Chapter 10
TCP Strategies

TCP gets reliability by acknowledging every byte sent. Does this mean two segments for every packet? It is possible to implement TCP like this, but performance would be poor

TCP uses a variety of strategies to improve performance, but always strictly keeping to the TCP protocol

TCP Strategies

Sliding Window

- As data arrives at its destination it is put into a buffer, reader for the application to read it: the TCP advertised window in a returning segment indicates how much buffer space is left
- The space depends on
  - how fast the sender is sending the data
  - how fast the application is reading the data

TCP Strategies

Sliding Window

- If the data arrives faster than it is read, the buffer will fill up
- The advertised window is how TCP tells the source to slow down
- It is a *sliding window* mechanism, used as a form of *flow control*

TCP Strategies

Sliding Window

- A sliding window describes the range of bytes the sender can transmit
- As the window gets smaller, the sender should send less
- As the window gets bigger, the sender can send more
- The sender recomputes the window every time it gets an ACK

TCP Strategies

Sliding Window

- The left hand edge of the window is defined by the latest ACK
- The right hand edge is then given by the advertised window
- The window is sent in every ACK segment
- As more ACK are received, the window *closes* as the left edge advances

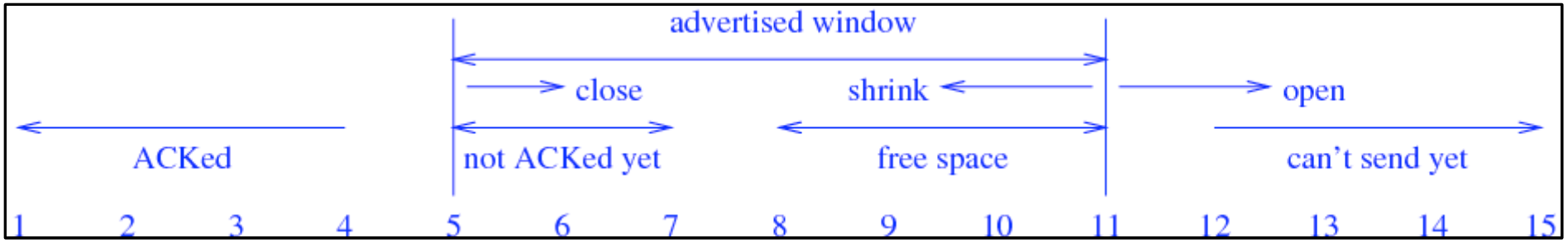
TCP Strategies

Sliding Window

- As the application reads data, the window *opens* as the right edge advances
- Rarely, the window can *shrink* (right edge recedes), perhaps if the buffer shrinks due to the memory being needed elsewhere

TCP Strategies

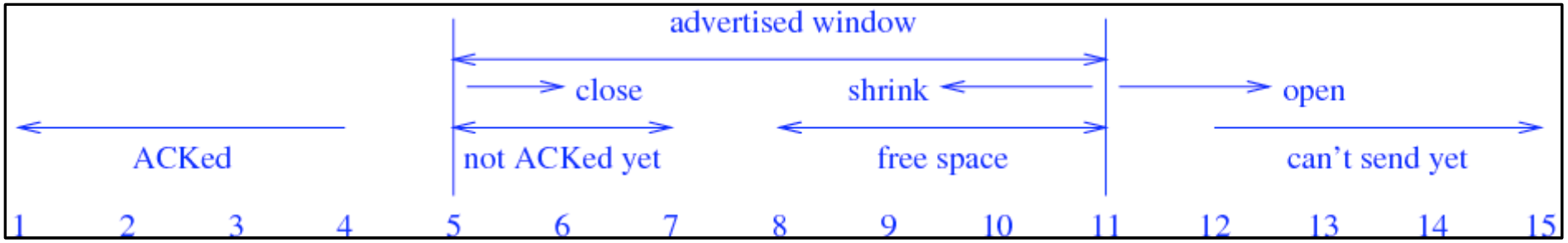
Sliding Window



- Bytes to the left of the window (1-4) have been ACKed and are safe
- Bytes to the right (12-) cannot be sent yet
- Bytes within the window are either not ACKed yet, or represent free space

