismen eingeführt, die zusammen einen zuverlässigen Datentransfer ermöglichen. ▶Tabelle 3.1 fasst diese Mechanismen zusammen. Nun, da wir sie alle in Aktion gesehen haben und in der Lage sind, das Gesamtbild zu erkennen, möchten wir Sie dazu anregen, diesen Abschnitt erneut zu lesen. Sie sollen erkennen, wie diese Techniken schrittweise zusammengefügt wurden, um immer komplexere (und realistischere) Modelle des Kanals zu unterstützen, der Absender und Empfänger verbindet, sowie um die Leistung der Protokolle zu steigern.

Mechanismus	Einsatzzweck, Kommentare
Prüfsumme	Wird verwendet, um Bitfehler in einem gesendeten Paket zu erkennen.
Timer	Wird verwendet, um ein Paket nochmals zu übertragen, möglicherweise weil das Paket (oder das zugehörige ACK) auf dem Kanal verloren ging. Weil Timeouts auftreten können, wenn ein Paket verzögert wird, aber nicht verloren geht (frühzeitiger Timeout), oder weil ein Paket beim Empfänger eingetroffen sein kann, aber das ACK verloren ging, können doppelte Kopien eines Paketes beim Empfänger ankommen.
Sequenznummer	Wird für die fortlaufende Nummerierung von Datenpaketen verwendet, die vom Sender zum Empfänger laufen. Lücken in den Sequenznummern der erhaltenen Pakete erlauben es dem Empfänger, ein verlorenes Paket zu erkennen. Pakete mit doppelten Sequenznummern ermöglichen es dem Empfänger, doppelte Kopien eines Paketes zu erkennen.
Acknowledgment (ACK)	Wird vom Empfänger verwendet, um dem Absender mitzuteilen, dass ein Paket oder ein Satz von Paketen richtig empfangen worden ist. Acknowledgments tragen normalerweise die Sequenznummer des Paketes oder der Pakete, die bestätigt werden. Acknowledgments können je nach Protokoll einzeln oder kumulativ sein.
Negatives Acknowledgment (NAK)	Wird vom Empfänger verwendet, um dem Absender mitzuteilen, dass ein Paket nicht richtig empfangen wurde. Ein negatives Acknowledgment enthält normalerweise die Sequenznummer des Paketes, das nicht richtig empfangen wurde.
Pipelining, Sendefenster	Der Sender darf mehrere unbestätigte Pakete senden, deren Sequenznummern innerhalb eines gegebenen Bereiches, dem Sendefenster, liegen müssen. Indem mehrere Pakete gesendet werden dürfen, aber noch nicht bestätigt werden müssen, kann die Auslastung im Vergleich zum Stop-and-Wait-Ansatz gesteigert werden. Wir werden später sehen, dass die Fenstergröße von verschiedenen Parametern beeinflusst wird, etwa der Fähigkeit des Empfängers, Nachrichten zu empfangen und zu puffern oder der (Über-)Lastsituation im Netz.

Tabelle 3.1: Zusammenfassung von Mechanismen für die zuverlässige Datenübertragung und ihrer Verwendung

Beenden wir unsere Diskussion der zuverlässigen Datentransferprotokolle, indem wir eine verbleibende Annahme in unser zugrunde liegendes Kanalmodell einfügen. Erinnern Sie sich daran, dass wir davon ausgegangen sind, dass Pakete nicht innerhalb des Kanals zwischen Sender und Empfänger umsortiert werden können. Dies ist im Allgemeinen eine vernünftige Voraussetzung, besonders dann, wenn Sender und Empfänger durch eine einzelne physikalische Leitung verbunden sind. Ist jedoch der „Kanal", der die beiden verbindet, ein komplexes Netzwerk, dann können Pakete umgeordnet werden. Eine Auswirkung umgeordneter Pakete besteht darin, dass alte Kopien eines Paketes mit einer Sequenznummer oder einer Acknowledgment-Nummer von *x* auftauchen können, obwohl weder das Sender- noch das Empfängerfenster *x* enthält. Durch das Umordnen der Pakete erscheint der Kanal, als ob er im Wesentlichen Pakete puffert und spontan zu einem beliebigen Zeitpunkt in der Zukunft wieder ausspuckt. Weil Sequenznummern wiederverwendet werden können, muss besondere Sorgfalt darauf verwandt werden, sich vor solchen doppelten Paketen zu schützen. Bei dem in der Praxis durchgeführten Ansatz wird sichergestellt, dass eine Sequenznummer nicht wieder verwendet wird, bis der Absender „sicher" ist, dass zuvor gesandte Pakete mit der Sequenznummer *x* nicht länger im Netz sind. Dies erfolgt durch die Annahme, dass ein Paket höchstens eine gewisse feste Zeitspanne im Netzwerk überleben kann. Bei TCP für Hochgeschwindigkeitsnetze wird eine maximale Paketlebensdauer von etwa drei Minuten angenommen [RFC 1323]. [Sunshine 1978] beschreibt eine Methode für die Verwendung von Sequenznummern, welche das Problem des Umordnens völlig vermeidet.

3.5 Verbindungsorientierter Transport: TCP

Nun, da wir die zugrunde liegenden Prinzipien des zuverlässigen Datentransfers behandelt haben, wenden wir uns TCP zu – das verbindungsorientierte, zuverlässige Transportprotokoll des Internets. In diesem Abschnitt werden wir sehen, dass TCP, um zuverlässigen Datentransfer anbieten zu können, auf vielen der Grundlagen aufbaut, die wir im vorherigen Abschnitt erörtert haben, etwa Fehlererkennung, Übertragungswiederholungen, kumulative Bestätigungen, Timer und Header-Felder für Sequenznummern und Acknowledgment-Nummern. TCP wird in RFC 793, RFC 1122, RFC 1323, RFC 2018 und RFC 2581 definiert.

3.5.1 Die TCP-Verbindung

TCP heißt **verbindungsorientiert**, weil zwei Prozesse zunächst einen „Handshake" durchführen müssen, bevor ein Anwendungsprozess beginnen kann, Daten zum anderen zu senden – d.h., sie müssen zu Beginn eine Reihe von Segmenten austauschen, um die Parameter des folgenden Datentransfers auszuhandeln. Als Teil des TCP-Verbindungsaufbaus initialisieren beide Seiten der Verbindung mehrere TCP-Zustandsvariablen dieser Verbindung (von denen viele in diesem Abschnitt und in Abschnitt 3.7 erörtert werden).

Die TCP-„Verbindung“ ist keine durchgehende TDM- oder FDM-Leitung wie in einem leitungsvermittelten Netzwerk. Es handelt sich auch nicht um eine virtuelle Leitung (siehe Kapitel 1), weil der Zustand der Verbindung ausschließlich in den beiden Endsystemen gehalten wird. Da das TCP-Protokoll nur auf den Endsystemen läuft und nicht in den dazwischen liegenden Netzwerkelementen (Router und Switches der Sicherungsschicht), halten die Netzwerkelemente keinen TCP-Verbindungsstatus. In der Tat sind sich die zwischengeschalteten Router der TCP-Verbindungen gar nicht bewusst; sie sehen Datagramme, nicht Verbindungen.

Eine TCP-Verbindung bietet einen **Vollduplexdienst**: Besteht eine TCP-Verbindung zwischen Prozess A auf einem Host und Prozess B auf einem anderen Host, dann können Anwendungsschichtdaten sowohl von Prozess A zu Prozess B als auch von Prozess B zu Prozess A fließen. Eine TCP-Verbindung ist außerdem immer eine Punkt-zu-Punkt-Verbindung, besteht also zwischen einem einzelnen Sender und einem einzelnen Empfänger. Sogenanntes „Multicasting“ (siehe Abschnitt 4) – der Datentransfer von einem Sender zu vielen Empfängern in einer einzelnen Sendeoperation – ist mit TCP nicht möglich. Bei TCP sind drei Hosts einer zu viel!

Werfen wir nun einen Blick darauf, wie eine TCP-Verbindung aufgebaut wird. Nehmen Sie an, dass ein Prozess, der auf einem Host läuft, eine Verbindung zu einem anderen Prozess auf einem anderen Host initiieren will. Erinnern Sie sich daran, dass der Prozess, der die Verbindung initiiert, als *Client-Prozess* und der andere Prozess als Server-Prozess bezeichnet wird. Der Client-Anwendungsprozess informiert die Client-Trans-

Fallstudie

Vinton Cerf, Robert Kahn und TCP/IP

In den frühen 1970ern wuchs die Zahl der Paketvermittlungsnetze stark an, wobei das ARPAnet – der Vorläufer des Internets – nur eines von vielen Netzen war. Jedes dieser Netze hatte sein eigenes Protokoll. Zwei Forscher, Vinton Cerf und Robert Kahn, erkannten die Wichtigkeit, diese Netze zusammenzuschalten, und entwickelten ein netzwerkübergreifendes Protokoll, das sie TCP/IP (für Transmission Control Protocol/Internet Protocol) nannten. Während Cerf und Kahn ihr Protokoll zunächst als eine Einheit betrachteten, wurden später die Bestandteile TCP und IP getrennt. Cerf und Kahn veröffentlichten im Mai 1974 in den IEEE Transactions on Communications Technology [Cerf 1974] einen Artikel über TCP/IP.

Das TCP/IP-Protokoll, die Basis des heutigen Internets, wurde vor den modernen PCs und Workstations entworfen, bevor Ethernet und lokale Netzwerke Verbreitung gefunden hatten, vor dem Web, Audio-/Video-Streaming und Chat. Cerf und Kahn erkannten die Notwendigkeit eines Netzwerkprotokolls, das einer Vielzahl von noch zu definierenden Anwendungen eine solide Basis bot, und gleichzeitig beliebige Kombinationen von Hosts und Sicherungsschichtprotokollen ermöglichte.

2004 erhielten Cerf und Kahn den ACM Turing Award, den „Nobelpreis der Informatik“, für ihre „Pionierarbeiten im Internetworking, einschließlich des Entwurfs und der Implementation der grundlegenden Kommunikationsprotokolle des Internets, TCP/IP, und für ihre inspirierte Führungsrolle im Bereich Computernetzwerke“.

portschicht zunächst, dass er eine Verbindung zu einem Prozess im Server einleiten will. Erinnern Sie sich an Abschnitt 2.7: Ein Java Client-Programm erreicht dies durch den Befehl

```
Socket clientSocket = new Socket("hostname", portNumber);
```

wobei `hostname` der Name des Servers ist und `portNumber` den Prozess auf dem Server festlegt. Die Transportschicht auf dem Client fährt dann fort, eine TCP-Verbindung mit der TCP-Instanz auf dem Server einzuleiten. Am Ende dieses Abschnittes erörtern wir detailliert das Verfahren, mit dem die Verbindung aufgebaut wird. Jetzt genügt uns zu wissen, dass der Client zuerst ein spezielles TCP-Segment sendet. Der Server antwortet mit einem zweiten speziellen TCP-Segment und der Client antwortet zuletzt wieder mit einem dritten speziellen Segment. Die ersten beiden Segmente tragen keine Nutzlast, d.h. keine Anwendungsschichtdaten. Das dritte dieser Segmente kann Nutzlast tragen, muss es aber nicht. Weil drei Segmente zwischen beiden Hosts ausgetauscht werden, wird dieses Verfahren zum Eröffnen einer Verbindung oft **Drei-Wege-Handshake** *(three-way handshake)* genannt.

Ist einmal eine TCP-Verbindung hergestellt, können beide Anwendungsprozesse einander Daten senden. Lassen Sie uns das Senden von Daten vom Client-Prozess an den Server-Prozess betrachten. Der Client-Prozess überträgt einen Datenstrom durch den Socket (die Tür des Prozesses), wie in Abschnitt 2.7 beschrieben. Sobald die Daten durch diese Tür gehen, sind sie in Händen des auf dem Client laufenden TCP. Wie ▶Abbildung 3.28 zeigt, packt TCP diese Daten in den **Sendepuffer** der Verbindung, einen der Puffer, die während des anfänglichen Drei-Wege-Handshakes reserviert wurden. Von Zeit zu Zeit holt TCP Teile der Daten aus dem Sendepuffer. Interessanterweise ist die TCP-Spezifikation [RFC 793] sehr zurückhaltend hinsichtlich der Zeitpunkte, zu denen TCP gepufferte Daten senden sollte und legt nur fest, dass TCP „diese Daten in Segmenten nach eigenem Gutdünken senden“ sollte. Die Datenmenge, die auf einmal in ein Segment gepackt werden darf, wird durch die **maximale Segmentgröße** (**MSS**, *maximum segment size*) beschränkt. Die *MSS* wird normalerweise gesetzt, indem zuerst die Länge des größtmöglichen Rahmens der Sicherungsschicht, der vom lokalen sendenden Host ausgesandt werden kann, bestimmt wird (die sogenannte **maximale Übertragungseinheit**, **MTU**, *maximum transmission unit*).

Danach wird die *MSS* so gesetzt, dass ein TCP-Segment (nach dem Verkapseln in einem IP-Datagramm) in einen einzelnen Rahmen der Sicherungsschicht passt. Häufig benutzte Werte für die MTU sind 1.460 Byte, 536 Byte und 512 Byte. Es gab auch Vorschläge, wie die Pfad-MTU bestimmt werden könnte – der größte Rahmen der Sicherungsschicht, der über alle Verbindungen zwischen Quelle und Ziel versendet werden kann [RFC 1191] – und wie die *MSS* basierend auf dieser Pfad-MTU gewählt werden kann. Beachten Sie, dass die *MSS* die maximale Menge an Anwendungsschichtdaten pro Segment ist, nicht aber die Maximalgröße des TCP-Segmentes einschließlich der Header. (Diese Terminologie ist verwirrend, wir müssen aber mit ihr leben, da sie sich eingebürgert hat.)

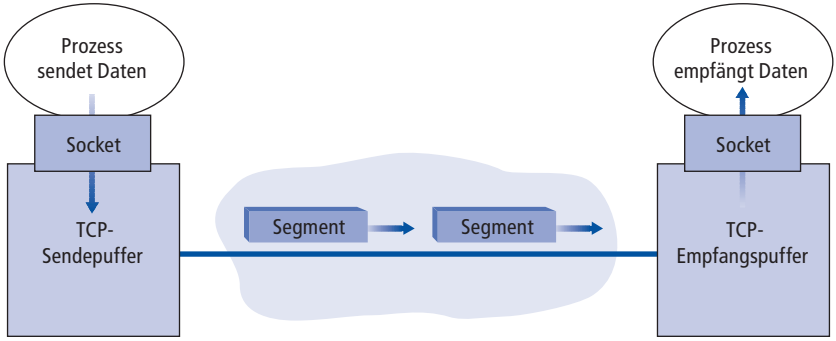


Abbildung 3.28: Puffer im TCP-Sender und -Empfänger

TCP ergänzt jeden Block von Client-Daten um einen TCP-Header und bildet dadurch **TCP-Segmente**. Die Segmente werden zur Netzwerkschicht hinuntergereicht, in der sie dann in Netzwerkschicht-IP-Datagramme verkapselt werden. Diese IP-Datagramme werden dann ins Netz gesandt. Sobald TCP am anderen Ende der Verbindung ein Segment erhält, werden die Daten des Segmentes in den Eingangspuffer der zugehörigen TCP-Verbindung eingefügt, wie ▶Abbildung 3.28 zeigt. Die Anwendung liest den Datenstrom aus diesem Puffer. Jede Seite der Verbindung besitzt ihren eigenen Sende- und ihren eigenen Empfangspuffer. (Betrachten Sie das Applet zur Online-Flusskontrolle auf *http://www.awl.com/kurose-ross*, das eine Animation der Sende- und Empfangspuffer enthält.)

Wir ersehen aus dieser Diskussion, dass eine TCP-Verbindung aus zwei Hälften besteht: Puffer, Variablen und eine Socket-Verbindung zu einem Prozess in einem Host sowie einem weiteren Satz dieser Elemente im anderen Host. Wie bereits erwähnt, sind keinerlei Puffer oder Variablen in den Komponenten im Inneren des Netzwerkes (den Routern und Switches der Sicherungsschicht) mit der Verbindung zwischen den Hosts assoziiert.

3.5.2 TCP-Segmentstruktur

Nachdem wir einen Blick auf die TCP-Verbindung geworfen haben, wollen wir nun die Struktur der TCP-Segmente betrachten. Das TCP-Segment besteht aus Header-Feldern und einem Datenfeld. Letzteres enthält einen Teil der Anwendungsdaten. Wie eben erwähnt, begrenzt die *MSS* die Maximalgröße des Datenfelds eines Segments. Wenn TCP eine große Datei, etwa eine Abbildung auf einer Webseite, sendet, zerlegt es normalerweise die Datei in Stücke der Größe *MSS* (außer dem letzten Stück, das in der Regel kleiner als die *MSS* ist). Interaktive Anwendungen senden jedoch oft Datenstücke, die kleiner als die *MSS* sind; zum Beispiel enthält beim Remote-Login mit Telnet das Datenfeld im TCP-Segment oft nur ein Byte. Weil der TCP-Header normalerweise 20 Byte groß ist (12 Byte mehr als der UDP-Header), sind von Telnet verschickte Segmente durchaus manchmal nur 21 Byte groß.

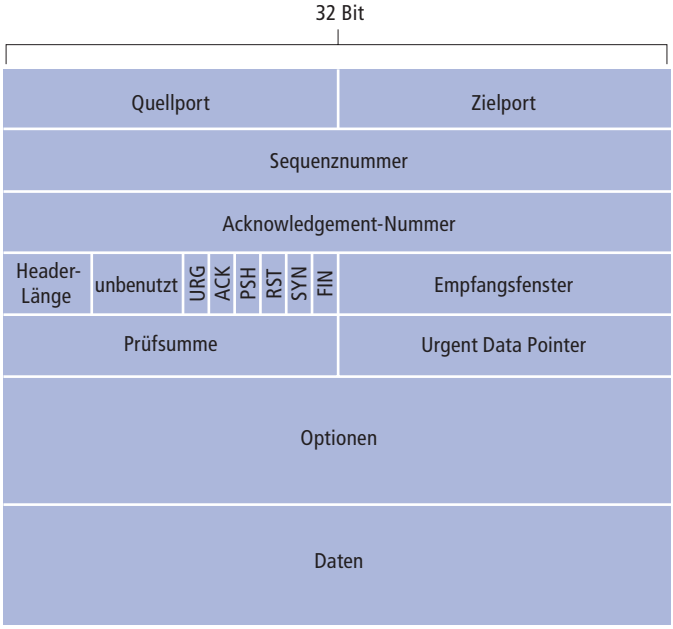


Abbildung 3.29: TCP-Segmentstruktur

▶Abbildung 3.29 zeigt die Struktur des TCP-Segmentes. Wie bei UDP beinhaltet der Header **Quell- und Zielportnummern**, die für das Multiplexing/Demultiplexing der Daten von/zu Anwendungen der nächsthöheren Schicht verwendet werden. Ebenfalls wie bei UDP enthält der Header ein **Prüfsummenfeld**. Ein TCP-Segment-Header enthält zudem die folgenden Felder:

- Das 32-Bit-**Sequenznummernfeld** *(sequence number field)* und das 32-Bit-**Acknowledgment-Nummern-Feld** (*acknowledgment number field)* werden vom TCP-Sender und Empfänger verwendet, um den – später diskutierten – zuverlässigen Datentransferdienst zu realisieren.
- Das 16-Bit-**Empfangsfenster-Feld** *(receive window field)* wird für die Flusskontrolle verwendet. Wir sehen bald, dass es benutzt wird, um dem Sender die Anzahl von Bytes mitzuteilen, die ein Empfänger zu akzeptieren bereit ist.
- Das 4-Bit-**Header-Längenfeld** *(header length field)* gibt die Länge des TCP-Headers in 32-Bit-Worten an. Der TCP-Header kann aufgrund des TCP-Options-Feldes unterschiedlich lang sein. (Normalerweise ist das Optionsfeld leer, so dass die Länge des typischen TCP-Headers 20 Byte beträgt.)
- Die Länge des **Optionsfelds** *(options field)* ist, wie eben bereits erwähnt, variabel. Dieses Feld wird benötigt, wenn Absender und Empfänger die zu verwendende maximale Segmentgröße *(MSS)* aushandeln, aber z.B. auch für einen Faktor zur Fensterskalierung in Hochgeschwindigkeitsnetzen. Außerdem gibt es eine Option für Zeitstempel. In RFC 854 und RFC 1323 finden Sie zusätzliche Details.

- Das **Flag-Feld** *(flag field)* umfasst 6 Bit. Das **ACK-Bit** gibt an, dass der im Acknowledgment-Feld eingetragene Wert gültig ist, d.h., das Segment enthält eine Bestätigung für ein Segment, das erfolgreich empfangen worden ist. Die **RST-**, **SYN-** und **FIN-Bits** werden für den Auf- und Abbau der Verbindung benutzt, wie wir am Ende dieses Abschnittes noch sehen werden. Ein gesetztes **PSH-Bit** zeigt an, dass der Empfänger die Daten sofort an die Anwendungsschicht weiterreichen soll. Schließlich zeigt noch das **URG-Bit**, dass in diesem Segment Daten vorliegen, welche die Instanz der Anwendungsschicht der sendenden Seite als „dringend" gekennzeichnet hat. Die Position des letzten Bytes dieser dringenden Daten wird durch das 16 Bit lange **Urgent-Data-Pointer-Feld** *(Zeiger auf dringende Daten)* gekennzeichnet. TCP muss die Instanz der Empfängerseite über das Vorliegen dringender Daten informieren und ihr einen Hinweis auf das Ende der dringenden Daten liefern. (In der Praxis werden das PSH- und das URG-Flag sowie der Urgent Data Pointer nicht verwendet. Wir erwähnen diese Felder nur der Vollständigkeit halber.)

Sequenznummern und Acknowledgment-Nummern

Zwei der wichtigsten Felder im TCP-Segment-Header sind das Sequenznummernfeld und das Feld für die Acknowledgment-Nummer. Diese Felder sind ein kritischer Teil des zuverlässigen Datentransferdienstes von TCP. Aber bevor wir diskutieren, wie diese Felder für den zuverlässigen Datentransfer verwendet werden, lassen Sie uns zunächst erklären, was TCP eigentlich in diese Felder einträgt.

TCP betrachtet Daten als einen unstrukturierten, aber geordneten Strom von Bytes. Die Verwendung von Sequenznummern durch TCP spiegelt diese Betrachtungsweise wider, denn die Sequenznummern nummerieren den Strom der gesendeten Bytes und nicht die Folge der gesendeten Segmente. Die **Sequenznummer eines Segmentes** ist deshalb die Position des ersten Bytes des Segmentes im Bytestrom. Betrachten wir ein Beispiel. Nehmen wir an, dass ein Prozess in Host A einen Datenstrom an einen Prozess in Host B über eine TCP-Verbindung senden will. TCP in Host A nummeriert implizit jedes Byte im Datenstrom. Nehmen wir weiter an, dass der Datenstrom aus einer Datei der Länge 500.000 Byte besteht, dass die *MSS* 1.000 Byte beträgt und dass das erste Byte des Datenstroms die Nummer 0 hat. Wie ▶Abbildung 3.30 zeigt, erzeugt TCP 500 Segmente aus dem Datenstrom. Das erste Segment erhält die Sequenznummer 0 zugeteilt, das zweite Segment die Sequenznummer 1.000, das dritte Segment die Sequenznummer 2.000 usw. Jede Sequenznummer steht im Sequenznummernfeld im Kopf des entsprechenden TCP-Segmentes.

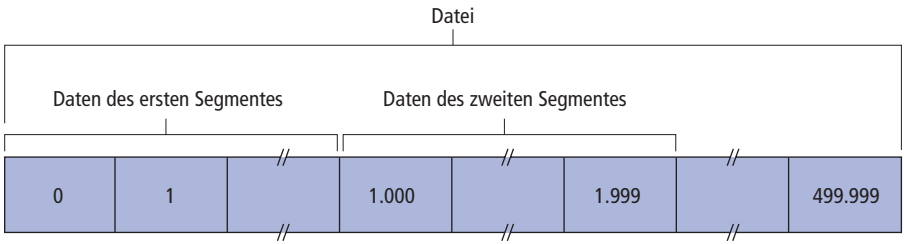


Abbildung 3.30: Das Aufteilen von Daten in TCP-Segmente

Betrachten wir nun die Acknowledgment-Nummern. Sie sind etwas komplexer als die Sequenznummern. Erinnern Sie sich daran, dass TCP Vollduplex bietet, so dass Host A Daten von Host B erhalten kann, während er gleichzeitig Daten an Host B sendet (als Teil derselben TCP-Verbindung). Jedes der Segmente, die von Host B ankommen, hat eine Sequenznummer für die von B zu A fließenden Daten. *Die Acknowledgment-Nummer, die Host A für sein Segment verwendet, ist die Sequenznummer des nächsten Bytes, das Host A von Host B erwartet.* Ein Blick auf einige Beispiele hilft beim Verständnis der Abläufe. Nehmen Sie an, dass Host A alle Bytes von 0 bis 535 erhalten hat und dabei ist, ein Segment an Host B zu senden. Host A wartet auf Byte 536 und alle nachfolgenden Bytes im Datenstrom von Host B. So schreibt Host A die 536 in das Acknowledgment-Nummernfeld des Segmentes, das er an B schickt.

Als anderes Beispiel nehmen Sie an, dass Host A ein Segment von Host B erhält, das die Bytes 0 bis 535 enthält, während ein anderes Segment die Bytes 900 bis 1.000 enthält. Aus irgendeinem Grund hat Host A die Bytes 536 bis 899 noch nicht erhalten. In diesem Beispiel wartet Host A immer noch auf Byte 536 (und folgende), um den Datenstrom von B wieder zusammensetzen zu können. Daher wird das nächste Segment von A an B die Zahl 536 im Acknowledgment-Nummernfeld enthalten. Weil TCP nur Bytes bis zum ersten fehlenden Byte im Strom bestätigt, heißt es, dass TCP **kumulative Acknowledgments** *(cumulative acknowledgments)* verwendet.

Dieses letzte Beispiel bringt auch ein wichtiges, aber subtiles Thema aufs Tablett. Host A hat das dritte Segment (Bytes 900 bis 1.000) vor dem Erhalten des zweiten Segmentes (Bytes 536 bis 899) empfangen. Auf diese Art kam das dritte Segment außer der Reihe an. Das Schwierige daran: Was tut ein Host, wenn er für eine TCP-Verbindung Segmente außer der Reihe erhält? Interessanterweise machen die RFCs von TCP hierfür keine Vorgaben und überlassen die Entscheidung den Menschen, die TCP implementieren. Grundsätzlich gibt es zwei Wahlmöglichkeiten: Entweder (1) der Empfänger verwirft sofort Segmente, die nicht in der richtigen Reihenfolge eintreffen (was, wie wir früher diskutiert haben, das Empfängerdesign vereinfachen kann) oder (2) der Empfänger speichert die erhaltenen Bytes zwischen und wartet auf die fehlenden Bytes, um die Lücke zu schließen. Eindeutig ist letztere Variante hinsichtlich der Nutzung der Netzwerkbandbreite effizienter. Dies ist auch der in der Praxis verwendete Ansatz.

In Abbildung 3.30 nahmen wir an, dass die anfängliche Sequenznummer 0 sei. In Wahrheit wählen beide Seiten einer TCP-Verbindung zufällig eine Anfangssequenznummer aus. Dies wird getan, um die Wahrscheinlichkeit zu reduzieren, dass ein Segment, das aufgrund einer früheren, bereits beendeten Verbindung zwischen zwei Hosts immer noch im Netz herumgeistert, mit einem gültigen Segment aus einer späteren Verbindung zwischen denselben beiden Hosts verwechselt werden kann [Sunshine 1978].

Telnet: eine Fallstudie für Sequenznummern und Acknowledgment-Nummern

Telnet, definiert in RFC 854, ist ein beliebtes Anwendungsschichtprotokoll für das Einloggen auf entfernten Rechnern. Es läuft über TCP und wurde dafür entworfen, zwischen jedem beliebigen Host-Paar zu funktionieren. Im Gegensatz zu den in Kapitel 2 erörterten Anwendungen für die Übertragung größerer Mengen von Daten

ist Telnet eine interaktive Anwendung. Wir erörtern hier ein Telnet-Beispiel, das gut die TCP-Sequenznummern und Acknowledgment-Nummern verdeutlicht. Wir halten fest, dass viele Benutzer mittlerweile das SSH-Protokoll dem Telnet-Protokoll vorziehen, da die Daten einer Telnet-Verbindung (einschließlich der Kennworte) nicht verschlüsselt werden, wodurch Telnet anfälliger für Angriffe ist (wie in Abschnitt 8.7 diskutiert wird).

