Consider a server transferring a file to a client after fragmenting it into packets. The server faces the nontrivial problem of choosing the rate at which to inject these packets into the network. If it sends all the packets at once, it may send them faster than the network or the client can process them, leading to packet loss. Because the server must retransmit lost packets, it is better off sending packets at a rate that both the network and the client can handle. If, however, the server sends packets slower than the highest sustainable rate, transferring the file takes longer than necessary. Thus, the server should carefully choose its transmission rate to neither overflow nor underflow the network or the client.

Flow control refers to the set of techniques that enable a data source to match its transmission rate to the currently available service rate at a receiver and in the network. Besides this primary goal, a flow control mechanism should meet several other, sometimes mutually contradictory objectives. It should be simple to implement, use the least possible network resources (in terms of bandwidth and buffers at multiplexing points), and work effectively even when used by many sources (that is, scale well). If possible, each member of the ensemble of flow-controlled sources sharing a scarce resource should restrict its usage to its fair share. Finally, the ensemble of sources should be stable, which means, loosely speaking, that when the number of sources is fixed, the transmission rate of each source settles down to an equilibrium value. Stability also implies that, if a new source becomes active, existing active sources adjust their transmission rates so that, after a brief transient period, the system settles down to a new equilibrium.
This range of objectives allows for many interesting trade-offs. For example, we can trade simplicity for fairness, designing a scheme that is simple to implement, but does not guarantee a fair share to every source. Other, more complex trade-offs are also possible, and we will see many of them later in this chapter. The variety of choices available to the designer has led to many different flow-control schemes being proposed in the literature (practically every networking conference in the last decade has one or more papers on flow control). Some schemes described in the literature are only paper proposals, whereas others have been implemented in real networks and are widely used. In general, the more widely used a flow-control algorithm, the better it has been studied, and the more it deals with implementation difficulties. In this chapter, we will study flow-control schemes that either illustrate an important control mechanism or are widely used, or both. We will also evaluate the pros and cons of these schemes, based on how well they satisfy the objectives just described.

We can implement flow control at the application, transport, network, or datalink layer of a protocol stack. The choice of layer depends on the situation at hand. The most common design is to place end-to-end flow control at the transport layer, and hop-by-hop (link-level) flow control in the datalink layer. However, other arrangements are possible and, depending on the situation, are just as correct. In this chapter, we will study the abstract flow control problem, without worrying too much about layering.

We mention in passing that flow control is often confused with congestion control. Congestion refers to a sustained overload of intermediate network elements. Thus, flow control is one mechanism for congestion control. We study other techniques for congestion control in Chapters 9 and 14.

This chapter is organized as follows. We present a model for flow control in Section 13.1, and a taxonomy of flow control schemes in Section 13.2. We can divide flow control techniques into three broad categories: open loop, closed loop, and hybrid. In Section 13.3, we study open-loop flow control, and in Section 13.4 we study closed-loop flow control. Finally, Section 13.5 discusses hybrid flow control. In each section, we study a handful of representative schemes that cover an illustrative range of flow-control techniques.

13.1 Model

We will study flow control in the context of a single source sending a stream of packets on a connection to a single destination or sink, over a path with many switches or routers. We assume that the sink acknowledges every packet. We model each network element, such as a switch, router, or multiplexing point, as a server that serves a certain number of packets from that connection per second. If the scheduling discipline is rate allocation, the service rate refers to the rate allocated to the connection (see Section 9.5.1). Otherwise, it is the instantaneous service rate available to the connection at that server. We call the slowest server along the path the bottleneck server. Low has shown that, for the purpose of flow control, we can ignore all but the bottleneck server [Low 92]. Thus, the flow control model reduces to the one shown in Figure 13.1.

We can view flow control as rate matching with delays. The bottleneck server removes data from its buffer at a variable rate. The source must match this rate so that the buffer neither overflows nor underflows. The problem is that we know the bottleneck server's current drain rate only after a delay, and the new source rate takes effect only after another delay. The sum of these delays is the round-trip time (RTT), which is the time taken for a packet to traverse the path from the source to the sink and for its acknowledgment to return. (We can exclude the portion of the path from the bottleneck server to the sink if the bottleneck server directly informs the source of its current service rate.) The round-trip time is the fundamental time constant in all feedback flow-control mechanisms, because it is the minimum time required for a source to learn of the effect of its control actions.

Perhaps a more intuitive model of flow control is to imagine a tank of water from which water drains at a variable rate. A source must control its flow so that the tank neither overflows nor empties. If the source's control actions take effect immediately, the problem is straightforward, because the source can simply increase its flow whenever the tank is in danger of emptying, and decrease it when it nears an overflow. However, if the tank is situated, say, a hundred kilometers away, a change in the source's flow rate takes effect only after some time. Moreover, the source knows about the current water level only after a delay. The fundamental time constant in the system is the sum of the times taken for the source's changes to take effect, and for the source to learn of the effect of its change (the round-trip time).

Flow control is a variant of the classical control problem. In classical control, a controller is allowed to change its input to a black box and observe the corresponding output. Its aim is to choose an input as a function of the observed outputs, so that the system state conforms to some

\[ \text{Figure 13.1: A reduced model for flow control.} \]

\[ \text{The source sends data at a rate } \dot{\alpha} \text{. The single server has a rate } \mu \text{ equal to the slowest server along the path, and a buffer as large as the slowest server's buffer. The delay } D \text{ is the sum of the delays on the data and acknowledgment path (this is also called the round-trip time).} \]

---

1We do not study flow control for multicast connections, because this is a difficult open problem that is still an area of active research.
Chapter 13: Flow Control

13.2 Classification

We can classify flow control schemes into open-loop, closed-loop, and hybrid schemes. In open-loop flow control, a source describes its traffic to the network with a few parameters. During call establishment, the network reserves resources (such as bandwidth and buffers) corresponding to these parameters. During data transmission, if the source shapes its traffic to match its traffic's description, network overload, and thus congestion, is avoided. The difficult problem in open-loop flow control is choosing the right set of parameters to describe a source adequately. Once this is done, the actual flow control (or regulation, as it is usually called) is straightforward. We study open-loop flow control in Section 13.3.

In closed-loop schemes, a source dynamically adapts its flow to match its current share of network resources. As this share increases and decreases, a source should send faster or slower. There are many interesting and hard problems associated with closed-loop schemes. For example, how should the network inform a source that its service rate has changed? In an implicit feedback scheme, the source infers a change in its service rate by measuring its current performance. Once it receives this information, the source must decide how best to react to the current system state. This depends on its flow-control strategy. We study closed-loop flow control schemes in Section 13.4.

Finally, hybrid schemes combine aspects of open- and closed-loop flow control. For example, a source may reserve some minimum resources during call setup, but may be given a larger share if the network is idle. Thus, the source must do call setup, as in open-loop flow control, but also adapt to the network state, as in closed-loop flow control. We study such schemes in Section 13.5.

13.3 Open-loop flow control

In open-loop flow control, during call establishment, a source describes its behavior with a set of parameters called its traffic descriptor and negotiates bandwidth and buffer reservations with network elements along the path to its destination. The network operator prescribes the descriptor's parameters, and each source decides parameter values that best describe its traffic. During call setup, each network element examines this description and decides whether it can support the call. If it can, it forwards the setup request to the next element along the path. Otherwise, it negotiates the parameters down to an acceptable value, or blocks the call. In the data transmission phase, the source shapes its traffic to match its descriptor, and each network element schedules traffic from admitted calls to meet the bandwidth, delay, and loss guarantees it makes to them. The hard problems in open-loop flow control are (a) choosing a descriptor at a source, (b) choosing a scheduling discipline at intermediate network elements, and (c) admitting calls so that their performance objectives are met (call admission control). We study the choice of descriptors in Section 13.3.1, scheduling disciplines in Chapter 9, and call admission control in Chapter 14.

In open-loop flow control, a source has to capture its entire future behavior with a handful of parameters, because the network's admission-control algorithm uses these parameters to decide whether to admit the source or not. Thus, open-loop flow control works best when a source can describe its traffic well with a small number of parameters, and when it needs to obtain quality-of-service guarantees from the network. If either of these conditions fails to apply, the source is better off with closed-loop or hybrid flow control.

13.3.1 Traffic Descriptors

A traffic descriptor is a set of parameters that describes the behavior of a data source. Typically, it is a behavior envelope, that is, it describes the worst possible behavior of a source, rather than its exact behavior. A descriptor plays three roles besides describing source traffic. First, it forms the basis of a traffic contract between the source and the network: the source agrees not to violate the descriptor, and in turn, the network promises a particular quality of service. If a source violates its part of the contract, the network cannot guarantee it a performance bound. Second, the descriptor is the input to a regulator, a device through which a source can pass data before it enters the network. To ensure that the source never violates its traffic descriptor, a regulator delays traffic in a buffer when the source rate is higher than expected. Third, the descriptor is also the input to a policer, a device supplied by the network operator that ensures that the source meets its portion of the contract. A policer delays or drops source traffic that violates the descriptor. The regulator and policer are identical in the way they identify descriptor violations: the difference is that a regulator typically delays excess traffic, while a policer typically drops it.