TCP Strategies

Sliding Window



- unACKed bytes (5-7) are those that have been read, but an ACK not yet sent
- The free space (8-11) is the actual amount of bytes that can be buffered
- The sender can compute this as the window minus the number of unACKed bytes

TCP Strategies

Sliding Window

- It is not unusual for the window to reduce to 0
- The sender will have to wait before sending more data
- When the receiver is ready it will send a duplicate ACK with the same sequence number, but a non-zero window: this is a *window update* segment

TCP Strategies

Sliding Window

- Complications arise if this ACK gets lost: the *Persist Timer* (see later) is used here

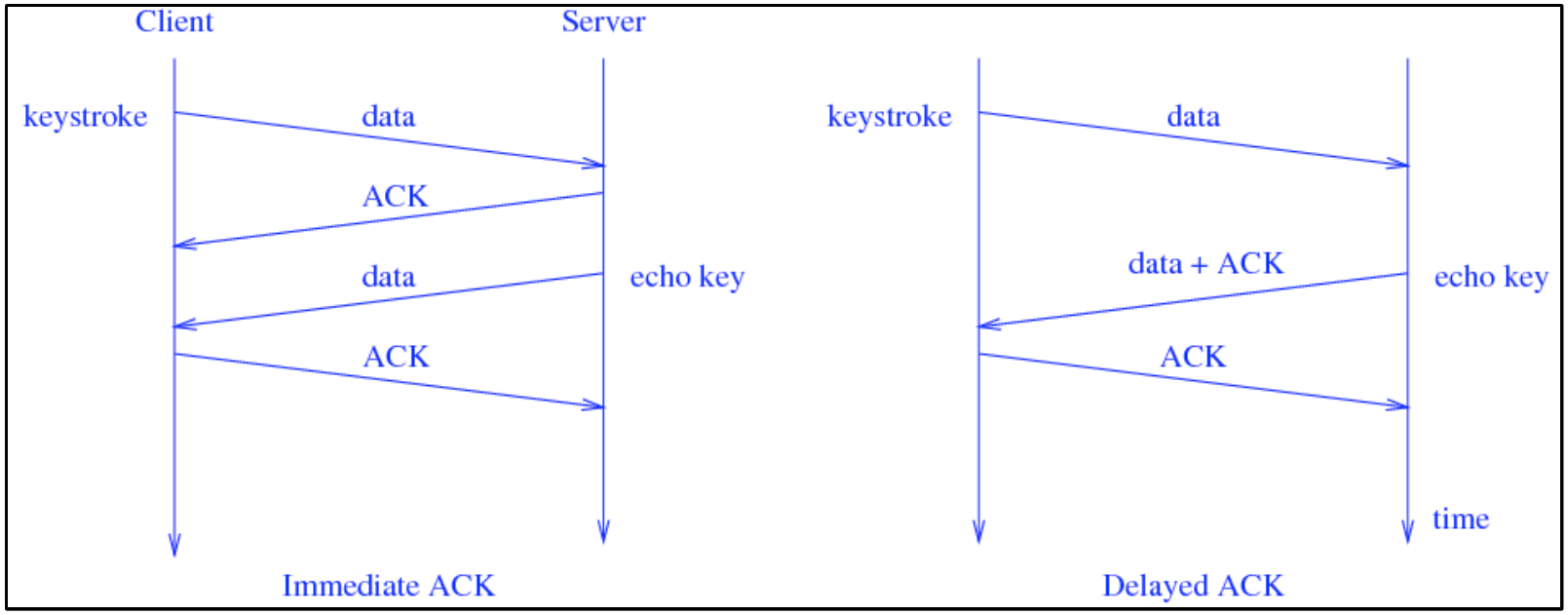
TCP Strategies

Delayed ACKs

- Instead of immediately ACKing every segment, we can slightly delay it and piggyback it on returning data