Nehmen Sie an, dass Host A eine Telnet-Sitzung mit Host B aufbaut. Weil Host A die Sitzung initiiert, wird er als Client bezeichnet und Host B als Server. Jedes vom Benutzer (am Client) eingetippte Zeichen wird dem entfernten Host zugesandt. Dieser schickt eine Kopie jedes Zeichens zurück, das auf dem Bildschirm des Telnet-Benutzers angezeigt wird. Dieses „echo back“ *(zurückgesandtes Echo)* stellt sicher, dass vom Telnet-Benutzer gesehene Zeichen auch wirklich schon am entfernten Standort empfangen und verarbeitet worden sind. Jedes Zeichen durchquert zwischen dem Zeitpunkt, in dem der Benutzer die Taste anschlägt, und der Zeit, zu dem es auf dem Monitor des Benutzers erscheint, zweimal das Netz.

Nehmen wir nun an, dass ein Benutzer eine einzelne Taste drückt und sich dann einen Kaffee holen geht. Betrachten wir die TCP-Segmente, die zwischen dem Client und dem Server ausgetauscht werden. Wie in ▶Abbildung 3.31 gezeigt nehmen wir an, dass die anfänglichen Sequenznummern 42 und 79 seitens des Clients bzw. des Servers sind. Erinnern Sie sich daran, dass die Sequenznummer eines Segmentes die

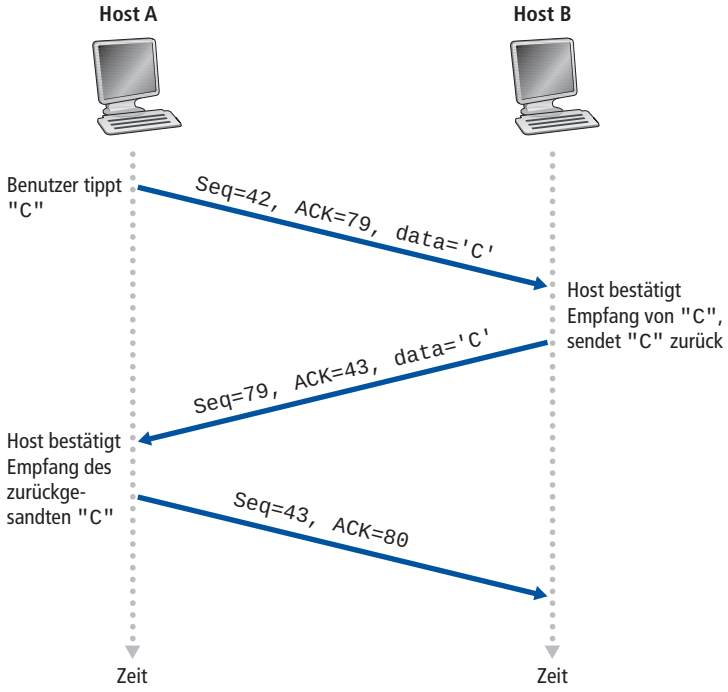


Abbildung 3.31: Sequenznummern und Acknowledgement-Nummern für eine einfache Telnet-Anwendung über TCP

Sequenznummer des ersten Bytes im Datenfeld ist. Auf diese Art hat das erste vom Client gesandte Segment die Sequenznummer 42; das erste vom Server gesandte Segment hat die Sequenznummer 79. Erinnern Sie sich daran, dass die Acknowledgment-Nummer die Sequenznummer des nächsten Daten-Bytes ist, auf das der Host wartet. Nachdem die TCP-Verbindung hergestellt ist, aber bevor irgendwelche Daten verschickt wurden, wartet der Client auf Byte 79 und der Server auf Byte 42.

Wie in Abbildung 3.31 gezeigt, werden drei Segmente geschickt. Das erste Segment geht vom Client zum Server und enthält die 1 Byte-ASCII-Darstellung des Buchstabens „C“ in seinem Datenfeld. Dieses erste Segment hat zudem, wie wir gerade beschrieben haben, den Eintrag 42 in seinem Sequenznummernfeld. Außerdem, weil der Client noch nicht alle Daten vom Server erhalten hat, steht im Acknowledgment-Nummernfeld des ersten Segmentes die 79.

Das zweite Segment wird vom Server an den Client gesandt. Es erfüllt einen doppelten Zweck. Zunächst einmal bestätigt es die Daten, die der Server erhalten hat. Indem er eine 43 ins Acknowledgment-Feld einträgt, sagt der Server dem Client, dass er erfolgreich alles bis einschließlich Byte 42 erhalten hat und jetzt auf Byte 43 wartet. Der zweite Zweck dieses Segmentes ist es, den Buchstaben „C“ zurückzuübertragen. Daher transportiert das zweite Segment die ASCII-Darstellung von „C“ in seinem Datenfeld. Dieses zweite Segment besitzt die Sequenznummer 79, die anfängliche Sequenznummer des Datenflusses vom Server zum Client dieser TCP-Verbindung, da dies das allererste Datenbyte ist, das der Server sendet. Beachten Sie, dass das Acknowlegment für Daten, die der Client zum Server geschickt hat, in einem Segment transportiert wird, das Daten des Servers an den Client enthält; daher nennt man diese Art der Bestätigung **piggybacked** *(huckepack)* auf dem vom Server versandten Segment.

Das dritte Segment wird wieder vom Client an den Server geschickt. Sein einziger Zweck besteht darin, die Daten zu bestätigen, die er vom Server erhalten hat. (Erinnern Sie sich daran, dass das zweite Segment Daten vom Server für den Client enthielt – den Buchstaben „C“.) Dieses dritte Segment hat ein leeres Datenfeld (das heißt, die Bestätigung reist nicht „huckepack“ auf irgendwelchen Daten des Clients an den Server). Im Acknowledgment-Nummernfeld des Segmentes steht der Wert 80, weil der Client den Strom von Bytes bis zur Byte-Sequenznummer 79 erhalten hat und er jetzt auf Bytes ab 80 wartet.

3.5.3 Schätzen der Rundlaufzeit und Timeouts

TCP verwendet, genau wie unser `rdt`-Protokoll in Abschnitt 3.4, einen Timeout-/Übertragungswiederholungsmechanismus, um verlorene Segmente wiederherzustellen. Obwohl dies eigentlich einfach ist, tauchen viele subtile Detailfragen auf, wenn wir diesen Timeout-/Übertragungswiederholungsmechanismus in einem realen Protokoll wie TCP implementieren möchten. Die vielleicht offensichtlichste dieser Fragen betrifft die Länge der Timeout-Intervalle.

Ganz eindeutig sollte der Timeout größer sein als die Rundlaufzeit der Verbindung *(RTT)*, d.h. die Zeit zwischen dem Absenden eines Segmentes und seiner Bestäti-

gung. Ansonsten würden unnötige Übertragungswiederholungen ausgesandt. Aber wie viel größer sollte sie sein? Wie kann die *RTT* überhaupt bestimmt werden? Sollte es einen eigenen Timer für jedes einzelne unbestätigte Segment geben? So viele Fragen! Unsere Diskussion in diesem Abschnitt basiert auf der Arbeit von Jacobson zu TCP [Jacobson 1988] sowie auf den aktuellen IETF-Empfehlungen für die Verwaltung von TCP-Timern [RFC 2988].

Schätzung der Rundlaufzeit

Beginnen wir unsere Betrachtung der TCP-Timerverwaltung, indem wir uns anschauen, wie TCP die Rundlaufzeit zwischen Absender und Empfänger bestimmt. Dies geschieht folgendermaßen: Die sogenannte *SampleRTT* eines Segmentes ist der Zeitraum zwischen dem Senden des Segmentes (das heißt der Übergabe an IP) und dem Empfang der Bestätigung für das Segment. Statt eine *SampleRTT* für jedes gesendete Segment zu messen, führen die meisten TCP-Implementierungen immer nur eine *SampleRTT*-Messung auf einmal durch. Das heißt, zu einem beliebigen Zeitpunkt wird die *RTT* für eines der gesendeten, aber gegenwärtig unbestätigten Segmente gemessen, wodurch einmal in jeder *RTT* ein neuer Wert von *SampleRTT* bestimmt wird. Auch bestimmt TCP nie eine *SampleRTT* für ein Segment, das nochmals übertragen worden ist; es misst *SampleRTT* nur bei Segmenten, die zum ersten Mal gesendet worden sind. (In einer Übungsaufgabe am Ende des Kapitels sollen Sie darüber nachdenken, warum dies sinnvoll ist.)

Offensichtlich schwanken die *SampleRTT*-Werte von Segment zu Segment aufgrund wechselnder Lastsituationen im Netzwerk und der veränderlichen Systemlast auf den Endsystemen. Wegen dieser Schwankungen kann ein einzelner *SampleRTT*-Wert vollkommen untypisch sein. Um die typische *RTT* abzuschätzen, bildet man daher sinnvollerweise eine Art Durchschnitt der *SampleRTT*-Werte. TCP merkt sich einen Durchschnitt, genannt *EstimatedRTT*, der *SampleRTT*-Werte. Wenn eine neue *SampleRTT* gemessen wird, aktualisiert TCP den Wert *EstimatedRTT* entsprechend der folgenden Formel:

$$\text{EstimatedRTT} = (1 - \alpha)\ \text{EstimatedRTT} + \alpha\ \text{SampleRTT}$$

Diese Formel hat die Form einer Programmieranweisung – der neue Wert von *EstimatedRTT* ist eine gewichtete Kombination des vorherigen Wertes von *EstimatedRTT* und der neuen *SampleRTT*. Der empfohlene Wert für α ist 0,125 (also 1/8) [RFC 2988], wodurch diese Formel folgendermaßen lautet:

$$\text{EstimatedRTT} = 0{,}875\ \text{EstimatedRTT} + 0{,}125\ \text{SampleRTT}$$

Beachten Sie, dass *EstimatedRTT* ein gewichteter Durchschnitt der *SampleRTT*-Werte ist. Wie eine Übungsaufgabe am Ende dieses Kapitels zeigen wird, legt dieser gewichtete Durchschnitt mehr Gewicht auf neuere Messungen als auf alte Messungen. Dies ist nur natürlich, da neuere Messungen den gegenwärtigen Lastzustand im Netz besser wiedergeben. In Statistiken wird ein solcher Durchschnitt ein **exponentiell gewichteter gleitender Durchschnitt** (**EWMA,** *exponential weigthed moving*

Von der Theorie zur Praxis

TCP bietet zuverlässigen Datentransfer mittels positiver Acknowledgments und Timer auf dieselbe Weise, die wir in Abschnitt 3.4 kennengelernt haben. TCP bestätigt Daten, die richtig empfangen wurden, und es überträgt Segmente erneut, sofern es davon ausgehen muss, dass Segmente oder ihre entsprechenden Bestätigungen verloren gegangen sind oder Übertragungsfehler aufgetreten sind. Bestimmte Versionen von TCP verwenden auch einen impliziten NAK-Mechanismus – mit dem **Fast-Retransmit-Mechanismus** (*schnelle Übertragungswiederholung*) werden drei duplizierte ACKs eines gegebenen Segmentes als implizites NAK für das folgende Segment aufgefasst. Dies stößt dann noch vor dem Timeout die erneute Übertragung des Segmentes an. TCP verwendet Sequenznummern, anhand derer der Empfänger verlorene oder duplizierte Segmente erkennen kann. Genau wie im Fall unseres zuverlässigen Datentransferprotokolls, `rdt3.0`, kann TCP nicht sicher entscheiden, ob ein Segment oder sein ACK verloren gegangen, verändert oder übermäßig verzögert ist. Der Sender reagiert auf jede dieser Möglichkeiten auf dieselbe Weise: mit einer erneuten Übertragung des betreffenden Segmentes.

TCP benutzt auch Pipelining, so dass der Sender zu jedem beliebigen Zeitpunkt mehrere übertragene, aber noch nicht bestätigte Segmente vorliegen hat. Wie wir oben gesehen haben, kann Pipelining den Durchsatz einer Sitzung deutlich erhöhen, wenn das Verhältnis der Segmentgröße zur Rundlaufzeit klein ist. Die Zahl der ausstehenden, unbestätigten Segmente, die einem Sender erlaubt sind, wird von TCPs Mechanismen zur Flusskontrolle und zur Überlastkontrolle bestimmt. Die TCP-Flusskontrolle wird am Ende dieses Abschnittes diskutiert. Die TCP-Überlastkontrolle ist das Thema in Abschnitt 3.7. Im Moment müssen wir nur wissen, dass der TCP-Sender Pipelining verwendet.

average) genannt. Der Begriff „exponentiell“ in EWMA bedeutet, dass das Gewicht eines gegebenen *SampleRTT*-Wertes exponentiell schnell abfällt, wenn aktuellere Werte hinzukommen. In den Übungsaufgaben sollen Sie den exponentiellen Term in *EstimatedRTT* ableiten.

▶ Abbildung 3.32 zeigt die *SampleRTT*- und *EstimatedRTT*-Werte für einen Wert von $\alpha = 1/8$ für eine TCP-Verbindung zwischen *gaia.cs.umass.edu* (in Amherst, Massachusetts) zu *fantasia.eurecom.fr* (im Süden Frankreichs). Deutlich erkennbar ist, wie die Schwankungen der *SampleRTT* bei der Berechnung der *EstimatedRTT* geglättet werden.

Zusätzlich zu einer Schätzung der *RTT* ist auch ein Maß der Variabilität der *RTT* sinnvoll. [RFC 2988] definiert die Schwankungsbreite der *RTT*, *DevRTT*, als Abschätzung, wie sehr die *SampleRTT* typischerweise von der *EstimatedRTT* abweicht:

$$\text{DevRTT} = (1 - \beta)\,\text{DevRTT} + \beta\,|\,\text{SampleRTT} - \text{EstimatedRTT}\,|$$

Beachten Sie, dass *DevRTT* ein EWMA der Differenz von *SampleRTT* und *EstimatedRTT* ist. Wenn die *SampleRTT*-Werte nur geringe Schwankungen aufweisen, dann wird auch die *DevRTT* klein sein. Gibt es andererseits viele Schwankungen, wird *DevRTT* groß. Der empfohlene Wert für β ist 0,25.

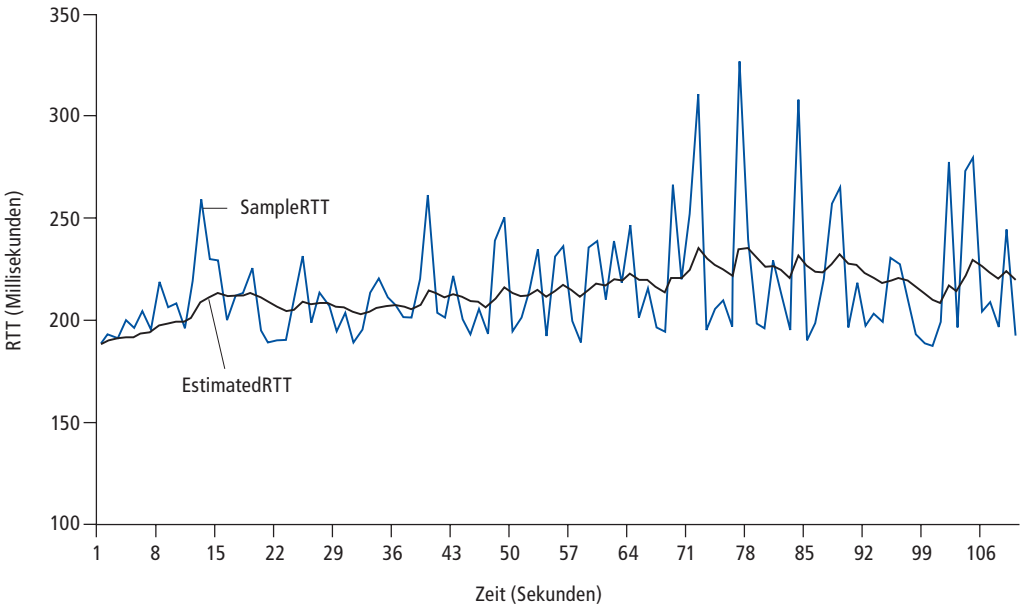


Abbildung 3.32: RTT-Samples und RTT-Estimates

Setzen und Verwalten des Retransmission Timeout-Intervalls

Bei gegebenen Werten von *EstimatedRTT* und *DevRTT*, welcher Wert soll für das TCP-Timeout-Intervall verwendet werden? Klarerweise sollte das Intervall größer oder gleich *EstimatedRTT* sein, ansonsten würden unnötige Übertragungswiederholungen gesendet. Aber das Timeout-Intervall sollte nicht allzu viel größer sein als *EstimatedRTT*, denn im Fall eines Segmentverlustes würde TCP die Neuübertragung des Segmentes nicht schnell genug durchführen, wodurch große Verzögerungen bei der Datenübertragung auftreten würden. Es ist deshalb wünschenswert, den Timeout mit *EstimatedRTT* plus einer zusätzlichen Zeitspanne anzusetzen. Die Zeitspanne sollte groß sein, wenn viele Schwankungen in den *SampleRTT*-Werten auftreten; sie sollte klein sein, wenn nur kleine Schwankungen vorliegen. Der Wert von *DevRTT* sollte daher an dieser Stelle ins Spiel kommen. Alle diese Überlegungen wurden bei der Methode in Betracht gezogen, mit der TCP das Zeitintervall für die Übertragungswiederholung bestimmt:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4\ \text{DevRTT}$$

3.5.4 Zuverlässiger Datentransfer

Erinnern Sie sich daran, dass die Netzwerkschicht des Internets (also IP) unzuverlässig ist. IP garantiert keine Datagrammzustellung, garantiert keine Ankunft der Datagramme in der korrekten Reihenfolge und garantiert nicht die Integrität der Daten in den Datagrammen. Bei IP können die Puffer der Router überlaufen und deshalb Datagramme ihren Zielort niemals erreichen, Datagramme können in der falschen Reihen-

folge ankommen und Bits in den Datagrammen können verändert worden sein (sind also von 0 auf 1 gesprungen oder umgekehrt). Weil Transportschichtsegmente von IP-Datagrammen über das Netz transportiert werden, können Transportschichtsegmente ebenfalls von diesen Problemen betroffen sein.

TCP erzeugt einen **zuverlässigen Datentransferdienst** *(reliable data transfer service)* oberhalb des unzuverlässigen Best-Effort-Service von IP. Der zuverlässige Datentransferdienst von TCP stellt sicher, dass der Datenstrom, den ein Prozess aus dem Eingangspuffer seiner TCP-Instanz ausliest, nicht verändert wurde, keine Lücken oder Duplikate aufweist und in der richtigen Reihenfolge vorliegt. Das bedeutet, der Byte-Strom ist genau derselbe Byte-Strom, der vom Endsystem auf der anderen Seite der Verbindung ausgesandt wurde. Der zuverlässige Datentransferdienst von TCP verwendet viele der Grundlagen, die wir in Abschnitt 3.4 untersucht haben.

In unserer früheren Betrachtung zuverlässiger Datentransfertechniken war es konzeptionell am einfachsten, anzunehmen, dass ein einzelner Timer zu jedem übertragenden, aber noch nicht bestätigten Segment gehört. Obwohl sich das in der Theorie großartig anhört, kann die Verwaltung der Timer beträchtlichen Aufwand verursachen. Daher verwenden die empfohlenen TCP-Timer-Managementverfahren [RFC 2988] nur einen einzelnen Timer für die Wiederholung von Übertragungen, selbst dann, wenn mehrere übertragene, aber noch nicht bestätigte Segmente existieren. Das in diesem Abschnitt beschriebene TCP-Protokoll folgt dieser Empfehlung der Verwendung eines einzelnen Timers.

Wir werden in zwei aufeinander aufbauenden Schritten diskutieren, wie TCP zuverlässigen Datentransfer realisiert. Wir zeigen zuerst eine extrem vereinfachte Beschreibung eines TCP-Senders, der nur Timeouts verwendet, um verlorene Segmente wiederherzustellen; danach zeigen wir eine vollständigere Beschreibung, die außer Timeouts auch doppelte Acknowledgments verwendet. In der folgenden Diskussion nehmen wir an, dass Daten nur in eine Richtung, von Host A zu Host B, gesandt werden und dass Host A eine große Datei schickt.

▶ Abbildung 3.33 zeigt eine extrem vereinfachte Beschreibung eines TCP-Senders. Wir sehen, dass es drei Hauptereignisse gibt, die im TCP-Sender mit Datenübertragung und dem Wiederholen von Übertragungen verbunden sind: von den Anwendungen erhaltene Daten, Timeouts und eintreffende ACKs. Bis zum Auftreten des ersten größeren Ereignisses erhält TCP Daten von der Anwendung, verkapselt diese Daten in einem Segment und reicht das Segment an IP weiter. Beachten Sie, dass jedes Segment eine Sequenznummer beinhaltet, nämlich, wie in Abschnitt 3.5.2 beschrieben, die Position des ersten Datenbytes im Bytestrom. Beachten Sie auch, dass, sofern der Timer nicht schon wegen eines anderen Segmentes läuft, TCP den Timer startet, sobald es ein Segment an IP weiterreicht. (Es ist hilfreich, sich den Timer als mit dem ältesten unbestätigten Segment verbunden zu denken.) Das Ablaufintervall für diesen Timer ist das Timeout-Intervall, das mit *EstimatedRTT* und *DevRTT* berechnet wird Abschnitt 3.5.3.

Das zweite größere Ereignis ist der Timeout. TCP antwortet auf das Timeout-Ereignis durch nochmaliges Übertragen des Segmentes, das den Timeout verursacht hat. TCP startet dann den Timer neu.

```
/* Nehmen Sie an, dass der Sender nicht durch TCP-Fluss- oder -Überlastkontrolle
eingeschränkt wird, die Größe von oben kommender Daten kleiner als die MSS ist und der
Datentransfer nur in eine Richtung verläuft. */

NextSeqNum = InitialSeqNumber
SendBase = InitialSeqNumber

loop (forever) {
   switch(event)

      event: Daten von der oberhalb liegenden Anwendung erhalten
         erzeuge TCP-Segment mit Sequenznummer NextSeqNum
         if (Timer läuft derzeit nicht)
            starte Timer
         gib Segment an IP weiter
         NextSeqNum=NextSeqNum+length(data)
         break;

      event: Timeout
         übertrage noch nicht bestätigtes Segment mit kleinster Sequenznummer erneut
         starte Timer
         break;

      event: ACK eingegangen, Inhalt des ACK-Felds hat den Wert y
         if (y > SendBase) {
            SendBase=y
            if (es gibt noch unbestätigten Segmente)
               starte Timer
            }
            break;

} /* Ende von loop forever */
```

Abbildung 3.33: Ein vereinfachter TCP-Sender

Das dritte größere Ereignis, das vom TCP-Sender bewältigt werden muss, ist die Ankunft eines Acknowledgment-Segmentes (ACK) vom Empfänger (genauer ein Segment, das einen gültigen ACK-Wert beinhaltet.) Tritt dieses Ereignis ein, vergleicht TCP den ACK-Wert *y* mit seiner Variablen *SendBase*. Die TCP-Zustandsvariable *SendBase* ist die Sequenznummer des ältesten unbestätigten Bytes. (Daher ist *SendBase* – 1 die Sequenznummer des letzten Bytes, von dem der Sender weiß, dass es korrekt und in der richtigen Reihenfolge beim Empfänger eingetroffen ist.) Wie früher gezeigt,

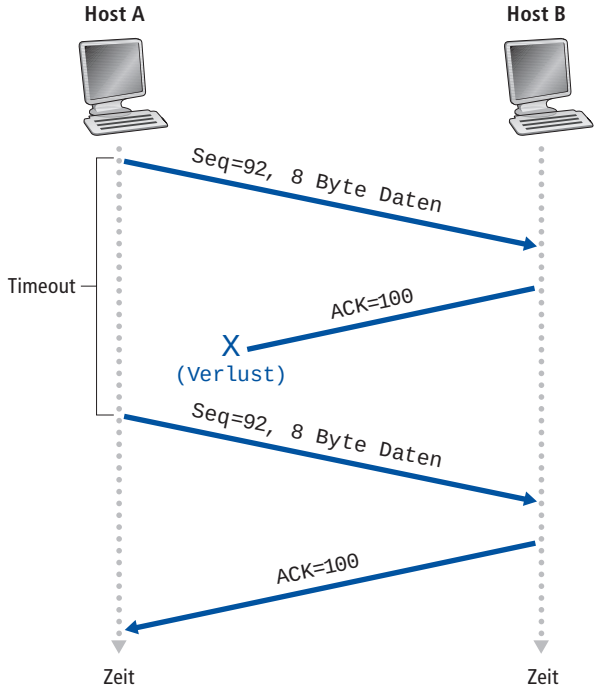


Abbildung 3.34: Erneute Übertragung aufgrund eines verloren gegangenen Acknowledgments

verwendet TCP kumulative Acknowledgments, so dass y den Erhalt aller Bytes vor Byte Nummer y bestätigt. Ist $y >$ *SendBase*, dann bestätigt ACK ein oder mehrere zuvor unbestätigte Segmente. Dadurch aktualisiert der Sender seine Variable *SendBase*; er startet auch den Timer neu, wenn es gegenwärtig irgendwelche noch nicht bestätigten Segmente gibt.

Einige interessante Szenarien

Wir haben gerade in einer äußerst vereinfachten Version beschrieben, wie TCP zuverlässigen Datentransfer realisiert. Aber sogar diese extrem vereinfachte Version hat viele Tücken. Um ein gutes Gefühl zu bekommen, wie dieses Protokoll arbeitet, wollen wir jetzt einige einfache Szenarien betrachten. ▶Abbildung 3.34 beschreibt das erste Szenario, in dem Host A ein Segment an Host B sendet. Nehmen Sie an, dass dieses Segment Sequenznummer 92 hat und 8 Byte Daten enthält. Nach dem Senden dieses Segmentes wartet Host A auf ein Segment von B mit der Acknowledgment-Nummer 100. Obwohl das Segment von A durch B empfangen wird, geht die Bestätigung von B nach A verloren. In diesem Fall tritt das Timeout-Ereignis ein und Host A überträgt dasselbe Segment erneut. Natürlich erkennt Host B die Übertragungswiederholung anhand der Sequenznummer und weiß, dass dieses Segment Daten enthält, die schon empfangen worden sind. Daher verwirft TCP in Host B die Bytes des nochmals übertragenen Segmentes.

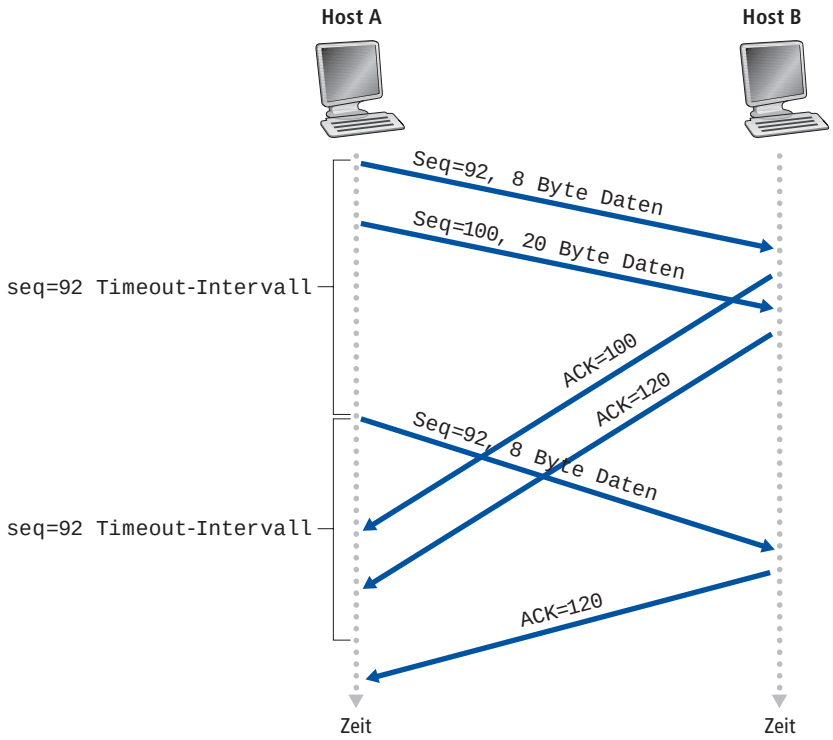


Abbildung 3.35: Segment 100 wird nicht erneut übertragen

In einem zweiten, in ▶Abbildung 3.35 gezeigten Szenario sendet Host A zwei Segmente direkt hintereinander. Das erste Segment hat die Sequenznummer 92 und enthält 8 Byte an Daten; das zweite Segment hat Sequenznummer 100 und 20 Byte an Daten. Nehmen Sie an, dass beide Segmente intakt bei Host B eintreffen und dieser zwei separate Acknowledgments für jedes dieser Segmente zurücksendet. Das erste dieser Acknowledgments hat die Nummer 100, das zweite hat Acknowledgment-Nummer 120. Nehmen Sie nun an, dass keine der Bestätigungen vor dem Timeout bei Host A ankommt. Wenn das Timeout-Ereignis auftritt, überträgt Host A das erste Segment mit Sequenznummer 92 erneut und startet den Timer neu. Sofern das ACK für das zweite Segment vor dem neuen Timeout ankommt, wird das zweite Segment nicht nochmals 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 Segmentes 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.