A practical traffic descriptor must have these important properties [Verma 91]:

- **Representativity:** The descriptor must adequately represent the long-term behavior of the traffic, so that the network does not reserve too little or too much.
- **Verifiability:** The network must be able to verify that a source is obeying its promised traffic specification quickly, cheaply, and preferably in hardware.
- **Preservability:** The network may inadvertently modify source traffic behavior as it travels along its path. Thus, the amount of resources allocated to a channel may change along the path. The network must be able either to preserve the traffic characteristics along the path, or to calculate the resource requirements of the modified traffic stream.
Coming up with good traffic descriptors is difficult because of these conflicting requirements. For example, the series of times at which a source places data onto a connection is a representative, verifiable, and preservable traffic descriptor. However, this time series is potentially very long and, for interactive traffic sources, is unknown. Thus, the descriptor is unusable. In contrast, if we choose the source's peak rate as its descriptor, the descriptor is usable, verifiable, and preservable, but not representative, because resource reservation at the peak rate is wasteful if a source rarely generates data at this rate.

Several traffic descriptors have been proposed in the literature. They are roughly equivalent, though there are subtle differences in their ease of use and descriptive power [Rathgeb 91]. We study three common descriptors: peak rate, average rate, and linear bounded arrival process. For each descriptor, we also study the corresponding regulator.

13.3.2 Peak rate

The peak rate is the highest rate at which a source can ever generate data during a call. A trivial bound on the peak rate of a connection is just the speed of the source's access link, because this is the instantaneous peak rate of the source during actual packet transmission. With this definition, a source on a 10-Mbps Ethernet that generates one 100-byte packet per second can be said to have a peak rate of 10 Mbps! Although accurate, this definition is not satisfactory because it does not give a true picture of a source's traffic load. Instead, we measure the peak rate in one of two ways. For networks with fixed-size packets, the peak rate is the inverse of the closest spacing between the starting times of consecutive packets. For variable-sized packets, we must specify the peak rate along with a time window over which we measure this peak rate. Then, the peak rate bounds the total number of packets generated over all windows of the specified size.

Example 13.1
(a) If all packets on a connection are 50 bytes long, and the closest packet spacing is 10 ms, what is the peak rate? (b) If the peak rate of a connection is 8 Mbps over all intervals of 15 ms, what is the largest amount of data that can be generated in 75 ms? (c) In 70 ms?

Solution: (a) The peak rate is 5000 bytes/s. (b) The largest amount allowed in 75 ms is 8 Mbps * 75 ms = 600,000 bits. (c) Since traffic is specified only over an interval of 15 ms, the worst-case amount of data generated in 70 ms is also 600,000 bits (for example, all the data could be generated in the first 5 ms of every consecutive 15-ms interval).

13.3.3 Average rate

The key problem with the peak rate is that it is subject to outliers. The motivation behind average-rate descriptors is that averaging the transmission rate over a period of time reduces the effect of outliers. Researchers have proposed two types of average-rate mechanisms. Both mechanisms use two parameters, $t$ and $a$, defined as follows:

$$ t = \text{time window over which the rate is measured} $$

$$ a = \text{the number of bits that can be sent in a window of time } t $$

In the jumping-window descriptor, a source claims that over consecutive windows of length $t$ seconds, no more than $a$ bits of data will be transmitted. The term "jumping window" refers to the fact that a new time interval starts immediately after the end of the earlier one. The jumping-window descriptor is sensitive to the choice of the starting time of the first window.

In the moving-window scheme, the time window moves continuously, so that the source claims that over all windows of length $t$ seconds, no more than $a$ bits of data will be injected into the network. The moving-window scheme (also called the $(r, T)$ model in reference [Golestani 90]) removes the dependency on the starting time of the first window. It also enforces a tighter bound on spikes in the input traffic.

An average-rate regulator is identical to a variable-packet-size peak-rate regulator, because both restrict the maximum amount of information that can be transmitted in a given interval of time. For a jumping-window descriptor, at time 0, a counter is initialized to 0 and is incremented by the packet size of each departing packet. Every $t$ seconds, the counter is reset to 0. When a packet arrives, the regulator computes whether sending the packet would result in too much data being sent in the current window. This test reduces to testing whether the sum of the current counter value and the current packet
size is larger or smaller than \( a \). Depending on the result, the regulator either forwards the packet immediately or buffers it until the next time window.

In one technique for building a moving-window descriptor, besides the counter described earlier, the regulator stores the departure time and packet size of every departing packet. The test for delaying or forwarding a packet reflects the number of bits sent in the past \( t \) seconds. If the regulator decides to delay a packet, it can determine the earliest time it can transmit the packet by examining the list of packet departure times. This technique is not used in practice because it requires the regulator to store a lot of information, which is hard to do at high speed. A second technique for building a moving-window regulator, which is the one used in practice, is the leaky-bucket regulator described in the next subsection (13.3.4).

**EXAMPLE 13.2**

An average-rate descriptor is specified with \( a = 100 \text{ Kbytes} \), \( t = 1 \text{ s} \). Packet arrival times and sizes are \((0.2 \text{ s}, 20 \text{ Kbytes})\), \((0.25 \text{ s}, 40 \text{ Kbytes})\), \((0.5 \text{ s}, 20 \text{ Kbytes})\), \((0.6 \text{ s}, 20 \text{ Kbytes})\), \((0.8 \text{ s}, 10 \text{ Kbytes})\), \((1.0 \text{ s}, 30 \text{ Kbytes})\), \((1.7 \text{ s}, 30 \text{ Kbytes})\), \((1.9 \text{ s}, 30 \text{ Kbytes})\).

What are the departure times with the jumping-window and moving-window regulators?

**Solution:** With a jumping window, the packet arriving at time 0.8 s is delayed to the second window. With a moving window, the packet arriving at time 0.8 s is delayed to time 1.0. The packet arriving at time 1.0 is delayed until a time \( x \) such that no more than 100 Kbytes have been sent in the interval \([x - t, x]\) i.e., 1.25 s. At this time, the effects of the first two packets are erased and the packet arriving at time 1.0 can be sent. For both regulators, the last two packets depart as soon as they arrive, because the jumping-window counter is not exceeded, and by time 1.7, the moving-window counter is 40 Kbytes.

### 13.3.4 Linear bounded arrival processes

**Linear bounded arrival processes, or LBAPs,** are a popular class of source descriptors [Cruz 87]. An LBAP-constrained source bounds the number of bits it transmits in any interval of length \( t \) by a linear function of \( t \). We characterize this linear function by two parameters, \( \rho \) and \( a \), so that:

\[
\text{Number of bits transmitted in any interval of length } t \leq \rho t + a \tag{13.2}
\]

\( \rho \) corresponds roughly to the long-term average rate *allocated* by the network to the source (which may be substantially larger than the source's true average rate), and \( a \) the longest burst a source may send, given the choice of \( \rho \), while still obeying the LBAP descriptor. In other words, assuming for the moment that \( \rho \) is the source's average rate, an LBAP characterizes a source that has an intrinsic long-term average rate \( \rho \), but can sometimes deviate from this rate, as specified by \( a \). Since an LBAP is a generalization of the average-rate descriptor, it is also insensitive to outliers.

A variant of an LBAP descriptor uses four parameters (and therefore is less general than an LBAP) [FV 90]. The parameters are as follows:

\[
S = \text{The largest possible packet size}
\]

\[
\frac{X_{\text{min}}}{X_{\text{ave}}} = \text{The smallest interval between the starting time of two consecutive packets (the peak rate is defined to be } S/X_{\text{min}} \text{)}
\]

\[
I = \text{The averaging interval}
\]

\[
\frac{X_{\text{ave}}}{X_{\text{ave}}} = \text{The mean interval between two consecutive packets (the average rate over all intervals of length } I \text{ is at most } S/X_{\text{ave}}\text{)}
\]

This characterization has an additional limit on the peak rate, so that the rate at which the source generates a burst is limited. Moreover, unlike the basic LBAP descriptor, it models burstiness at the level of a packet, because it explicitly constrains the largest packet size. The inclusion of the \( X_{\text{min}} - X_{\text{ave}} - I - S \) model in the class of LBAP descriptors can be seen by noticing that the largest number of bits that the source can send in an interval of length \( I \) in the \( X_{\text{min}} - X_{\text{ave}} - I - S \) model is bounded by:

\[
\left( \frac{I + 2}{I} \right) \left( \frac{1}{X_{\text{ave}}} \right) S \tag{13.4}
\]

because \((I + 2)/I\) bounds the number of intervals of length \( I \) that can be spanned by an interval of time \( t \), and in each such interval, the source can transmit at most \( \lfloor X_{\text{ave}}/I \rfloor S \) bits. This is a linear function of \( I \).