TCP Strategies

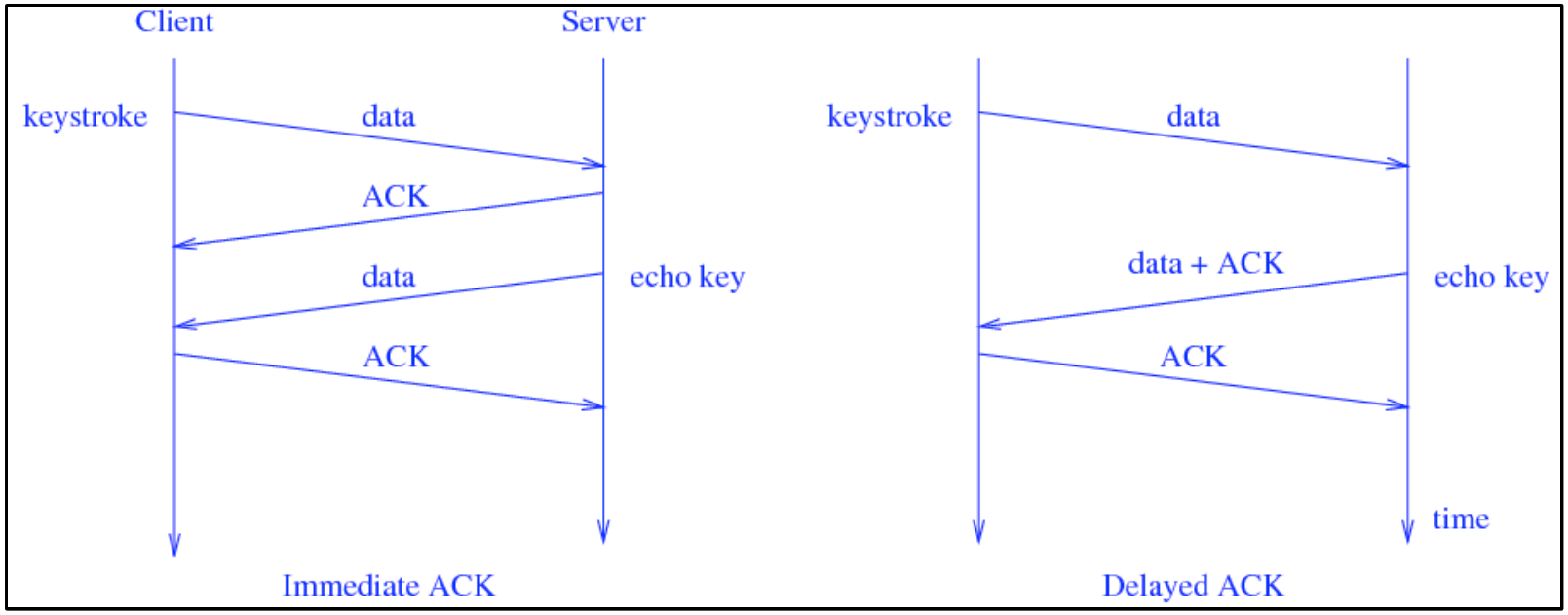
Delayed ACKs



- For example, when remotely logged in to a machine each keystroke is echoed back to your screen

