而 β 可用以下伪代码计算出来。其中，B 是最新拥塞事件前的吞吐量，B_{last} 是倒数第二个拥塞事件前的吞吐量，$RTT_{\min}$ 是 RTT 的最小值，$RTT_{\max}$ 是 RTT 的最大值。

$$\textbf{If} \left| \frac{B - B_{\text{last}}}{B} \right| > 0.2$$

$$\textbf{Then } \beta \leftarrow 0.5$$

$$\textbf{Else } \beta \leftarrow \frac{RTT_{\min}}{RTT_{\max}}$$

基于 ns-3 的模拟结果 H-TCP

在与 NewReno 相同的条件下，H-TCP 的模拟结果如图 4.13 所示。可以看到，在 3.0 秒附近，状态迁移到 Open 之后，*cwnd* 的增量值随着时间而逐渐变大（图 4.13✪）。在这种情况下，即使是长肥管道，也可以有效地利用带宽，但这样会导致拥塞发生时大量数据包被废弃的问题。

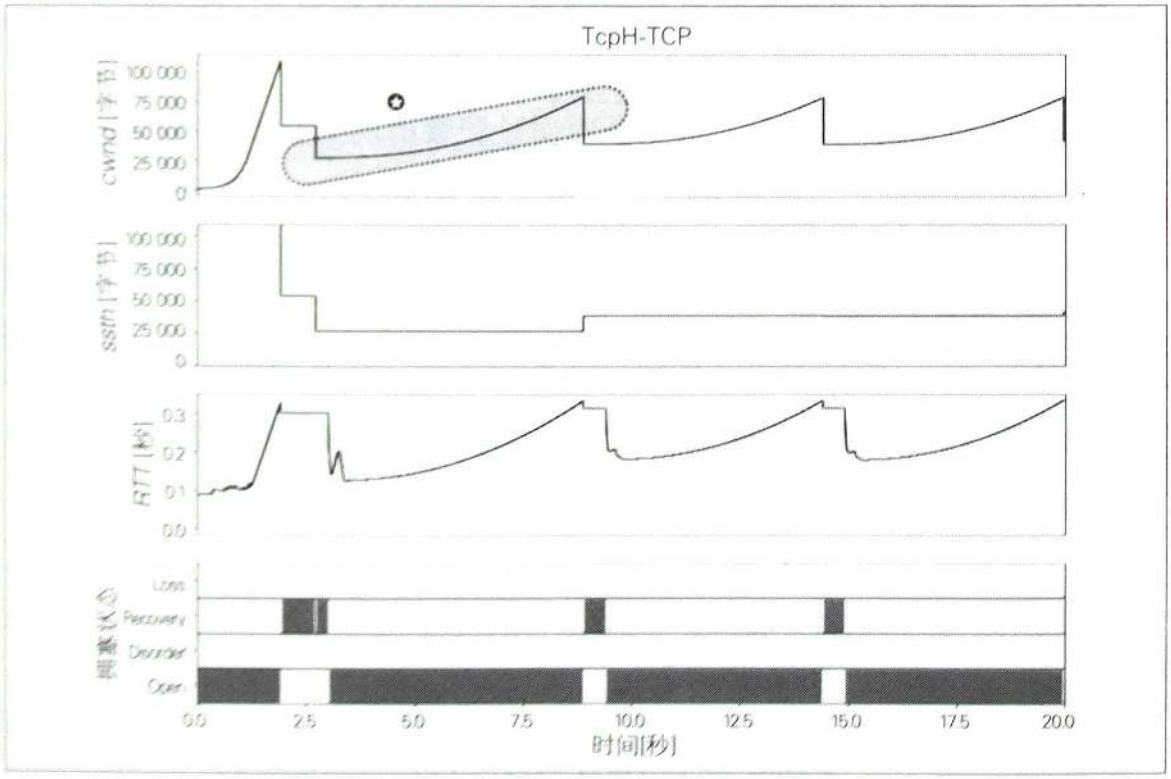

图 4.13 H-TCP 的模拟结果

Hybla 面向 RTT 较大的通信链路的基于丢包的拥塞控制算法

Hybla 于 2005 年提出，是一个基于丢包的拥塞控制算法。众所周知，随着 *RTT* 逐渐增大，NewReno 算法的 *cwnd* 和吞吐量会急剧减小。

——计算公式 Hybla

于是，为了能在卫星通信这种 *RTT* 很大的通信链路上也能有一个不太低的吞吐量，Hybla 像下面这样修改了在 Open 状态下的计算公式。

$$\begin{aligned}&\textbf{If } cwnd \leqslant ssthresh\\&\quad\textbf{Then } cwnd \leftarrow cwnd + (2^{\rho} - 1)\cdot MSS\\&\quad\textbf{Else } cwnd \leftarrow cwnd + \frac{\rho^2 \cdot MSS}{cwnd}\end{aligned}$$

上面公式中的 ρ 是使用参数 RTT_0 归一化后的 *RTT*，如下所示。

$$\rho = \frac{RTT}{RTT_0}$$

——基于 ns-3 的模拟结果 Hybla

在与 NewReno 相同的条件下，Hybla 的模拟结果如图 4.14 所示。在 Slow Start 状态下，快速地增大 *cwnd*（图 4.14❶），结果就是迁移到了 Loss 状态（❷）。通过 4.4 节介绍的 ns-3，使用不同大小的 *RTT* 值进行模拟，并观察 Hybla 的运行，便可以更加深入地理解 Hybla。