A leaky-bucket regulator regulates an LBAP descriptor [Turner 86]. Intuitively, the regulator collects tokens in a bucket, which fills up at a steady drip rate. Each token is permission for the source to send a certain number of bits into the network. When a packet arrives at the regulator, the regulator sends the packet if the bucket has enough tokens. Otherwise, the packet waits either until the bucket has enough tokens or until the packet is discarded. If the bucket is already full of tokens, incoming tokens overflow and are not available to future packets. Thus, at any time, the largest burst a source can send into the network is roughly proportional to the size of the leaky bucket.

More formally, a leaky bucket accumulates fixed-size tokens in a token bucket and transmits a packet only if the sum of the token sizes in the bucket adds up to the packet's size (Figure 13.2). On a packet departure, the regulator removes tokens corresponding to the packet size from the token bucket. The regulator periodically adds tokens to the bucket (at a rate \( \rho \)). However, the bucket overflows if the number of tokens crosses some
threshold, called its depth, $\sigma$. A leaky bucket limits the size of a transmitted burst to a little more than the bucket's depth (since tokens may arrive while the bucket's worth of packets are being transmitted), and over the long term, the rate at which packets depart the regulator is limited by the rate at which tokens are added to the bucket. The regulator delays a packet if it does not have sufficient tokens for transmission. Typically, we initialize the bucket to be full.

**EXAMPLE 13.3**

Tokens of size 100 bytes are added to a leaky-bucket regulator of capacity 500 bytes twice a second. (a) What is the average rate, peak rate, and largest burst size of the regulated traffic stream? (b) Can this regulator handle a packet of size 700 bytes? (c) If a packet of size 400 bytes arrives when the bucket contains tokens worth 200 bytes, and there are no other packets awaiting service, what is the least and most delay it could have before transmission?

**Solution:** (a) The average rate is $200$ bytes/s = 1.6 Kbps. The largest burst size is 500 bytes. The peak rate is unbounded, because a burst of up to 500 bytes can be transmitted arbitrarily fast. (b) No, because the packet will never have enough tokens to be transmitted. (c) If the packet arrives just before the arrival of a token, it need wait for only a little over 0.5 s; if it arrives just after the token, it has to wait for 1 s.

A leaky bucket can be used both as a peak-rate and a moving-window average rate regulator, because they are both special cases of an LBAP. If the token replenishment interval corresponds to the peak rate, and the token bucket size is set to one token, then the leaky bucket is a peak-rate regulator. Similarly, setting the token-bucket limit to one token and replenishing the bucket at the average rate makes it a moving-window average rate regulator. (In both cases, with variable-sized packets, we have to be careful that the token is at least as large as the largest packet.)

In a common variant of the leaky-bucket regulator, the token bucket is augmented with a peak-rate regulator, so that packet bursts arrive into the network no faster than this peak rate. This allows us to control the average rate, the peak rate, and the largest burst from a source.

Note that a leaky bucket regulator has both a token bucket and a data buffer (if it did not have a data buffer, it would be a policer). Packets that arrive to the regulator that cannot be sent immediately are delayed in the data buffer. Intuitively, the larger the token bucket, the smaller the data buffer need be, because an arriving packet uses a token and depletes the regulator, instead of being delayed in the data buffer. In fact, Berger and Whitt have shown that the performance of a leaky bucket with a data buffer and a token bucket (in terms of the packet loss rate from the regulator) depends only on the sum of the token-bucket size and the data-buffer size [BW 92]. In other words, confirming our intuition, a larger token bucket size exactly offsets a smaller data buffer.

**Choosing LBAP parameters**

Given an intrinsically limited traffic source, such as a stored compressed video, we would like to come up with an LBAP descriptor for the source that is minimal in the sense that no other descriptor has both a smaller $\sigma$ and a smaller $p$. If a source must pay for the resources it consumes, a minimal descriptor is likely to have the smallest price. Unfortunately, the minimal LBAP descriptor for a source is not unique. Given the size of the data buffer at the regulator and the maximum loss allowed at the regulator, each choice of the token arrival rate has a corresponding minimum burst size so that the loss parameter is met. To see this, consider a source that has some intrinsic peak rate $P$ and average rate $A$ measured over a long interval. If the token arrival rate $\rho$ is less than or equal to $A$, then the regulator buffer grows without bound, and $\sigma$ must be infinite if we want to avoid packet loss. If $\rho > P$ then there are always tokens on hand when a packet arrives, and $\sigma$ can be as small as one maximal-sized packet (Figure 13.3). As we increase $\rho$ in the range $[A, P]$, the minimum $\sigma$ needed to meet the loss bound decreases. Any $\rho$ and its corresponding $\sigma$ is an equivalent minimal descriptor of the source. The set of all $(\sigma, \rho)$ pairs that form the minimal LBAP descriptors for a source are described by the $(\sigma, \rho)$ curve for the source.
describes it "best." However, for many common sources, the "knee," which makes the choice of parameters straightforward. A knee in the indicates that for descriptors that are slightly away from the knee, either the parameter rapidly increases. Thus, the optimal choice of the parameters is the value at the knee (Figure 13.3). LBAP descriptors are popular in practice and also in academic papers, because they correctly model the fact that even a "smooth" source may have periods in which it is bursty. However, they do not accurately represent sources that have occasional very large bursts. For such sources, the regulator delay is to be small, and the loss probability low, the chosen to be fairly large so that the burst is drained away into the network. Unfortunately, this makes the network more expensive, because it has to size internal buffers to be large enough to handle large bursts. A better solution is to renegotiate the LBAP descriptor just before the burst (if the occurrence of a burst can be predicted, or is already known, as with stored video), so that we increase the drain rate to handle the burst, then decrease it back to the long-term average rate [GKT 95]. If a burst lasts long enough, we can start renegotiation after detecting the start of the burst. However, this does not perform as well, because the regulator's data buffer fills while the renegotiation procedure, which can take more than one round-trip time, is going on. Dynamic renegotiation of traffic descriptors is still an area of active research (see Section 14.7).

### 13.4 Closed-loop flow control

In open-loop flow control, a source specifies a traffic descriptor during call establishment, and, during data transfer, ensures that its traffic meets this description. Even if the network load changes while the call is in progress, an admitted source need not change its descriptor or its traffic, because each network element reserves sufficient resources to meet the source's performance requirements.

In closed-loop flow control, we assume that network elements do not reserve sufficient resources for the call, either because they do not support resource reservation, or because they overbook resources to get additional statistical multiplexing gain. In Chapter 9, we studied how a network element can use a scheduling discipline to dynamically allocate transmission rates to an ensemble of feedback flow-controlled sources. Given such an allocation, the aim of closed-loop flow control is to adjust a source's transmission rate dynamically, in response to feedback signals, so that the ensemble of sources does not overload the network. If closed-loop flow control is ineffective, sources either suffer excessive packet loss or underutilize network resources.

#### Taxonomy of closed-loop flow-control schemes

In general, a traffic source must control its transmission rate not only in response to the receiver's state, but also in response to the network state. The first generation of flow-control protocols did not explicitly consider network state; they simply matched the source rate to the service rate at the destination (Table 13.1). The three important protocols in this generation, which we will study in Sections 13.4.1–13.4.3, are on-off, stop-and-wait, and static-window flow control.

The second generation of protocols changes the source rate in response to both the sink state and the network state. We can categorize these protocols in three complementary ways:

- **Implicit versus explicit state measurement**: With explicit measurement, a network element uses an explicit control message to communicate the current sustainable data rate to every source. In an implicit measurement scheme, a source uses performance measurements to dynamically infer its share of network bandwidth. Explicit schemes can control a source's rate more precisely than implicit schemes, because a source has better information. On the other hand, they require both a communication and a computation overhead. Ideally, we would like a scheme that has as little overhead as an implicit scheme, but performs nearly as well as an explicit scheme (also see the discussion in Section 6.3.11).

<table>
<thead>
<tr>
<th>On-off (13.4.1)</th>
<th>Stop-and-wait (13.4.2)</th>
<th>Static window (13.4.3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit</td>
<td>Dynamic window</td>
<td>End-to-end</td>
</tr>
<tr>
<td>Implicit</td>
<td>Dynamic rate</td>
<td>Hop-by-hop</td>
</tr>
</tbody>
</table>

Table 13.1: Taxonomy of closed-loop flow-control schemes.
• **Dynamic window versus dynamic rate:** We define the error-control window to be the number of packets sent from a source, but yet to be acknowledged. Because the source must stop after it has a window’s worth of packets in flight (see Section 12.4.7 for more details), by limiting the error-control window size, we automatically limit the source’s transmission rate. Thus, we can use the error-control window for flow control. To distinguish between these distinct uses of a window, we will call the window used for flow control the **flow-control window** or the transmission window. In an adaptive-window scheme, we indirectly control a source’s transmission rate by modifying its transmission window. In an adaptive-rate scheme, we directly control the source rate. Every time a source sends a packet, it sets a timer with a timeout value equal to the inverse of the current transmission rate, and transmits the next packet when the timer expires.