TCP Strategies

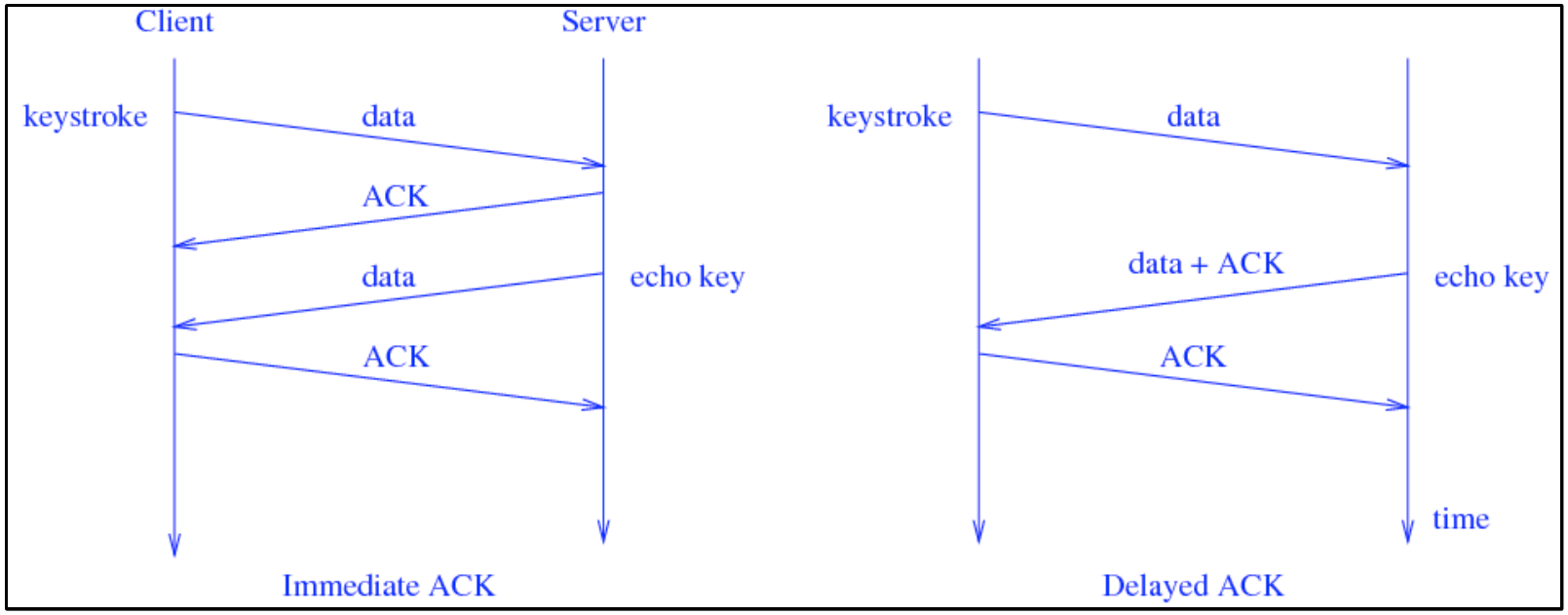
Delayed ACKs



- An immediate ACK would use four segments

TCP Strategies

Delayed ACKs



- A delayed ACK piggybacking on the echoed key uses just three segments

TCP Strategies

Delayed ACKs

- The same time is taken for the exchange, but fewer segments are sent
- Important to reduce the traffic on a heavily loaded network
- Also reduces the chance of a lost segment

TCP Strategies

Delayed ACKs

- We can also ACK more than one segment at a time
- If we receive, say, three segments in the time we are delaying, we can simply ACK the last segment: this implicitly ACKs the previous two segments: an ACK indicates which byte we are expecting next
- The reduces traffic again

TCP Strategies

Delayed ACKs

- So how long to delay an ACK?
- If too long, the sender might think the segment was lost and resend
- If too short, we do not get so many free piggybacks
- A typical implementation will delay for up to 200ms

TCP Strategies

Delayed ACKs

- The TCP specification says not to delay for more than 500ms
- This is the first of many *timers* associated with TCP
- Each time you receive a segment you (i.e., the TCP software) set a timer for that segment that expires after 200ms

TCP Strategies

Delayed ACKs

- If the segment has not already been ACKed (e.g., on a returning data segment), ACK it when the timer expires
- Many implementations have a single global timer that fires every 200ms rather than a timer per segment
- When the timer goes off, all unACKed segments are ACKed

TCP Strategies

Delayed ACKs

- Not so good as per-segment timers, but much easier to implement
- If you receive and out-of-sequence segment (the sequence number is not the one you are expecting), e.g., a segment was lost, you must not delay, but send an ACK immediately
- This will probably be a *duplicate ACK* of one sent earlier. This is to inform the sender as soon as possible that something has gone wrong

TCP Strategies

Nagle's Algorithm

- When sending keystrokes over a network there is a lot of wasted bandwidth
- A keystroke could be one byte
- This is sent in a TCP segment that has 20 bytes of header
- This is sent in a IP datagram with 20 bytes of header
- And so on down the layers

TCP Strategies

Nagle's Algorithm

- So we are sending (for the sake of argument) a 41 byte packet for each byte of data
- Such a packet is called a *tinygram*
- The proliferation of tinygrams causes additional congestion in a network
- Nagle created a strategy for reducing this
- It applies to the sender (client) rather than the receiver (server)

TCP Strategies

Nagle's Algorithm

"a TCP connection can have only *one* outstanding unACKed small segment: no additional small segments can be sent until that ACK has been received"

- Any small waiting segments should be collected together into a single larger segment that is sent when the ACK is received

TCP Strategies

Nagle's Algorithm

- This segment can also be sent if you collect enough small segments to fill a segment, or have exceeded half the destination's window size
- This leaves open the definition of “small”
- Variants choose anything from “1 byte” to “any segment shorter than the maximum segment size”