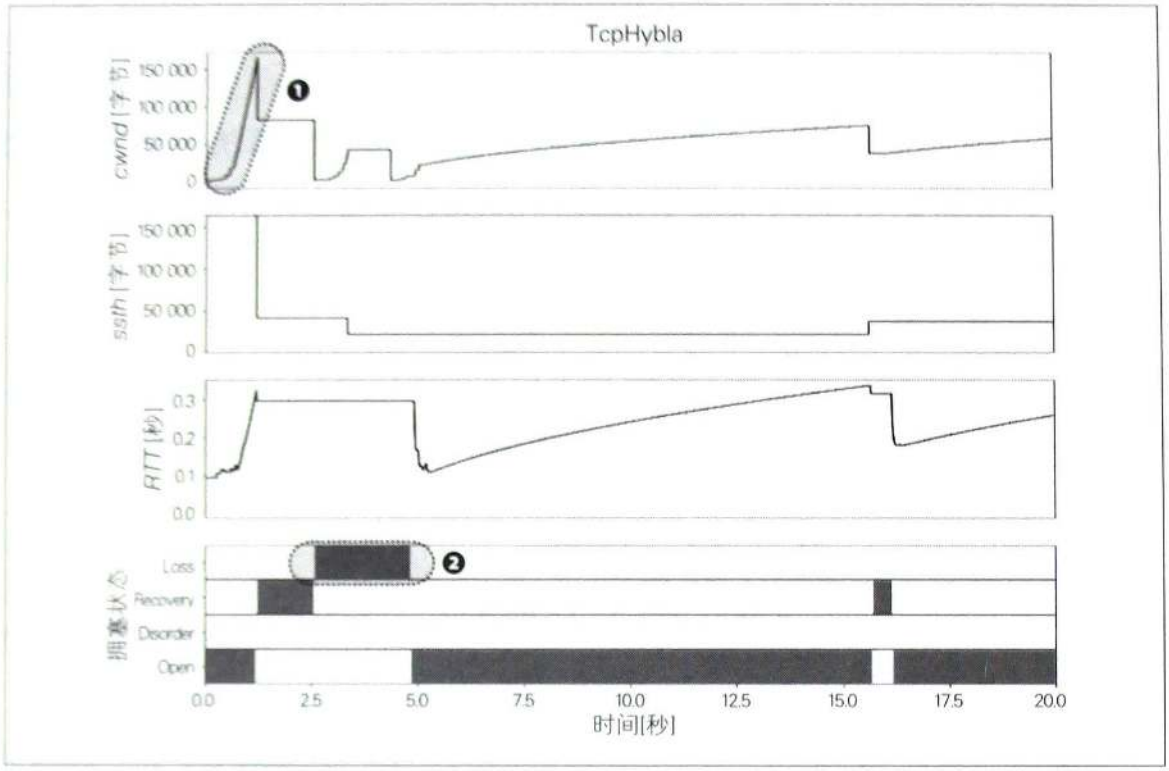

图 4.14 Hybla 的模拟结果

Illinois 与 BIC 形成对比的混合型拥塞控制算法

Illinois 于 2006 年提出，是一个面向长肥管道的混合型拥塞控制算法。以往的面向长肥管道的基于丢包的拥塞控制算法，*cwnd* 在增大的同时，其每次的增量也逐渐变大，因此一旦拥塞发生，就会有大量数据包被废弃。

而 Illinois 会根据 *RTT* 的值调整 AIMD 的参数 α 和 β，以此实现在拥塞极有可能发生时控制住 *cwnd* 的增量。想必读者可以很容易地察觉到，在问题定义和解决上，BIC 和 Illinois 采用的方法比较相似。

计算公式 Illinois

在 Congestion avoidance 状态下，Illinois 每次接收到 ACK 后，就会计算队列时延（详见 5.1 节）d_a 及其对应的上限值 d_m。此处，T_{max} 是最大的 *RTT*，而 T_{min} 是最小的 *RTT*，T_a 则代表最近 *cwnd* 次的 *RTT* 的平均近似值。

$$d_a \leftarrow T_a - T_{\min}$$
$$d_m \leftarrow T_{\max} - T_{\min}$$

以上述的 d_a 和 d_m 为基础，计算中间参数 κ。在这里，$0 < \alpha_{\min} \leqslant 1 \leqslant \alpha_{\max}$、$0 < \beta_{\min} \leqslant \beta_{\max} \leqslant \frac{1}{2}$、$W_{thresh} > 0$、$0 \leqslant \eta_1 < 1$ 和 $0 \leqslant \eta_2 < \eta_3 \leqslant 1$ 都是事先设置好的参数。下面的计算公式虽然看起来比较复杂，但它们主要是为了使 α 和 β 的计算函数能够成为连续函数而进行调整之后的结果[①]。

$$d_1 \leftarrow \eta_1 d_m$$
$$d_2 \leftarrow \eta_2 d_m$$
$$d_3 \leftarrow \eta_3 d_m$$
$$\kappa_1 \leftarrow \frac{(d_m - d_1)\alpha_{\min}\alpha_{\max}}{\alpha_{\max} - \alpha_{\min}}$$
$$\kappa_2 \leftarrow \frac{(d_m - d_1)\alpha_{\min}}{\alpha_{\max} - \alpha_{\min}} - d_1$$
$$\kappa_3 \leftarrow \frac{\beta_{\min} d_3 - \beta_{\max} d_2}{d_3 - d_2}$$
$$\kappa_4 \leftarrow \frac{\beta_{\max} - \beta_{\min}}{d_3 - d_2}$$