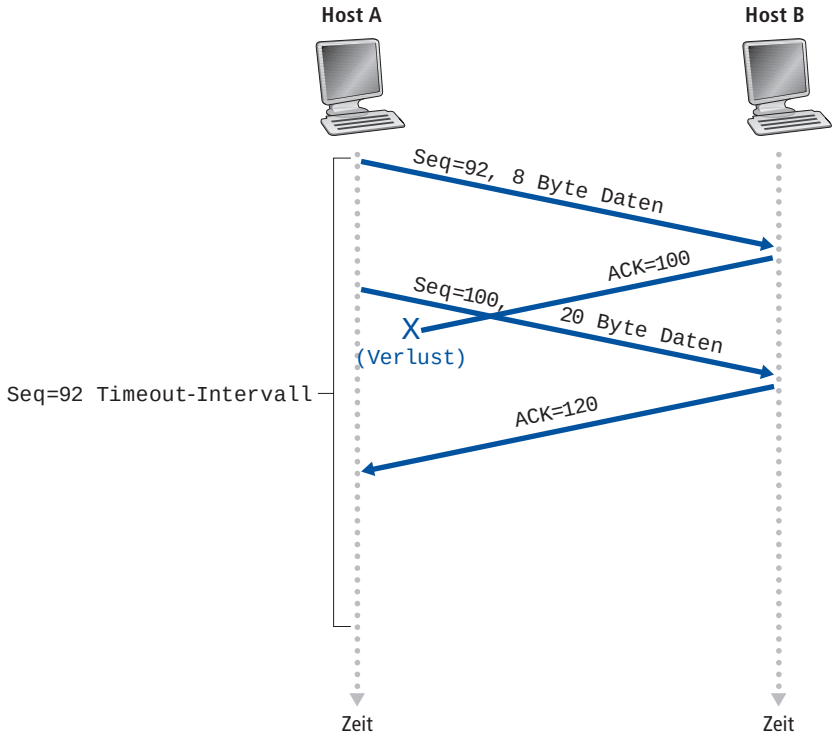


Abbildung 3.36: Das kumulative Acknowledgment verhindert die erneute Übertragung des ersten Segmentes

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 handelt TCP politisch korrekt und 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 Segmentes 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 ACKs)* 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 1122; RFC 2581]. 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 Segment fehlt. Diese Lücke könnte das Ergebnis eines verlorenen Segmentes 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.

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.)

Weil ein Sender oft eine große Anzahl von Segmenten direkt hintereinander sendet, wird es beim Verlust eines Segmentes 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 2581], wobei er das fehlende Segment nochmals überträgt, bevor der Timer des Segmentes abläuft. Dies wird in Abbildung 3.37 gezeigt, in der das zweite Segment verloren

Ereignis	Aktion des TCP-Empfängers
Ankunft des Segmentes in der richtigen Reihenfolge mit der erwarteten Sequenznummer. Alle Daten bis zur erwarteten Sequenznummer sind bereits bestätigt.	Verzögertes ACK. Wartet bis zu 500 ms auf die Ankunft eines anderen Segmentes in richtiger Reihenfolge. Wenn das nächste Segment nicht in diesem Zeitintervall eintrifft, wird ein ACK gesendet.
Ankunft eines Segmentes 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 Segmentes 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 Segmentes, 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: Empfehlung für die Erzeugung von TCP ACK [RFC 1122; RFC 2581]

gegangen ist und nochmals übertragen wird, bevor sein Timer abläuft. Bei TCP mit schneller Übertragungswiederholung ersetzt der folgende Codeschnipsel das ACK-Empfangs-Ereignis 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 15 Jahren Erfahrung mit TCP-Timern sind, sollten Sie davon überzeugen, dass dies wirklich zutrifft!

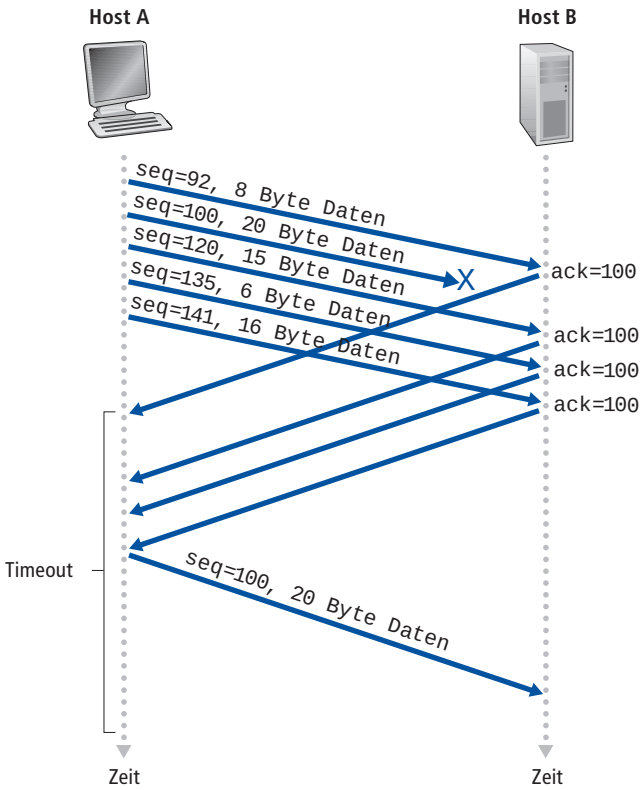


Abbildung 3.37: Fast Retransmit: erneute Übertragung des fehlenden Segmentes, bevor der Timer des Segmentes abläuft

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 folgen-

den 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 eingetroffen sind, 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 Fehlerbehebungsmechanismus 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, 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 erbringt 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 dieses Empfangsfenster dem Sender eine Vorstellung davon, wie viel freier Pufferplatz beim Emp-

fänger verfügbar ist. Weil TCP Vollduplex ist, 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 allokiert einen Eingangspuffer für diese Verbindung, dessen Größe durch RcvBuffer festgelegt wird. Ab und zu liest der Anwendungsprozess in Host B aus dem Puffer. Definieren Sie 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:

$$\text{LastByteRcvd} - \text{LastByteRead} \leq \text{RcvBuffer}$$

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

$$\text{RcvWindow} = \text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}]$$

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

Wie verwendet die Verbindung die Variable *RcvWindow* für die Flusskontrolle? Host B sagt Host A, wie viel freien Speicherplatz er im Verbindungspuffer besitzt, indem er den aktuellen Wert von *RcvWindow* in das RcvWindow-Feld jedes Segmentes schreibt, das er an Host A sendet. Zu Beginn setzt Host B *RcvWindow* = *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 *RcvWindow*, 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

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{RcvWindow}$$

Dieses Vorgehen zieht ein kleines technisches Problem nach sich. Um es zu erkennen, nehmen Sie an, dass der Eingangspuffer von Host B vollläuft, so dass *RcvWindow* = 0. Nachdem er Host A mitgeteilt hat, dass *RcvWindow* = 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 RcvWindow-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

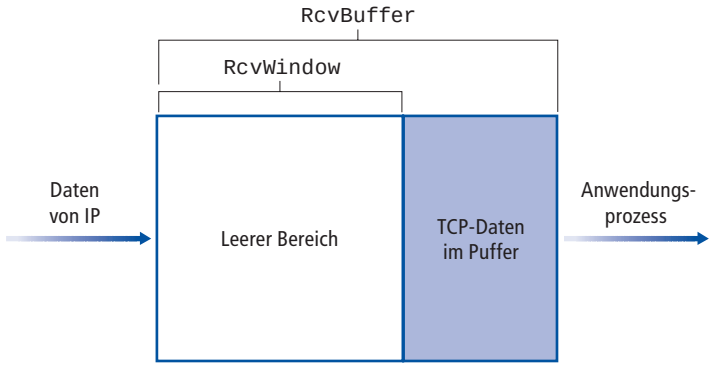


Abbildung 3.38: Das Empfangsfenster (RcvWindow) und der Eingangspuffer (RecBuffer)

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 *RcvWindow*, der nicht null ist.

Weblink

Die Webseite dieses Buches bietet ein interaktives Java-Applet, das das Funktionieren des TCP-Empfangsfensters 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, 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 unglaublich populären 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 Client-Seite sendet zuerst ein spezielles TCP-Segment zur Server-Seite. Dieses spezielle Segment enthält keine Anwendungsschichtdaten. Aber eines der Flag-Bits im Kopf des Segmentes (Abbildung 3.29), das SYN-Bit, ist 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-Segmentes. Dieses Segment wird in ein IP-Datagramm gepackt und an den Server gesandt. Es gab beträchtliches Interesse daran, 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 Serverhost ankommt (vorausgesetzt, dass es ankommt!), holt der Server das TCP-SYN-Segment aus dem Datagramm heraus, allokiert TCP-Puffer und Variablen für die Verbindung und sendet dem Client-TCP ein Antwortsegment zu. (Wir werden später 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. Zuerst ist das SYN-Bit auf 1 gesetzt. Als Zweites wird das Acknowledgment-Feld des TCP-Segment-Headers auf *client_isn* + 1 gesetzt. Zuletzt 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 Ihr SYN-Paket erhalten, das eine Verbindung mit Ihrer 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 Erhalten des SYNACK-Segmentes allokiert 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

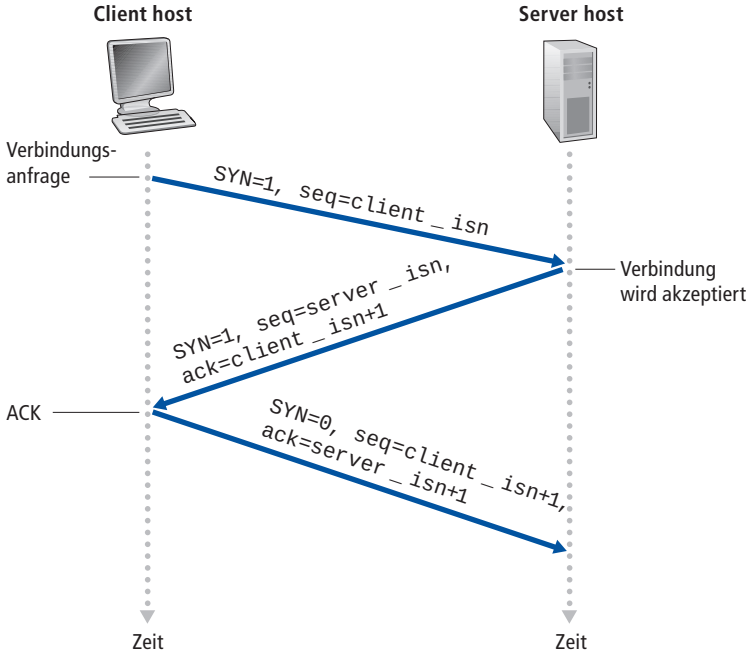


Abbildung 3.39: Segmentaustausch beim Drei-Wege-Handshake

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.

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 Segmentes 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.

Während der Lebensdauer einer TCP-Verbindung durchläuft das TCP-Protokoll, das in jedem Host läuft, verschiedene **TCP-Zustände**. ▶Abbildung 3.41 zeigt eine typische

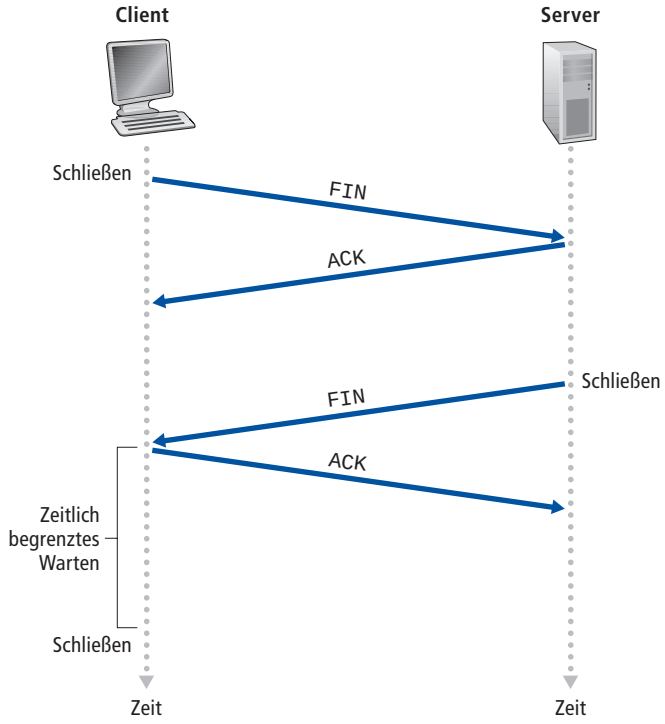


Abbildung 3.40: Schließen einer TCP-Verbindung

Sequenz von TCP-Zuständen, die vom Client-seitigen TCP durchlaufen werden. Es beginnt im Zustand CLOSED *(geschlossen)*. Die Anwendung auf der Client-Seite initiiert eine neue TCP-Verbindung (durch Erzeugen eines Socket-Objektes wie in unseren Java-Beispielen von Kapitel 2). Dadurch sendet der Client ein SYN-Segment an den Server. Nachdem das SYN-Segment gesandt wurde, beginnt für das Client-seitige TCP der Zustand „SYN_SENT“ *(SYN gesendet)*. Im SYN-SENT-Zustand wartet TCP auf ein Segment vom Server-seitigen TCP, welches eine Bestätigung für das vorherige Segment des Clients beinhaltet und das SYN-Bit auf 1 setzt. Nachdem es solch ein Segment erhalten hat, befindet sich TCP auf dem Client im Zustand ESTABLISHED *(Verbindung aufgebaut)*. In diesem Zustand kann der Client TCP-Segmente senden und erhalten, die (von Anwendungen erzeugte) Nutzdaten enthalten.

Nehmen Sie an, die Client-Anwendung entscheidet, die Verbindung schließen zu wollen. (Beachten Sie, dass auch der Server die Verbindung schließen könnte.) Dann sendet der Client ein TCP-Segment mit dem auf 1 gesetzten FIN-Bit und tritt in den FIN_WAIT_1-Zustand ein. In diesem Zustand wartet TCP auf ein Segment vom Server mit einem Acknowledgment. Wenn es dieses Segment erhält, tritt es in den FIN_WAIT_2-Zustand ein. In diesem wartet es auf ein weiteres Segment vom Server, mit einem auf 1 gesetzten FIN-Flag; nach dessen Erhalt bestätigt das Client-seitige TCP das Segment des Servers und tritt in den TIME_WAIT-Zustand ein. Nun sendet

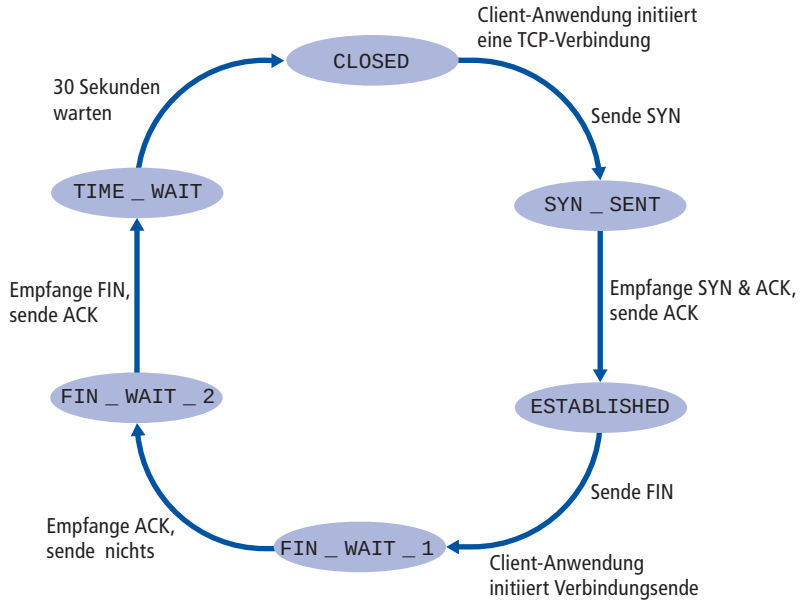


Abbildung 3.41: Eine typische Sequenz von TCP-Zuständen, die vom Client-seitigen TCP durchlaufen werden

der Client die abschließende Bestätigung erneut, falls das ACK verloren ging. Die Zeit, die im TIME_WAIT-Zustand verbracht wird, hängt von der Implementierung ab, typische Werte sind 30 Sekunden, 1 Minute und 2 Minuten. Nach dieser Wartezeit ist die Verbindung formell geschlossen und alle Ressourcen auf der Client-Seite (einschließlich der Portnummern) werden freigegeben.

▶ Abbildung 3.42 erläutert die Abfolge von Zuständen, die normalerweise von dem TCP der Server-Seite durchlaufen werden, vorausgesetzt, dass der Client den Verbindungsabbau beginnt. Die Übergänge sind selbst erklärend. In den beiden Zustandsübergangsdiagrammen haben wir nur gezeigt, wie eine TCP-Verbindung normalerweise aufgebaut und geschlossen wird. Wir haben nicht beschrieben, was in bestimmten pathologischen Szenarien geschieht, beispielsweise, wenn beide Seiten einer Verbindung zufällig zur gleichen Zeit einen Abbau einleiten. Wenn Sie sich für dieses und andere weiterführende Themen rund um TCP interessieren, empfehlen wir Ihnen das umfassende Buch von [Stevens 1994].

Unsere bisherige Diskussion ging davon aus, dass sowohl Client als auch Server bereit sind, zu kommunizieren, d.h., dass der Server auf dem Port lauscht, auf den der Client sein SYN-Segment schickt. Überlegen wir, was geschieht, wenn ein Host ein TCP-Segment erhält, dessen Portnummern oder Quell-IP-Adresse zu keinem der aktiven Sockets im Host passt. Nehmen Sie z.B. an, dass ein Host ein TCP-SYN-Paket mit dem Zielport 80 erhält, aber der Host keine Verbindungen auf Port 80 akzeptiert (das heißt, es läuft kein Webserver auf Port 80). Dann sendet der Host ein spezielles Reset-Segment an die Quelle. Dieses TCP-Segment hat das RST-Flag-Bit auf eins gesetzt

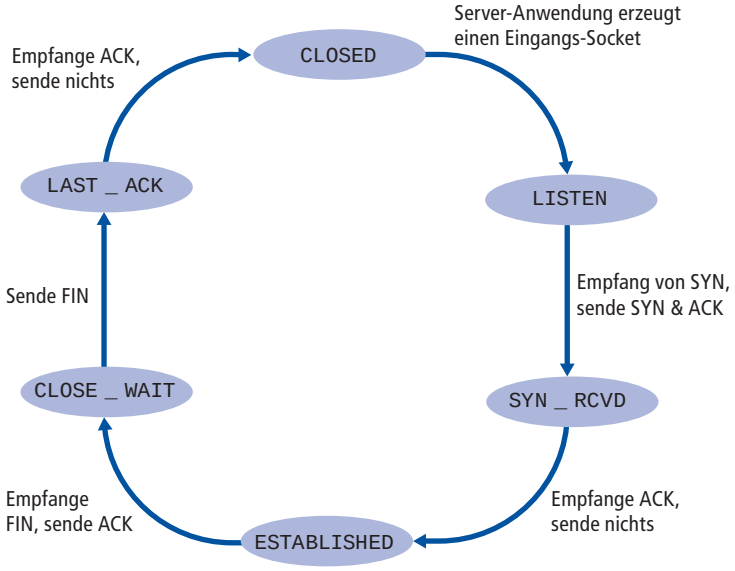


Abbildung 3.42: Eine typische Sequenz von TCP-Zuständen, die vom Server-seitigen TCP durchlaufen werden

(siehe Abschnitt 3.5.2). Sendet ein Host ein Reset-Segment, teilt er der Quelle mit: „Ich habe keinen Socket für dieses Segment. Bitte senden Sie das Segment nicht noch mal“. Erhält ein Host ein UDP-Paket, dessen Zielportnummer nicht zu einem aktiven UDP-Socket passt, sendet der Host ein spezielles ICMP-Datagramm, das in Kapitel 4 diskutiert wird.

Nachdem wir uns nun ein gutes Verständnis der TCP-Verbindungsverwaltung angeeignet haben, wollen wir erneut das nmap-Werkzeug zum Port-Scanning betrachten und uns etwas genauer anschauen, wie es funktioniert. Um einen bestimmten TCP-Port, z. B. Port 6789, auf einem Zielhost zu untersuchen, sendet nmap an diesen Host ein TCP-SYN-Segment mit Zielport 6789. Es gibt drei mögliche Ergebnisse:

- *Das Quellsystem erhält ein TCP-SYNACK-Segment vom Zielhost.* Da dies bedeutet, dass eine Anwendung mit TCP-Port 6789 auf dem Zielhost läuft, gibt nmap die Meldung „open“ *(offen)* zurück.
- *Das Quellsystem erhält ein TCP-RST-Segment vom Zielhost.* Dies bedeutet, dass das SYN-Segment den Zielhost zwar erreicht hat, aber der Zielhost keine Anwendung auf TCP-Port 6789 ausführt. Nun weiß aber der Angreifer mindestens, dass Segmente, die an Port 6789 des Hosts geschickt werden, nicht von irgendeiner Firewall auf dem Pfad zwischen Quell- und Zielhost blockiert werden. (Firewalls erörtern wir in Kapitel 8.)
- *Die Quelle erhält nichts.* Dies bedeutet wahrscheinlich, dass das SYN-Segment von einer dazwischenliegenden Firewall abgefangen wurde und den Zielhost nie erreicht hat.

Fokus Sicherheit

Der SYN-Flood-Angriff

Wir haben bei unserer Diskussion des Drei-Wege-Handshakes von TCP gesehen, dass ein Server Verbindungsvariablen und Puffer als Antwort auf ein empfangenes SYN allokiert und initialisiert. Der Server sendet dann ein SYNACK und wartet auf ein ACK-Segment vom Client, den dritten und letzten Schritt des Handshakes vor dem vollständigen Aufbau einer Verbindung. Sendet der Client kein ACK, um den dritten Schritt des Drei-Wege-Handshakes durchzuführen, beendet der Server schließlich (oft erst nach einer Minute oder mehr) die halboffene Verbindung und gibt die reservierten Ressourcen wieder frei.

Dieses TCP-Verbindungsmanagement-Protokoll ermöglicht einen klassischen DoS-Angriff, nämlich den **SYN-Flood-Angriff**. Bei diesem Angriff sendet der Angreifer eine große Zahl von TCP-SYN-Segmenten, ohne den dritten Schritt des Handshakes durchzuführen. Der Angriff kann verstärkt werden, wenn SYNs von mehreren Quellen gesendet werden, in Form eines DDoS-SYN-Flood-Angriff (DDoS = Distributed Denial of Service). Mit dieser Schwemme von SYN-Segmenten können die Verbindungsressourcen des Servers schnell erschöpft sein, da sie zwar für halboffene Verbindungen reserviert, aber nie tatsächlich verwendet werden. Sind die Ressourcen des Servers erschöpft, wird rechtmäßigen Clients der Dienst versagt. Solche SYN-Flood-Angriffe [CERT SYN 1996] waren unter den ersten von CERT dokumentierten DoS-Angriffen [CERT 2007].

SYN-Flooding ist ein potenziell verheerender Angriff. Glücklicherweise gibt es eine wirkungsvolle Verteidigung, die als **SYN-Cookies** bezeichnet wird [Skoudis 2006; Cisco SYN 2007; Bernstein 2007]. Mittlerweise in den meisten größeren Betriebssystemen eingesetzt, funktionieren SYN-Cookies wie folgt:

- Erhält der Server ein SYN-Segment, weiß er nicht, ob das Segment von einem rechtmäßigen Benutzer kommt oder Teil eines SYN-Flutenangriffs ist. Also erzeugt der Server keine halboffene TCP-Verbindung für dieses SYN. Stattdessen erzeugt der Server eine initiale TCP-Sequenznummer, die eine komplexe Funktion (eine Hash-Funktion) der Quell- und Ziel-IP-Adressen und der Portnummern des SYN-Segmentes sowie einem nur dem Server bekannten geheimen Wert ist. (Der Server verwendet denselben geheimen Wert für eine große Anzahl von Verbindungen.) Diese sorgfältig gestaltete initiale Sequenznummer ist der sogenannte Cookie. Der Server sendet dann ein SYNACK-Paket mit dieser speziellen Sequenznummer. *Wichtig ist dabei, dass sich der Server nicht an den Cookie oder irgendeine andere Zustandsinformation des SYN-Segmentes erinnert.*
- Handelt es sich um einen rechtmäßigen Client, dann gibt er ein ACK-Segment zurück. Der Server muss beim Erhalten dieses ACK sicherstellen, dass das ACK zu einem früher übersandten SYN gehört. Wie kann dies geschehen, wenn der Server keine Aufzeichnungen über die erhaltenen SYN-Segmente behält? Vielleicht haben Sie schon vermutet, dass es mithilfe des Cookies geschieht. Der Wert im Acknowledgment-Feld eines legitimen ACK ist gleich der Sequenznummer im SYNACK plus eins (Abbildung 3.39). Der Server führt dann dieselbe Funktion aus und verwendet dabei dieselben Felder im ACK-

Segment sowie seinen geheimen Wert. Ist das Ergebnis der Funktion plus eins dasselbe wie die Acknowledgment-Nummer, schließt der Server daraus, dass das ACK zu einem früheren SYN-Segment passt und daher gültig ist. Der Server allokiert dann eine vollständig offene Verbindung, zusammen mit einem Socket.

- Gibt andererseits der Client kein ACK-Segment zurück, dann hat das ursprüngliche SYN keinen Schaden angerichtet, da der Server keine Ressourcen dafür allokiert hat!

SYN-Cookies eliminieren wirksam die Bedrohung durch einen SYN-Flood-Angriff. Eine Abwandlung dieses Angriffes besteht darin, den böswilligen Client für jedes SYNACK-Segment, das der Server generiert, ein gültiges ACK-Segment zurückgeben zu lassen. Dies bewirkt, dass der Server *vollständig* offene TCP-Verbindungen herstellt, selbst wenn sein Betriebssystem SYN-Cookies verwendet. Werden Zehntausende Clients eingesetzt (DDoS-Angriff), von denen jeder eine andere Quell-IP-Adresse hat, wird es für den Server schwierig, zwischen legitimen und böswilligen Quellen zu unterscheiden. Daher ist die Verteidigung gegen diesen Completed-Handshake-Angriff schwieriger als der klassische SYN-Flood-Angriff.

Nmap ist ein mächtiges Werkzeug, mit dem sich gut hinter die Kulissen blicken lässt und das nicht nur offene TCP-Ports, sondern auch offene UDP-Ports, Firewalls und ihre Konfigurationen, und sogar die Versionen von Anwendungen und Betriebssystemen herausfinden kann. Das meiste davon erfolgt durch Manipulation der TCP-Verbindungsmanagement-Segmente [Skoudis 2006]. Sitzen Sie zufällig in der Nähe einer Linux-Maschine, können Sie nmap jetzt sofort durch einfaches Eintippen von „nmap“ auf der Kommandozeile ausprobieren. Für andere Betriebssysteme können Sie nmap von *http://insecure.org/nmap* herunterladen.