TCP Strategies

Nagle's Algorithm

- This is a very simple strategy and reduces the number of tinygrams without introducing extra perceived delay (over that delay there is there already)
- The faster ACKs come back, the more tinygrams can be sent
- When there is congestion, so ACK return more slowly, fewer tinygrams are sent

TCP Strategies

Nagle's Algorithm

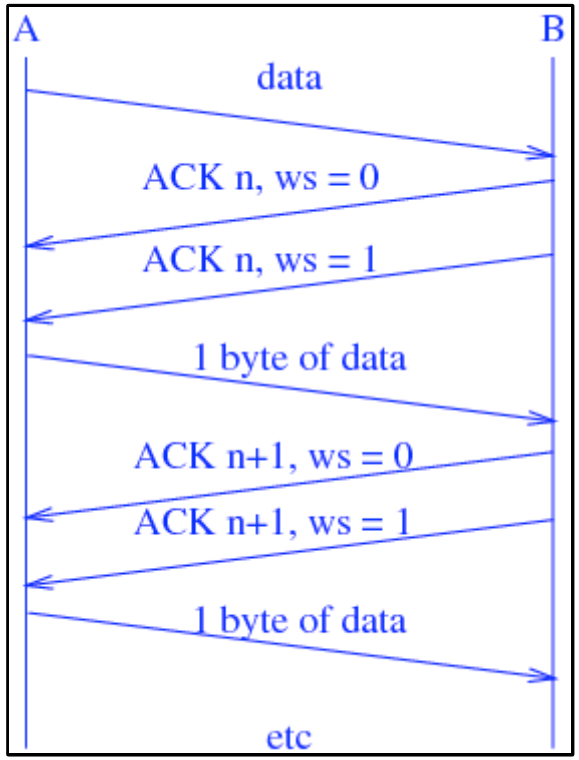
- Nagle can reduce the number of segments significantly when the network is heavily loaded
- Sometimes buffering up tinygrams is not a good idea: e.g., in a graphical interface over a network, each mouse movement becomes a tinygram. Buffering the segments would cause the cursor to jump erratically. Nagle can be turned off for such cases

TCP Strategies

Silly Window Syndrome

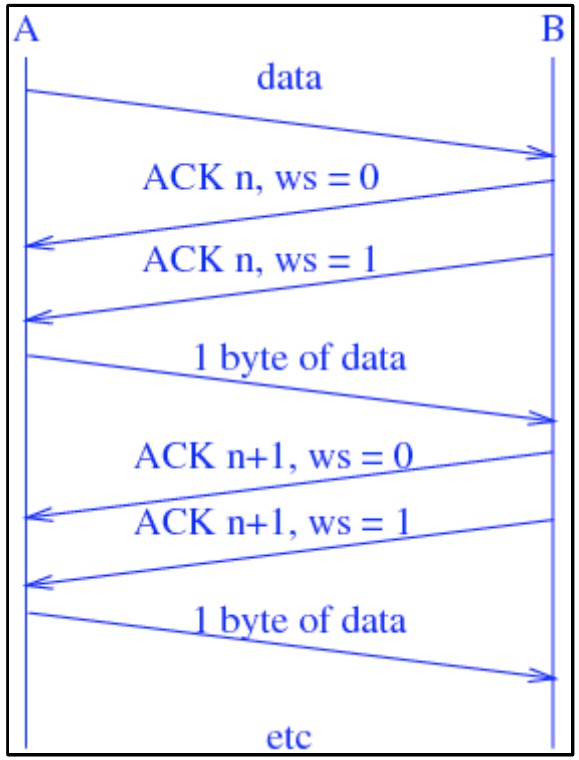
- Another problem with tinygrams is manifested as *silly window syndrome*

TCP Strategies



- A is sending data to B, but B is reading only one byte at a time
- B's buffer fills, and B ACKs with a window of 0
- B reads a byte
- B sends a window update segment, size 1

TCP Strategies



- A get this and sends as much data as possible, i.e., 1 byte
- B ACKs with window 0
- B reads a byte
- B sends an update, size 1
- A sends 1 byte
- and so on

TCP Strategies

Silly Window Syndrome

- We are back to the two segment per byte high overhead: this is silly window syndrome
- Better is for B not to send an update of 1, but wait until there is more space