A dynamic-window scheme has the disadvantage that the window is used for both error control and rate control. This coupling is often problematic. For example, if a receiver has only a few buffers to hold out-of-order packets, and error control is based on selective retransmission, the error-control window must be small. Unfortunately, this limits the maximum transmission rate from the source. We comment on some other disadvantages of window-based flow control at the end of Section 13.4.5, after we have more context for these comments.

Window-based control has two main advantages over rate-based flow control. First, it is easier to implement, because it does not require a fine-grained timer, which can be expensive in some systems. Second, a window automatically limits the damage a source can inflict on a network. After transmitting a window’s worth of packets, a source stops. With rate-based flow control, a source may continue to send packets into the network if it fails to receive rate-throttling information. Thus, we must carefully engineer rate-based flow control to be robust to packet loss and corruption.

• **Hop-by-hop versus end-to-end control:** We can automatically make a first-generation flow-control scheme responsive to network state (in addition to receiver state) by implementing it between every adjacent pair of network elements. For example, stop-and-wait becomes a second-generation flow-control scheme if we use it not just between a source and a sink, but also between every pair of adjacent routers. This change typically improves performance: the control delay is smaller, and each element only needs to react to a change in the next element, which is typically more effective than responding to changes in all elements along the path. Besides, by limiting the buffer buildup at each element, a hop-by-hop scheme more evenly distributes buffer usage. Hop-by-hop schemes, however, make the network elements more complicated. Moreover, unless hop-by-hop control is done per-connection, it can be unfair [MK 92].

Because these design elements are complementary, we can come up with eight possible combinations. Note that there are no implicit hop-by-hop closed-loop flow control schemes. An implicit scheme tries to minimize the work done within the network by guessing the available service rate in the network. A hop-by-hop scheme requires a considerable amount of work to be done at each switch, so the additional overhead for explicit control is small. Thus, hop-by-hop schemes tend to use explicit control.

In Sections 13.4.4–13.4.11 we will study representatives of each combination as shown in Table 13.2. Table 13.5 in Section 13.4.12 is a summary of these schemes and a road map.

Of these schemes, we will study the DECbit, TCP, and ATM Forum EERC schemes in more detail than the others for two reasons. First, they are widely implemented on a variety of platforms. Second, the ideas in these schemes form the basis of many other schemes that are in the literature, but are not covered here.

### 3.4.1 On–Off

In an on–off flow control, the receiver sends the transmitter an **On** signal when it can receive data, and an **Off** signal when it can accept no more data. The transmitter sends as fast as it can when it is in the On state, and is idle when it is in the Off state.

**Evaluation**

On–off control is effective when the delay between the receiver and the sender is small. It works poorly when the propagation delay between the sender and receiver is large, because the receiver needs to buffer all the data that arrive before the Off signal takes effect. If the Off packet is delayed or lost, the receiver continues to receive data at the source’s peak rate, leading to potential buffer overflow and loss. Moreover, intermediate network elements are subjected to abrupt data bursts from the sources, making packet loss in the network more likely.

On–off control is primarily used over short serial lines or LANs, where propagation delays are small and packet losses are rare. It is the basis for the XON/XOFF protocol used to control serial input–output devices such as printers and mice.

<table>
<thead>
<tr>
<th>Table 13.2: Representative closed-loop flow-control schemes.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheme</td>
</tr>
<tr>
<td>DECbit (13.4.4)</td>
</tr>
<tr>
<td>TCP-Tahoe and TCP-Reno (13.4.5)</td>
</tr>
<tr>
<td>Credit-based (13.4.11)</td>
</tr>
<tr>
<td>ATM Forum EERC (13.4.9)</td>
</tr>
</tbody>
</table>
13.4.2 Stop-and-wait

In the stop-and-wait protocol, one of the earliest attempts at flow control, a source sends a single packet and waits for an acknowledgment before sending the next packet. If it receives no acknowledgment for some time, it times out and retransmits the packet.

Stop-and-wait simultaneously provides error control and flow control. It provides error control because if a packet is lost, the source repeatedly retransmits it until the receiver acknowledges it. It provides flow control because the sender waits for an acknowledgment before sending a packet. Thus, stop-and-wait forces the sender to slow down to a rate slower than can be supported at the receiver.

Evaluation

Stop-and-wait is useful in networks where the propagation delay is small, but is inefficient otherwise (see Figure 13.4). In the figure, each vertical line corresponds to a network element. Time increases along the vertical axis. We represent a packet transmission as a parallelogram: the slope of the parallelogram represents the link delay, and the width is inversely proportional to the transmission rate of the link (thus, the width is also proportional to the time a packet spends on a link). It is clear from the figure that a source must wait for one round-trip time to elapse after it sends a packet before it can send the next one. Thus, the best possible throughput is one packet per round-trip time. This decreases rapidly as the propagation delay, relative to the packet transmission time, increases.

Example 13.4

What is the peak throughput achievable by a source employing stop-and-wait flow control when the maximum packet size is 1000 bytes, and the network spans (a) 10 km, (b) 5000 km?

Solution: Assuming a direct fiber-optic line between endpoints, the speed-of-light propagation delay is \( \frac{1}{(0.7 \times 3 \times 10^8) \text{ s/km}} \), since the speed of light in fiber is approximately \( 0.7c \), where \( c \) (the speed of light in vacuum) \( = 3 \times 10^8 \text{ km/s} \). This works out to \( 4.76 \mu \text{s/km} \). For the purposes of this example, we will ignore the effects of queuing and switching delays. (a) For a 10 km line, the round-trip delay is thus \( 2 \times 47.6 \mu \text{s} = 95.2 \mu \text{s} \). The maximum possible throughput is thus \( \frac{1 \text{ packet/RTT}}{1000 \times 8 \text{ bits/95.2 \mu s}} = 84.03 \text{ Mbps} \). (b) Since the link is 500 times longer, the maximum speed goes down by a factor of 500 to \( \frac{84.03}{500} \text{ Mbps} = 0.168 \text{ Mbps} \).

13.4.3 Static window

Stop-and-wait flow control has a peak throughput of one packet per round-trip time (RTT). This is inefficient if the propagation delay, and thus the RTT, is large. In static-window flow control, we allow a source to send up to \( w \) packets before it stops and waits for an acknowledgment. In other words, we have:

\[
\text{Transmission window} = w \tag{13.5}
\]

When a source that has \( w \) packets outstanding receives an ack, we allow it to transmit another packet, because the left edge of the sliding window slides one step forward. It takes at least one round-trip time for the source to receive an ack for a packet. In this time, it could have sent at most \( w \) packets. Thus, its maximum achievable throughput is \( \frac{w}{\text{RTT packets/s}}, \) a factor of \( w \) better than stop-and-wait. Figure 13.5 illustrates a source allowed to have three packets outstanding at any time, i.e., has a window of size three. Thus, its throughput is almost triple that of a corresponding stop-and-wait source.

How large should \( w \) be? Let:

\[
\text{Bottleneck service rate along the path} = \mu \text{ packets/s} \tag{13.6}
\]

\[
\text{Round-trip time} = R \text{ s}
\]

The source's sending rate is at most \( \frac{w}{R} \) packets/second. If this is to keep the bottleneck fully utilized, we must have:

\[
\frac{w}{R} \geq \mu \Rightarrow w \geq R \mu \tag{13.7}
\]
The problem with static-window flow control is that the optimal window size depends on the bottleneck service rate and the round-trip time. However, not only do the bottleneck rate and the round-trip time differ from connection to connection, but they also vary with time even for the same connection. Choosing a single static window size suitable for all connections is therefore impossible. Unfortunately, a poor choice of \( w \) can lead to severe performance problems.

If \( w \) is too small, a source may have data to send, and the bottleneck may have spare capacity, but the window size prevents the source from using the available capacity. If \( w \) is too large, from Equation 13.8, \( w = \frac{R}{L} \) packets are buffered at the bottleneck; if the bottleneck does not have sufficient buffering, packets may be lost. We avoid these problems in the second generation of flow-control protocols by dynamically varying a connection's window size to be always close to the current optimal.

### Example 13.5

Compute the optimal window size when packet size is 53 bytes, the RTT is 60 ms, and bottleneck bandwidth is (a) 1.5 Mbps (the standard T1 trunk speed), (b) 155 Mbps (the standard OC-3 trunk speed).

**Solution:** (a) The bottleneck rate in packets/sec is 1.5 Mbps/53 * 8 bits/packet = 3337.7 pkts/s. Thus, the optimal window is 3337.7 * 0.06 = 212.3 packets.

(b) Similarly, at OC3 rates, the bottleneck rate is 365,566 pkts/s, and the optimal window is 21,933 packets.