通过上面公式中的 κ_1 和 κ_2，计算 AIMD 的 α 值。d_a 越大，拥塞可能性越高，α（每个 *RTT* 的 *cwnd* 的增加量）也就越小。与上面类似，像下面公式这样设计 κ_1 和 κ_2 的值，也是为了让 α 成为一个连续函数。

$$\textbf{If } d_a \leqslant d_1$$
$$\textbf{Then } \alpha \leftarrow \alpha_{\max}$$
$$\textbf{Else } \alpha \leftarrow \frac{\kappa_1}{\kappa_1 + d_a}$$

此外，使用前述公式中的 κ_3 和 κ_4，计算 AIMD 中的 β。d_a 越大，拥

① 详细的导出过程请参考“TCP-Illinois: A loss- and delay-based congestion control algorithm for high-speed networks”一文。

塞可能性越高，β（状态迁移至 Recoery 时 *cwnd* 的减小比例）也就越大。与上面类似，像下面公式这样设计 κ_3 和 κ_4 的值，也是为了让 β 成为一个连续函数。

$$
\begin{aligned}
&\textbf{If } d_a \leqslant d_2 \\
&\quad \textbf{Then } \beta \leftarrow \beta_{\min} \\
&\quad \textbf{Else If } d_a < d_3 \\
&\qquad \textbf{Then } \beta \leftarrow \kappa_3 + \kappa_3 d_a \\
&\qquad \textbf{Else } \beta \leftarrow \beta_{\max}
\end{aligned}
$$

基于 ns-3 的模拟结果 Illinois

在与 NewReno 相同的条件下，Illinois 的模拟结果如图 4.15 所示。从图中可以看出，在进入 Congestion avoidance 状态之前，由于不计算 d_a 的值，所以 Illinois 的行为与 NewReno 一样。随后可以看到，从 Recovery 状态迁移到 Open 状态（Congestion avoidance 状态）之后，*cwnd* 的增长逐渐放缓（图 4.15✪）。

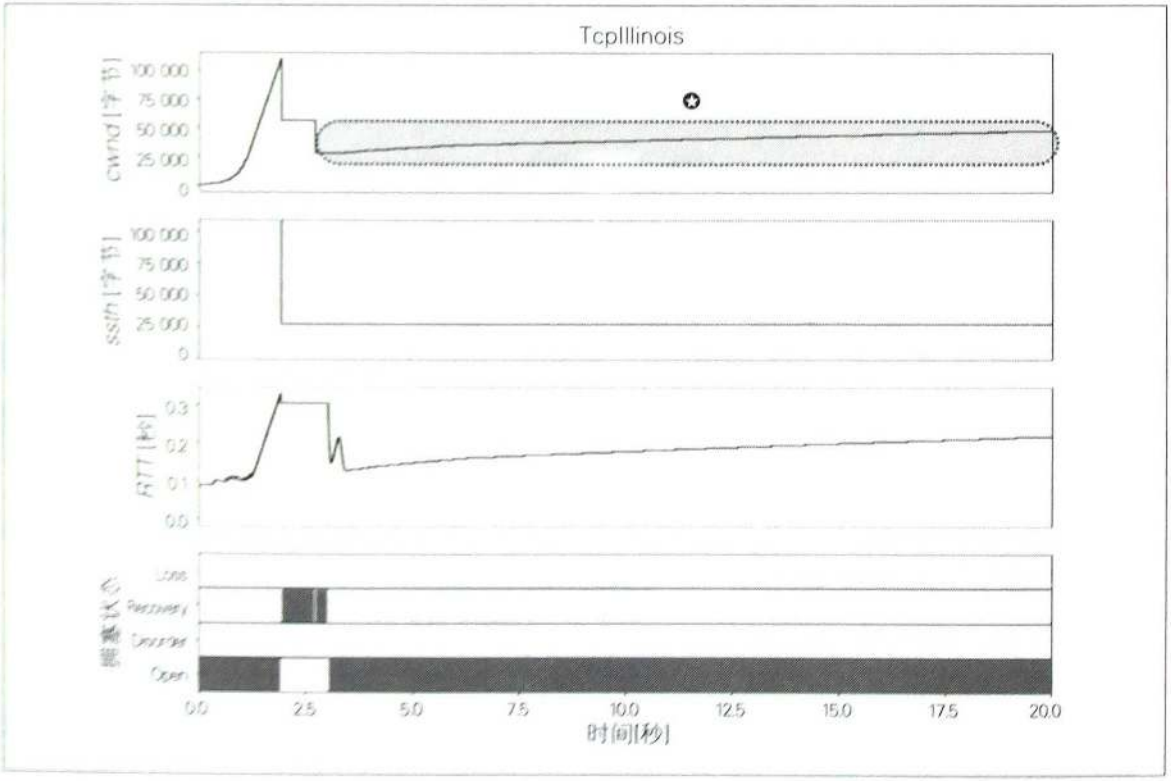

图 4.15 Illinois 的模拟结果

另外，由于基于同样的问题设定而被开发出来的 BIC 算法，自最初的 Open 状态开始就比 NewReno 要积极，所以在这个模拟环境下，BIC 能比 Illinois 更有效地利用带宽。如果使用 4.4 节介绍的 ns-3，使用不同的条件来比较 BIC 和 Illinois，大家应该能有一个更深入的理解。

YeAH 拥有两种模式、面向长肥管道的混合型拥塞控制算法

YeAH（Yet Another Highspeed）于 2007 年提出，是一个面向长肥管道的混合型拥塞控制算法。YeAH 通过分别使用 Slow 和 Fast 两种模式，可以同时满足以下所有条件。

- 长肥管道下带宽利用率高
- 规避 *cwnd* 急剧增长给网络带来的过多负担
- 可以与 Reno 公平地分享带宽（**与 Reno 的亲和性**）
- 可以与 *RTT* 不同的网络流公平地分享带宽（***RTT* 公平性**）
- 在随机丢包方面鲁棒性高
- 即使存在缓冲区较小的链路，也可以发挥较高的性能