TCP Strategies

Silly Window Syndrome

- Clarke's algorithm to avoid SWS is

  never send an update for a window of 1

  only advertise a new window when there is enough space for a full segment, or the buffer is half empty

TCP Strategies

Nagle and Silly Window Syndrome

- If we take “small” in Nagle to mean “less than the segment size”, then Nagle and SWS fit together naturally

TCP Strategies

Congestion Control

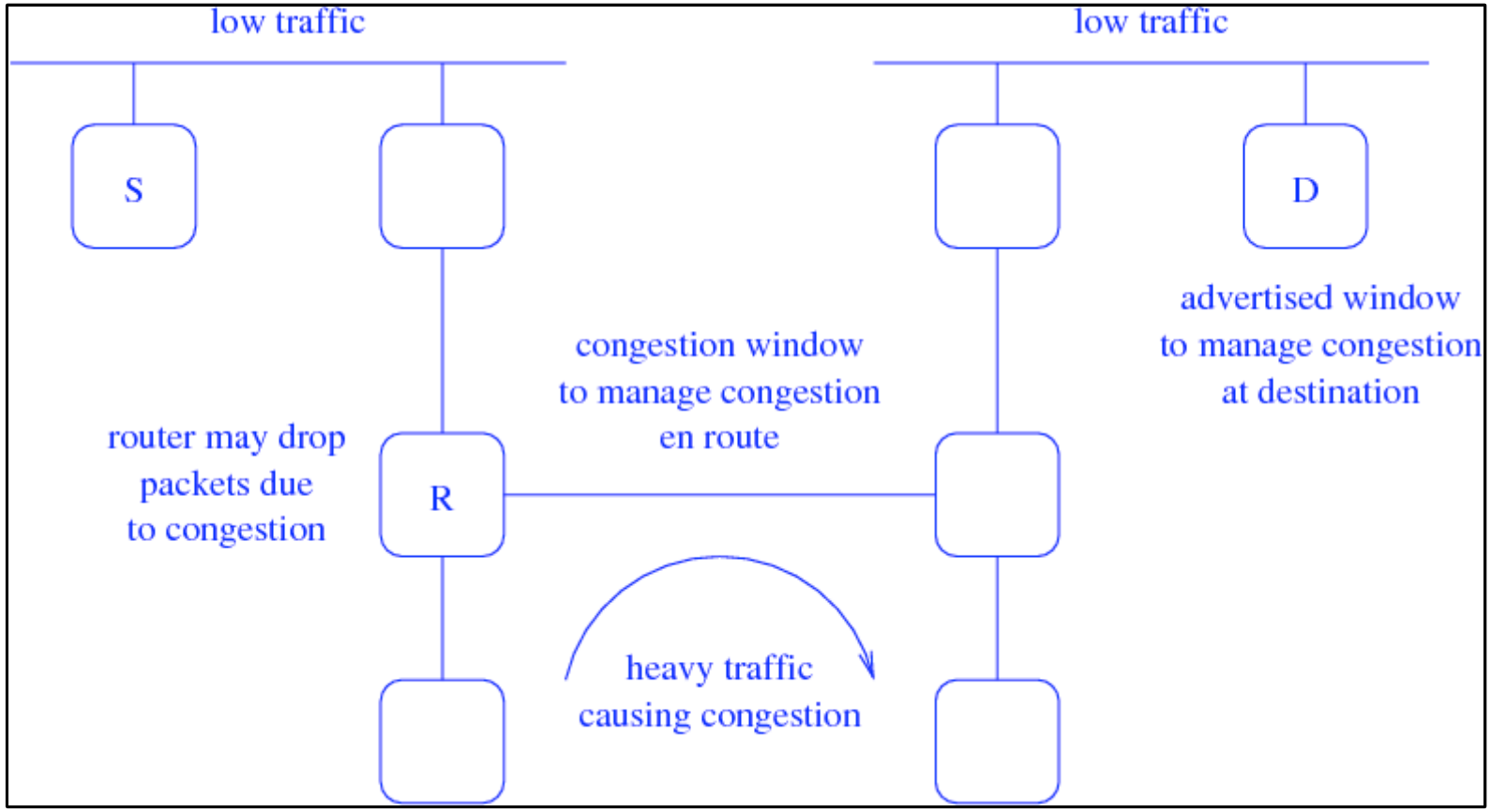
- Congestion happens when more data is sent to a network than it can handle
- There are several strategies in TCP to help deal with and avoid congestion
- The first issue is how to spot congestion, given that it might be happening in a part of the network many hops away