### DECbit flow control

The key idea behind the DECbit scheme [RJ88, RJ90] is that every packet header carries a bit that can be set by an intermediate network element that is experiencing congestion (i.e., a sustained queue buildup). The receiver copies the bit from a data packet to its acknowledgment, and sends the acknowledgment back to the source (Figure 13.6). The source modifies its transmission-window size based on the series of bits it receives in the
acknowledgment headers as follows: The source increases its windows until it starts building queues at the bottleneck server (because its window size is larger than the optimal window size), causing that server to set bits on the source's packets. When this happens, the source reduces its window size, and bits are no longer set. The propagation delay \( \rho \) or the bottleneck service rate \( \mu \) changes, the source-window size adapts to this change and oscillates about a new optimal point. Note that the scheme does not require any particular scheduling discipline at multiplexing points. In our discussion, we assume, as the authors did, that this is first-come-first-served. We now study the scheme in more detail.

The important elements in the DECbit scheme are (a) when bits get set, and for which connections, and (b) how these bits are interpreted by the source. We examine these two issues in turn.

Router actions

In the DECbit scheme, each network element monitors packet arrivals from each source to compute its bandwidth demand (a measure of the bandwidth the source is using) and the mean aggregate queue length (the mean length of the queue shared by all the sources). Monitoring these values may be expensive in practice, in which case a source may choose to use the nonselective DECbit scheme described at the end of this subsection. The network element computes source demands and queue lengths as averages over queue regeneration cycles (Figure 13.7). A queue regeneration cycle has a busy and an idle component. The busy period starts when the queue size goes from zero to one (we count the packet being transmitted as part of the queue) and ends when the queue size goes back to zero. Thus, during the busy period, the queue always has data to send. The idle period spans the time when the queue is idle; it ends when the next busy period starts. A source's demand is the mean number of packets it transmits during the current (partial) and previous regeneration cycles. The network element similarly computes the aggregate mean queue length over the current and previous regeneration cycles.

Measuring source demands and the mean queue length over regeneration cycles may seem, at first, to introduce unnecessary complexity. Notice, however, that the two metrics, by definition, must be computed over some time interval. The choice, then, is only of how long this interval should be. The measurement interval should represent a balance between sensitivity to the current system state (which argues for short measurement intervals) and stability in the measurement, to prevent the control mechanism from reacting to a transient burst of packet arrivals (which argues for longer measurement intervals). Experiments show that averages computed over regeneration cycles provide a good balance between the two objectives [RJ 90].

Although a server could compute averages over the immediately past complete regeneration cycle, and use these to set bits until the end of the current cycle, this does not work well if the current regeneration cycle is long and measures during the past cycle are outdated. The source avoids this by measuring demands and queue lengths over the immediately past and current (partial) regeneration cycle, so that as the current regeneration period increases in length, its contribution to the system performance measures becomes increasingly important.

A server takes control actions if the measured mean queue length crosses one of two thresholds. The first threshold is set to 1.0. If the mean queue length exceeds this value, the server always has at least one packet in service, so that it is 100% utilized, and in danger of being swamped by the sources. Thus, it sets bits on packets from sources whose demand is larger than their max-min fair share, as defined in Section 9.2.2. This causes the server to reduce their window size, and thus their rate, relieving the load on the server. The second threshold is set to 2.0. If the mean queue length exceeds this value, not only is the server 100% utilized, but its efforts at setting bits have not decreased the queue size. The server therefore goes into panic mode and sets bits on all packets. The idea is that the server should rarely enter this state, and if it does, it should get out of it as soon as possible, irrespective of fairness.

Source actions

A source keeps track of the bits it receives in the ack headers and uses this to dynamically adapt its flow control window. Notice that once a source changes its window size, it takes one RTT before this change propagates to every server along the path. For the control to be stable, the source should measure the network state for an additional RTT before it changes its window size again. Thus, a source should change its window once every two RTTs. Specifically, if, after changing the window size, the previous window size is \( p \) and the current window size is \( w \), the source should wait till it receives \( w+p \) acknowledgments, then examine the bits in the last \( w \) acks. It is recommended that a source reduce its window size if more than 50% of the bits are \( 1 \), and otherwise increase its window size. The control mechanism, however, is insensitive to this choice of threshold [RJ 90].

The window change policy is additive increase multiplicative decrease (AIMD). When the window increases, it increases by an additive factor, but when it decreases, it decreases by a multiplicative factor.
Nonselective DECbit

In the scheme we have studied thus far, the router must compute a demand per source, which is expensive in terms of both storage and computation. In an alternative scheme, called nonselective DECbit, the router measures only the mean aggregate queue length, which does not need per-source storage and therefore is less expensive. In this scheme, if the mean queue length exceeds 1.0, then the server sets bits on all packets. This reduces the aggregate queue length rapidly, but can be unfair to sources that have their bits set although their demand is less than their fair share. It is also unfair to connections that have longer RTTs, because they take longer to recover from a rapid window decrease. Source actions for the selective DECbit and nonselective DECbit scheme are identical.

Evaluation

The DECbit scheme has several useful properties. It requires only one additional bit in the packet header and does not require per-connection queuing at servers. Endpoints can implement the scheme in software, without additional hardware support. Moreover, many simulations, and experience in the field, have shown that the control is stable [RJ 90]. Figure 13.8 shows the behavior of the scheme in a simple scenario (from [RJ 90]).

However, the DECbit scheme has two serious flaws. First, it assumes that the endpoints are cooperative. If a source chooses to ignore the bits it receives in its acks, it can drive the server to the panic state, so that all other sources sharing the server are affected [DKS 89]. Thus, a single malicious or misbehaving source can affect the performance of all other sources. As we saw in Chapter 9, to control misbehaving or malicious sources, a network must provide either per-connection queuing, or per-connection policing using a traffic descriptor.

The second problem with DECbit is that it has a very conservative window increase policy. If the initial window size is 5 and the optimal window size is 200, the source will take 390 RTTs to reach this value, because the window increases by only 1 every two RTTs. If a source does not have much data to send, it finishes its transmission before reaching its optimal window size! Thus, DECbit performs poorly in networks where the bandwidth-delay product is large, which is expected to be the case for future wide-area networks.

EXAMPLE 13.6

If the RTT is 60 ms, which corresponds to a cross-continental path, the service rate available to a source is 5 Mbps, and its mean packet size is 500 bytes, its optimal window is $5 \times 60 \times 3 = 300,000$ bits = 300,000/4000 = 75 packets. Thus, the source takes 150 RTTs = $150 \times 60 \times 3 = 9$ s to reach its optimal window size. If its transfer size is smaller than about 5 Mbytes, it would complete its transmission before reaching this value.

13.4.5 TCP flow control

The flow-control scheme in TCP, designed by Jacobson and Karels, is similar to the DECbit scheme, but differs in one important detail [Jacobson 88]. Instead of receiving explicit congestion information from network elements, a source dynamically adjusts its flow control window in response to implicit signals of network overload. Specifically, a source increases its window size until it detects a packet loss. At this point, the source reduces the window size, and the cycle repeats (Figure 13.9).

A source starts with a window of size 1 and increases it exponentially until the window size reaches a threshold, and linearly after that. We maintain the current window...
size in a floating-point number, the integral part of which is the window size (this is explained further in Example 13.7). In the exponential (or slow-start) phase, a source increases its window by 1 every time it receives an ack. If every packet is acked, this doubles the window size every RTT (in practice, some receivers send only one ack for every two packets, halving the increase rate). In the linear (or congestion-avoidance) phase, the source increases its window by \( \frac{1}{\text{current window}} \) every time it receives an ack. This increases the window by \( \frac{1}{1} \) every RTT. A threshold variable called the slow-start threshold \( \text{sssthresh} \) controls the transition from the exponential to the linear phase. The source initializes this variable to half the initial window size, and, on detecting a loss, resets it to half the current window size.

There are two widely used variants of TCP. The Tahoe version detects losses using timeouts. On a timeout, it decreases its flow control window to 1, sets \( \text{sssthresh} \) to half the current window size, and enters slow start. The window thus rises exponentially to half its previous value, then continues with a linear increase. The Reno version detects losses using both timeouts and the receipt of three acks with the same cumulative sequence number (the fast retransmit scheme explained in Section 12.4.7). On a timeout, a TCP-Reno source behaves in the same way as a TCP Tahoe source. However, on a fast retransmit, it decreases both \( \text{sssthresh} \) and its flow-control window to half its previous value. At the point, since the flow-control window is already as large as \( \text{sssthresh} \), the source goes directly into the linear increase phase, skipping the exponential increase phase. Moreover, after a fast retransmit, the source is allowed to send one packet for each duplicate cumulative ack received, even if this causes it to exceed the flow-control window size (this is called fast recovery). The intuition is that each cumulative ack, even if duplicate, signals the availability of network resources. By "inflating" the actual window beyond the nominal window, a source can exploit this capacity. This window inflation ceases when a source receives the first nonduplicate cumulative ack, indicating that the retransmission succeeded.

**EXAMPLE 13.7**

Let us trace the evolution of the window size of a source that is transmitting data over a connection with RTT 1. Assume that its initial value for \( \text{sssthresh} \) is 5, and the largest allowed flow control window is 10. We will also assume that the bandwidth-delay product on the connection is 4 (that is, the bottleneck service rate is 4 packets/second), and the bottleneck has a buffer size of 4 packets.