计算公式 YeAH

YeAH 会针对每个 *RTT* 计算以下公式中的 Q 和 L，并进行模式切换。其中，RTT_{base} 代表 *RTT* 的最小值，RTT_{min} 代表对应 *RTT* 中的 *RTT* 最小值。综上所述，Q 就是通信链路中缓存的数据量。

$$Q \leftarrow (RTT_{\text{min}} - RTT_{\text{base}}) \cdot \frac{cwnd}{RTT_{\text{min}}}$$

$$L \leftarrow \frac{RTT_{\text{min}} - RTT_{\text{base}}}{RTT_{\text{base}}}$$

当 Q 和 L 满足 $Q < Q_{\text{max}}$ 和 $L < \frac{1}{\phi}$ 时，YeAH 进入 Fast 模式，否则进入 Slow 模式。这里，Q_{max} 和 ϕ 是可设置的参数。

当处在 Fast 模式下时，YeAH 与 Scalable、H-TCP 算法一样，进行比较积极的拥塞控制行为。而在 Slow 模式下，YeAH 的行为与 NewReno 一致。但是，当 $Q > Q_{max}$ 时，为了减少通信链路中缓存的数据量，会针对每个 RTT 让 *cwnd* 减小 Q。这一点与 NewReno 不一样。

然而，当与 NewReno 等完全不考虑缓冲区[1]的拥塞控制算法共存时，YeAH 算法会占据所有减少的数据缓冲区，最终导致无法规避拥塞。因此，YeAH 会根据 Slow 模式和 Fast 模式各自运行的次数，判断当前是否在与这些完全不考虑缓冲区的拥塞控制算法在一起工作，并依据判断结果切换最终的工作模式。

基于 ns-3 的模拟结果 YeAH

在与 NewReno 相同的条件下，YeAH 的模拟结果如图 4.16 所示。由于 YeAH 自始至终处于 Slow 模式下，所以从图中无法看出与 NewReno 的明显差异。如果使用 4.4 节介绍的 ns-3，观察 YeAH 在长肥管道下的运行情况，大家应该能有一个更深入的理解。

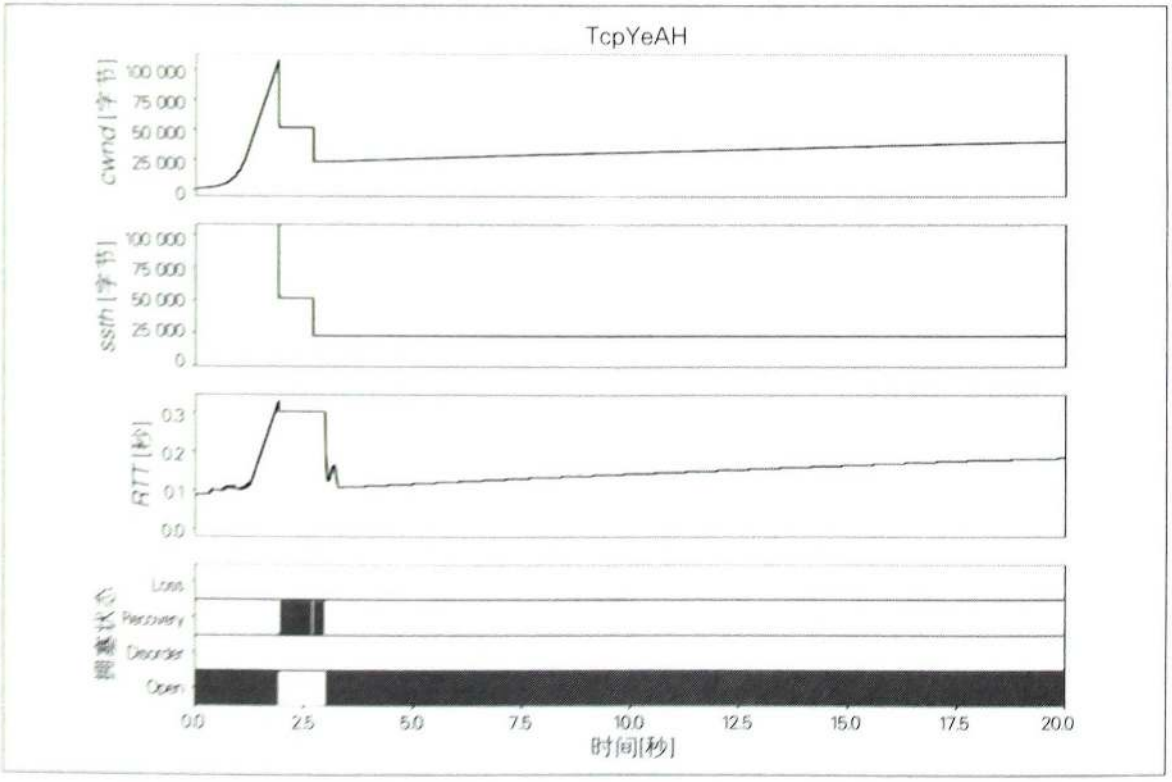

图 4.16 YeAH 的模拟结果

1) 原论文中将此行为描述为 greedy（贪婪）。

4.3 协议分析器 Wireshark 实践入门

拥塞控制算法的观察 ❶

百闻不如一见。本节，我们就使用协议分析器 Wireshark 来观察一下拥塞控制算法实际的运行情况。本节将使用 Wireshark 的 TCP Stream Graphs 功能来确认一下序列号、吞吐量、*RTT* 和发送窗口大小的变化情况。此外，本节也会介绍一下在虚拟机上更改拥塞控制算法的方法。