TCP Strategies

Congestion Control

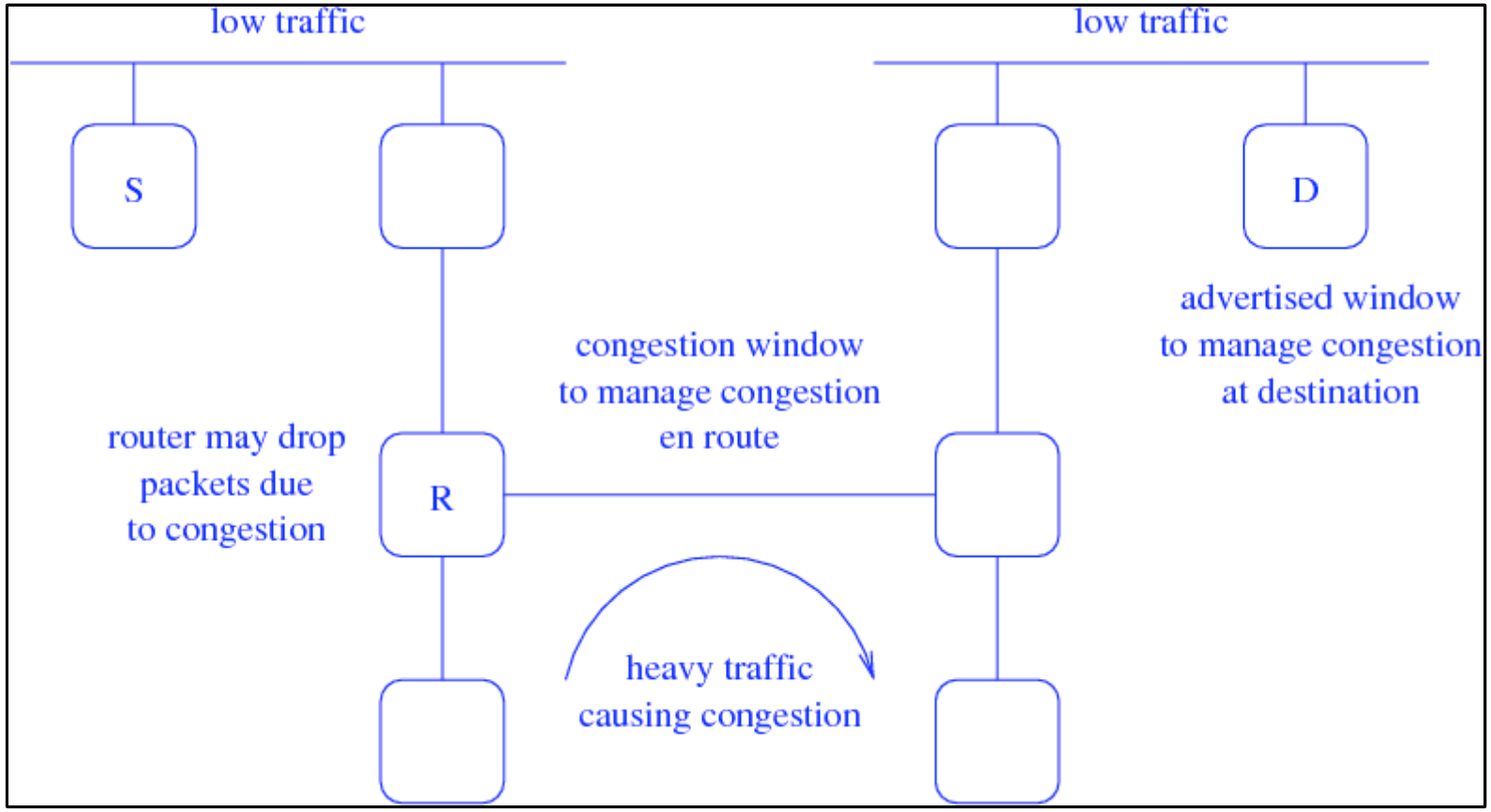
- We watch for segment loss
- Segments can be lost though poor transmission or being dropped at a congested routers or destination
- Poor transmission is unusual these days, so we assume loss is due to congestion (which is common these days)
- Thus TCP treats missing ACKs as a sign of congestion