Hiermit endet unsere Einführung in Zuverlässigkeit und Flusskontrolle bei TCP. In Abschnitt 3.7 kehren wir zu TCP zurück und sehen uns die TCP-Überlastkontrolle im Detail an. Bevor wir das jedoch tun, werden wir einen Schritt zurückgehen und die Grundlagen von Überlastkontrolle in einem allgemeineren Kontext betrachten.

3.6 Grundlagen der Überlastkontrolle

In den vorherigen Abschnitten haben wir sowohl die allgemeinen Grundlagen als auch spezifische TCP-Mechanismen kennengelernt, die für einen zuverlässigen Datentransferdienst angesichts von Paketverlust notwendig sind. Wir erwähnten bereits, dass in der Praxis solche Verluste normalerweise aus dem Überlaufen von Router-Puffern resultieren, während das Netz überlastet ist. Übertragungswiederholungen behandeln daher ein Symptom von Netzüberlast (den Verlust eines bestimmten Transportschichtsegmentes), gehen aber nicht die Ursache der Überlast an: Zu viele Quellen versuchen, Daten mit zu hoher Geschwindigkeit zu senden. Um die Ursache der Netz-

überlast zu beseitigen, werden Mechanismen gebraucht, um die Sender im Fall von Überlast zu drosseln.

In diesem Abschnitt behandeln wir das Problem der Überlastkontrolle in einem allgemeinen Kontext, wobei wir zu verstehen versuchen, warum Überlast nachteilig ist und wie sie sich in der Leistung der Anwendungen auf den oberen Schichten manifestiert. Wir werden verschiedene Ansätze kennenlernen, um Überlast zu vermeiden oder darauf zu reagieren. Diese eher allgemeine Betrachtung der Überlastkontrolle ist angebracht, weil sie, wie der zuverlässige Datentransfer, hoch oben auf unserer „Top Ten"-Liste der fundamental wichtigen Probleme im Bereich Netzwerke steht. Wir beenden diesen Abschnitt mit einer Diskussion der Überlastkontrolle des **Available-Bitrate-Dienstes** (**ABR**, *verfügbare Bitrate*) in **Asynchronous-Transfer-Mode**-Netzen (**ATM**). Der nachfolgende Abschnitt enthält dann eine detaillierte Betrachtung des TCP-Überlastkontrollalgorithmus.

3.6.1 Ursachen und Kosten von Überlast

Beginnen wir unsere allgemeine Diskussion der Überlastkontrolle mit einer Betrachtung dreier zunehmend komplexerer Szenarien, in denen Überlast auftritt. In jedem Fall sehen wir uns an, warum überhaupt Überlast auftritt und welche Kosten die Überlast verursacht (in Bezug auf unvollständig genutzte Ressourcen und geringere Leistung für die Endsysteme). Wir konzentrieren uns noch nicht darauf, wie wir auf Überlast reagieren oder sie vermeiden können, sondern auf das einfachere Thema, zu verstehen, was geschieht, wenn die Hosts ihre Übertragungsrate erhöhen und das Netzwerk überlastet wird.

Szenario 1: zwei Quellen, ein Router mit unendlichen Puffern

Wir beginnen mit dem vielleicht einfachsten Überlastszenario: Von zwei Hosts (A und B) geht je eine Verbindung aus. Die beiden Verbindungen teilen sich einen gemeinsamen Router zwischen Quelle und Ziel, wie ▶Abbildung 3.43 zeigt.

Lassen Sie uns annehmen, dass die Anwendung auf Host A Daten mit der durchschnittlichen Rate λ_{in} Byte/Sekunde sendet (zum Beispiel das Weiterleiten von Daten an das Transportschichtprotokoll über einen Socket). Dabei werden alle Daten nur einmal an den Socket gesendet. Das zugrunde liegende Transportschichtprotokoll ist einfach. Daten werden verkapselt und versendet; es gibt keine Fehlerkorrektur (zum Beispiel Übertragungswiederholung), Flusskontrolle oder Überlastkontrolle. Ignorieren wir die zusätzlichen Daten für Header-Informationen der Transportschicht und der niedrigeren Schichten, dann beträgt die Geschwindigkeit, mit der Host A Verkehr an den Router bringt, in diesem ersten Szenario λ_{in} Byte/Sekunde. Host B arbeitet auf ähnliche Weise und wir nehmen der Einfachheit halber an, dass er ebenfalls mit einer Rate λ_{in} Byte/Sekunde sendet. Die Pakete der Hosts A und B gehen durch einen gemeinsamen Router und verlassen diesen über dieselbe Verbindung der Kapazität *R*.

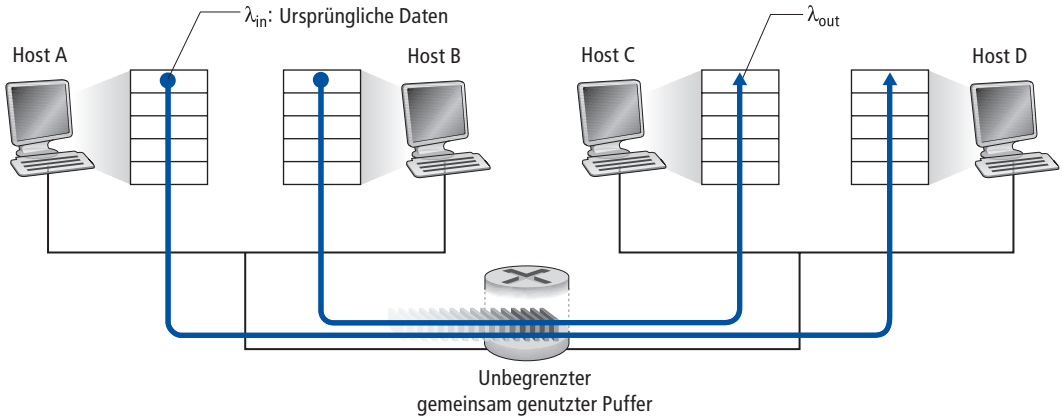


Abbildung 3.43: Überlastszenario 1: Zwei Verbindungen teilen sich einen einzelnen Router mit unendlichem Puffer

Der Router hat Puffer, die es ihm ermöglichen, eingehende Pakete zu speichern, wenn die Paketankunftsrate die Kapazität der ausgehenden Verbindung übersteigt. In diesem ersten Szenario nehmen wir an, dass der Router einen unendlich großen Puffer hat.

▶Abbildung 3.44 zeigt die Leistung der Verbindung des Hosts A in diesem ersten Szenario. Der linke Graph zeigt den **Durchsatz pro Verbindung** (die Anzahl der Bytes pro Sekunde beim Empfänger) als Funktion der Senderate. Für Senderaten zwischen 0 und $R/2$ ist der Durchsatz beim Empfänger gleich der Senderate der Quelle – alles, was vom Sender verschickt wird, wird vom Empfänger mit begrenzter Verzögerung empfangen. Steigt die Senderate jedoch über $R/2$, ist der Durchsatz weiterhin nur $R/2$. Diese Obergrenze für den Durchsatz ist eine Folge des Teilens der Verbindungskapazität zwischen beiden Verbindungen. Die Verbindung kann einfach keine Pakete mit einer stetigen Rate größer als $R/2$ an den Empfänger senden. Ganz gleich, wie hoch Host A und B ihre Senderaten setzen, keiner von ihnen wird jemals einen Durchsatz erleben, der größer ist als $R/2$.

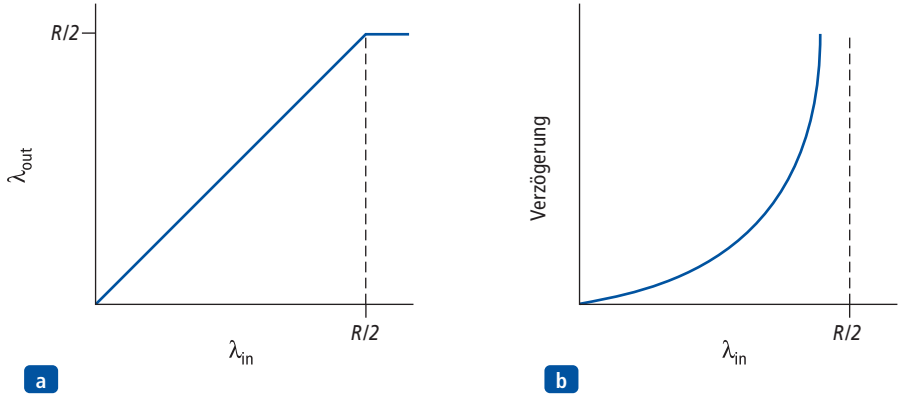


Abbildung 3.44: Überlastszenario 1: Durchsatz und Verzögerung als Funktion der Quelldatenrate

Ein Durchsatz pro Verbindung von $R/2$ scheint doch eigentlich eine gute Sache zu sein, weil die gesamte Verbindung genutzt wird, um Pakete an ihre Zielorte zu liefern. Der rechte Graph in Abbildung 3.44 zeigt jedoch die Folgen, wenn mit nahezu voll ausgelasteter Verbindungskapazität gearbeitet wird. Sobald sich die Senderate $R/2$ (von links) annähert, wird die durchschnittliche Verzögerung immer größer. Übersteigt die Senderate $R/2$, ist die durchschnittliche Zahl der Pakete in der Warteschlange des Routers unbegrenzt und die durchschnittliche Verzögerung zwischen Quelle und Ziel wird unendlich (unter der Voraussetzung, dass die Verbindungen über einen unendlichen Zeitraum mit dieser Senderate arbeiten und dass unendlich große Puffer verfügbar sind). Während also ein Gesamtdurchsatz von beinahe R vom Standpunkt des Durchsatzes durchaus ideal erscheinen mag, ist er vom Standpunkt der Verzögerung weit von einer Idealsituation entfernt. *Sogar in diesem (extrem) idealisierten Szenario haben wir schon einen Kostenfaktor eines überlasteten Netzwerkes gefunden: große Warteschlangenverzögerungen treten dort auf, wo sich die Paketempfangsrate der Verbindungskapazität annähert.*

Szenario 2: zwei Quellen und ein Router mit beschränktem Puffer

Lassen Sie uns jetzt Szenario 1 in zwei Aspekten etwas verändern Abbildung 3.45. Zunächst werden wir von einem begrenzten Routerpuffer ausgehen. Eine Folge dieser Annahme aus der realen Welt besteht darin, dass Pakete verworfen werden, wenn sie an einem bereits vollen Puffer ankommen. Zusätzlich nehmen wir nun auch an, dass jede Verbindung zuverlässig sein soll. Wird ein Paket, das ein Transportschichtsegment enthält, vom Router verworfen, überträgt es der Sender schließlich erneut. Weil Pakete nochmals übertragen werden können, müssen wir jetzt mit unserer Verwendung des Begriffs Senderate vorsichtiger sein. Insbesondere werden wir wieder die Rate, mit der die Anwendung ursprünglich Daten in den Socket sendet, als λ_{in} Byte/Sekunde bezeichnen. Die Rate, mit der die Transportschicht Segmente ins Netz sendet (die Originaldaten und erneut übertragene Daten), beträgt λ'_{in} Byte/Sekunde. Sie wird bisweilen als die **angebotene Last** *(offered load)* bezeichnet.

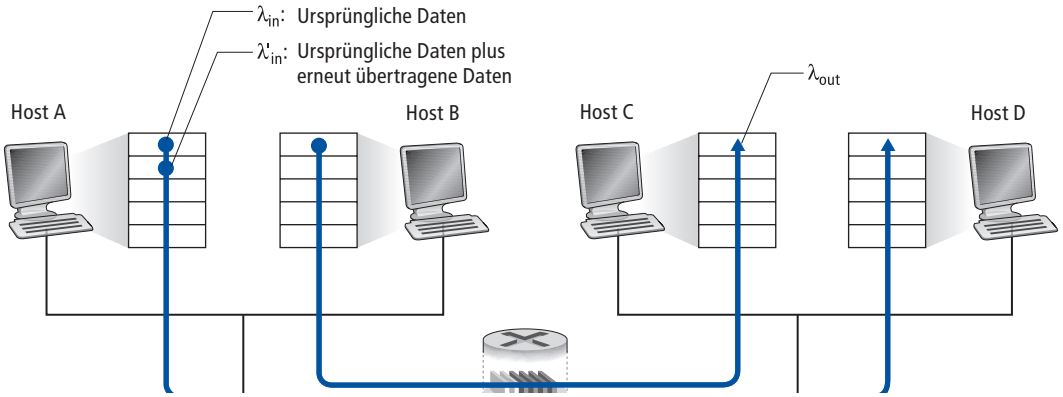


Abbildung 3.45: Überlastszenario 2: Zwei Verbindungen mit Übertragungswiederholungen teilen sich einen Router mit endlichem Puffer

Die in Szenario 2 erreichte Leistung hängt deutlich davon ab, wie die Übertragungswiederholung durchgeführt wird. Betrachten Sie zuerst den unrealistischen Fall, dass Host A auf irgendeine (magische) Weise bestimmen kann, ob ein Puffer im Router frei ist und nur dann Pakete sendet. In diesem Fall würde kein Verlust auftreten, λ_{in} wäre gleich λ'_{in} und der Durchsatz der Verbindung wäre gleich λ_{in}. Dieser Fall wird in ▶Abbildung 3.46 (a) dargestellt. Vom Standpunkt des Durchsatzes ist die Leistung ideal – alles, was verschickt wird, wird auch empfangen. Beachten Sie, dass die Senderate des durchschnittlichen Hosts in diesem Szenario $R/2$ nicht übersteigen kann, da wir voraussetzen, dass Paketverlust nie stattfindet.

Beachten Sie danach den etwas realistischeren Fall, in dem der Sender nur dann die Übertragung wiederholt, wenn er sicher weiß, dass ein Paket verloren gegangen ist. (Auch diese Annahme überspannt etwas die Realität. Jedoch wäre es möglich, dass der sendende Host seinen Timeout so groß setzt, dass ein nicht bestätigtes Paket praktisch als verloren gegangen betrachtet werden kann.) In diesem Fall könnte sich die Leistung etwa entsprechend Abbildung 3.46 (b) verhalten. Um zu erkennen, was hier geschieht, betrachten Sie den Fall, dass die angebotene Last λ'_{in} (die Rate der ursprünglichen Datenübertragung plus der erneuten Übertragungen) gleich $R/2$ ist. Gemäß Abbildung 3.46 (b) ist bei dieser angebotenen Last die Rate, mit der die Daten an die empfangende Anwendung übermittelt werden, $R/3$. Daher enthalten die 0,5 R gesendete Daten $0{,}333 \cdot R$ Byte/Sekunde (im Durchschnitt) an Originaldaten und $0{,}166 \cdot R$ Byte/Sekunde (im Durchschnitt) an wiederholt übertragenen Daten. *Wir sehen hier einen anderen Kostenfaktor eines überlasteten Netzwerkes – der Absender muss Übertragungen wiederholen, um wegen Pufferüberläufen verworfene (und somit verloren gegangene) Pakete zu kompensieren.*

Betrachten wir abschließend den Fall, dass beim Sender vorzeitig Timer auslaufen und so Übertragungswiederholungen für Pakete auftreten können, die zwar in der Warteschlange verzögert wurden, aber nicht wirklich verloren gegangen sind. In diesem Fall können sowohl das Originaldatenpaket als auch die Übertragungswiederholung den Empfänger erreichen. Natürlich braucht der Empfänger nur eine Kopie dieses Paketes und verwirft die Übertragungswiederholung. In diesem Fall war die vom Router geleistete Arbeit beim Weiterleiten der erneut übertragenen Kopie des Originalpaketes ver-

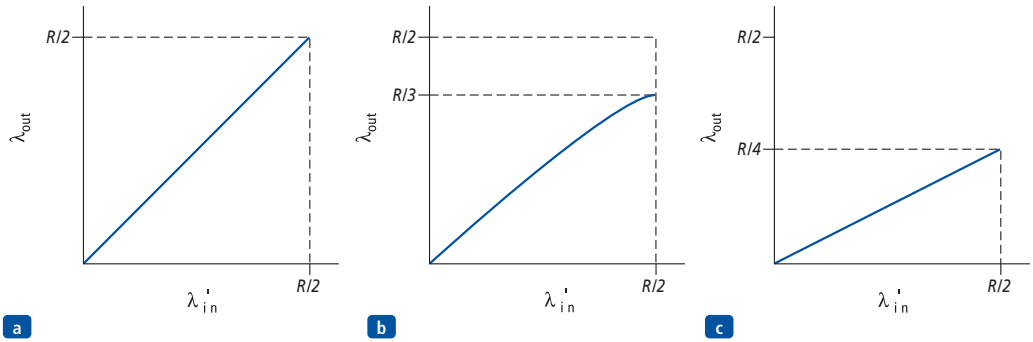


Abbildung 3.46: Szenario 2: Leistung bei endlichen Puffern

geudet, weil der Empfänger das Original dieses Paketes schon erhalten hatte. Der Router hätte die Übertragungskapazität der Verbindung besser verwendet, um stattdessen ein anderes Paket weiterzuleiten. *Hier zeigt sich noch ein weiterer Kostenfaktor eines überlasteten Netzwerkes – unnötige Übertragungswiederholungen durch den Absender bei großen Verzögerungen bewirken, dass ein Router seine Verbindungsbandbreite zum Weiterleiten unnötiger Kopien eines Paketes verwendet.* Abbildung 3.46 (c) zeigt den Durchsatz in Abhängigkeit von der angebotenen Last, wenn von jedem Paket angenommen wird, dass es (im Durchschnitt) zweimal vom Router weitergeleitet wird. Dann hat der Durchsatz einen asymptotischen Wert von $R/4$, während die angebotene Last gegen $R/2$ geht.

Szenario 3: vier Quellen, Router mit begrenzten Puffern und Multihop-Pfade

In unserem abschließenden Stauszenario übertragen vier Hosts Pakete, jeder über überlappende, zwei Router durchquerende Pfade, wie in ▶ Abbildung 3.47 dargestellt. Wir nehmen wieder an, dass jeder Host einen Timeout-/Übertragungswiederholungsmechanismus verwendet, um einen zuverlässigen Datentransferdienst durchzuführen, dass alle Hosts denselben Wert für λ_{in} haben, und dass alle Leitungen die Kapazität R Byte/Sekunde haben.

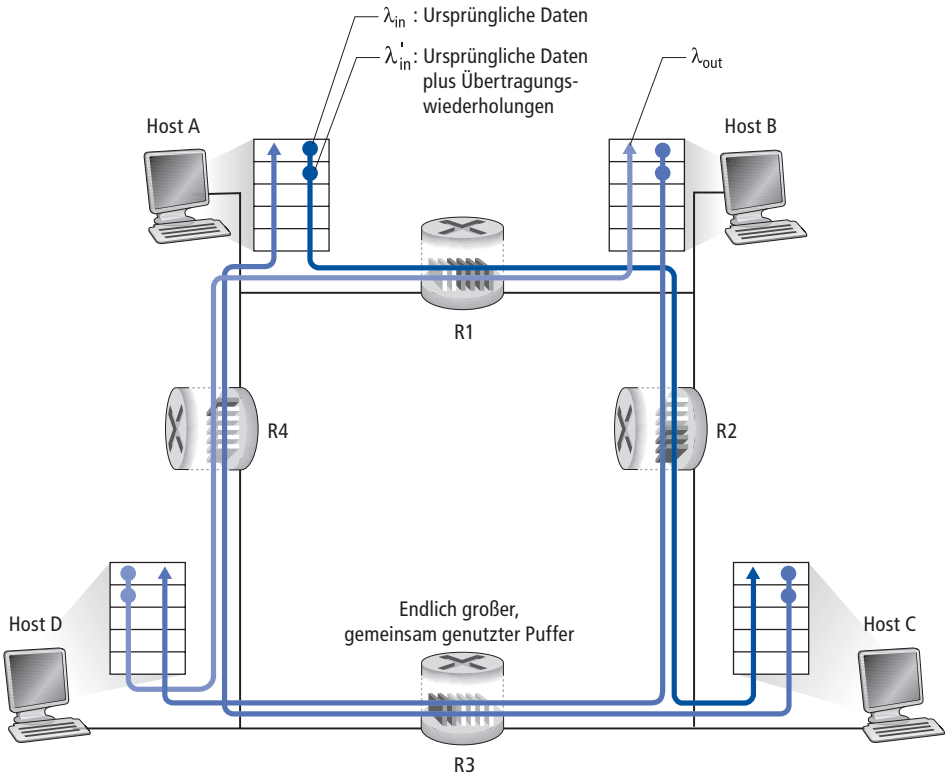


Abbildung 3.47: Vier Sender, Router mit endlichen Puffern und Multihop-Pfade

Betrachten Sie die Verbindung von Host A zu Host C, welche die Router R1 und R2 passiert. Die Verbindung A-C teilt sich Router R1 mit der Verbindung D-B und Router R2 mit der Verbindung B-D. Für äußerst kleine Werte λ_{in} sind Pufferüberläufe selten (wie in den Überlastszenarien 1 und 2) und der Durchsatz ist gleich der angebotenen Last. Für geringfügig größere Werte von λ_{in} ist der zugehörige Durchsatz ebenfalls größer, da weitere Originaldaten ins Netz gesendet und zum Zielort geliefert werden, während Pufferüberläufe immer noch selten sind. Daher führt bei kleinen Werten von λ_{in} eine Zunahme von λ_{in} zu einer Zunahme von λ_{out}.

Nachdem wir den Fall äußerst geringen Verkehrsaufkommens betrachtet haben, lassen Sie uns als Nächstes den Fall untersuchen, in dem λ_{in} (und daher λ'_{in}) extrem groß werden. Betrachten Sie Router R2. Der Verkehr von A nach C, der bei Router R2 ankommt (nachdem er von R1 weitergeleitet wurde), kann bei R2 eine Ankunftsrate von höchstens *R* haben, also der Kapazität der Leitung von R1 nach R2, ganz egal, welchen Wert λ_{in} hat. Wenn λ'_{in} für alle Verbindungen (einschließlich B-D) extrem groß ist, kann die Ankunftsrate des Verkehrs von B an D bei R2 viel größer sein als diejenige des Verkehrs von A zu C. Weil beide am Router R2 um den beschränkten Pufferplatz konkurrieren, verringert sich der A-C-Verkehr, der erfolgreich durch R2 geht (also nicht durch Pufferüberlauf verloren geht) in dem Maße, in dem die angebotene Last von B-D immer größer wird. Schließlich, wenn die angebotene Last gegen unendlich geht, wird ein leerer Puffer an R2 sofort von einem B-D-Paket gefüllt und der Durchsatz der Verbindung A-C an R2 *geht gegen null*. Dies wiederum *impliziert, dass der Ende-zu-Ende Durchsatz von A-C für den Fall intensiven Verkehrs gegen null geht.* Diese Überlegungen führen zum in ▶Abbildung 3.48 gezeigten Zusammenhang zwischen angebotener Last und erreichtem Durchsatz.

Der Grund für das Absinken des Durchsatzes bei zu sehr wachsender angebotener Last ist offensichtlich, wenn man das Maß an verschwendeter Arbeit betrachtet, die das Netz durchführen muss. Im Fall eines hohen Verkehrsaufkommens, wie er oben umrissen wurde, ist die vom ersten Router auf dem Weg geleistete Arbeit beim Weiterleiten des Paketes an den zweiten Router jedes Mal vergeudet, wenn das weitergeleitete Paket auf seinem weiteren Weg verworfen wird. Für das Netzwerk wäre es

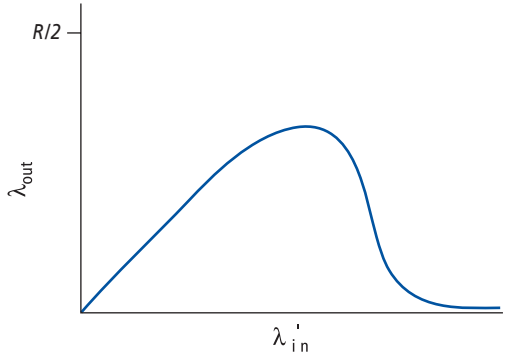


Abbildung 3.48: Szenario 3: Leistung bei endlichen Puffern und Multihop-Pfaden

genauso gut (oder genauer gesagt genauso schlecht) gewesen, wenn der erste Router das Paket einfach verworfen und gar nichts getan hätte. Genauer gesagt wäre die Übertragungskapazität, die der erste Router dafür verwendet, das Paket an den zweiten Router weiterzuleiten, viel profitabler genutzt worden, ein anderes Paket zu übertragen. (Beispielsweise könnte es eine geschickte Idee sein, bei der Auswahl des zu sendenden Paketes diejenigen zu bevorzugen, die bereits eine Reihe von Routern durchquert haben.) *Wieder erkennen wir einen Kostenfaktor beim Verwerfen eines Paketes aufgrund von Überlast: Wenn ein Paket unterwegs verworfen wird, dann ist die Übertragungskapazität verschwendet, die auf jeder der früheren Leitungen benötigt wurde, um es bis zu dem Punkt zu bringen, an dem es verworfen wird.*

3.6.2 Ansätze zur Überlastkontrolle

In Abschnitt 3.7 werden wir detailliert den spezifischen Ansatz von TCP zur Überlastkontrolle untersuchen. Hier besprechen wir die beiden in der Praxis genutzten generellen Ansätze und diskutieren spezifische Netzwerkarchitekturen und Überlastkontrollprotokolle, die diesen Ansätzen Leben einhauchen.

Im Wesentlichen können wir Überlastkontrollansätze dahingehend unterscheiden, ob die Netzwerkschicht der Transportschicht irgendwelche explizite Hilfe bietet:

- *Ende-zu-Ende-Überlastkontrolle.* In einem Ende-zu-Ende-Ansatz zur Überlastkontrolle stellt die Netzwerkschicht der Transportschicht *keine explizite* Hilfe für Überlastkontrollzwecke zur Verfügung. Sogar die Existenz von Überlast im Netz muss von den Endsystemen allein auf Basis von Beobachtungen des Netzwerkverhaltens erkannt werden (zum Beispiel Paketverluste und Verzögerung). Wir werden in Abschnitt 3.7 sehen, das TCP notwendigerweise diesen Ende-zu-Ende-Ansatz für die Überlastkontrolle verwenden muss, da die IP-Schicht den Endsystemen bezüglich Netzüberlast keinerlei Rückmeldung liefert. TCP-Segmentverlust (der durch einen Timeout oder drei doppelte Bestätigungen erkannt wird) wird als Hinweis auf Überlast betrachtet und TCP verringert in der Folge seine Fenstergröße. Wir werden außerdem auch einen neueren Vorschlag zur TCP-Überlastkontrolle kennenlernen, der steigende Rundlaufzeiten als Zeichen einer gestiegenen Netzüberlast verwendet.
- *Netzwerkunterstützte Überlastkontrolle.* Bei der netzwerkunterstützten Überlastkontrolle liefern Netzwerkschichtkomponenten (das heißt Router) dem Sender explizite Rückmeldungen bezüglich des Lastzustandes im Netz. Diese Rückmeldung kann ganz einfach sein, etwa ein einzelnes Bit, das Überlast auf einer Verbindung anzeigt. Dieser Ansatz wurde in der frühen IBM SNA- [Schwartz 1982] und in der DEC DECnet-Architektur [Jain 1989; Ramakrishnan 1990] verwendet. Er wurde vor kurzem auch für TCP/IP-Netzwerke vorgeschlagen [Floyd TCP 1994; RFC 3168] und wird zudem bei ATMs Available-Bit-Rate-Überlastkontrolle verwendet, wie wir unten noch sehen werden. Weitere, komplexere Rückmeldungen aus dem Inneren des Netzwerkes sind ebenfalls möglich. Zum Beispiel ermöglicht es eine Form

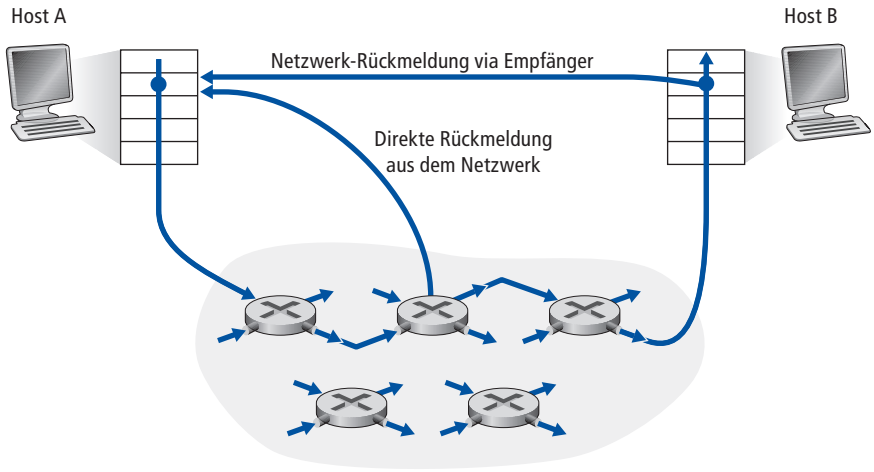


Abbildung 3.49: Zwei Pfade für die Rückmeldung, wenn das Netzwerk selbst die Überlast feststellt

der ATM-ABR-Überlastkontrolle, die wir in Kürze kennenlernen werden, einem Router, explizit den Sender über die Übertragungsgeschwindigkeit zu informieren, die er (der Router) auf einer ausgehenden Leitung bereitstellen kann. Das XCP-Protokoll [Katabi 2002] liefert jeder Quelle vom Router berechnete Rückmeldungen, die in den Paket-Header geschrieben werden. Aus ihnen lässt sich entnehmen, wie diese Quelle die Übertragungsrate erhöhen kann oder verringern sollte.