什么是 Wireshark

Wireshark 是最为主流的协议分析器之一。协议分析器是指解析网络中流量的设备与程序。这其中，既有可以运行在个人计算机上的轻量、免费的设备软件，也有工作在专用设备上、面向专业人士的天价设备软件。Wireshark 属于前者。由于 Wireshark 不仅免费，而且功能丰富，所以许多企业、非营利性组织、政府机构和学术机构将其作为默认标准使用。

下面，我们将使用 Wireshark 观察 TCP 拥塞控制算法的行为。

Wireshark 的环境搭建

Wireshark 支持 Windows、macOS 和 Linux，安装也十分简单方便[①]。

本书为了统一运行环境，将使用 VirtualBox 和 Vagrant 在虚拟机上构建 Ubuntu 环境。在笔者执笔时（2019 年 4 月 1 日），后文所述的 ns-3 安装向导暂时不支持 Ubuntu 18.04，因此本书使用 Ubuntu 16.04。本书的模拟主要是通过 Ubuntu 16.04 上启动的 Wireshark，使用 X Window System 在物理机上绘制图像，并以此观察 TCP 拥塞控制算法的行为。这里请再一次确认本书导言中介绍的 VirtualBox、Vagrant 和 X Server 的环境搭建是否完成。

① 详细的安装方法请参考 Wireshark 官方网站的介绍。

网络结构

网络结构如图 4.17 所示。在本书中，我们将安装在物理机上的操作系统称为宿主操作系统，将安装在虚拟机上的操作系统称为客户操作系统。本次模拟将搭建连接两台虚拟机的私有网络，并通过 Wireshark 进行数据抓包，观察通过 FTP 从第 1 台客户操作系统（guest1）向第 2 台客户操作系统（guest2）发送 100 MB 的数据文件时的网络流量情况。

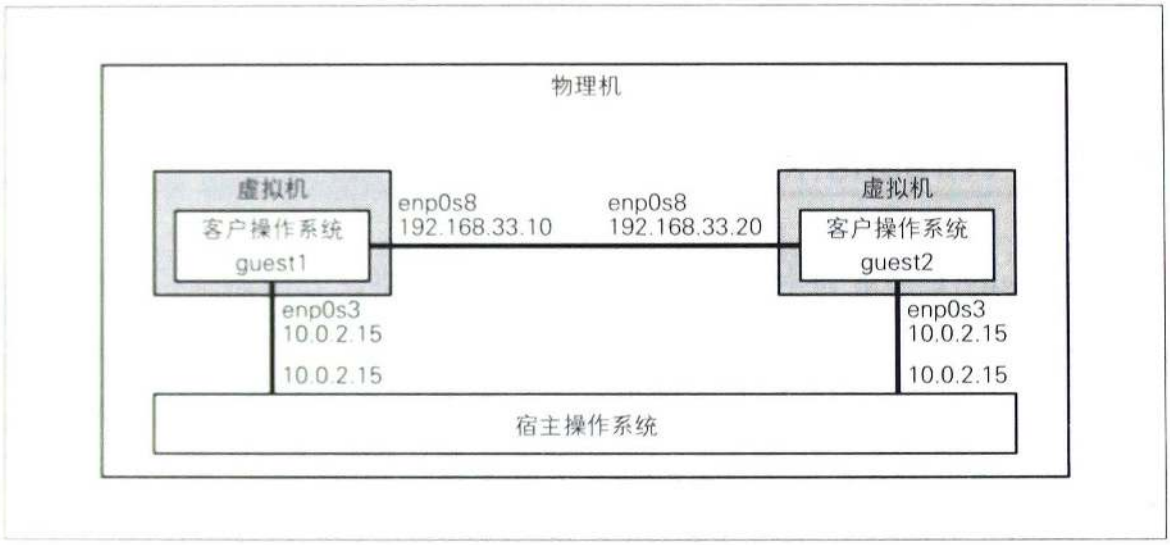

图 4.17 搭建用于 Wireshark 模拟的网络环境

设置

当确认已经准备好前述的所有环境之后，请将本书的 Github 仓库[1]克隆到任意目录中。然后，打开其中的 wireshark/vagrant 目录，运行命令 vagrant up。这样就可以在两台虚拟机上搭建 Ubuntu 16.04 的运行环境。

shell

```
$ git clone https://github.com/ituring/tcp-book.git
$ cd tcp-book/wireshark/vagrant
$ vagrant up
```

使用以下命令，通过 SSH 连接到客户操作系统上。在登录消息显示

1. URL https://github.com/ituring/tcp-book

之后，命令行提示会变成 vagrant@guest1:~$。

shell

```
$ vagrant ssh guest1

> Welcome to Ubuntu 16.04.5 LTS (GNU/Linux 4.4.0-139-generic x86_64)
>
> * Documentation:  部分省略
> * Management:     部分省略
> * Support:        部分省略
>
> Get cloud support with Ubuntu Advantage Cloud Guest:
> 部分省略
>
> 0 packages can be updated.
> 0 updates are security updates.
>
> New release '18.04.1 LTS' available.
> Run 'do-release-upgrade' to upgrade to it.

vagrant@guest1:~$
```