In the first RTT, the source sends packet 1 at time 0 and receives an ack at time 1. At the end of the first RTT, the source increases its window by 1 for this ack, doubling its window to 2. Thus, at time 1, it sends packets 2 and 3. As each ack arrives, the window increases by 1, so that at the end of the second RTT, the window is 4.

In the third RTT, the source sends packet 4, 5, 6, 7. At the end of the third RTT, when the ack for 4 arrives, the window increases to 5 and reaches the slow-start threshold. Thus, theacks for 5, 6, 7, which arrive during the fourth RTT, each contribute \( \frac{1}{2} \) to the window size, and when the ack for 7 is received, the window reaches 5.6 (only the integer portion is used for flow control).

In the fourth RTT, five packets can be outstanding, and the source transmits 8, 9, 10, 11, 12. When the source receives the ack for 9, during the fifth RTT, the window finally increases to 6, and acks for 10, 11, and 12 each contribute \( \frac{1}{2} \) to the window. Thus, at the end of the fifth RTT, the window reaches 6.5. In the absence of losses, the window increases slowly until it reaches nine packets during the eighth RTT. Of these, four are "in the pipeline," because the bandwidth delay product is 4. Four more can be buffered in the bottleneck buffer. Thus, one packet is lost at the bottleneck (the packet with sequence number 42). This causes the receiver to repeat the cumulative acknowledgment on packets sent during the ninth RTT, triggering a fast retransmission.

At the start of the ninth RTT, the source has nine packets outstanding and has a window size of 9.333. During the ninth RTT, the source receives acks for packets 34–41, which increases its window, at the end of the ninth RTT, to 10.2. Thus, the source, during the 9th RTT, sends ten packets, 43–52. The last ack received during the ninth RTT is the ack for 41, which was the last packet sent before a packet loss.

In the tenth RTT, the source receives acks for packets sent in the ninth RTT. The acks for packets 43–52 all carry the same cumulative ack number, that is, 41 (these acks increase the window from 10.2 to 10.5). On the third such duplicate, the source invokes fast retransmission (see Section 12.4.7) and retransmits 42. (Meanwhile, when it got the acks for 42 and 44, it transmitted 53 and 54.) In both TCP Tahoe and TCP-Reno, fast retransmission causes the source to set its \( \text{sssthresh} \) value to \( \frac{10.5}{2} = 5.25 \) and enters the linear increase phase. Thus, in the tenth RTT, a TCP-Reno source has five packets outstanding: 42 and 53–56. In TCP-Tahoe, the window drops to 1. It cannot send any more packets after 42. Thus, in the tenth RTT, a TCP-Tahoe source has three packets outstanding: 53, 54, and 42. In two round-trip times, the window comes back to 4, and the source reenters the linear increase phase.

---

**Figure 13.9:** Behavior of three TCP-Tahoe sources. Three sources start simultaneously, and the figure shows the evolution of their window sizes. Each source increases its window linearly in the congestion-avoidance stage. When the queue fills up, all three sources see one loss, and all of them drop their window size to 1. The window increases first exponentially, then linearly, till each source loses one packet each, and the cycle repeats. (From [ESC91].)
of their popularity, there is considerable experience in understanding their performance.

TCP flow-control algorithms are effective over a wide range of bandwidths, and because of their popularity, there is considerable experience in understanding their performance. TCP-Tahoe and TCP-Reno algorithms have motivations similar to the DECbit algorithm we studied in Section 13.4.4. Both use dynamic-window flow control. In either case, a source uses a conservative window-increase policy (exponential and then additive in TCP, additive in DECbit), and a multiplicative decrease policy. We know that an additive-increase, multiplicative-decrease policy is stable, unlike variations such as additive increase, additive decrease, or multiplicative increase, additive decrease, so this coincidence is not surprising [IC 89]. The main difference between the algorithms is that the TCP-Tahoe and -Reno algorithms do not require explicit information about the current congestion status from the network. They view the network as a black box, which they probe by increasing the transmission-window size and looking for packet loss. In contrast, with DECbit, the network explicitly communicates information about queue builds up to the sources. A second difference is that the two TCP algorithms do not filter information at the source, for example, by treating multiple packet losses over a time interval as an indication of congestion, instead of a single packet loss. This is because they operate the network close to overload, and unless they cut back the window size immediately after detecting a loss, the network is in danger of sustained overload.

Evaluation

TCP flow-control algorithms are effective over a wide range of bandwidths, and because of their popularity, there is considerable experience in understanding their performance. [DKS 89, ZSC 90, ZSC 91, Floyd 91, Mogul 93, Stevens 94, LM 95]. Moreover, they are about the best that one can do when the network does FIFO queuing (the common case), and an endpoint cannot make any assumptions about the structure of the network or its operation (for a slightly better scheme, see Section 13.4.6).

However, the algorithms have several weaknesses. First, they assume that any loss is an indication of an overload, and thus immediately reduce the window size. This is because most packet networks currently are nearly loss free, except for congestive losses. Wireless networks, however, have a different loss behavior, and multipath interference or shadowing effects can lead to frequent link-level errors. In such situations, assuming random loss, it can be shown the effective throughput of a TCP connection decreases in proportion to \( p(R_0)^2 \), where \( p \) is the probability of a random loss, and \( R_0 \) is the bandwidth-delay product expressed in packets [LM 95]. For a cross-country link where the first hop is over a wireless link, typical window sizes are around 15 packets (assuming a packet size of 500 bytes, a round-trip delay of 60 ms, and an available bandwidth of 1 Mbps). Thus, on this connection, the throughput degrades as 1/225p, where \( p \) is the probability of a random loss on the wireless link. As the available bandwidth increases, \( p \) increases linearly, but the degradation in throughput goes up as \( p^2 \), so the effective throughput decreases.

Many solutions have been proposed for this problem. One approach is to use link-level error control to make the link appear error free. The problem is that the link-level and TCP-level error control schemes must coordinate their actions, to avoid, for example, retransmission at both the link level and the TCP level. However, TCP is unaware of the link's actions, leading to many subtle problems. A second approach is to modify TCP's flow-control strategy to allow it to distinguish between congestive losses and link losses. The datalink layer informs TCP about link-level losses, which TCP retransmits without shutting down the flow-control window. This, too, is problematic because it is a layering violation. Moreover, it only works if the wireless hop is the first or last hop in the connection. The question of how best to modify TCP to deal well with noncongestive losses is still an area of active research.

A second problem with TCP is that a TCP source always assumes that a packet loss is due to its own large window size, and thus cuts back on its window. When used in networks with FCFS queuing, this leads to situations where the presence of a single malicious or misbehaved source causes all the TCP connections to back off immediately, giving a free hand to the misbehaving source. Using a round-robin-like algorithm at the queuing points can solve this problem [DKS 89], though only at the expense of per-connection queuing at multiplexing points.

Third, the algorithm detects overload using loss. Thus, each endpoint tries to increase its window until some buffer overflows (Figure 13.9). Even in the steady state, bottleneck buffers are kept full, so that the arrival of a burst of traffic is very likely to cause packet loss. Each lost packet represents wasted work, so using loss to measure the state of the network is inherently wasteful.

Fourth, for short transfers (which constitute the bulk of connections on the Internet), the algorithm is sensitive to the initial values of parameters such as ssthresh (the slow-
422 Chapter 13: Flow Control

13.4.6 TCP-Vegas

Recent research has addressed this issue by coming up with heuristics to estimate these parameters from the first few packets sent on a connection [Hoe 96].

Finally, a TCP endpoint underutilizes the bottleneck link if the bottleneck has so few buffers that a source loses packets when it is still in its exponential increase phase [LM 95]. This corresponds to roughly $R_u/3$ packets’ worth of buffering per connection at the bottleneck. Thus, if TCP is to work well, a network designer must budget for a minimum amount of buffering at each multiplexing point, increasing the cost of the network.

13.4.7 NETBLT

Because of the widespread use of TCP on the Internet, considerable effort has gone into improving it. One of the more successful attempts is TCP-Vegas [BP 95]. TCP-Vegas not only improves TCP's retransmission mechanism, by using a better fast-retransmission heuristic, but also improves TCP's congestion-avoidance strategy. Recall that in TCP-Reno and TCP-Tahoe, the congestion-avoidance mechanism is to increase the transmission-window size over time, decreasing it if there is a loss. The main idea in Vegas is that a source can compute the expected throughput of a connection as:

\[
\text{Expected throughput} = \frac{\text{Transmission window size}}{\text{Propagation delay}}
\]

The denominator is estimated by computing the smallest RTT seen so far, which is likely to be close to the true propagation delay. The numerator is easily available to the algorithm. Thus, over any period, TCP-Vegas can compute the expected throughput along the path. The source can also measure the actual throughput by measuring how many acks it receives in a round-trip time. If the actual throughput over the same period is less than the expected throughput, then the source adjusts its window size, and thus its transmission rate.