Bei der netzwerkunterstützten Überlastkontrolle wird die Überlastinformation normalerweise auf eine von zwei Arten vom Netz an den Sender zurückübertragen, wie ▶Abbildung 3.49 zeigt. Eine direkte Rückmeldung kann von einem Netzwerkrouter an einen Sender erfolgen. In diesem Fall wird die Benachrichtigung normalerweise in Form eines **Choke-Paketes** (*Drossel-Paket*) verschickt (im Grunde genommen besagt es „Ich bin überlastet!“). Bei der zweiten Form der Benachrichtigung markiert der Router ein Paket, das vom Sender zum Empfänger fließt, in einem speziellen Feld oder er aktualisiert einen Wert im Paket-Header, um die Überlast anzuzeigen. Beim Empfang eines derart markierten Paketes informiert der Empfänger den Sender über die Überlast. Beachten Sie, dass diese letztere Form der Benachrichtigung mindestens eine volle Rundlaufzeit in Anspruch nimmt.

3.6.3 Beispiel für netzwerkunterstützte Überlastkontrolle: ATM ABR

Wir beenden diesen Abschnitt mit einer kurzen Fallstudie über die Überlastkontrolle in ATM ABR – ein Protokoll, das einen netzwerkunterstützten Ansatz zur Überlastkontrolle benutzt. Wir betonen, dass unser Ziel hier *nicht* darin besteht, alle Aspekte der ATM-Architektur im Detail zu beschreiben, sondern vielmehr ein Protokoll zu erläutern, das einen deutlich anderen Ansatz zur Überlastkontrolle benutzt als TCP. In der Tat berühren wir nur jene wenigen Aspekte der ATM-Architektur, die wir für das Verständnis der ABR-Überlastkontrolle benötigen.

Im Wesentlichen beruht ATM auf dem Grundgedanken einer **virtuellen Leitung** (*virtual circuit*, *VC*). Erinnern Sie sich aus unserer Diskussion in Kapitel 1, dass dies bedeutet, dass jeder Switch auf dem Weg zwischen Quelle und Ziel den Zustand des VC speichert. Dieser Zustand für jeden VC ermöglicht es, dass ein Paket-Switch das Verhalten einzelner Sender im Auge behält (z.B. ihre durchschnittliche Übertragungsgeschwindigkeit) und so quellenspezifische Maßnahmen zur Überlastkontrolle ergreifen kann (wie dem Absender explizit zu signalisieren, sein Tempo zu reduzieren, wenn der Paket-Switch überlastet wird). Wegen dieses in Netzwerk-Switches vorhandenen Zustandes ist ATM ideal geeignet, netzwerkunterstützte Überlastkontrolle durchzuführen.

ABR wurde als elastischer Datentransferdienst auf eine Art und Weise gestaltet, die an TCP erinnert. Wenn das Netz unterbelastet ist, sollte ABR in der Lage sein, die zusätzlich verfügbare Bandbreite zu nutzen. Ist das Netz überlastet, sollte ABR seine Übertragungsgeschwindigkeit auf eine vorherbestimmte minimale Rate drosseln. Ein detailliertes Tutorial über ATM-ABR-Überlastkontrolle und -Verkehrsmanagement enthält [Jain 1996].

▶Abbildung 3.50 veranschaulicht den Rahmen der ATM-ABR-Überlastkontrolle. Für unsere Diskussion übernehmen wir die ATM-Terminologie (und benutzen z.B. den Begriff *Switch* anstelle von *Router* und den Ausdruck *Zelle* anstelle von *Paket*). Mit dem ATM-ABR-Dienst werden Datenzellen von der Quelle zum Ziel durch eine Reihe von dazwischenliegenden Switches gesendet. Mit den Datenzellen vermischt sind **Ressourcenverwaltungszellen** (**RM-Zellen**, *resource-management cells*); mithilfe dieser RM-Zellen lassen sich Informationen über Überlast zwischen den Hosts und Switches austauschen. Erreicht eine RM-Zelle ihren Zielort, wird sie zum Absender zurückgeschickt (nachdem das Ziel den Inhalt der RM-Zelle möglicherweise modifiziert hat). Es ist einem Switch auch möglich, eine RM-Zelle selbst zu erzeugen und direkt an eine Quelle zu schicken. RM-Zellen können daher dazu benutzt werden, um sowohl direkte Netzwerkrückmeldung zu geben als auch Rückmeldungen über den Empfänger weiterzuleiten, wie in ▶Abbildung 3.50 gezeigt.

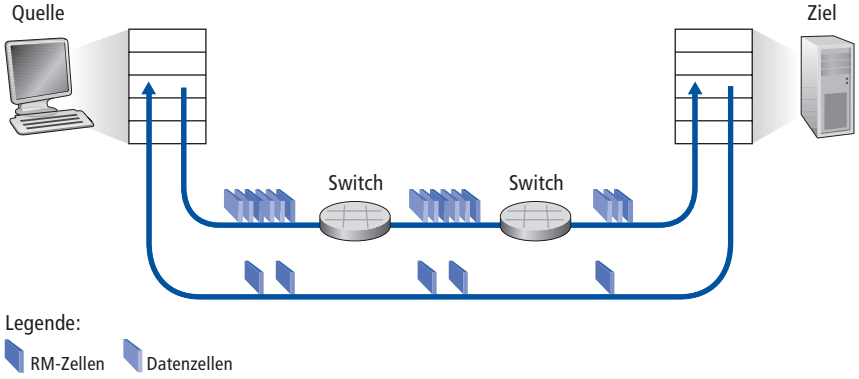


Abbildung 3.50: Überlastkontrolle für ATM-ABR-Dienste

ATM-ABR-Überlastkontrolle ist ein ratenbasierter Ansatz. Das heißt, der Absender berechnet eine maximale Rate, mit der er senden darf, und regelt sich entsprechend der Ergebnisse selbst. ABR unterstützt drei Mechanismen, um Überlastinformation von den Switches zu den Empfängern zu übermitteln:

- *EFCI-Bit*. Jede *Datenzelle* enthält ein Bit für das **explicit forward congestion indication** (**EFCI**, *explizite vorwärtsgerichtete Überlastwarnung*). Ein überlasteter Switch kann das EFCI-Bit in einer Datenzelle auf eins setzen, um dem Zielhost Überlast zu signalisieren. Das Ziel muss das EFCI-Bit in allen erhaltenen Datenzellen überprüfen. Kommt eine RM-Zelle am Zielort an und in der zuletzt erhaltenen Datenzelle war das EFCI-Bit auf eins gesetzt, dann setzt das Ziel das Überlasthinweis-Bit (das CI-Bit) der RM-Zelle auf eins und schickt dem Absender die RM-Zelle zurück. Mithilfe des EFCI-Bit bei Datenzellen und des CI-Bit in RM-Zellen kann ein Absender so von der Überlastung eines Netzwerk-Switches in Kenntnis gesetzt werden.
- *CI- und NI-Bits*. Wie oben erwähnt, sind die Sender-zu-Empfänger-RM-Zellen zwischen die Datenzellen eingestreut. Die Rate, mit der diese Zellen auftauchen, ist ein einstellbarer Parameter, der als Standardwert eine RM-Zelle pro 32 Datenzellen vorgibt. Diese RM-Zellen enthalten ein **Congestion-Indication-Bit** (**CI**, *Hinweis auf Überlast*) und ein **No-Increase-Bit** (**NI**, *keine Steigerung*), die von einem überlasteten Switch gesetzt werden können. Insbesondere kann ein Switch bei geringer Überlast das NI-Bit einer vorbeikommenden RM-Zelle auf eins setzen, während er bei schwerer Überlast das CI-Bit auf eins setzt. Erhält ein Zielhost eine RM-Zelle, schickt er dem Sender die RM-Zelle mit den unveränderten CI- und NI-Bits zurück (außer, wenn CI als Folge des oben beschriebenen EFCI-Mechanismus vom Ziel auf eins gesetzt wird).
- *Vorgeben expliziter Raten*. Jede RM-Zelle enthält ein 2 Byte großes **Explicit-Rate-Feld** (**ER**, *explizite Rate*). Ein überlasteter Switch kann den im ER-Feld enthaltenen Wert einer vorbeikommenden RM-Zelle senken. Auf diese Weise wird das ER-Feld insgesamt auf die minimale Rate gesetzt, die von allen Switches auf dem Weg zwischen Quelle und Ziel noch unterstützt wird.

Eine ATM-ABR-Quelle stellt die Rate, mit der sie senden kann, als Funktion der Werte in den CI-, NI- und ER-Werten in den zurückgeschickten RM-Zellen ein. Die Regeln für die Durchführung dieser Ratenanpassung sind ziemlich kompliziert und ein bisschen ermüdend. Wir verweisen den interessierten Leser auf [Jain 1996] für Details.

3.7 TCP-Überlastkontrolle

In diesem Abschnitt kehren wir zu unserer Betrachtung von TCP zurück. Wie wir in Abschnitt 3.5 gelernt haben, realisiert TCP einen zuverlässigen Transportdienst zwischen zwei Prozessen, die auf verschiedenen Hosts laufen. Ein anderer wesentlicher Bestandteil von TCP ist sein Überlastkontrollmechanismus. Wie im vorherigen

Abschnitt gezeigt, muss TCP Ende-zu-Ende-Überlastkontrolle anstatt netzwerkunterstützter Überlastkontrolle verwenden, da die IP-Schicht den Endsystemen hinsichtlich der aktuellen Netzlast keine explizite Rückmeldung liefert.

Der von TCP verwendete Ansatz besteht darin, dass jeder Sender die Rate, mit der er Verkehr in seine Verbindung überträgt, als Funktion der wahrgenommenen Überlast im Netz selbst einstellt. Erkennt ein TCP-Sender nur geringe Last auf dem Weg zwischen sich und dem Zielort, dann steigert er sein Sendetempo; nimmt der Sender allerdings Überlast wahr, dann reduziert er sein Sendetempo. Dies wirft drei Fragen auf. Zunächst einmal, wie begrenzt ein TCP-Sender die Rate, mit der er Daten über seine Verbindung sendet? Zweitens, wie erkennt ein TCP-Sender, dass es Überlast auf dem Pfad zwischen sich und dem Zielort gibt? Drittens, welchen Algorithmus sollte der Absender verwenden, um sein Sendetempo als Funktion der erkannten Überlast zwischen den Endpunkten zu ändern? Wir diskutieren diese drei Themen im Kontext des TCP-Reno-Überlastkontrollalgorithmus, der von den meisten modernen Betriebssystemen verwendet wird [Padhye 2001]. Um die Diskussion konkret zu halten, nehmen wir an, dass der TCP-Sender eine große Datei überträgt.

Schauen wir zunächst, wie ein TCP-Sender die Rate begrenzt, mit der er Verkehr in seine Verbindung einbringt. In Abschnitt 3.5 haben wir gesehen, dass jede Seite einer TCP-Verbindung aus einem Eingangspuffer, einem Sendepuffer und mehreren Variablen (LastByteRead, RcvWindow usw.) besteht. Der TCP-Überlastkontrollmechanismus sorgt dafür, dass jede Seite der Verbindung eine zusätzliche Variable verwaltet, das **Congestion Window** *(Überlastfenster)*. Das Congestion Window, kurz *CongWin*, beschränkt die Rate, mit der ein TCP-Sender Verkehr ins Netz senden kann. Konkret darf die Menge an unbestätigten Daten eines Senders nicht das Minimum von *CongWin* und *RcvWindow* übersteigen, d. h.:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \min\{\mathit{CongWin}, \mathit{RcvWindow}\}$$

Um uns auf die Überlastkontrolle (im Gegensatz zur Flusskontrolle) zu konzentrieren, gehen wir von nun an davon aus, dass der TCP-Eingangspuffer so groß ist, dass die Einschränkung durch das Empfangsfenster ignoriert werden kann; dadurch wird die Menge an unbestätigten Daten beim Sender alleine von *CongWin* begrenzt.

Die obige Einschränkung begrenzt die unbestätigten Daten des Senders und damit auch indirekt die Senderate des Absenders. Um das zu verstehen, stellen Sie sich eine Verbindung vor, bei der Verlust und Paketübertragungsverzögerungen vernachlässigbar sind. Grob geschätzt erlaubt die Beschränkung dem Sender am Anfang jeder *RTT*, *CongWin* Datenbytes in die Verbindung zu senden; am Ende der *RTT* erhält der Sender Acknowledgments für die Daten. *Daher ist die Senderate des Absenders grob CongWin/RTT Byte/Sekunde. Durch Einstellen des Wertes von CongWin kann der Absender deshalb die Rate einstellen, mit der er Daten über seine Verbindung sendet.*

Lassen Sie uns als Nächstes darüber nachdenken, wie ein TCP-Sender erkennt, dass es Überlast auf dem Pfad zwischen sich und dem Zielort gibt. Lassen Sie uns ein „Verlustereignis“ an einem TCP-Sender definieren, das entweder aus dem Auftreten eines Timeouts oder dem Erhalt von drei doppelten ACKs vom Empfänger besteht. (Erin-

nern Sie sich an unsere Diskussion in Abschnitt 3.5.4 des Timeout-Ereignisses in ▶Abbildung 3.33 und die anschließende Änderung, um schnelle Übertragungswiederholung beim Empfang von drei doppelten ACKs zu ermöglichen.) Gibt es exzessive Überlast, dann laufen ein (oder mehrere) Routerpuffer entlang des Pfads über, wodurch ein Datagramm (das ein TCP-Segment enthält) verworfen wird. Dieses verlorene Datagramm führt wiederum zu einem Verlustereignis beim Sender – entweder über einen Timeout oder den Erhalt von drei doppelten ACKs – was vom Absender als Hinweis auf Überlast auf dem Pfad zwischen Sender und Empfänger gewertet wird.

Nachdem wir überlegt haben, wie Überlast wahrgenommen wird, lassen Sie uns nun den optimistischeren Fall betrachten, in dem das Netz frei von Überlast ist, d.h., wenn kein Verlustereignis auftritt. In diesem Fall werden Acknowledgments für zuvor unbestätigte Segmente am TCP-Sender empfangen. Wie wir sehen werden, wertet TCP das Eintreffen dieser Bestätigungen als Hinweis, dass alles gut geht – dass Segmente, die ins Netz gesendet werden, erfolgreich am Ziel abgeliefert werden – und vergrößert auf diese Bestätigungen hin sein Congestion Window (und damit seine Übertragungsgeschwindigkeit). Beachten Sie, dass ziemlich langsam eintreffende Bestätigungen (z.B. wenn der Ende-zu-Ende-Pfad eine hohe Verzögerung aufweist oder eine Verbindung mit niedriger Bandbreite benutzt), dazu führen, dass das Congestion Window nur langsam wächst. Wenn andererseits Acknowledgments mit hoher Rate ankommen, dann wird das Congestion Window schneller vergrößert. Weil TCP Acknowledgments quasi als Taktgeber verwendet, um die Vergrößerung seines Congestion Windows zu steuern, wird TCP als **selbsttaktend** *(self-clocking)* bezeichnet.

Wir sind jetzt in der Lage, die Details jenes Algorithmus zu erörtern, den TCP-Sender zur Regulierung ihrer Senderate als Funktion der wahrgenommenen Überlast verwenden. Dieser Algorithmus ist der berühmte **TCP-Überlastkontroll-Algorithmus**. Der Algorithmus hat drei Hauptbestandteile: (1) Additive-Increase, Multiplicative-Decrease, (2) Slow Start und (3) Reaktion auf Timeout-Ereignisse.

Additive-Increase, Multiplicative-Decrease

Der Grundgedanke hinter der TCP-Überlastkontrolle ist, den Sender seine Senderate senken zu lassen, wenn ein Verlustereignis stattfindet (durch Verringerung der Größe seines Congestion Windows, *CongWin*). Da andere TCP-Verbindungen, die durch dieselben überfüllten Router gehen, wahrscheinlich ebenfalls Verlustereignisse erfahren, ist es wahrscheinlich, dass sie ebenfalls ihre Senderate verringern, indem sie ihre eigenen Werte von *CongWin* reduzieren. Insgesamt bedeutet dies, dass Quellen, deren Pfade durch überlastete Router gehen, alle die Geschwindigkeit, mit der sie Verkehr ins Netz senden, reduzieren, wodurch die überfüllten Router entlastet werden. Aber wie sehr sollte ein TCP-Sender sein Congestion Window verkleinern, wenn ein Verlustereignis auftritt? TCP verwendet den Ansatz des sogenannten „Multiplicative Decrease“ *(multiplikative Verkleinerung)*, durch den es den aktuellen Wert von *CongWin* nach einem Verlustereignis halbiert. Beträgt der Wert von *CongWin* eines TCP-Senders gegenwärtig 20 Kbyte und wird ein Verlust wahrgenommen, wird *CongWin*

daher auf 10 Kbyte halbiert. Tritt ein weiteres Verlustereignis ein, wird *CongWin* weiter auf 5 Kbyte reduziert. Möglicherweise fällt der Wert von *CongWin* weiter, allerdings darf er nicht unterhalb 1 *MSS* fallen. (Dies war die grobe Beschreibung, wie sich das Congestion Window nach einem Verlustereignis ändert. Wie wir bald sehen werden, sind die Dinge tatsächlich noch ein bisschen komplizierter.)

Nachdem wir beschrieben haben, wie ein TCP-Sender seine Senderate angesichts einer wahrgenommenen Überlast vermindert, ist es nur natürlich, nun darüber nachzudenken, wie TCP seine Senderate steigern sollte, wenn es keinen Stau erkennt, wenn also ACKs für frühere, noch zu bestätigende Daten eintreffen. Der Gedanke hinter einer Tempoerhöhung ist, dass es wahrscheinlich verfügbare (ungenutzte) Bandbreite gibt, wenn keine Überlast erkennbar ist, die von der TCP-Verbindung ebenfalls verwendet werden könnte. Unter diesen Umständen erhöht TCP allmählich sein Congestion Window und sucht auf dem Ende-zu-Ende-Pfad vorsichtig nach zusätzlich verfügbarer Bandbreite. Dies erfolgt seitens des TCP-Senders, indem er jedes Mal, wenn er eine Bestätigung erhält, den Wert von *CongWin* geringfügig erhöht, mit dem Ziel, *CongWin* um 1 *MSS* je *RTT* zu steigern [RFC 2581]. Es gibt mehrere Wege, das zu bewerkstelligen. In einem häufig benutzten Ansatz erhöht der TCP-Sender sein *CongWin* um *MSS*·(*MSS*/*CongWin*) Byte, sobald eine neue Bestätigung ankommt. Beträgt z.B. *MSS* 1.460 Byte und *CongWin* 14.600 Byte, dann werden zehn Segmente innerhalb einer *RTT* gesandt. Jedes ankommende ACK (wobei wir ein ACK pro Segment annehmen) steigert die Größe des Congestion Windows um 1/10 *MSS*. Dadurch wird nach Empfang der Bestätigungen aller zehn Segmente der Wert des Congestion Windows wie gewünscht um *MSS* zugenommen haben.

Zusammenfassend steigert ein TCP-Sender sein Tempo additiv, wenn er erkennt, dass der Ende-zu-Ende-Pfad frei von Überlast ist, aber er verringert sein Tempo multiplikativ, wenn er (über ein Verlustereignis) wahrnimmt, dass der Pfad überlastet ist. Deshalb wird die TCP-Überlastkontrolle oft als **Additive-Increase, Multiplicative-Decrease-Algorithmus** (**AIMD**) bezeichnet. Die Phase der linearen Zunahme des TCP-Überlastkontrollprotokolls ist als **Congestion Avoidance** *(Überlastvermeidung)* bekannt. Der Wert von *CongWin* durchläuft immer wieder Zyklen, in denen er linear zunimmt, um dann plötzlich auf die Hälfte seines aktuellen Werts zu fallen (wenn ein Verlustereignis stattfindet). Dies führt in langlebigen TCP-Verbindungen zu einem Sägezahnmuster wie in ▶Abbildung 3.51 gezeigt.

Slow Start

Zu Beginn einer TCP-Verbindung wird der Wert von *CongWin* normalerweise mit 1 *MSS* initialisiert [RFC 3390], was zu einer anfänglichen Senderate von grob *MSS*/*RTT* führt. Sind zum Beispiel *MSS* = 500 Byte und *RTT* = 200 ms, dann beträgt die resultierende anfängliche Senderate nur etwa 20 Kbps. Weil die zur Verfügung stehende Bandbreite der Verbindung viel größer als *MSS*/*RTT* ist, wäre es eine Schande, die Senderate nur linear ansteigen zu lassen und unmäßig lange zu warten, bis die Senderate ein akzeptables Maß erreicht hat. Anstatt also die Rate während dieser Anfangsphase linear zu steigern, erhöht ein TCP-Sender sein Tempo exponentiell, indem er den Wert von

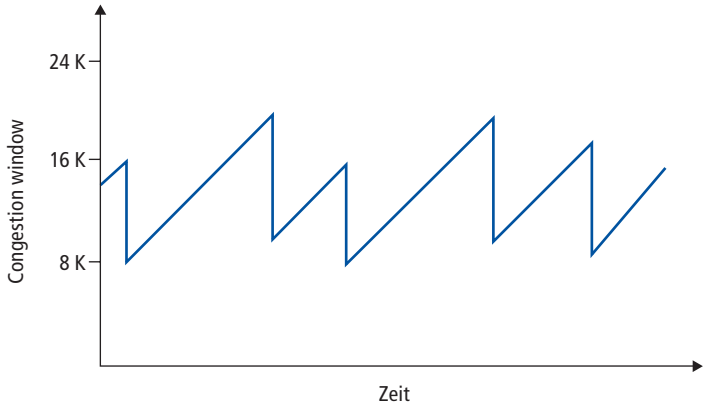


Abbildung 3.51: Überlastkontrolle mit Additive-Increase, Multiplicative-Decrease

CongWin während jeder *RTT* verdoppelt. Der TCP-Sender fährt mit dem exponentiellen Steigern der Senderate fort, bis ein Verlustereignis eintritt, wonach *CongWin* halbiert wird und von nun an, wie oben beschrieben, linear wächst. Während dieser Anfangsphase, die als **Slow Start** (**SS**, *langsamer Anfang*) bezeichnet wird, beginnt der TCP-Sender mit einer langsamen Übertragung (daher der Ausdruck Slow Start), steigert aber sein Sendetempo exponentiell. Der Sender sorgt jedes Mal, wenn ein gesendetes Segment bestätigt wird, für exponentielles Wachstum des Congestion Window, indem er es um eine *MSS* erhöht. Wie in ▶Abbildung 3.52 gezeigt wird, sendet TCP das erste Segment ins Netz und wartet auf eine Bestätigung. Wenn das Segment vor einem Verlustereignis bestätigt wird, steigert der TCP-Sender das Congestion Window um eine *MSS* und schickt zwei Segmente maximaler Größe ab. Werden diese Segmente wieder vor einem Verlustereignis bestätigt, steigert der Absender das Congestion Window um eine *MSS* für jedes der bestätigten Segmente, was zu einem Congestion Window von 4 *MSS* führt. Er schickt deshalb nun vier Segmente maximaler Größe ab. Dieses Verfahren geht so lange weiter, wie Bestätigungen eintreffen oder schließlich ein Verlustereignis stattfindet. Auf diese Art verdoppelt sich der Wert von *CongWin* in jeder *RTT* während der Slow-Start-Phase.

Reaktion auf Timeouts

Das Bild, das wir bisher von TCPs Congestion Window gemalt haben, ist das eines exponentiell schnellen Anwachsens (während des Slow Start), bis ein Verlustereignis eintritt, und dem darauf folgenden Sägezahnmuster des AIMD. Obwohl dieses Bild beinahe zutrifft, wären wir nachlässig, würden wir nicht erwähnen, dass die TCP-Überlastkontrolle in Wahrheit anders auf ein Verlustereignis reagiert, das mittels eines Timeout-Ereignisses wahrgenommen wird, als auf ein Verlustereignis, das durch den Erhalt von drei doppelten ACKs erkannt wird. Nach Letzteren verhält sich TCP wie gerade beschrieben: Das Congestion Window wird halbiert und wächst dann wieder linear. Aber nach einem Timeout beginnt ein TCP-Sender mit einer erneuten Slow-Start-Phase – das heißt, er setzt das Congestion Window auf 1 *MSS* zurück,

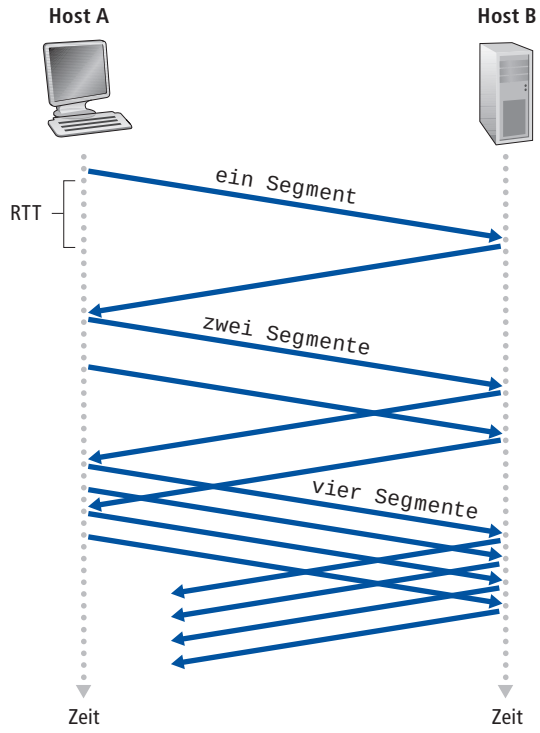


Abbildung 3.52: TCP Slow Start

danach wächst das Congestion Window exponentiell an. Das exponentielle Wachstum des Congestion Windows setzt sich fort, bis *CongWin* die Hälfte des Wertes erreicht, den es vor dem Timeout-Ereignis hatte. Ab diesem Zeitpunkt wächst *CongWin* linear weiter, genau wie im Fall der drei doppelten ACK.

TCP erreicht diese komplexere Dynamik, indem es eine weitere Variable namens **Threshold** *(Schwellwert)* speichert, welche die Fenstergröße bestimmt, bei der der Slow Start endet und die AIMD-basierte Congestion-Avoidance-Phase beginnt. Die Variable Threshold wird anfangs auf einen großen Wert gesetzt (in der Praxis 65 Kbyte [Stevens 1994]), so dass sie zu Beginn keine Auswirkungen hat. Jedes Mal, wenn ein Verlustereignis eintritt, wird der Wert von Threshold auf die Hälfte des aktuellen Wertes von *CongWin* gesetzt. Betrug die Größe des Congestion Window direkt vor einem Verlustereignis 20 Kbyte, dann wird der Wert von Threshold auf 10 Kbyte gesetzt und behält diesen Wert bis zum nächsten Verlustereignis.