Wireshark 的启动与关闭

下面启动 Wireshark。

shell

```
vagrant@guest1:~$ wireshark
```

当看到如图 4.18 一样的画面时，表示 Wireshark 已经启动完成，准备工作也就完成了。此时请先登出，并关闭虚拟机。

shell

```
vagrant@guest1:~$ exit
$ vagrant halt
```

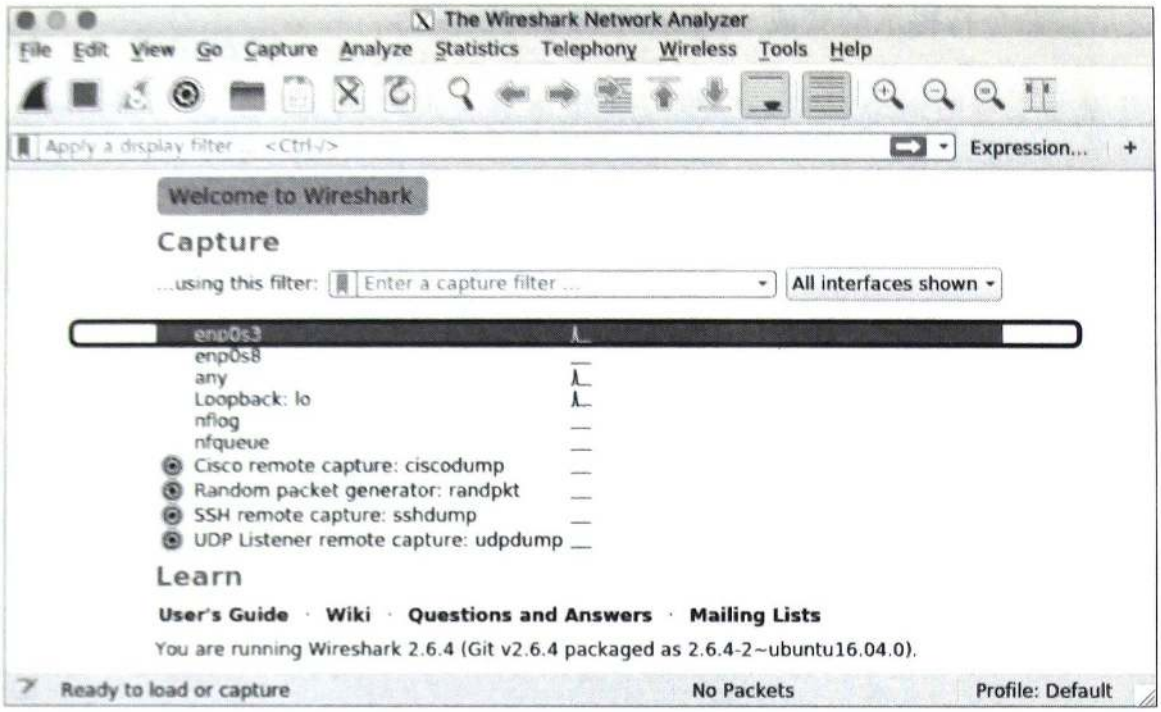

图 4.18 启动 Wireshark

使用 Wireshark 进行 TCP 首部分析

在开始观察拥塞控制算法之前，我们先来学习一下使用 Wireshark 分析 TCP 首部的方法吧。首先，启动虚拟机。

shell

```
$ vagrant up
```

本书的模拟过程需要同时启动 Wireshark 和 FTP 命令，因此要像图 4.19 一样，**同时打开两个 shell**。然后，分别在两个不同的 shell 中登录 `guest1`。

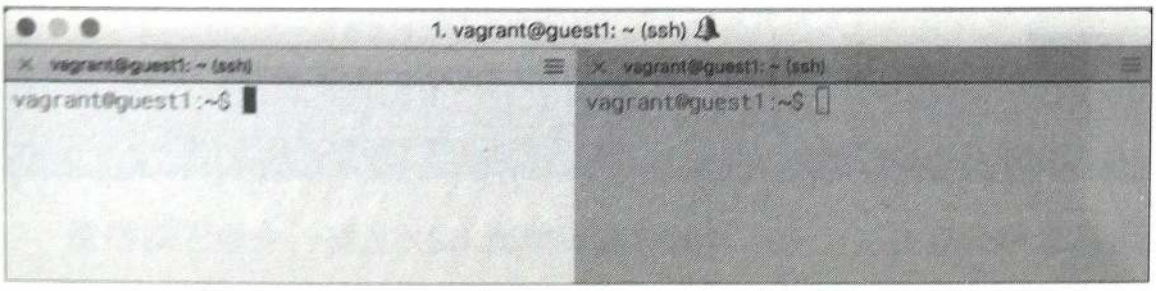

图 4.19 在两个 shell 下通过 SSH 连接 guest1

ell

```
$ vagrant ssh guest1
vagrant@guest1:~$
```

在第 1 个 Shell 中运行 Wireshark[①]。

shell

```
vagrant@guest1:~$ wireshark
```

——选择要抓包的网卡接口和观察数据包

选择要抓包的网卡接口。如前面的图 4.17 所示，enp0s3 是面向宿主操作系统的接口，而 enp0s8 是面向 guest2 的网卡接口。这里就说明一下选择 enp0s3（参考前面的图 4.18）并使用 Wireshark 进行数据包分析的方法[②]。

当选择 enp0s3 之后，就可以在如图 4.20 所示的界面中实时看到进出 enp0s3 的数据包情况。Wireshark 的默认界面主要分为上、中、下 3 个部分。请看图 4.20，上面的❶是进出 enp0s3 的各个数据包的概览，中间的❷是❶中选择的数据包的详细信息，而下面的❸则是以二进制表示的具体内容。

从结果可以看出，经过 enp0s3 的是宿主操作系统和客户操作系统之间连续收发的数据包，这些数据包展示在❶中，并不断增多。另外，点击图中❹旁边的矩形按钮[③]之后，就可以暂时停止抓包。