In practice, the source sends a distinguished packet and records the transmission time of this packet. When the source receives the ack for the packet, it computes the expected and actual throughput (call them $e$ and $a$) as described in the previous paragraph. If $e < a$, this means that the \(\text{Propagation delay} \) value is too large, and it adjusts this value. If $e = a$, then $(e-a)RTT$ packets transmitted in the previous RTT are still in the bottleneck buffer. The source does not change its window if this number lies in a range $\alpha$ to $\beta$, where these are user-selected low and high watermarks. Otherwise, the source adjusts the transmission window so that in the next RTT the expected number of packets in the buffer reaches this value.

Evaluation

TCP-Vegas has been simulated and tested in the field [BP 95, ADLY 95]. Current experience shows that it performs better than TCP-Tahoe or -Reno. A serious concern is whether TCP-Vegas connections sharing a link with TCP-Reno connections gain better performance at the expense of TCP-Reno. Small-scale simulations and experiments indicate that this is probably not the case. Given the popularity of TCP, incremental improvements in its performance are worth the effort. TCP-Vegas, though not an ideal flow-control scheme, is a step in the right direction.
source multiplicatively reduces its rate to $\beta r$, where $\beta < 1$. On the other hand, if $r^* = r$, the source additively increases its rate to $r + \alpha$.

**Evaluation**

NETBLT's main contribution to flow control is in its separation of flow and error control. The original design ignored bottlenecks in the network, and the revised algorithm adapts only slowly to changing capacity in the bottleneck, because a source takes control decisions only once for each buffer's worth of packets. Thus, the algorithm tends to either overflow or underflow the bottleneck buffer. Nevertheless, rate-based flow control, an idea first embodied in NETBLT, is a rich and interesting area for research. When carefully implemented, rate-based flow control has the potential to provide a complete solution to the flow-control problem.

### 13.4.8 Packet-pair

Packet-pair flow control improves on NETBLT's performance by better estimating the bottleneck capacity in the network [Keshav 91, Keshav 97]. Moreover, it predicts the future service rate in the network and corrects for incorrect past predictions. These allow it to maintain a certain number of packets in the bottleneck queue precisely (its setpoint) despite variations in the bottleneck service rate.

Unlike other flow-control schemes, packet-pair explicitly assumes that all the bottlenecks in the networks serve packets in round-robin order. Thus, when two packets belonging to the same connection enter a server back-to-back, an interval that is inversely proportional to the connection's service rate at the bottleneck separates them when they leave (Figure 13.10). Note that the separation is largest at the bottleneck server. Thus, if the receiver measures the packet separation (or if the source measures the acknowledgment separation), the receiver (or source) can directly determine the bottleneck service rate. A packet-pair source sends all packets as pairs (except if the source only has a single packet to send). Thus, a source can update its estimate of the bottleneck service rate with every pair of acks it receives. If the bottleneck service rate changes with time, the source automatically detects and adapts to the change.

Assume, for the moment, that all packets are the same size. Let $\mu(k)$ be the bottleneck service rate detected by the kth ack pair. Packet-pair does exponential averaging of the time series $\mu_1, \mu_2, \ldots, \mu_k$ to predict $\hat{\mu}(k+1)$ (see a description in Section 12.4.6), so that:

$$\hat{\mu}(k+1) = \alpha \cdot \hat{\mu}(k) + (1 - \alpha) \cdot \mu(k), \quad 0 < \alpha < 1$$  \hspace{1cm} (13.11)

The averaging factor, $\alpha$, is altered dynamically to simultaneously eliminate spikes in the measured $\mu(k)$ and quickly latch on to "long-term" changes [KK 92].

Given the round-trip propagation delay $R$ and the number of packets outstanding (that is, transmitted, but not acknowledged) $S$, packet-pair estimates $X$, the number of packets in the bottleneck buffer, as:

$$X = S - R \cdot \hat{\mu}(k+1),$$  \hspace{1cm} (13.12)

because of the $S$ packets, $Ra(k + 1)$ are "in the pipe," so the rest must be in the bottleneck buffer. If the setpoint is $B$, then the source adjusts the actual transmission rate so that it reaches the setpoint in approximately one round-trip time. Thus, the transmission rate $\lambda(k+1)$ is given by:

$$\lambda(k+1) = \hat{\mu}(k+1) + (B - X)/R$$

It can be shown both analytically and through simulation that this choice of $\lambda$ results in a control system that is stable and does not oscillate [Keshav 91, ABB 93]. In practice, $\lambda$ is updated using this equation on the arrival of each ack pair. When a source sends a pair of packets, it uses the latest value of $\lambda$ to set a timer for the next pair of packets.

The hard part in packet-pair is in computing the number of outstanding packets, $S$, accurately in the presence of packet losses. If a source does not decrement $S$ to account for lost packets, the estimate for $X$ is too large, and thus $\lambda$ is too small, leading to underutilization of the bottleneck. Fortunately, we can reliably estimate $S$ using the SMART algorithm presented in Section 12.4.7. Simulations show that by carefully accounting for packet and ack losses, packet-pair, with SMART, works extremely well even with loss rates as high as 10% [KM 97].
Evaluation

Exhaustive simulations show that packet-pair flow control is stable in a variety of scenarios. For example, Figure 13.11 (from [Keshav 97]) plots the sequence number versus time for four packet-pair sources that share a common bottleneck server. We see that all four sources make progress at equal rates. Moreover, there are no pauses due to timeouts and retransmissions. Indeed, this is the best performance that one can expect. It has been shown that for many scenarios, packet-pair's performance is nearly as good as an implementable optimal flow-control scheme that assumes that network elements have infinite buffers [KKM 93, Keshav 97]. Packet-pair scales well with network size and does not require any explicit information from network elements. Thus, it is a good choice for flow control in networks of round-robin servers.

Unfortunately, most current networks do not provide round-robin-like servers (though the situation is rapidly changing). In such networks, a scheme such as the EERC scheme described in Section 13.4.9 can be used to explicitly convey rate and buffer size information to an endpoint. If one or both quantities are available, a source can plug them into the equations given earlier.

13.4.9 ATM Forum end-to-end rate-control scheme

| Implicit | Dynamic window | Hop-by-hop |

The ATM Forum has adopted an explicit end-to-end rate-based flow-control (EERC) scheme similar to the DECbit explicit end-to-end dynamic window flow-control scheme we studied in Section 13.4.4, for controlling available bit-rate (ABR) traffic (see Section 14.4 [BF 95]). In this scheme, each source periodically sends a resource management (RM) cell with a request for an allocation of a particular transmission rate in the next time interval. Each network element (or server) computes a rate allocation per-source according to the max-min fairness criterion described in Section 13.4.4. It writes this rate in the RM cell, which carries it back to the source. At the source, the rate in the RM cell is used to dynamically modify its transmission rate. The scheme therefore is similar to the DECbit scheme, except that a bottlenecked server is allowed to return more information to the sender, and the sender uses this to adapt a rate rather than a window.

We begin a detailed description of this scheme with a table of acronyms (Table 13.3), which the ATM Forum liberally uses in its documents. Starred acronyms refer to fields negotiated at the time of call establishment.

<table>
<thead>
<tr>
<th>RM cell</th>
<th>Resource management cell</th>
<th>A probe cell periodically sent by a source that returns with the currently available service rate.</th>
</tr>
</thead>
<tbody>
<tr>
<td>NRM</td>
<td>Number of resource management cells</td>
<td>A source sends an RM cell every NRM data cells.</td>
</tr>
<tr>
<td>ER</td>
<td>Explicit rate</td>
<td>A field in the RM cell that initially carries the rate a source requests and is modified by intermediate network elements.</td>
</tr>
<tr>
<td>ACR</td>
<td>Allowed cell rate</td>
<td>The rate at which the source actually transmits.</td>
</tr>
<tr>
<td>PCR*</td>
<td>Peak cell rate</td>
<td>The highest rate at which a source can ever transmit.</td>
</tr>
<tr>
<td>ICR*</td>
<td>Initial cell rate</td>
<td>The initial rate at which a source transmits.</td>
</tr>
<tr>
<td>MCR*</td>
<td>Minimum cell rate</td>
<td>The minimum transmission rate. A source is guaranteed to have a rate of at least MCR always available.</td>
</tr>
<tr>
<td>RIF*</td>
<td>Rate increase factor</td>
<td>An increase factor used for additive increase. If there is no congestion, a source increases its rate by RIF times PCR.</td>
</tr>
<tr>
<td>RDF*</td>
<td>Rate decrease factor</td>
<td>A decrease factor used for multiplicative decrease in the allowed cell rate.</td>
</tr>
<tr>
<td>EFCl</td>
<td>Explicit forward congestion indication or notification bit</td>
<td>A bit in the header of every cell that can be set by a congested network element.</td>
</tr>
<tr>
<td>Cl bit</td>
<td>Congestion indication bit</td>
<td>A bit in an RM cell (instead of a cell header) that carries the value of the EFCl field from the destination to the source.</td>
</tr>
</tbody>
</table>

Table 13.3: Acronyms used in the ATM Forum end-to-end rate-control scheme. Starred acronyms refer to fields negotiated at the time of call establishment.
### Table 13.5: Comparison of closed-loop flow-control schemes.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Easy to Implement</th>
<th>Works only when the RTT is small and the receiver has many buffers. Intermediate switches are heavily loaded.</th>
<th>Source does not send a packet until the receiver acknowledges the previous one.</th>
<th>Source can keep more than one packet outstanding.</th>
<th>Uses only one bit in the packet header when congested, causing cooperating sources to reduce their window size using an AIMD policy.</th>
<th>Assumed cooperative sources. Startup is too conservative in environments with large bandwidth-delay products. Endpoint algorithm can be implemented in software. Does not assume fair-share scheduling at switches. Minimal buffering needed at switches.</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table 13.5: Continued.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 13.5 Continued.