Nachdem wir die Variable Threshold beschrieben haben, können wir jetzt genau angeben, wie *CongWin* sich nach einem Timeout-Ereignis verhält. Wie oben gezeigt, beginnt ein TCP-Sender nach einem Timeout-Ereignis mit der Slow-Start-Phase. Während dieser Zeit steigert er den Wert von *CongWin* exponentiell, bis er Threshold erreicht. Dann beginnt TCP mit der Congestion-Avoidance-Phase, in der *CongWin* wie zuvor beschrieben linear anwächst.

Zustand	Ereignis	Reaktion der TCP-Überlastkontrolle	Kommentar
Slow Start (SS)	ACK für zuvor unbestätigte Daten empfangen	*CongWin* = *CongWin* + *MSS*, Wenn (*CongWin* > Threshold), setze Zustand auf „Congestion Avoidance"	Führt zu einer Verdopplung von *CongWin* in jeder *RTT*.
Congestion Avoidance (CA)	ACK für zuvor unbestätigte Daten empfangen	*CongWin* = *CongWin* + *MSS*·(*MSS*/*CongWin*)	Additive Increase, resultiert in einer Zunahme von *CongWin* um 1 *MSS* jede *RTT*.
SS oder CA	Verlustereignis entdeckt durch drei doppelte ACKs	Threshold = *CongWin*/2, *CongWin* = Threshold, setze Zustand auf „Congestion Avoidance"	Fast Recovery, implementiert Multiplicative Decrease. *CongWin* kann nicht unter 1 *MSS* fallen.
SS oder CA	Timeout	Threshold = *CongWin*/2, *CongWin* = 1 *MSS*, setze Zustand auf „Slow Start"	Erneute Slow-Start-Phase
SS oder CA	Doppeltes ACK empfangen	Erhöhe den Zähler für doppelte ACKs für das bestätigte Segment	*CongWin* und Threshold werden nicht verändert.

Tabelle 3.3: TCP-Überlastkontrolle [RFC 2581], unter der Voraussetzung, dass der anfängliche Wert von *CongWin* gleich *MSS* ist, dass der Anfangswert von Threshold groß ist (z. B. 65 Kbyte [Stevens 1994]) und dass der TCP-Sender in der Slow-Start-Phase beginnt. Die dargestellten Zustandswerte sind die Zustände des TCP-Senders unmittelbar bevor die beschriebenen Ereignisse eintreten. Siehe [RFC 2581] für zusätzliche Details

Unsere Diskussion des TCP-Überlastkontrollalgorithmus ist in ▶ Tabelle 3.3 zusammengefasst. An dieser Stelle liegt die Frage auf der Hand, warum sich die TCP-Überlastkontrolle nach einem Timeout-Ereignis anders verhält als nach dem Erhalt dreier doppelter ACKs. Genau gefragt, warum benimmt sich ein TCP-Sender nach einem Timeout-Ereignis sehr vorsichtig und verringert sein Congestion Window auf 1 *MSS*, während er nach dem Erhalten dreier doppelter ACKs sein Congestion Window nur auf die Hälfte reduziert? Interessanterweise kappte eine frühe Version von TCP, bekannt als **TCP Tahoe,** ihr Congestion Window bedingungslos auf 1 *MSS* und begann nach jeder Art von Verlustereignis mit der Slow-Start-Phase. Die neuere Version von TCP, **TCP Reno**, vermeidet die Slow-Start-Phase nach drei doppelten ACKs. Die Motivation hinter dieser Änderung ist, dass trotz des Verlustes eines Paketes die Ankunft dreier doppelter ACKs anzeigt, dass einige Segmente (genauer drei zusätzliche Segmente nach dem verlorenen Segment) beim Absender angekommen sind. Daher zeigt sich das Netz, im Gegensatz zu dem Fall eines Timeouts, fähig, zumindest einige Segmente zuzustellen, selbst wenn andere wegen Überlast verloren gehen. Dieses Weglassen der Slow-Start-Phase nach dem dritten doppelten ACK wird **Fast Recovery** *(schnelle Erholung)* genannt.

▶ Abbildung 3.53 zeigt die Veränderung des Congestion Windows von TCP sowohl für TCP Reno als auch für TCP Tahoe. In dieser Abbildung ist Threshold anfangs

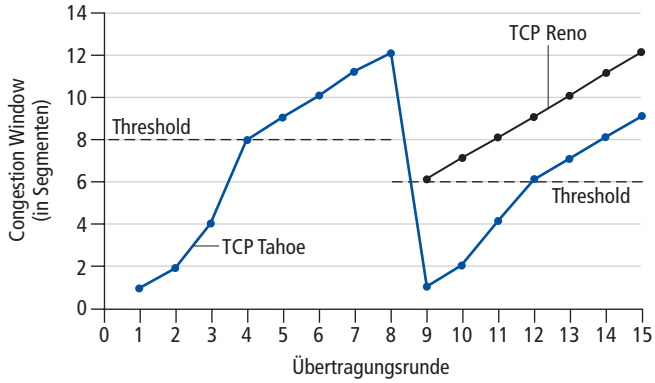


Abbildung 3.53: Entwicklung des Überlastkontrollfensters bei TCP (sowohl Tahoe als auch Reno)

gleich 8 *MSS*. Während der ersten acht Übertragungsrunden ergreifen Tahoe und Reno identische Maßnahmen. Das Congestion Window steigt während des Slow Start exponentiell an und erreicht den Wert von Threshold bei der vierten Übertragungsrunde. Das Congestion Window steigt danach linear, bis drei doppelte ACKs direkt nach der achten Übertragung auftreten. Beachten Sie, dass das Congestion Window beim Eintritt dieses Verlustereignisses gleich 12 *MSS* ist. Threshold wird dann auf 0,5 *CongWin* = 6 *MSS* gesetzt. Bei TCP Reno wird das Congestion Window auf *CongWin* = 6 *MSS* gesetzt und wächst dann linear. Unter TCP Tahoe wird das Congestion Window auf 1 *MSS* gesetzt und wächst exponentiell, bis Threshold erreicht ist. Dieser Überlastkontrollalgorithmus beruht auf der Arbeit von V. Jacobson [Jacobson 1988]; eine Reihe von Änderungen an Jacobsons ursprünglichem Algorithmus werden in [Stevens 1994] und in [RFC 2581] beschrieben. Wie oben erwähnt, verwenden die meisten TCP-Implementierungen heute den Reno-Algorithmus. Mittlerweile wurden viele Variationen des Reno-Algorithmus vorgeschlagen [RFC 3782; RFC 2018].

Der als TCP Vegas vorgeschlagene Algorithmus [Brakmo 1995; Ahn 1995] versucht Überlast zu vermeiden, während ein guter Durchsatz stets erhalten bleibt. Der Grundgedanke von Vegas ist (1) Überlast in den Routern zwischen Quelle und Ziel zu entdecken, schon bevor Paketverlust auftritt und (2) lineares Absenken der Rate, sobald dieser kurz bevorstehende Paketverlust erkannt wird. Der unmittelbar bevorstehende Paketverlust wird durch Beobachten der *RTT* vorhergesagt. Je länger die *RTT* der Pakete, desto größer die Überlast in den Routern.

Makroskopische Beschreibung des TCP-Durchsatzes

Angesichts des Sägezahnverhaltens von TCP liegt die Frage nach dem durchschnittlichen Durchsatz (also der durchschnittlichen Senderate) einer langlebigen TCP-Verbindung auf der Hand. In dieser Analyse ignorieren wir die Slow-Start-Phasen, die nach Timeout-Ereignissen auftreten. (Diese Phasen sind normalerweise sehr kurz, da der Sender schnell die exponentielle Phase verlässt.) Während eines gegebenen Rundlauf-Intervalls ist die Rate, mit der TCP Daten sendet, eine Funktion des Congestion

Windows und der aktuellen *RTT*. Beträgt die Fenstergröße *w* Bytes und die aktuelle Rundlaufzeit *RTT* Sekunden, dann ist die Übertragungsgeschwindigkeit von TCP grob *w*/*RTT*. TCP sucht dann nach zusätzlicher Bandbreite, indem es *w* um 1 *MSS* jede *RTT* steigert, bis ein Verlustereignis auftritt. Bezeichnen wir mit *W* den Wert, den *w* hat, wenn ein Verlustereignis stattfindet. Unter der Annahme, dass *RTT* und *W* während der Dauer der Verbindung ungefähr konstant bleiben, reicht die Spanne der TCP-Übertragungsraten von *W*/(2 *RTT*) bis *W*/*RTT*.

Diese Annahmen führen zu einem extrem vereinfachten makroskopischen Modell für das Verhalten von TCP im Gleichgewicht. Das Netzwerk verwirft ein Paket der Verbindung, wenn das Tempo auf *W*/*RTT* steigt; die Rate wird dann halbiert, wonach sie um *MSS*/*RTT* jedes *RTT* steigt, bis *W*/*RTT* wieder erreicht wird. Dieser Prozess wiederholt sich immer wieder. Weil der Durchsatz von TCP (also seine Rate) zwischen den beiden Extrema linear ansteigt, haben wir

$$\text{durchschnittlicher Durchsatz einer Verbindung} = \frac{0{,}75 \cdot \mathrm{W}}{RTT}$$

Mithilfe dieses extrem idealisierten Modells für die Gleichgewichtsdynamik von TCP können wir auch einen interessanten Ausdruck ableiten, der die Verlustrate einer Verbindung auf deren verfügbare Bandbreite bezieht [Mahdavi 1997]. Diese Ableitung ist in den Übungsaufgaben umrissen. Ein weiterentwickeltes Modell, das empirisch mit gemessenen Daten übereinstimmt, beschreibt [Padhye 2000].

Die Zukunft von TCP

Wir müssen uns ständig vor Augen halten, dass sich die TCP-Überlastkontrolle im Lauf vieler Jahre entwickelt hat und bis heute weiterentwickelt. Eine Zusammenfassung der TCP-Überlastkontrolle bis zu den späten 1990er enthält [RFC 2581]; für eine Diskussion jüngerer Entwicklungen siehe [Floyd 2001]. Was für das Internet gut war, als der Großteil der TCP-Verbindungen SMTP, FTP und Telnet-Datenverkehr transportierte, ist nicht unbedingt für das HTTP-dominierte Internet von heute oder für ein zukünftiges Internet mit Diensten, die noch gar nicht bekannt sind, geeignet.

Die Notwendigkeit einer ständigen Weiterentwicklung von TCP kann durch Betrachtung der Hochgeschwindigkeits-TCP-Verbindungen verdeutlicht werden, die für Grid-Computing-Anwendungen erforderlich sind [Foster 2002]. Betrachten Sie zum Beispiel eine TCP-Verbindung mit 1.500 Byte-Segmenten und einer *RTT* von 100 ms und nehmen Sie an, dass wir durch diese Verbindung Daten mit 10 Gbps senden wollen. Folgen wir [RFC 3649], so stellen wir fest, dass bei Verwendung der obigen TCP-Durchsatzformel das Congestion Window für einen 10 Gbps-Durchsatz die durchschnittliche Größe von 83.333 Segmenten haben müsste. Das sind viele Segmente und wir müssen befürchten, dass eines dieser 83.333 Segmente unterwegs verloren gehen könnte. Was würde im Fall eines Verlustes geschehen? Oder, anders herum gefragt, welcher Bruchteil der gesendeten Segmente dürfte verloren gehen, damit der in ▶Tabelle 3.3 skizzierte TCP-Überlastkontrollalgorithmus immer noch die gewünsch-

ten 10 Gbps erreichen kann? In den Übungsaufgaben für dieses Kapitel werden Sie durch die Ableitung einer Formel geführt, die den Durchsatz einer TCP-Verbindung als Funktion der Verlustrate *(L)*, der Rundlaufzeit *(RTT)* und der maximalen Segmentgröße *(MSS)* darstellt:

$$\text{durchschnittlicher Durchsatz einer Verbindung} = \frac{1{,}22 \cdot \text{MSS}}{RTT\sqrt{L}}$$

Mithilfe dieser Formel sehen wir, dass, um einen Durchsatz von 10 Gbps zu erreichen, der heutige TCP-Überlastkontrollalgorithmus nur eine Segmentverlustwahrscheinlichkeit von $2 \cdot 10^{-10}$ tolerieren kann (anders ausgedrückt ein Verlustereignis alle 5.000.000.000 Segmente) – das ist eine sehr niedrige Rate! Diese Beobachtung hat eine Reihe von Forschern dazu motiviert, neue Versionen von TCP speziell für solche Hochgeschwindigkeitsumgebungen zu entwerfen. [Jin 2004; RFC 3649; Kelly 2003] beschreiben Ergebnisse dieser Bemühungen.

3.7.1 Fairness

Stellen Sie sich K TCP-Verbindungen vor, von denen jede über einen anderen Ende-zu-Ende-Pfad läuft, die aber alle eine gemeinsame Engpassleitung mit der Übertragungsgeschwindigkeit R bps passieren. (Mit Engpassleitung meinen wir, dass für jede Verbindung alle anderen Leitungen entlang des Pfades nicht überlastet sind und im Vergleich zur Engpassleitung reichlich freie Übertragungskapazität haben.) Nehmen Sie an, dass jede Leitung eine große Datei überträgt und dass es keinen UDP-Verkehr über die Engpassleitung gibt. Ein Überlastkontrollmechanismus wird als fair bezeichnet, wenn die durchschnittliche Übertragungsgeschwindigkeit jeder Verbindung ungefähr R/K ist; das bedeutet, jede Verbindung bekommt einen gleich großen Anteil der Leitungsbandbreite.

Ist der TCP-AIMD-Algorithmus fair, insbesondere wenn verschiedene TCP-Verbindungen zu unterschiedlichen Zeitpunkten beginnen und daher verschiedene Fenstergrößen haben können? [Chiu 1989] liefert eine elegante und intuitive Erklärung, warum die TCP-Überlastkontrolle gegen einen gleichen Anteil der Bandbreite einer Engpassleitung für alle konkurrierenden TCP-Verbindungen konvergiert.

Berücksichtigen wir den einfachen Fall zweier TCP-Verbindungen, die sich eine einfache Leitung mit Übertragungsrate R teilen, wie in ▶Abbildung 3.54 gezeigt. Nehmen Sie an, dass die beiden Verbindungen dieselbe *MSS* und *RTT* haben (d.h., wenn sie dieselbe Congestion-Window-Größe besitzen, dann erzielen sie auch denselben Durchsatz), dass sie eine große Datenmenge senden und dass keine andere TCP-Verbindung und keine UDP-Datagramme diese gemeinsame Leitung passieren. Ignorieren Sie auch die Slow-Start-Phase von TCP und nehmen Sie an, dass die TCP-Verbindungen die ganze Zeit im Congestion-Avoidance-Modus (AIMD) arbeiten.

▶Abbildung 3.55 skizziert den von beiden TCP-Verbindungen erreichten Durchsatz. Wenn TCP die Bandbreite der Leitung gleichmäßig zwischen den beiden Verbindun-

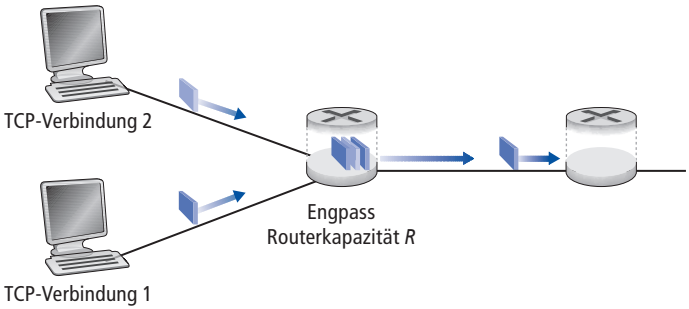


Abbildung 3.54: Zwei TCP-Verbindungen teilen sich eine Engpassleitung

gen aufteilt, dann sollte der realisierte Durchsatz auf dem 45°-Pfeil (gleicher Bandbreitenanteil) liegen, der vom Ursprung ausgeht. Idealerweise sollte die Summe der beiden Durchsätze gleich R sein. (Natürlich ist die Situation, in der beide Verbindungen eine identische Datenrate von 0 bps erhalten, nicht wirklich wünschenswert!) Also sollte das Ziel darin bestehen, die erreichten Durchsätze in Abbildung 3.55 irgendwo in der Nähe des Kreuzungspunktes der Linie gleicher Bandbreitenanteile und der Linie der vollständigen Bandbreitennutzung anzusiedeln.

Nehmen Sie an, dass die TCP-Fenstergrößen zu einem gegebenen Zeitpunkt so sind, dass die Durchsätze der Verbindungen 1 und 2 Punkt A in Abbildung 3.55 entsprechen. Weil der Betrag der von beiden Verbindungen gemeinsam belegten Leitungsbandbreite geringer als R ist, findet kein Verlust statt und beide Verbindungen steigern ihre Fenstergröße um 1 *MSS* pro *RTT* als Resultat des Congestion-Avoidance-Algorithmus von TCP. Dadurch verläuft der gemeinsame Durchsatz der beiden Verbindun-

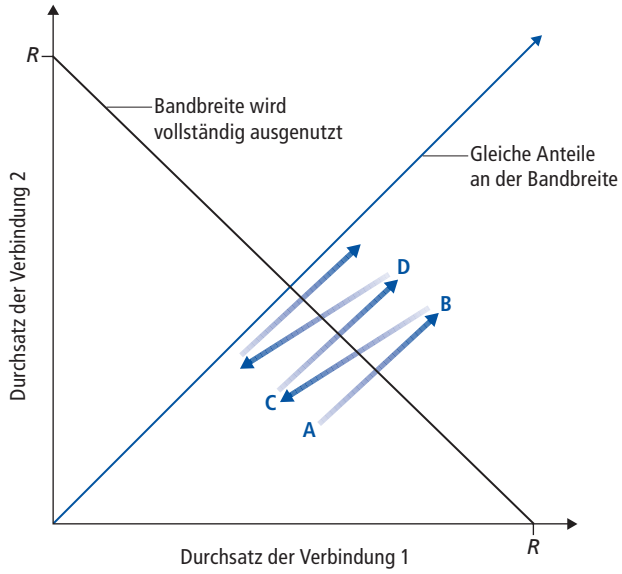


Abbildung 3.55: Durchsatz der TCP-Verbindungen 1 und 2

gen entlang einer 45°-Linie (gleicher Zuwachs für beide Verbindungen), die am Punkt A beginnt. Irgendwann wird die von beiden Verbindungen gemeinsam beanspruchte Bandbreite größer als *R* sein und es treten Paketverluste auf. Nehmen Sie an, dass Verbindungen 1 und 2 einen Paketverlust erfahren, sobald sie Durchsätze haben, die von Punkt B gekennzeichnet sind. Verbindungen 1 und 2 halbieren dann beide ihre Fenstergröße. Die entstehenden Durchsätze erreichen so Punkt C, etwa in der Mitte der Strecke, die Punkt B mit dem Ursprung verbindet. Weil die gemeinsame Bandbreitenverwendung im Punkt C geringer als *R* ist, steigern die beiden Verbindungen ihre Durchsätze wieder entlang einer 45 Grad-Linie, die bei C beginnt. Irgendwann wird es wieder Paketverluste geben, zum Beispiel an Punkt D, und die beiden Verbindungen halbieren ihre Fenstergrößen wieder usw.

Sie sollten sich davon überzeugen, dass die von den zwei Verbindungen genutzte Bandbreite um die Linie gleicher Bandbreitenanteile schwankt. Sie sollten sich auch davon überzeugen, dass die beiden Verbindungen zu diesem Verhalten konvergieren, ohne Rücksicht darauf, wo sie sich in der zweidimensionalen Darstellung ursprünglich befinden! Obwohl eine Reihe von idealisierten Annahmen diesem Szenario zu Grunde liegt, vermittelt es immer noch ein intuitives Gefühl dafür, warum TCP dazu führt, dass mehrere Verbindungen gleiche Teile der Bandbreite erhalten.

In unserem idealisierten Szenario haben wir vorausgesetzt, dass nur TCP-Verbindungen die Engpassleitung durchqueren, dass die Verbindungen denselben *RTT*-Wert haben, und dass nur eine einzelne TCP-Verbindung pro Quelle-Ziel-Paar existiert. In der Praxis ist die Lage normalerweise komplizierter und die Verbindungen können sehr ungleiche Anteile der Engpassbandbreite erhalten. Insbesondere dann, wenn sich mehrere Verbindungen einen gemeinsamen Engpass teilen, sind jene Sitzungen mit kleinerer *RTT* in der Lage, die verfügbare Bandbreite dieser Verbindung schneller aufzufüllen, sobald diese verfügbar wird (d.h., sie vergrößern ihr Congestion Window schneller). Dadurch genießen sie einen höheren Durchsatz als jene Verbindungen mit größerer *RTT* [Lakshman 1997].

Fairness und UDP

Wir haben gerade gesehen, wie die TCP-Überlastkontrolle die Übertragungsrate einer Anwendung mittels des Congestion-Window-Mechanismus reguliert. Viele Multimedia-Anwendungen wie Internettelefonie und Videokonferenzen laufen aus diesem Grund oft nicht über TCP – sie wollen vermeiden, dass ihre Übertragungsrate gedrosselt wird, selbst wenn das Netz sehr überlastet ist. So ziehen es diese Anwendungen vor, UDP zu verwenden, das keine eingebaute Überlastkontrolle hat.

Wenn sie UDP nutzen, pumpen Anwendungen ihre Audio- und Videodaten mit konstanter Rate ins Netz und verlieren gelegentlich Pakete, statt ihre Rate in Zeiten der Überlast auf ein „faires“ Niveau zu reduzieren und keine Pakete zu verlieren. Aus der Perspektive von TCP sind Multimedia-Anwendungen, die über UDP laufen, nicht fair – sie arbeiten weder mit den anderen Verbindungen zusammen noch passen sie ihre Übertragungsraten an. Weil TCP seine Übertragungsrate angesichts wach-

sender Überlast (und der damit einhergehenden Verluste) vermindert, während UDP darauf keine Rücksicht nimmt, können die UDP-Quellen den TCP-Verkehr möglicherweise verdrängen. Daher befasst sich heutzutage ein Forschungszweig mit der Entwicklung von Überlastkontrollmechanismen für das Internet, die den UDP-Verkehr genau daran hindern [Floyd 1999; Floyd 2000; Kohler 2006].

Fairness und parallele TCP-Verbindungen

Aber selbst wenn wir den UDP-Verkehr dazu zwingen könnten, sich fair zu verhalten, wäre das Problem der Fairness immer noch nicht völlig gelöst. Dies liegt daran, dass es keine Möglichkeiten gibt, eine TCP-basierte Anwendung davon abzuhalten, mehrere parallele Verbindungen zu verwenden. Zum Beispiel verwenden Webbrowser oft mehrere parallele TCP-Verbindungen, um mehrere Objekte einer Webseite zu übertragen. (Die verwendete Zahl von Verbindungen ist in den meisten Browsern konfigurierbar.) Wenn eine Anwendung mehrere parallele Verbindungen verwendet, steht ihr ein größerer Anteil der Bandbreite in einem überlasteten Link zur Verfügung. Betrachten Sie als Beispiel eine Leitung der Rate R, über die neun Anwendungen je eine TCP-Verbindung unterhalten. Kommt eine neue Anwendung und damit eine neue TCP-Verbindung hinzu, dann bekommt jede Verbindung ungefähr dieselbe Übertragungsgeschwindigkeit von $R/10$. Verwendet die neu hinzukommende Anwendung aber elf parallele TCP-Verbindungen statt einer einzelnen, dann erhält die neue Anwendung einen unfairen Anteil von mehr als $R/2$. Weil der Webdatenverkehr einen so großen Anteil der Übertragungen im Internet ausmacht, sind solche parallelen Verbindungen nicht ungewöhnlich.

ZUSAMMENFASSUNG

Wir haben dieses Kapitel mit der Betrachtung der Dienste begonnen, die ein Transportschichtprotokoll erbringen kann, um Anwendungen zu vernetzen. Auf der einen Seite kann das Transportschichtprotokoll sehr einfach sein und den Anwendungen nur die allernotwendigsten Dienste anbieten, nämlich eine Multiplexing/Demultiplexing-Funktion für kommunizierende Prozesse. Das UDP-Protokoll des Internets ist ein Beispiel für ein solches schlichtes Transportschichtprotokoll. Auf der anderen Seite kann ein Transportschichtprotokoll den Anwendungen eine Reihe von Garantien geben, darunter die zuverlässige Zustellung von Daten, Verzögerungsgarantien und Bandbreitengarantien. Dennoch werden die Dienste, die ein Transportprotokoll erbringen kann, oft vom Dienstmodell des zugrunde liegenden Netzwerkschichtprotokolls eingeschränkt. Wenn das Netzwerkschichtprotokoll den Segmenten der Transportschicht keine Beschränkung der maximalen Verzögerung und keine Minimalbandbreite garantieren kann, dann kann auch das Transportschichtprotokoll den zwischen Prozessen ausgetauschten Nachrichten keine Verzögerungs- oder Bandbreitengarantien anbieten.

Wir haben in **Abschnitt 3.4** erfahren, dass ein Transportschichtprotokoll zuverlässigen Datentransfer liefern kann, selbst wenn die zugrunde liegende Netzwerkschicht unzuverlässig ist. Wir haben gesehen, dass es im Zusammenhang mit zuverlässigen Datentransfers viele Feinheiten zu beachten gilt, aber dass die Aufgabe sich durch sorgfältiges Kombinieren von Acknowledgments, Timern, Übertragungswiederholungen und Sequenznummern lösen lässt.

Obwohl wir den zuverlässigen Datentransfer in diesem Kapitel behandelt haben, sollten wir im Hinterkopf behalten, dass zuverlässiger Datentransfer generell von Sicherungsschicht-, Netzwerkschicht-,

Transportschicht-, oder Anwendungsschichtprotokollen angeboten werden kann. Jede der oberen vier Schichten des Protokollstapels kann Acknowledgments, Timer, Übertragungswiederholungen und Sequenznummern implementieren und der darüber liegenden Schicht zuverlässigen Datentransfer bieten. In der Tat haben Ingenieure und Informatiker über die Jahre hinweg unabhängig voneinander Sicherungsschicht-, Netzwerkschicht-, Transportschicht- und Anwendungsschichtprotokolle entworfen und implementiert, die zuverlässigen Datentransfer ermöglichen (wobei viele dieser Protokolle mittlerweile wieder still und leise in der Versenkung verschwunden sind).

In **Abschnitt 3.5** betrachteten wir TCP, das verbindungsorientierte und zuverlässige Transportschichtprotokoll des Internets, aus der Nähe. Wir haben erfahren, dass TCP komplex ist und Mechanismen für die Verbindungsverwaltung, die Flusskontrolle, die Schätzung der Rundlaufzeit und den zuverlässigen Datentransfer enthält. In der Tat ist TCP sogar noch viel komplexer als unsere Beschreibung – wir haben absichtlich nicht die große Zahl von Patches, Fixes und Verbesserungen erörtert, die in unterschiedlichen TCP-Versionen implementiert sind. Diese ganze Komplexität wird jedoch vor der Netzanwendung verborgen. Wenn ein Client auf einem Host Daten zuverlässig an einen Server auf einem anderen Host senden will, öffnet er einfach einen TCP-Socket zum Server und pumpt Daten in dieses Socket. Die Client-Server-Applikation muss sich glücklicherweise der TCP-Komplexität nicht bewusst sein.

In **Abschnitt 3.6** haben wir die Überlastkontrolle auf breiter Grundlage eingeführt und in **Abschnitt 3.7** gezeigt, wie TCP sie durchführt. Wir haben erfahren, dass Überlastkontrolle für das Wohl des Netzes unbedingt notwendig ist. Ohne Überlastkontrolle kann ein Netz leicht in eine festgefahrene Lage geraten, in der sehr wenige oder gar keine Daten erfolgreich ihren Empfänger erreichen. In **Abschnitt 3.7** haben wir auch erfahren, dass TCP einen Ende-zu-Ende-Überlastkontrollmechanismus besitzt, der seine Übertragungsrate additiv steigert, wenn er davon ausgehen kann, dass der Pfad der TCP-Verbindung frei von Überlast ist, und die Übertragungsrate multiplikativ vermindert, sobald Verluste auftreten. Dieser Mechanismus ist auch bestrebt, jeder durch eine überlastete Verbindung gehenden TCP-Verbindung einen gleich großen Anteil der Verbindungsbandbreite zur Verfügung zu stellen. Wir haben auch in einiger Detailtiefe den Einfluss des TCP-Verbindungaufbaus und des Slow Start auf die Latenzzeit betrachtet. Wir haben beobachtet, dass in vielen wichtigen Szenarien der Verbindungsaufbau und Slow Start signifikant zur Ende-zu-Ende-Verzögerung beitragen. Während sich die TCP-Überlastkontrolle über die Jahre hinweg entwickelt hat, ist sie auch heute noch ein Bereich intensiver Forschung und sie wird sich wahrscheinlich auch in den kommenden Jahren weiterentwickeln.