——数据包的 TCP 首部分析

本次模拟主要关注 TCP 的运行，因此在图 4.20❶的数据包列表中，我们选择❶的 Protocol（协议）列为 TCP 的一个数据包。当选中了图 4.20❷的 Source（发送方 IP 地址）列为宿主操作系统 10.0.2.2，而 Destination（目的地 IP 地址）列为客户操作系统 10.0.2.15 的数据包（图 4.20❸）时，显示的详细结果如图 4.21 所示。

---

① 由于之后还要使用第 2 个 shell，所以先保持登录状态。

② Wireshark 的相关教程，请参考 Wireshark user's guide。

③ 当鼠标悬停在上面时，会提示“停止抓包”。按钮的实际颜色是红色的。

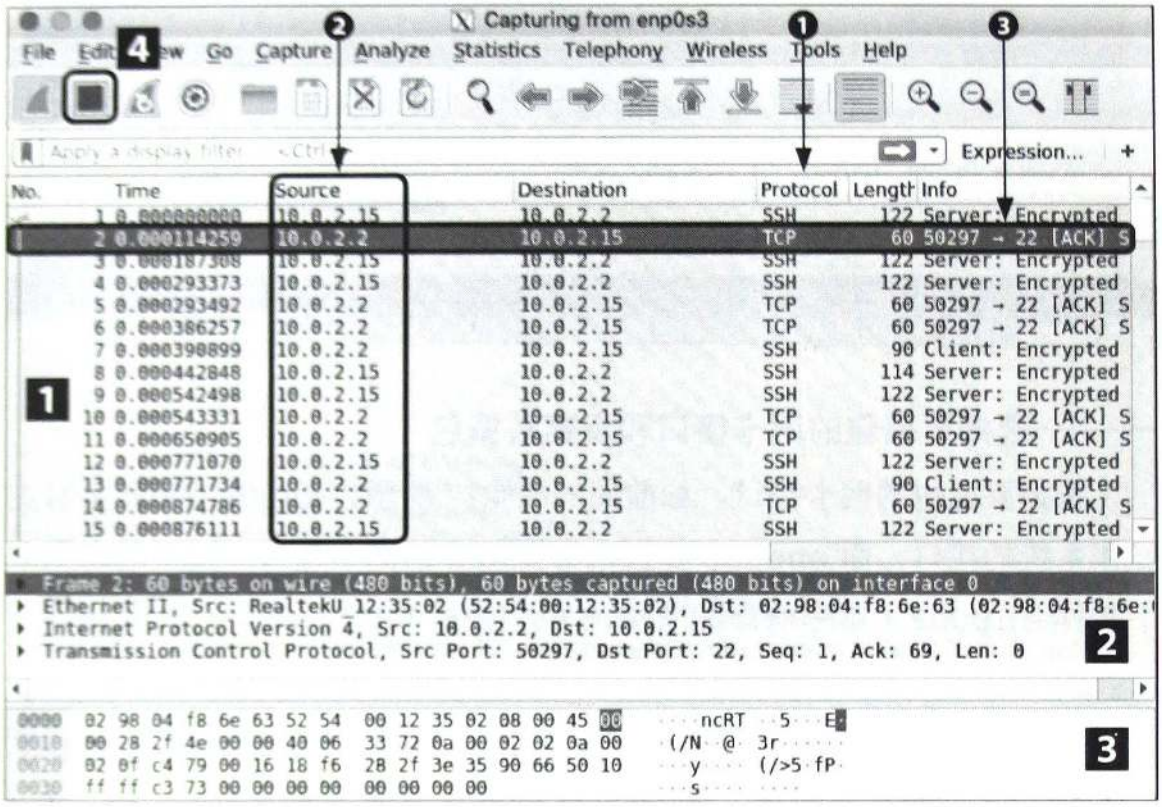

图 4.20 面向宿主操作系统的网络接口 enp0s3 的抓包结果

```
▸ Frame 2: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
▸ Ethernet II, Src: RealtekU_12:35:02 (52:54:00:12:35:02), Dst: 02:98:04:f8:6e:63 (02:98:04:f8:6e:
▸ Internet Protocol Version 4, Src: 10.0.2.2, Dst: 10.0.2.15
▾ Transmission Control Protocol, Src Port: 50297, Dst Port: 22, Seq: 1, Ack: 69, Len: 0
    Source Port: 50297
    Destination Port: 22
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 1    (relative sequence number)
    [Next sequence number: 1    (relative sequence number)]
    Acknowledgment number: 69    (relative ack number)
    0101 .... = Header Length: 20 bytes (5)
  ▾ Flags: 0x010 (ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
      .... .... ..0. = Syn: Not set
      .... .... ...0 = Fin: Not set
      [TCP Flags: ·····A····]
    Window size value: 65535
    [Calculated window size: 65535]
    [Window size scaling factor: -1 (unknown)]
    Checksum: 0xc373 [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
  ▾ [SEQ/ACK analysis]
      [This is an ACK to the segment in frame: 1]
      [The RTT to ACK the segment was: 0.000114259 seconds]
  ▾ [Timestamps]
      [Time since first frame in this TCP stream: 0.000114259 seconds]
      [Time since previous frame in this TCP stream: 0.000114259 seconds]
```

图 4.21 使用 Wireshark 进行 TCP 首部分析（放大上面的图 4.20 2）

我们来看一下图 4.21 中数据包的各层信息。从图中可以看出，此数据包在数据链路层使用以太网，在网络层使用 IP 协议，然后在传输层使用 TCP。点击 TCP 之后，可以展开 TCP 层的详细信息。

例如，我们可以看出 Source Port（发送方端口号）是 50297，Destination Port（目的地端口号）是 22。Sequence number（序列号）是 1，Acknownlegement number（确认应答号）是 69，Flags（标志位）中只有 ACK 被置位了，Window size value（接收窗口大小）是 65535。使用 Wireshark，便可以像这样分析各种数据包的首部信息。