<table>
<thead>
<tr>
<th>Type</th>
<th>Control</th>
<th>Switches adjust per-connection window per-link to ensure no loss at downstream switch and full utilization of bottleneck link.</th>
<th>Rapid reaction to available capacity at the bottleneck link. No assumptions about source behavior.</th>
<th>Assumes fair scheduling at switches, and per-connection queuing. Requires about 4 RTTs worth of buffers at each switch.</th>
</tr>
</thead>
<tbody>
<tr>
<td>H W E</td>
<td></td>
<td>Source adjust periodically injects a resource management cell into its cell stream. Switches write the sustainable fair share into this cell, which is carried back to the source.</td>
<td>Works well over a wide range of bandwidth-delay products. Sources know the exact fair share. Allows room for innovation in source flow-control algorithms. Interoperates with DiffServ switches. Does not require fair-share scheduling. Switches do not need per-connection state. Does not assume cooperative sources.</td>
<td>Interleaved RM cells mix the data and control paths. Only feasible in hardware at the endpoints. Bandwidths are complex. Fair-share computation is explicit and can be expensive. Source rate needs to be policed.</td>
</tr>
</tbody>
</table>

### 13.5 Hybrid flow control

In open-loop flow control, a source reserves capacity according to its expected traffic, whereas in closed-loop flow control, the source must adapt to changing network conditions. In hybrid control, a source reserves some minimum capacity, but may obtain more if other sources are inactive. We have already seen an example of hybrid control in the ATM Forum EERC scheme, where the network guarantees a source a minimum cell rate, but the source may get an allowed cell rate that could be larger. Thus, the source must go through admission control during the call-setup phase, during which every switch must test whether the minimum cell rate is available. Subsequently, sources adjust their demand in response to the bandwidth available in the network. We can also modify other closed-loop schemes to perform hybrid control.

Hybrid control schemes not only inherit the problems of open-loop and closed-loop control, but also introduce some new ones. As in open-loop control, a source must find descriptors for its traffic that satisfy the criteria in Section 13.3. However, note that source descriptors in hybrid control can be looser than in open-loop control, because its descriptor does not limit a source. In other words, a source that asks for too little may not notice it, because the actual capacity available to it is likely to be much greater. Hybrid controlled sources must obey all appropriate closed-loop control mechanisms, and thus hybrid control inherits all the problems and constraints of closed-loop control discussed in Section 13.4.

A new problem introduced by hybrid control is that of resource partitioning at switches [GF 95]. Recall that a network operator can choose to have some or all of the network resources (bandwidth and buffers) available for reservation by open-loop control sources. With a hybrid flow-control scheme, the operator must decide what fraction of the resource can be reserved, and what fraction should be contended for. Clearly, if too large a fraction is reserved, then, during congestion, purely closed-loop controlled sources would get very little bandwidth. On the other hand, if too small a fraction is reserved, only a few hybrid-controlled sources can be admitted. Appropriate partitioning depends on the pricing policy for hybrid and closed-loop flow-controlled sources and is still an open research area.

Despite these problems, hybrid control has a strong advantage: a guaranteed minimum resource allocation to an admitted call, even when the network is overloaded. Thus, a hybrid-controlled source, once admitted, knows that even in the worst case, it has some minimum bandwidth guaranteed to it, and in the average case, it will obtain substantially more bandwidth. This is a desirable property for applications that involve voice and video transfer, because they must have at least a minimum bandwidth to provide utility to users. For example, if an application can provide degraded, but acceptable, voice transfer at 8 Kbps, and can provide excellent voice quality at 16 Kbps, then it might ask for a minimum cell rate of 4 Kbps, but contend for, and obtain, a higher rate on average. In the presence of congestion (which we presume is a rare event), the application can still provide utility to its users. If it did not have this minimum guaranteed rate, congestion would cause the application to become unusable.
To sum up, hybrid control has one main advantage, that is, the ability to guarantee a minimum service rate to admitted calls even in the worst case, and all the problems of open-loop and closed-loop flow control. Nevertheless, this advantage seems worth the trouble, and it is likely that future networks will provide some form of hybrid flow control.

13.6 Summary

Like error control, flow control can be performed at many layers of the protocol stack. This chapter describes a set of flow-control techniques that we can apply in a variety of situations. We studied the three forms of flow control: open-loop, closed-loop, and hybrid. We examined some problems in open-loop control, particularly in choosing appropriate flow descriptors. We also studied several closed-loop schemes that use (a) implicit or explicit rate measurements, (b) hop-by-hop or end-to-end control, and (c) dynamic-window or dynamic-rate control. These schemes make widely varying assumptions about their environment of operation, and thus are not directly comparable. Table 13.5 is a summary of these schemes. Finally, we looked at some problems and advantages with hybrid flow control.

Review Questions

1. What is the bottleneck rate in the equivalent reduced flow model?
2. What is the flow-control problem?
3. What are the three types of flow-control schemes?
4. What is a traffic descriptor, and what purposes does it serve?
5. What are regulators and policers? How do they differ?
6. What is a jumping-window regulator?
7. What is an LBAP?
8. What parameters control the performance of a Leaky Bucket regulator?
9. What is a \((\sigma, \rho)\) curve, and how does it help in selecting a traffic descriptor?
10. What information is conveyed in an explicit state-measurement packet?
11. Can the on-off scheme be implemented hop-by-hop?
12. When does a stop-and-wait scheme have poor performance?
13. What is the optimal choice for a static-window size?
14. When does a DECbit router set bits, and on which sources?
15. When does a DECbit source reduce its window, and by how much?
16. How long is a queue regeneration cycle?
17. What are the two problems with the nonselective DECbit algorithm?
18. What does the \(s_{thresh}\) value in TCP-Tahoe determine?
19. To what value does TCP-Reno reduce its window size on a timeout?
20. How does TCP-Vegas determine how many packets are in the bottleneck buffer?
21. What are the MCR and ICR values in the ATM Forum EERC scheme?

Exercises

13.1. A source is on an Ethernet with a propagation delay of 100 \(\mu s\) and a service rate of 10 Mbps. This is connected to a T1 line via a router with propagation delay 30 ms. The destination is connected via a symmetric arrangement. Assuming 1-ms service times at the routers, what are the equivalent delay and service rates for a reduced-flow model that describes this path?
13.2. If the peak rate of a connection is 1.5 Mbps, and the packet size is fixed at 53 bytes, how close in time can two packets be?
13.3. A hypothetical source is cell-smooth with an intercell spacing of 0.1 ms, except that once every second, it emits two cells instead of one. If the service rate is exactly 10,000 cells/s, what is the queuing delay at the peak-rate regulator 1 h after the source starts? Suggest two ways this delay can be eliminated.
13.4. Give an example to show that the jumping window-descriptor is sensitive to the choice of the starting time of the first window.
13.5. Packets arrive to a moving-window regulator as follows: (40 bytes, 0.1 s), (20 bytes, 0.2 s), (40 bytes, 0.3 s), (60 bytes, 0.5 s). If the regulator has a constraint of 100 bytes over 1-s intervals, what is the output stream?
13.6. If a token is 200 bytes long, the token bucket is 1Kbytes, and tokens arrive at the rate of 1 every second, what is the least and most delay suffered by a packet of size 800 bytes if it arrives when the token bucket has 100 bytes in it?
13.7. If a DECbit source has bits set on all its packets when it reaches a window size of 16, what is the optimal window size if the RTT is 1, and the optimal window size is 16, what is the utilization of the bottleneck line (assuming this is the only source) when the source has a window size of \(w\)? Use this to compute the mean utilization of the bottleneck if it only carries this single source.
13.8. If a TCP-Tahoe source has a packet loss when its window size is 16, what is the range in which its window size oscillates? If the RTT is 1, and the optimal window size is 16, what is the utilization of the bottleneck link when the source has a window size of \(w\)? Use this to compute the mean utilization of the bottleneck if it only carries a single source.
13.9. Is it possible in the ATM Forum EERC scheme for the CI bit to be set to 1, but the ER field to be larger than the source’s ACR? What action does a source take when this happens?