Unsere Diskussion von Internettransportprotokollen in diesem Kapitel hat sich auf UDP und TCP konzentriert – die zwei Arbeitspferde der Internet-Transportschicht. Zwei Jahrzehnte an Erfahrung mit diesen Protokollen haben jedoch gezeigt, dass es Situationen gibt, in denen keines von beiden ideal geeignet ist. Wissenschaftler arbeiten daher an der Entwicklung zusätzlicher Transportschichtprotokolle, von denen mehrere mittlerweile der IETF als neue Standards vorgeschlagen wurden.

Das Datagram Congestion Control Protocol (DCCP) [RFC 4340] stellt einen nachrichtenorientierten, UDP-ähnlichen unzuverlässigen Dienst mit geringem Overhead bereit, unterstützt aber eine Form der Überlastkontrolle, die von der Anwendung gewählt werden kann und die mit TCP kompatibel ist. Benötigt eine Anwendung zuverlässigen oder halbzuverlässigen Datentransfer, dann würde dies innerhalb der Anwendung selbst realisiert, vielleicht unter Verwendung der Mechanismen, die wir in **Abschnitt 3.4** studiert haben. DCCP soll in Anwendungen wie Multimedia-Streaming (siehe *Kapitel 7*) eingesetzt werden, die einen Kompromiss zwischen zeitnaher und zuverlässiger Zustellung eingehen können, die aber auch auf Überlast im Netz reagieren sollen.

Das Stream Control Transmission Protocol (SCTP) [RFC 2960; RFC 3286] ist ein zuverlässiges, nachrichtenorientiertes Protokoll, das es ermöglicht, mehrere verschiedene „Ströme" der Anwendungsschicht durch eine einzelne SCTP-Verbindung zu multiplexen (ein als „Multidatenstrom" bekannter Ansatz). Vom Standpunkt der Zuverlässigkeit aus werden die verschiedenen Ströme innerhalb der Verbindung gesondert behandelt, so dass Paketverlust in einem Strom nicht die Zustellung von Daten in anderen Strömen beeinflusst. SCTP ermöglicht auch die Übertragung von Daten über zwei ausgehende Wege, etwa wenn ein Host mit zwei oder mehr Netzen verbunden ist. Es erlaubt optional die Zustellung von Daten auch außerhalb der Reihenfolge und noch einiges mehr. Die Algorithmen der Fluss- und Überlastkontrolle in SCTP sind im Grunde genommen die gleichen wie in TCP.

Das TCP Friendly Rate Control Protocol (TFRC) [RFC 2448] ist eher ein Überlastkontrollmechanismus als ein vollständiges Transportschichtprotokoll. Es spezifiziert eine Vorgehensweise zur Überlastkontrolle, die in anderen Transportprotokollen wie DCCP verwendet werden könnte. (In der Tat ist einer der beiden in DCCP verfügbaren und von den Anwendungen wählbaren Mechanismen TFRC.) Das Ziel von TFRC ist das Ausgleichen der „Sägezahn"-Struktur (Abbildung 3.51) der TCP-Überlastkontrolle, während es langfristig eine Senderate aufrechterhält, die nur wenig von TCP abweicht. Mit seinem glatteren Sendeverhalten als TCP ist TFRC für Multimedia-Anwendungen wie IP-Telefonie oder Multimedia-Streaming geeignet, bei denen eine gleichmäßige Rate wichtig ist. TFRC ist ein „gleichungsbasiertes" Protokoll, das die gemessene Paketverlustrate als Eingabewert für eine Gleichung verwendet [Padhye 2000]. Diese schätzt ab, wie hoch der TCP-Durchsatz wäre, wenn eine TCP-Sitzung dieselbe Verlustrate erleiden würde. Diese Rate wird dann als Zielsenderate für TFRC verwendet.

Nur die Zukunft wird erweisen, ob sich DCCP, SCTP oder TFRC verbreiten werden. Während diese Protokolle eindeutig gegenüber TCP und UDP verbesserte Eigenschaften und Möglichkeiten bieten, haben sich TCP und UDP über die Jahre hinweg als „gut genug" erwiesen. Ob „besser" über „gut-genug" gewinnen wird, hängt von einem komplexen Zusammenspiel technischer, sozialer und wirtschaftlicher Faktoren ab.

In *Kapitel 1* haben wir gesagt, dass ein Computernetzwerk in den „Rand des Netzwerkes" und das „Innere des Netzwerkes" aufgeteilt werden kann. Der Rand des Netzwerkes umfasst alles, was in den Endsystemen geschieht. Nachdem wir jetzt die Anwendungsschicht und die Transportschicht kennen, ist unsere Diskussion des Netzwerkrandes beendet. Es wird Zeit, das Innere zu erkunden! Diese Reise beginnt im nächsten Kapitel, in dem wir die Netzwerkschicht studieren, und setzt sich fort in *Kapitel 5*, in dem wir die Sicherungsschicht betrachten.

Lösungshinweise

Aufgaben

Verständnisfragen

ABSCHNITTE 3.1–3.3

R1. Nehmen Sie an, dass die Netzwerkschicht den folgenden Dienst erbringt: Die Netzwerkschicht im Quellhost erhält ein Segment der Maximalgröße 1.200 Byte und eine Zielhostadresse aus der Transportschicht. Die Netzwerkschicht garantiert dann, das Segment an die Transportschicht des Zielhosts zu liefern. Gehen Sie davon aus, dass viele Anwendungsprozesse auf dem Zielhost laufen können.

a. Entwerfen Sie das einfachste mögliche Transportschichtprotokoll, das Anwendungsdaten zum gewünschten Prozess auf dem Zielhost bringt. Nehmen Sie an, dass das Betriebssystem des Zielhosts jedem laufenden Anwendungsprozess eine Portnummer von 4 Byte Länge zugeteilt hat.

b. Modifizieren Sie dieses Protokoll so, dass es dem Zielprozess eine „Antwortadresse“ liefert.

c. Muss in Ihren Protokollen die Transportschicht irgendetwas im Inneren des Computernetzwerkes tun?

R2. Stellen Sie sich einen Planeten vor, auf dem jeder zu einer sechsköpfigen Familie gehört. Jede Familie lebt in ihrem eigenen Haus, jedes Haus hat eine eindeutige Adresse und jede Person in einem gegebenen Haus hat einen eindeutigen Namen. Nehmen Sie an, dass dieser Planet einen Postdienst hat, der Briefe vom Quellhaus zum Zielhaus transportiert. Der Postdienst erfordert es, dass (i) der Brief in einem Umschlag steckt und (ii) die Adresse des Zielhauses (und nichts anderes) deutlich auf dem Umschlag steht. Nehmen Sie an, dass ein Mitglied jeder Familie für das Sammeln und Verteilen der Briefe für die anderen Familienmitglieder verantwortlich ist. Die Briefe liefern nicht unbedingt einen Hinweis auf die Empfänger der Briefe.

a. Lassen Sie sich von der Lösung für Problem R1 oben inspirieren und beschreiben Sie ein Protokoll, das die Postbeauftragten in den Familien nutzen können, um Briefe an das richtige Familienmitglied zuzustellen.

b. Muss in Ihrem Protokoll der Postdienst jemals den Umschlag öffnen und den Brief selbst anschauen, um seinen Dienst zu erbringen?

R3. Betrachten Sie eine TCP-Verbindung zwischen Host A und Host B. Nehmen Sie an, dass die TCP-Segmente, die von Host A an Host B übertragen werden, die Quellportnummer *x* und die Zielportnummer *y* haben. Was sind die Quell- und Zielportnummern der Segmente, die von Host B zu Host A transportiert werden?

R4. Erklären Sie, aus welchen Gründen sich ein Anwendungsentwickler dafür entscheiden könnte, eine Anwendung über UDP statt TCP auszuführen.

R5. Warum wird Sprach- und Videoverkehr im heutigen Internet oft über TCP statt UDP transportiert? (*Hinweis*: Die Antwort, die wir suchen, hat nichts mit der Überlastkontrolle von TCP zu tun.)

R6. Ist es möglich, dass eine Anwendung sogar dann in den Genuss zuverlässigen Datentransfers kommen kann, wenn sie über UDP läuft? Wenn ja, wie?

R7. Nehmen Sie an, dass ein Prozess in Host C einen UDP-Socket mit Portnummer 6789 hat. Nehmen Sie an, dass sowohl Host A als auch Host B ein UDP-Segment mit Zielportnummer 6789 an Host C senden. Werden beide Segmente an denselben Socket auf Host C gerichtet? Wenn ja, wie kann der Prozess auf Host C wissen, dass diese beiden Segmente von zwei verschiedenen Hosts stammten?

R8. Nehmen Sie an, dass auf Host C ein Webserver auf Port 80 läuft. Nehmen Sie auch an, dass dieser Webserver persistente Verbindungen verwendet und gegenwärtig Anfragen von zwei verschiedenen Hosts A und B erhält. Kommen alle Anfragen über denselben Socket auf Host C an? Wenn sie verschiedene Sockets passieren, haben dann beide Sockets Port 80? Diskutieren und begründen Sie Ihre Antworten.

ABSCHNITT 3.4

R9. Warum mussten wir in unseren `rdt`-Protokollen Sequenznummern einführen?

R10. Warum mussten wir in unseren `rdt`-Protokollen Timer einführen?

R11. Nehmen Sie an, dass die Rundlaufzeit zwischen Absender und Empfänger konstant und dem Absender bekannt ist. Vorausgesetzt, Pakete können verloren gehen – wäre immer noch ein Timer im Protokoll `rdt 3.0` notwendig? Erläutern Sie Ihre Antwort.

R12. Verwenden Sie das Go-Back-N-Applet auf unserer Buch-Website.

a. Lassen Sie die Quelle fünf Pakete senden und halten Sie die Animation an, bevor irgendeines der fünf Pakete den Zielort erreicht. Dann löschen Sie das erste Paket und lassen Sie die Animation weiterlaufen. Beschreiben Sie, was geschieht.

b. Wiederholen Sie den Versuch, lassen Sie aber jetzt das erste Paket den Zielort erreichen und löschen Sie die erste Bestätigung. Beschreiben Sie wieder, was geschieht.

c. Versuchen Sie schließlich, sechs Pakete zu senden. Was geschieht?

R13. Wiederholen Sie R12, nun aber mit dem Java-Applet zu Selective Repeat. Wie unterscheiden sich Selective Repeat und Go-Back-N?

ABSCHNITT 3.5

R14. Richtig oder falsch?

a. Host A schickt Host B eine große Datei über eine TCP-Verbindung. Nehmen Sie an, dass Host B keine Daten hat, die er an Host A schicken möchte. Host B kann deshalb keine Bestätigungen an Host A schicken,

weil Host B keine Datenpakete sendet, bei denen er die Acknowledgments per Piggybacking mitschicken kann.

b. Die Größe des TCP RcvWindow ändert sich während der gesamten Dauer der Verbindung nie.

c. Nehmen Sie an, dass Host A eine große Datei über eine TCP-Verbindung an Host B sendet. Die Anzahl von unbestätigten Bytes, die A schickt, kann die Größe des Eingangspuffers nicht übersteigen.

d. Nehmen Sie an, dass Host A eine große Datei über eine TCP-Verbindung an Host B schickt. Beträgt die Sequenznummer für ein Segment dieser Verbindung m, dann muss die Sequenznummer für das anschließende Segment auf jeden Fall $m + 1$ sein.

e. Das TCP-Segment hat in seinem Header ein Feld für RcvWindow.

f. Nehmen Sie an, dass die letzte Sample*RTT* in einer TCP-Verbindung gleich 1 Sekunde ist. Der aktuelle Wert von TimeoutInterval für die Verbindung muss notwendigerweise ebenfalls gleich 1 Sekunde sein.

g. Nehmen Sie an, dass Host A ein Segment mit Sequenznummer 38 und 4 Byte Daten über eine TCP-Verbindung an Host B sendet. Im gleichen Segment lautet die Acknowledgment-Nummer auf jeden Fall 42.

R15. Nehmen Sie an, dass Host A direkt hintereinander zwei TCP-Segmente an Host B über eine TCP-Verbindung schickt. Das erste Segment hat Sequenznummer 90, das zweite hat Sequenznummer 110.

a. Wie viele Daten sind im ersten Segment?

b. Nehmen Sie an, dass das erste Segment verloren geht, aber das zweite bei B ankommt. Wie lautet die Acknowledgment-Nummer in der Bestätigung, die Host B an Host A sendet?

R16. Betrachten Sie das in Abschnitt 3.5 erörterte Telnet-Beispiel. Einige Sekunden nachdem der Benutzer den Buchstaben „C“ eingegeben hat, tippt er den Buchstaben „R“ ein. Wie viele Segmente werden nach dem Drücken der Taste „R“ versandt und was wird in die Felder für Sequenznummer und Acknowledgments der Segmente eingetragen?

ABSCHNITT 3.7

R17. Nehmen Sie an, dass zwei TCP-Verbindungen an irgendeiner Engpassleitung mit Rate R bps anliegen. Jede der beiden Verbindungen muss eine riesige Datei in derselben Richtung über die Engpassleitung versenden. Die Übertragungen der Dateien starten zur gleichen Zeit. Welche Übertragungsgeschwindigkeit würde TCP jeder der Verbindungen geben?

R18. Richtig oder falsch? Betrachten Sie die Überlastkontrolle in TCP. Wenn der Timer der Quelle ausläuft, wird der Wert für Threshold halbiert.

Übungsaufgaben

P1. Nehmen Sie an, dass Client A eine Telnet-Sitzung mit Server S initiiert. Etwa zur selben Zeit initiiert auch Client B eine Telnet-Sitzung mit Server S. Legen Sie mögliche Quell- und Zielportnummern fest für

a. die von A an S gesandten Segmente.

b. die von B an S gesandten Segmente.

c. die von S an A gesandten Segmente.

d. die von S an B gesandten Segmente.

e. Wenn A und B verschiedene Hosts sind, ist es möglich, dass die Quellportnummer in den Segmenten von A zu S dieselbe ist wie in denen von B zu S?

f. Was ist, wenn es sich bei A und B um denselben Host handelt?

P2. Betrachten Sie ▶Abbildung 3.5. Wie lauten die Quell- und Zielportnummern in den Segmenten, die vom Server zurück zu den Prozessen der Clients fließen? Welches sind die IP-Adressen in den Netzwerkschichtdatagrammen, die die Transportschichtsegmente enthalten?

P3. UDP und TCP verwenden das 1er-Komplement für ihre Prüfsummen. Nehmen Sie an, dass Sie die folgenden drei 8 Bit-Zahlen haben: 01010101, 01110000, 01001100. Was ist das 1er-Komplement der Summe dieser 8 Bit-Zahlen? (Berücksichtigen Sie, dass Sie in dieser Aufgabe 8 Bit-Summen verwenden sollen, obwohl UDP und TCP 16 Bit-Worte zur Berechnung der Prüfsumme benutzen.) Zeigen Sie den Rechenweg. Warum benutzt UDP das 1er-Komplement der Summe, statt sie direkt zu verwenden? Mit dem Schema des 1er-Komplements, wie kann der Empfänger Fehler feststellen? Ist es möglich, dass ein 1 Bit-Fehler unentdeckt bleibt? Was wäre mit einem 2 Bit-Fehler?

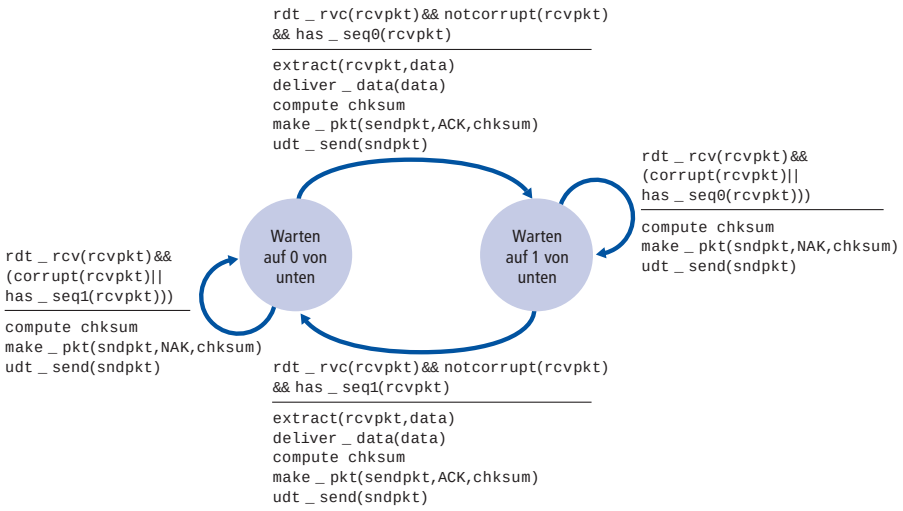
P4. a. Nehmen Sie an, dass Sie die folgenden zwei Bytes haben: 00110100 und 01101001. Was ist das 1er-Komplement dieser beiden Bytes?

b. Nehmen Sie an, dass Sie die folgenden zwei Bytes haben: 11110101 und 00101001. Was ist das 1er-Komplement diesen beiden Bytes?

c. Verändern Sie in jedem Byte aus Aufgabe a ein Bit so, dass sich das 1er-Komplement nicht ändert.

P5. Nehmen Sie an, dass der UDP-Empfänger die Prüfsumme für das erhaltene UDP-Segment berechnet und feststellt, dass es mit dem Wert im Prüfsummenfeld übereinstimmt. Kann der Empfänger absolut sicher sein, dass keine Bitfehler aufgetreten sind? Begründen Sie Ihre Antwort.

P6. Denken Sie an unsere Motivation für die Korrektur des `rdt2.1`-Protokolls. Zeigen Sie, dass der Empfänger in der folgenden Abbildung, wenn er mit dem Sender in ▶Abbildung 3.11 zusammenarbeitet, Sender und Empfänger in einen Zustand gegenseitiger Blockade führen kann, in dem jeder auf ein Ereignis wartet, das nie stattfindet.



P7. Im Protokoll `rdt3.0` haben die ACK-Pakete, die vom Empfänger zum Absender fließen, keine Sequenznummern (obwohl sie ein ACK-Feld haben, das die Sequenznummer des Paketes enthält, das sie bestätigen). Warum benötigen unsere ACK-Pakete keine Sequenznummern?

P8. Zeichnen Sie die FSM für die Empfängerseite des Protokolls `rdt3.0`.

P9. Geben Sie Schritt für Schritt die Operationen des Protokolls `rdt3.0` an, wenn Datenpakete und Acknowledgment-Pakete zerstört werden. Verwenden Sie dazu ein Zeit-Ablauf-Diagramm wie in ▶Abbildung 3.16.

P10. Betrachten Sie einen Kanal, der Pakete verlieren kann, aber eine bekannte Maximalverzögerung hat. Modifizieren Sie Protokoll `rdt2.1`, um Sender-Timeouts und Übertragungswiederholungen hinzuzufügen. Argumentieren Sie qualitativ, warum Ihr Protokoll über diesen Kanal richtig kommunizieren kann.

P11. Die Absenderseite von `rdt3.0` ignoriert einfach (das heißt, sie ergreift keinerlei Maßnahmen) alle erhaltenen Pakete, die entweder fehlerhaft sind oder den falschen Wert im Feld acknum eines Acknowledgment-Paketes haben. Nehmen Sie an, `rdt3.0` würde in dieser Situation einfach das aktuelle Datenpaket nochmals übertragen. Würde das Protokoll immer noch funktionieren? (*Hinweis*: Denken Sie daran, was geschehen würde, wenn es nur Bitfehler gäbe; es gäbe keine Paketverluste, aber vorzeitige Timeouts könnten auftreten. Stellen Sie sich vor, wie oft das n-te Paket gesandt wird, wenn n gegen unendlich geht.)

P12. Betrachten Sie das `rdt3.0`-Protokoll. Zeichnen Sie ein Diagramm, das zeigt, dass das Alternierende-Bit-Protokoll nicht richtig funktionieren kann (arbeiten Sie klar heraus, was genau nicht funktioniert), wenn die Netzwerkver-

bindung zwischen Absender und Empfänger Nachrichten umordnen kann. (Das heißt, dass zwei Nachrichten, die sich auf dem Medium zwischen dem Absender und Empfänger fortpflanzen, in ihrer Reihenfolge vertauscht werden können.) Ihr Diagramm sollte den Absender auf der linken Seite und den Empfänger auf der rechten Seite zeigen, die Zeitachse sollte nach unten weisen und Sie sollten den Austausch von Daten (D) und Bestätigungsnachrichten (A) einzeichnen. Geben Sie die mit jedem Daten- oder Bestätigungssegment verbundene Sequenznummer an.

P13. Stellen Sie sich ein zuverlässiges Datentransferprotokoll vor, das nur negative Bestätigungen verwendet. Nehmen Sie an, dass der Absender nur selten Daten sendet. Wäre ein reines NAK-Protokoll einem Protokoll, das ACKs verwendet, vorzuziehen? Warum? Nehmen Sie jetzt an, dass der Absender viele Daten zu senden hat und die Ende-zu-Ende-Verbindung nur wenige Verluste erfährt. Wäre in diesem zweiten Fall ein NAK-Protokoll einem Protokoll, das ACKs verwendet, vorzuziehen? Warum oder warum nicht?

P14. Betrachten Sie das Beispiel in ▶Abbildung 3.17. Wie groß müsste die Fenstergröße sein, damit die Kanalauslastung höher als 90 Prozent ist?

P15. Im generischen SR-Protokoll, das wir in Abschnitt 3.4.4 studiert haben, sendet der Absender eine Nachricht, sobald sie verfügbar ist (wenn sie im Fenster ist), ohne auf eine Bestätigung zu warten. Nehmen Sie an, dass wir ein SR-Protokoll entwickeln wollen, das zwei Nachrichten auf einmal sendet. Das heißt, der Absender sendet ein Nachrichtenpaar und sendet das nächste Nachrichtenpaar nur dann, wenn er weiß, dass beide Nachrichten des ersten Paares richtig empfangen worden sind.

Nehmen Sie an, dass der Kanal Nachrichten verlieren kann, aber keine Nachrichten korrumpiert oder umordnet. Gestalten Sie ein Fehlerkontrollprotokoll für die zuverlässige Übertragung von Nachrichten in einer Richtung. Geben Sie eine FSM-Beschreibung des Absenders und Empfängers an. Beschreiben Sie das Format der zwischen Absender und Empfänger übertragenen Pakete. Wenn Sie irgendwelche Prozeduraufrufe verwenden, die sich von jenen in Abschnitt 3.4 unterscheiden (zum Beispiel `udt_send`, `start_timer`, `rdt_rcv`), beschreiben Sie deren Aktionen eindeutig. Geben Sie ein Beispiel an (in Form eines Zeit-Ablauf-Diagramms analog zu ▶Abbildung 3.16), das zeigt, wie Ihr Protokoll den Verlust eines Paketes korrigiert.

P16. Betrachten Sie ein Szenario, in dem Host A Pakete simultan an die Hosts B und C senden will. A ist mit B und C durch einen gemeinsamen Kanal verbunden – ein von A gesandtes Paket wird vom Kanal gleichzeitig sowohl zu B als auch zu C transportiert. Nehmen Sie an, dass der Kanal, der A, B und C verbindet, unabhängig Pakete verlieren und korrumpieren kann (z.B. könnte ein von A gesandtes Paket richtig von B empfangen werden, nicht aber von C). Gestalten Sie ein Stop-and-Wait-ähnliches Fehlerkontrollprotokoll für die zuverlässige Übertragung von Paketen von A zu B und C, so dass A keine

neuen Daten von der darüberliegenden Schicht entgegennimmt, bis er weiß, dass sowohl B als auch C das gegenwärtige Paket korrekt erhalten haben. Geben Sie FSM-Beschreibungen von A und C an. (*Hinweis:* Die FSM für B sollte im Grunde genommen die gleiche sein wie für C.) Geben Sie zudem eine Beschreibung des verwendeten Paketformates an.

P17. Betrachten Sie ein Szenario, in dem Host A und Host B Nachrichten an Host C senden wollen. Die Hosts A und C sind durch einen Kanal verbunden, der Nachrichten verlieren und korrumpieren (aber nicht umordnen) kann. Die Hosts B und C sind durch einen anderen Kanal (unabhängig von dem, der A und C verbindet) mit denselben Eigenschaften verbunden. Die Transportschicht bei Host C sollte beim Abliefern von Nachrichten an die darüberliegende Schicht zwischen den Daten von A und B abwechseln (das heißt, sie sollte zuerst die Daten eines Paketes von A zustellen, dann die Daten eines Paketes von B usw.). Gestalten Sie ein Stop-and-Wait-ähnliches Fehlerkontrollprotokoll für das zuverlässige Übertragen von Paketen von A und B zu C mit abwechselnder Zustellung an C, wie oben beschrieben. Geben Sie FSM-Beschreibungen von A und C an. (*Hinweis:* Die FSM für B sollte im Grunde genommen die gleiche sein wie für A.) Geben Sie zudem eine Beschreibung des verwendeten Paketformates an.

P18. Betrachten Sie das GBN-Protokoll mit einer Größe des Absenderfensters von 3 und einem Sequenznummernbereich von 1024. Gehen Sie davon aus, dass zum Zeitpunkt t das nächste vom Empfänger erwartete Paket die Sequenznummer k hat. Nehmen Sie an, dass das Medium keine Nachrichten umordnet. Beantworten Sie die folgenden Fragen:

a. Was sind die möglichen Sequenznummernbereiche im Fenster des Absenders zum Zeitpunkt t? Begründen Sie Ihre Antwort.

b. Welches sind alle möglichen Werte des ACK-Feldes in allen möglichen Nachrichten, die sich zum Zeitpunkt t gerade unterwegs zurück zum Absender befinden? Begründen Sie Ihre Antwort.

P19. Nehmen Sie an, dass wir zwei Systeme A und B haben. B hat einen Vorrat an Datennachrichten, die entsprechend folgender Konventionen an A gesandt werden. Wenn A eine Anfrage von der darüberliegenden Schicht erhält, die nächste Datennachricht (D) von B zu bekommen, muss A an B eine Request-Nachricht (R) auf dem A-zu-B-Kanal schicken. Nur wenn B eine R-Nachricht erhält, darf es A eine Datennachricht (D) auf dem B-zu-A-Kanal zurückschicken. A soll der darüberliegenden Schicht genau eine Kopie jeder D-Nachricht liefern. R-Nachrichten können auf dem A-zu-B-Kanal verloren gehen (aber nicht korrumpiert werden); D-Nachrichten werden, sobald gesandt, immer richtig geliefert. Die Verzögerung auf beiden Kanälen ist unbekannt und variabel.

Gestalten Sie ein Protokoll (geben Sie eine FSM-Beschreibung an), das geeignete Mechanismen enthält, um den verlustgefährdeten A-zu-B-Kanal abzusichern, und, wie oben diskutiert, das Übertragen von Nachrichten an

die darüberliegende Schicht des Systems A durchführt. Verwenden Sie nur jene Mechanismen, die absolut notwendig sind.

P20. Betrachten Sie die GBN- und SR-Protokolle. Nehmen Sie an, dass der Bereich der Sequenznummern die Größe k hat. Was ist das größte zulässige Absenderfenster, mit dem das Auftreten von Problemen wie dem in ▶Abbildung 3.27 für jedes dieser Protokolle vermieden wird?

P21. Beantworten Sie, ob die folgenden Fragen wahr oder falsch sind, und rechtfertigen Sie kurz Ihre Antwort:

a. Mit dem SR-Protokoll ist es möglich, dass der Absender ein ACK für ein Paket erhält, das außerhalb seines gegenwärtigen Fensters liegt.

b. Mit GBN ist es möglich, dass der Absender ein ACK für ein Paket erhält, das außerhalb seines gegenwärtigen Fensters liegt.

c. Das Alternierende-Bit-Protokoll ist das Gleiche wie das SR-Protokoll mit einer Größe für Absender- und Empfängerfenster von eins.

d. Das Alternierende-Bit-Protokoll ist das Gleiche wie das GBN-Protokoll mit einer Größe für Absender- und Empfängerfenster von eins.