当确认完这些信息之后，先把 Wireshark 暂时关闭。虽然接下来会出现如图 4.22 一样的提示框，但是选中并按下“Stop and Quit without Saving”就可以了。

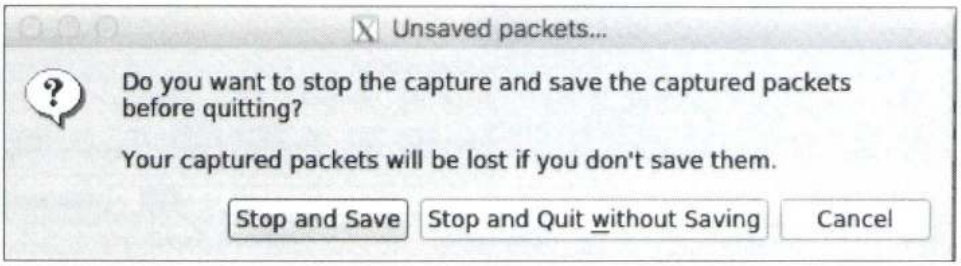

图 4.22 关闭 Wireshark 时屏幕显示的消息

通过 Wireshark 观察拥塞控制算法

接下来，我们就一起来看一下 TCP 拥塞控制的情况吧。本次的模拟过程是通过 FTP 从 guest1 的 enp0s8（192.168.33.10）向 guest2 的 enp0s8（192.168.33.20）发送 100 MB 的文件。这里再次提一下，本次模拟同时进行 Wireshark 抓包和 FTP 文件发送，因此需要用两个 shell 登录 guest1。请务必再次确认一下你的环境配置情况。

确认所用的拥塞控制算法 sysctl 命令（Ubuntu）

首先看一下 guest1 所用的拥塞控制算法。Ubuntu 可以通过以下的 sysctl 命令来确认。

shell

```
vagrant@guest1:~$ sysctl net.ipv4.tcp_congestion_control
> net.ipv4.tcp_congestion_control = reno  ← 拥塞控制算法是Reno
```

我们可以看到，这里使用的拥塞控制算法是 Reno。接下来，先使用第 1 个 shell 启动 Wireshark。

shell

```
vagrant@guest1:~$ wireshark
```

选择接口和确认数据包的收发情况

前面选择了与宿主操作系统对应的网络接口 `enp0s3`，这次就要选择与另一个客户操作系统（`guest2`）对应的网络接口 `enp0s8`。此时，如图 4.23 所示，我们可以看到两个接口间没有收发任何数据包。

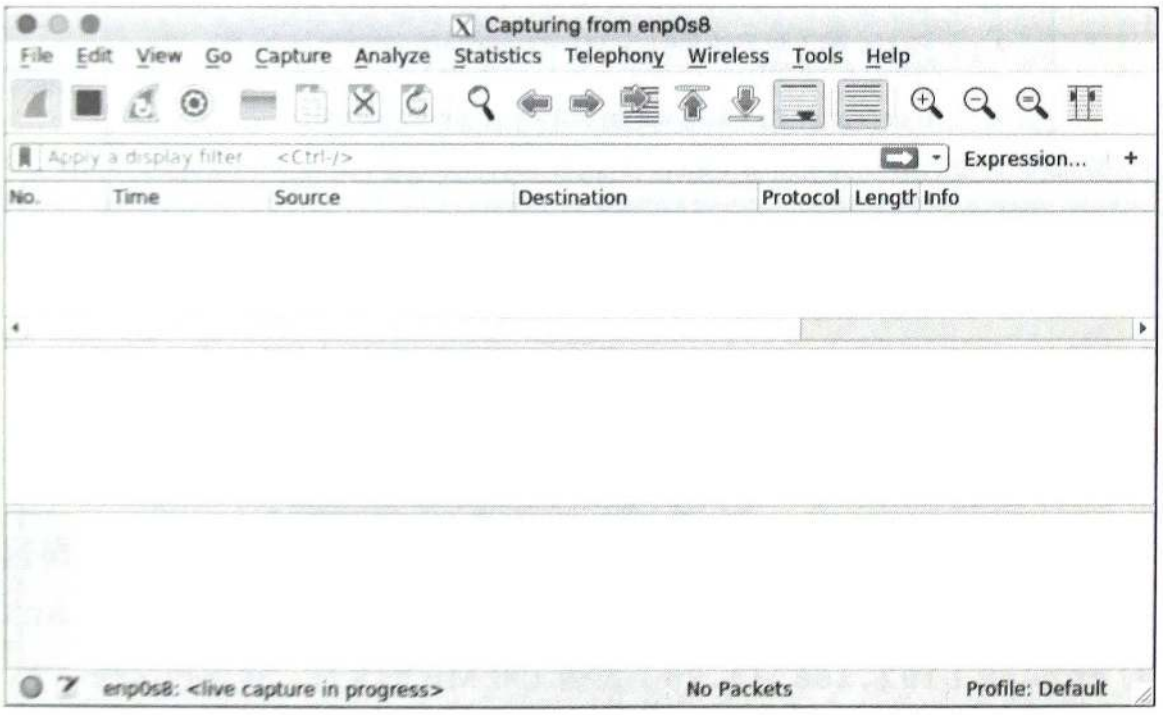

图 4.23 enp0s8 的初始状态

接下来，使用第 2 个 shell 运行下页这行命令，开始 FTP 发送。此时，建议如图 4.24 一样，同时查看两个 shell 和 Wireshark 窗口，这样可以实时地观察数据包的动向。