P22. Wir haben gesagt, dass eine Anwendung UDP als Transportprotokoll wählen kann, weil UDP den Applikationen eine bessere Kontrolle (als TCP) ermöglicht, welche Daten wann in einem Segment gesendet werden.

a. Warum hat eine Anwendung mehr Kontrolle darüber, welche Daten in einem Segment übertragen werden?

b. Warum hat eine Anwendung mehr Kontrolle darüber, wann das Segment gesendet wird?

P23. Sie versuchen, eine riesige Datei von L Byte von Host A zu Host B zu übertragen. Gehen Sie von einer *MSS* von 1.460 Byte aus.

a. Bei welchem Maximalwert von L sind die TCP-Sequenznummern nicht erschöpft? Erinnern Sie sich daran, dass das TCP-Sequenznummernfeld eine Größe von 4 Byte hat.

b. Bestimmen Sie für den Wert von L, den Sie eben berechnet haben, wie lange es dauert, die Datei zu senden. Nehmen Sie an, dass an jedes Segment insgesamt 66 Byte für Transport-, Netzwerk- und Sicherungsschicht-Header angefügt werden, bevor die entstehenden Pakete über eine 10 Mbps-Verbindung übertragen werden. Ignorieren Sie Flusskontrolle und Überlastkontrolle, so dass A die Segmente kontinuierlich und direkt hintereinander absenden kann.

P24. Host A und B kommunizieren über eine TCP-Verbindung und Host B hat bereits von A alle Bytes bis zum Byte 248 erhalten. Nehmen Sie an, dass Host A dann zwei Segmente direkt hintereinander an B sendet. Diese Segmente enthalten 40 bzw. 60 Byte an Daten. Das erste Segment trägt die Sequenznummer 249, die Quellportnummer 503 und die Zielportnummer 80. Host B sendet jedes Mal, wenn er ein Segment von Host A erhält, eine Bestätigung.

a. Wie lauten die Sequenznummer, die Quellportnummer und die Nummer des Zielports im zweiten von Host A an B geschickten Segment?

b. Wenn das erste Segment vor dem zweiten Segment ankommt, wie lauten die Acknowledgment-Nummer, die Quellportnummer und die Zielportnummer in der Bestätigung des ersten ankommenden Segmentes?

c. Wenn das zweite Segment vor dem ersten Segment eintrifft, wie lautet dann die Acknowledgment-Nummer bei der Bestätigung des ersten eingetroffenen Segmentes?

d. Nehmen Sie an, dass die zwei von A geschickten Segmente in der richtigen Reihenfolge bei B ankommen, die erste Bestätigung verloren geht und die zweite Bestätigung nach dem ersten Timeout-Intervall ankommt. Zeichnen Sie ein Zeit-Ablauf-Diagramm (analog zu ▶Abbildung 3.16), welches diese Segmente und alle anderen Segmente und Bestätigungen zeigt, die übertragen werden. (Nehmen Sie an, dass es keinen zusätzlichen Paketverlust gibt.) Nennen Sie die Sequenznummer und die Anzahl der Datenbytes für jedes Segment in Ihrer Abbildung. Nennen Sie die Acknowledgment-Nummer für jede Bestätigung.

P25. Host A und B seien direkt mit einer 200 Mbps-Leitung verbunden. Es gibt eine TCP-Verbindung zwischen den beiden Hosts und Host A sendet über diese Verbindung eine riesige Datei an Host B. Host A kann Anwendungsdaten mit 100 Mbps über die Verbindung senden, aber Host B seinen TCP-Eingangspuffer maximal mit einem Rate von 50 Mbps auslesen. Beschreiben Sie die Wirkung der TCP-Flusskontrolle.

P26. SYN-Cookies wurden in Abschnitt 3.5.6 erörtert.

a. Warum ist es notwendig, dass der Server eine spezielle Anfangssequenznummer beim SYNACK verwendet?

b. Nehmen Sie an, dass ein Angreifer weiß, dass ein Zielhost SYN-Cookies verwendet. Kann der Angreifer halb offene oder vollständig offene Verbindungen dadurch schaffen, dass er einfach ein ACK-Paket an das Ziel sendet? Warum oder warum nicht?

P27. Betrachten Sie das TCP-Verfahren für das Abschätzen der *RTT*. Nehmen Sie an, dass $\alpha = 0{,}1$ ist. Sei $SampleRTT_1$ die jüngste *SampleRTT*, $SampleRTT_2$ die vorletzte *SampleRTT* usw.

a. Nehmen Sie für eine gegebene TCP-Verbindung an, dass vier Acknowledgments mit zugehörigen Sample-RTTs, nämlich $SampleRTT_4$, $SampleRTT_3$, $SampleRTT_2$ und $SampleRTT_1$, eingetroffen sind. Geben Sie *EstimatedRTT* in Abhängigkeit von den vier Sample-RTTs an.

b. Verallgemeinern Sie Ihre Formel für n Sample-RTTs.

c. Lassen Sie in der Formel in Teil (b) n gegen unendlich gehen. Kommentieren Sie, warum dieses Durchschnittsberechnungsverfahren als exponentieller gleitender Durchschnitt bezeichnet wird.

P28. In Abschnitt 3.5.3 haben wir TCPs Abschätzung der RTT erörtert. Warum lässt TCP die *SampleRTT* für wiederholt übertragene Segmente nicht in die Berechnung miteinfließen?

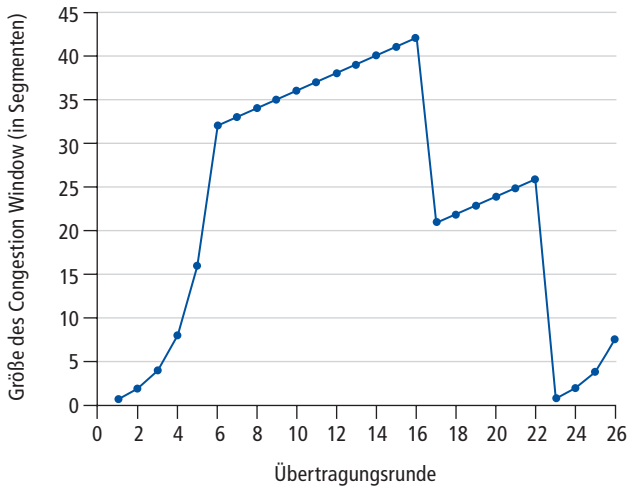
P29. Welche Beziehung besteht zwischen der Variablen *SendBase* in Abschnitt 3.5.4 und der Variablen *LastByteRcvd* in Abschnitt 3.5.5?

P30. Welche Beziehung besteht zwischen der Variablen *LastByteRcvd* in Abschnitt 3.5.5 und der Variablen *y* in Abschnitt 3.5.4?

P31. In Abschnitt 3.5.4 haben wir gesehen, dass TCP wartet, bis es drei doppelte ACKs erhalten hat, bevor ein Fast Retransmit durchgeführt wird. Warum, denken Sie, haben sich die TCP-Designer gegen einen Fast Retransmit schon nach dem ersten doppelten ACK für ein Segment entschieden?

P32. Betrachten Sie ▶Abbildung 3.46 (b). Wenn λ'_{in} über $R/2$ wächst, kann dann λ_{out} über $R/3$ ansteigen? Begründen Sie Ihre Antwort. Betrachten Sie nun ▶Abbildung 3.46 (c). Wenn λ'_{in} über $R/2$ steigt, kann dann, unter der Annahme, dass ein Paket im Durchschnitt zweimal vom Router an den Empfänger weitergeleitet wird, λ_{out} über $R/4$ anwachsen? Begründen Sie Ihre Antwort.

P33. Betrachten Sie folgenden Verlauf der TCP-Fenstergröße über die Zeit.



Nehmen Sie an, dass TCP Reno das Protokoll ist, welches das oben gezeigte Verhalten aufweist, und beantworten Sie die folgenden Fragen. In allen Fällen sollten Sie eine kurze Diskussion liefern, die Ihre Antwort begründet.

a. Identifizieren Sie die Zeitintervalle, in denen TCPs Slow Start aktiv ist.

b. Identifizieren Sie die Zeitintervalle, in denen TCP im Zustand Congestion Avoidance ist.

c. Wird nach der 16. Übertragungsrunde ein Segmentverlust anhand dreier doppelter ACKs oder eines Timeouts erkannt?

d. Wird nach der 22. Übertragungsrunde ein Segmentverlust anhand dreier doppelter ACKs oder eines Timeouts erkannt?

e. Wie lautet der Anfangswert von *Threshold* in der ersten Übertragungsrunde?

f. Wie lautet der Wert von *Threshold* in der 18. Übertragungsrunde?

g. Wie lautet der Wert von *Threshold* in der 24. Übertragungsrunde?

h. In welcher Übertragungsrunde wird das 70. Segment gesandt?

i. Nehmen Sie an, ein Paketverlust wird nach der 26. Runde durch das Eintreffen dreier doppelter ACK erkannt. Welche Werte haben die Größe des Congestion Window und *Threshold* danach?

P34. Beziehen Sie sich auf ▶Abbildung 3.55, welche die Konvergenz des AIMD-Algorithmus von TCP erläutert. Nehmen Sie an, dass TCP statt Multiplicative Decrease die Fenstergröße um einen konstanten Betrag verringert. Würde der entstehende AIAD-Algorithmus ebenfalls zu einer fairen Bandbreitenaufteilung konvergieren? Begründen Sie Ihre Antwort mit einem Diagramm analog zu ▶Abbildung 3.55.

P35. In Abschnitt 3.5.4 haben wir das Verdoppeln des Timeout-Intervalls nach einem Timeout-Ereignis erörtert. Dieser Mechanismus ist eine Form der Überlastkontrolle. Warum braucht TCP zusätzlich zu diesem sich verdoppelnden Timeout einen fensterbasierten Überlastkontrollmechanismus (wie in Abschnitt 3.7 untersucht)?

P36. Host A sendet eine riesige Datei an Host B über eine TCP-Verbindung. Auf dieser Verbindung gibt es nie Paketverlust und die Timer laufen nie aus. Bezeichnen Sie die Übertragungsrate des Links, der Host A mit dem Internet verbindet, mit R bps. Nehmen Sie an, dass der Prozess auf Host A in der Lage ist, Daten an sein TCP-Socket mit einer Rate von S bps zu übergeben, wobei $S = 10\ R$. Nehmen Sie weiter an, dass der TCP-Empfangspuffer groß genug ist, die ganze Datei zu halten, während der Sendepuffer nur ein Prozent der Datei speichern kann. Was würde den Prozess in Host A daran hindern, stetig Daten an sein TCP-Socket mit Rate S bps weiterzureichen? Die TCP-Flusskontrolle? Die TCP-Überlastkontrolle? Oder etwas anderes? Führen Sie die Berechnung aus.

P37. Stellen Sie sich vor, eine große Datei von einem Host an einen anderen über eine TCP-Verbindung zu senden, bei der keine Paketverluste auftreten.

a. Nehmen Sie an, dass TCP für seine Überlastkontrolle AIMD ohne Slow Start verwendet. Unter der Voraussetzung, dass *CongWin* jedes Mal um 1 *MSS* steigt, wenn ein Schub von ACKs empfangen wird, und bei ungefähr konstanter Rundlaufzeit, wie lange dauert es, bis *CongWin* von 1 *MSS* auf 6 *MSS* gestiegen ist (wenn wir den Fall ohne Verlustereignisse betrachten)?

b. Wie groß ist der durchschnittliche Durchsatz (in Abhängigkeit von *MSS* und *RTT*), für diese Verbindung bis zum Zeitpunkt $t = 5\ RTT$?

P38. Erinnern Sie sich an die makroskopische Beschreibung des TCP-Durchsatzes. Im Zeitintervall, in dem sich die Rate der Verbindung zwischen $W/(2\ RTT)$ und W/RTT bewegt, geht nur ein Paket verloren (ganz am Ende des Zeitabschnittes).

a. Zeigen Sie, dass die Verlustrate (der Bruchteil der verloren gegangenen Pakete) gegeben ist durch:

$$L = \frac{1}{\frac{3}{8}W^2 + \frac{3}{4}W}$$

b. Verwenden Sie das obige Ergebnis, um zu zeigen, dass, wenn eine Verbindung die Verlustrate L besitzt, ihre Durchschnittsrate ungefähr durch

$$\approx \frac{1{,}22\ MSS}{RTT\sqrt{L}}$$

gegeben ist.

P39. In unserer Diskussion der Zukunft von TCP in Abschnitt 3.7 merkten wir an, dass TCP nur eine Segmentverlustwahrscheinlichkeit von $2 \cdot 10^{-10}$ tolerieren kann (entsprechend einem Verlustereignis pro 5000000000 Segmente), um einen Durchsatz von 10 Gbps zu erreichen. Zeigen Sie die Ableitung für diesen Wert für die in Abschnitt 3.7 gegebenen Werte von *RTT* und *MSS*. Wenn TCP eine 100 Gbps-Verbindung auslasten müsste, wie groß wäre dann der tolerierbare Verlust?

P40. In unserer Diskussion der TCP-Überlastkontrolle in Abschnitt 3.7 haben wir implizit angenommen, dass der TCP-Sender immer Daten zu senden hätte. Betrachten Sie jetzt den Fall, dass der TCP-Sender eine große Datenmenge sendet und dann bei t_1 in einen Ruhezustand verfällt (da er kein Folgepaket zu senden hat). TCP bleibt für einen relativ langen Zeitraum untätig und sendet dann Folgepakete ab t_2. Was sind die Vor- und Nachteile, wenn TCP die Werte von *CongWin* und Threshold von t_1 verwendet, sobald es in t_2 Daten zu senden beginnt? Welche Alternative würden Sie empfehlen? Warum?

P41. In dieser Aufgabe untersuchen wir, ob UDP oder TCP ein gewisses Maß an Endpunktauthentifikation liefern.

a. Betrachten Sie einen Server, der eine Anfrage durch ein UDP-Paket erhält und diese Anfrage mit einem UDP-Paket beantwortet (wie das zum Beispiel von einem DNS-Server gemacht wird). Wenn ein Client mit IP-Adresse X seine Adresse in Adresse Y verändert (spooft), wohin sendet der Server seine Antwort?

b. Nehmen Sie an, dass ein Server ein SYN mit IP-Quelladresse Y empfängt und nach dem Antworten mit einem SYNACK ein ACK mit IP-Quelladresse Y mit der richtigen Acknowledgment-Nummer erhält. Unter der Annahme, dass der Server eine zufällige anfängliche Sequenznummer wählt und es keinen Man-in-the-Middle gibt, kann der Server sicher

sein, dass der Client wirklich unter Y zu finden ist (und nicht aufgrund von Spoofing unter irgendeiner anderen Adresse X)?

P42. In dieser Aufgabe betrachten wir die von der Slow Start-Phase von TCP verursachte Verzögerung. Stellen Sie sich einen Client und einen Webserver vor, die direkt durch eine Leitung der Rate R verbunden sind. Nehmen Sie an, dass der Client ein Objekt anfragt, dessen Größe gleich 15 S ist, wobei S die maximale Segmentgröße *(MSS)* ist. Bezeichnen Sie die Rundlaufzeit zwischen Client und Server (die als konstant angenommen wird) als *RTT*. Bestimmen Sie die Zeit, die benötigt wird, um ein Objekt herunterzuladen (einschließlich TCP-Verbindungsaufbau) – ignorieren Sie dabei Protokoll-Header – wenn

a. $4\ S/R > S/R + RTT > 2\ S/R$

b. $S/R + RTT > 4\ S/R$

c. $S/R > RTT$.

Diskussion

D1. Was bedeutet TCP-Verbindungs-Hijacking *(Verbindungsentführung)*? Wie könnte es durchgeführt werden?

D2. In Abschnitt 3.7 haben wir gesehen, dass eine Client-Server-Applikation unfairerweise viele parallele Verbindungen gleichzeitig aufbauen kann. Was kann getan werden, um das Internet wirklich fair zu machen?

D3. Lesen Sie wissenschaftliche Veröffentlichungen, um zu erfahren, was unter TCP-freundlich (TCP friendly) zu verstehen ist. Lesen Sie auch das Interview mit Sally Floyd am Ende dieses Kapitels. Verfassen Sie eine Beschreibung der TCP-Fairness von einer Seite Länge.

D4. Am Ende des Abschnittes 3.7.1 erörterten wir die Tatsache, dass eine Anwendung mehrere TCP-Verbindungen öffnen und einen höheren Durchsatz erhalten kann (oder äquivalent eine geringere Datentransferzeit). Was würde geschehen, wenn alle Anwendungen versuchten, ihre Leistung durch Verwenden von mehreren Verbindungen zu verbessern? Nennen Sie einige der Schwierigkeiten, die entstehen, wenn eine Komponente im Inneren des Netzes bestimmen soll, ob eine Anwendung mehrere TCP-Verbindungen verwendet.

D5. Welche Funktionalität hat nmap über das TCP- und UDP-Port-Scanning hinaus? Sammeln Sie mit Wireshark (oder einem anderen Packet Sniffer) Packettraces des nmap-Nachrichtenaustausches. Verwenden Sie diese Traces, um zu erklären, wie manche der erweiterten Funktionen funktionieren.

D6. Lesen Sie wissenschaftliche Literatur zu SCTP [RFC 2960; RFC 3286]. Für welche Anwendungen stellen sich die Designer von SCTP dessen Einsatz vor? Welche Merkmale von SCTP wurden hinzugefügt, um den Bedürfnissen dieser Anwendungen entgegenzukommen?

Programmieraufgaben

Implementieren eines zuverlässigen Transportprotokolls

In dieser Programmieraufgabe schreiben Sie den Code für beide Seiten einer Transportschicht (Sender und Empfänger), welche einen einfachen zuverlässigen Datentransfer ermöglicht. Es gibt zwei Versionen dieser Aufgabe: mit dem Alternierenden-Bit-Protokoll und GBN. Diese Aufgabe sollte Spaß machen – Ihre Implementierung unterscheidet sich nur wenig von dem, was in der wirklichen Welt benötigt würde.

Wahrscheinlich haben Sie nicht mehrere Computer mit einem Betriebssystem, das Sie nach Belieben modifizieren können, zur Hand. Deshalb soll Ihr Code in einer simulierten Hardware-/Softwareumgebung arbeiten. Allerdings ist die Programmierschnittstelle Ihrer Routinen – der Code, der Ihre Instanzen von oberhalb und unterhalb aufrufen würde – nahe an dem, was in einer tatsächlichen UNIX-Umgebung abläuft. (Tatsächlich sind die in dieser Programmieraufgabe beschriebenen Software-Schnittstellen viel realistischer als die Endlosschleifensender und -empfänger, die viele Lehrbücher beschreiben.) Das Starten und Stoppen von Timern wird ebenfalls simuliert und Timer-Interrupts werden dafür sorgen, dass ihre Routinen zum Timer-Handling aufgerufen werden.

Die komplette Aufgabenstellung sowie den Code, den Sie mit Ihrem eigenen Code zusammen kompilieren müssen, finden Sie auf der Website dieses Buches.

Weblink

Wireshark-Experiment: Eine Untersuchung von TCP

In dieser Übung verwenden Sie Ihren Webbrowser, um auf eine Datei über einen Webserver zuzugreifen. Wie in früheren Experimenten verwenden Sie Wireshark, um die Pakete aufzuzeichnen, die an Ihrem Computer ankommen. Im Gegensatz zu früheren Experimenten werden Sie auch in der Lage sein, ein Wireshark-lesbares Paket-Trace vom gleichen Webserver herunterzuladen, von dem Sie die Datei herunterladen. In diesem Server-Trace finden Sie die Pakete, die von Ihrem eigenen Zugriff auf den Webserver generiert wurden. Sie werden die client- und serverseitigen Traces analysieren, um einige Aspekte von TCP zu untersuchen. Insbesondere werden Sie die Leistung der TCP-Verbindung zwischen Ihrem Computer und dem Webserver beurteilen. Sie verfolgen das Fensterverhalten von TCP und schließen auf Paketverlust, Übertragungswiederholung, Flusskontrolle, Überlastkontrollverhalten und geschätzte Rundlaufzeit.

Wie bei allen Wireshark-Experimenten finden Sie die ausführliche Beschreibung dieses Experimentes auf der Website zum Buch.

Weblink

Wireshark-Experiment: Eine Untersuchung von UDP

In diesem Experiment zeichnen Sie Datenverkehr auf und analysieren Ihre bevorzugte Anwendung, die UDP verwendet (zum Beispiel DNS oder eine Multimedia-Anwen-

dung wie Skype). Wie wir in Abschnitt 3.3 gelernt haben, ist UDP ein einfaches, schlichtes Transportprotokoll. Sie untersuchen die Header-Felder sowohl im UDP-Segment als auch bei der Prüfsummenberechnung.

Weblink

Wie bei allen Wireshark-Experimenten finden Sie die ausführliche Beschreibung dieses Experimentes auf der Website zum Buch.

Interview mit Sally Floyd

Sally Floyd ist Wissenschaftlerin am ICSI-Center for Internet Research, einem Institut, das sich mit Fragen des Internets und von Computernetzwerken im Allgemeinen befasst. In der Industrie ist sie für ihre Arbeiten zum Internet-Protokolldesign bekannt, insbesondere zu zuverlässigem Multicast, Überlastkontrolle (TCP), Packet Scheduling (RED) und Protokollanalyse. Sally hat einen Bachelor in Soziologie der University of California, Berkeley sowie einen Master und einen Doktortitel in Informatik von derselben Universität.

Wieso haben Sie sich für das Studium der Informatik entschieden?

Nachdem ich meinen Bachelor in Soziologie hatte, musste ich mir einen Weg suchen, für mich selbst aufzukommen. Ich machte schließlich einen zweijährigen Kurs in Elektronik an meinem örtlichen College und verbrachte danach zehn Jahre mit Arbeiten in den Bereichen Elektronik und Informatik. Dies beinhaltete acht Jahre als Systemingenieurin an den Computern, welche die schnellen Transitzüge der Bay Area steuern. Ich beschloss, etwas mehr formale Informatik zu lernen, und bewarb mich später an der Graduiertenschule des Informatikfachbereiches der UC Berkeley.

Warum haben Sie sich dafür entschieden, sich auf Netzwerke zu spezialisieren?

In der Graduiertenschule begann ich, mich für theoretische Informatik zu interessieren. Ich arbeitete zuerst an der probabilistischen Analyse von Algorithmen und später an Computerlerntheorie. Ich arbeitete auch einen Tag im Monat am LBL (Lawrence Berkeley Laboratory), wo mein Büro gegenüber dem von Van Jacobson lag, der damals an TCP-Überlastkontrollalgorithmen arbeitete. Van fragte mich, ob ich während des Sommers an der Analyse von Algorithmen für netzwerkbezogene Probleme arbeiten wollte, was die Untersuchung unerwünschter Synchronisation von periodischen Routing-Nachrichten beinhaltete. Das klang für mich interessant, so dass ich diese Analyse den Sommer über durchführte.

Nachdem ich meine Doktorarbeit beendet hatte, bot mir Van eine Vollzeitstelle an, bei der ich die Arbeit über Netzwerke fortsetzen sollte. Ich hatte nicht unbedingt beabsichtigt, mich jahrelang mit Netzwerken zu beschäftigen, aber ich empfand Netzwerkforschung als zufriedenstellender als theoretische Informatik. Ich stellte fest, dass ich in der angewandten Welt glücklicher bin, in der die Folgen meiner Arbeit greifbarer sind.

Was war Ihre erste Stelle in der Computerindustrie? Was brachte sie mit sich?

Meine erste Stelle war von 1975 bis 1982 bei BART (Bay Area Rapid Transit), wo ich an den Computern arbeitete, welche die BART-Züge steuerten. Ich begann als Technikerin, wartete und reparierte die verschiedenen verteilten Computersysteme, die daran beteiligt sind, das BART-System am Laufen zu halten.

Das beinhaltete ein Zentralrechnersystem und verteilte Minicomputersysteme zur Kontrolle der Zugbewegung, ein System von DEC-Computern für die Darstellung von Anzeigen und Zugzielen auf den Anzeigetafeln und ein System von Modcomp-Computern für die Beschaffung von Informationen von den Fahrkartenschaltern. Meine letzten Jahre bei BART verbrachte ich mit einem Gemeinschaftsprojekt von BART und LBL, in dem der Ersatz von BARTs alterndem Zugsteuer-Rechnersystem gestaltet werden sollte.

Was stellt in Ihrem Job die größten Herausforderungen dar?

Die aktuelle Forschung ist der herausforderndste Teil. Derzeit beinhaltet ein Forschungsthema die Untersuchung von Themen der Überlastkontrolle für Anwendungen wie Multimedia-Ströme. Ein zweites Thema betrachtet Probleme, die einer ausführlicheren Kommunikation zwischen Routern und Endknoten im Wege stehen. Dies beinhaltet IP-Tunnel und MPLS-Pfade, Router oder andere zwischengeschaltete Systeme, die Pakete mit IP-Optionen verwerfen, komplexe Schicht-2-Netzwerke und Potenziale für Netzwerkangriffe. Ein drittes Thema betrifft die Erforschung, wie unsere Auswahl an Modellszenarien in der Analyse, Simulation und bei Experimenten unsere Bewertung der Leistung von Überlastkontrollmechanismen beeinflusst. Weitere Informationen zu diesen Themen sind auf den Webseiten zu DCCP, Quick Start und TMRG zu finden, die über *http://www.icir.org/floyd* erreichbar sind.

Was ist Ihre Sicht auf die Zukunft der Netzwerke und des Internets?

Eine Möglichkeit besteht darin, dass die typische Überlast, mit der der Internetverkehr zu kämpfen hat, weniger schwerwiegend wird, da die verfügbare Bandbreite schneller wächst als der Bedarf. Ich sehe den Trend in Richtung weniger schwerer Überlast, obwohl mittelfristig wachsende Überlast mit gelegentlichen Netzzusammenbrüchen wegen zu starker Überlast auch nicht unmöglich scheint.

Die Zukunft des Internets selbst, oder der Internetarchitektur, sehe ich nicht klar. Viele Faktoren können zu raschen Änderungen beitragen, daher ist es schwer vorherzusagen, wie das Internet oder die Internetarchitektur sich entwickeln werden oder wie erfolgreich diese Entwicklung die vielen potenziellen Fallen auf ihrem Weg vermeiden kann.

Ein bekannter negativer Trend ist die wachsende Schwierigkeit, Änderungen an der Internetarchitektur vorzunehmen. Die Internetarchitektur ist nicht mehr ein schlüssiges Ganzes und die verschiedenen Bestandteile wie Transportprotokolle, Routermechanismen, Firewalls, Lastverteilung, Sicherheitsmechanismen usw. dienen manchmal entgegengesetzten Zwecken.

Welche Menschen haben Sie beruflich inspiriert?

Richard Karp, der Betreuer meiner Doktorarbeit an der Graduiertenschule, hat mir im Grunde genommen gezeigt, wie man Wissenschaft betreibt, und Van Jacobson, mein „Gruppenleiter" bei LBL, war für mein Interesse an Netzwerken und für einen Großteil meines Verständnisses der Internetinfrastruktur verantwortlich. Dave Clark hat mich durch seine klare Sicht der Internetarchitektur inspiriert und durch seine Rolle in der Entwicklung dieser Architektur durch Forschung, Schreiben und Beteiligung an der IETF und anderen öffentlichen Foren. Deborah Estrin inspirierte mich durch ihre Konzentration und Effektivität und ihre Fähigkeit, bewusst zu entscheiden, woran sie arbeiten möchte und warum.

Einer der Gründe, warum ich die letzten zehn Jahre in der Netzwerkforschung genossen habe, ist, dass so viele Leute in diesem Feld arbeiten, die ich mag und respektiere und die mich inspirieren. Sie sind intelligent, arbeiten hart und engagieren sich stark für die Entwicklung des Internets. Sie leisten beeindruckende Arbeit und sind gute Begleiter auf ein Bier und ein freundschaftliches Streitgespräch (oder eine Versöhnung) nach einem Tag voller Besprechungen.