TCP Strategies



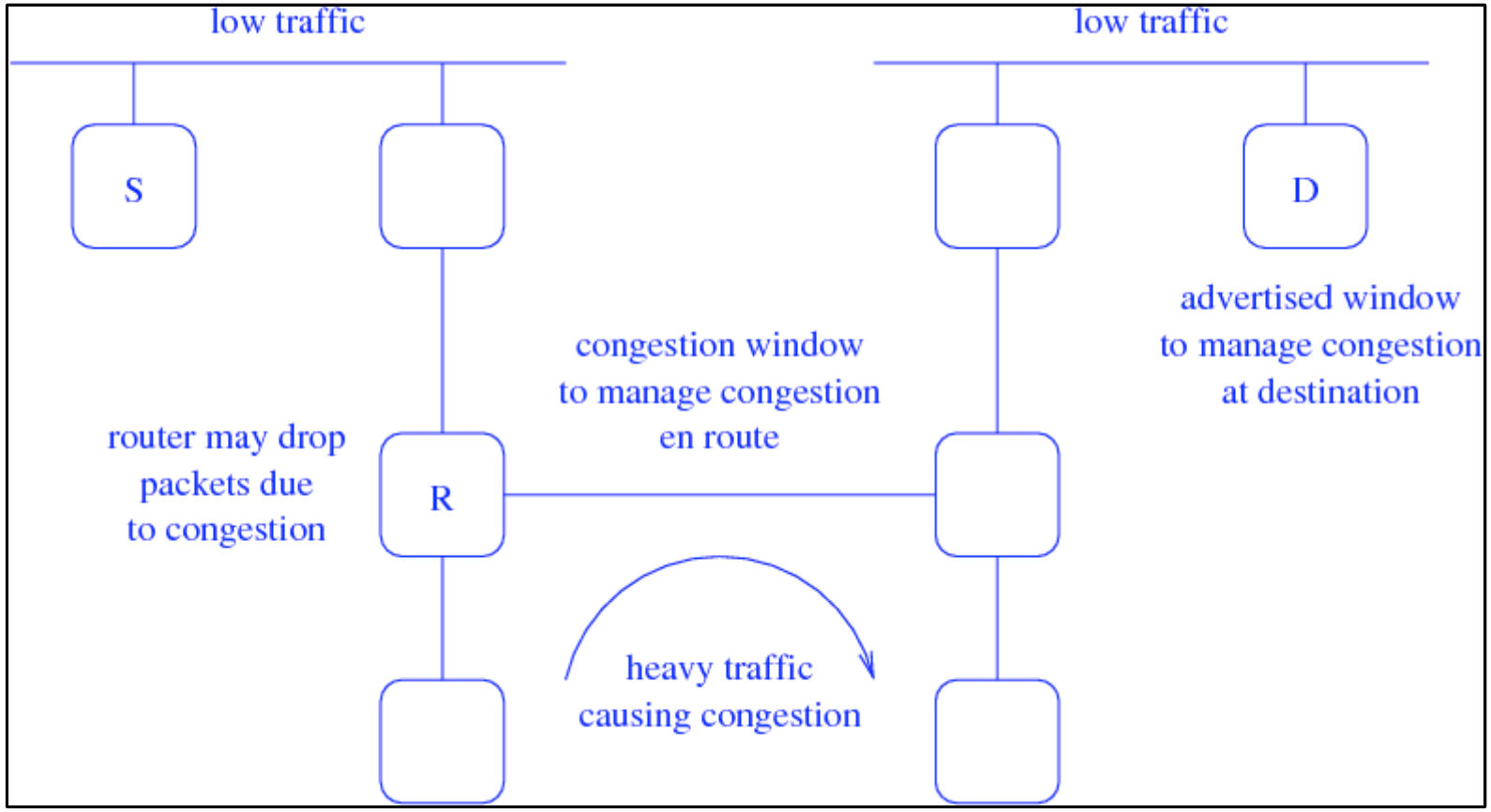
- Congestion can happen in a router due to lack of capacity in an onward link

TCP Strategies



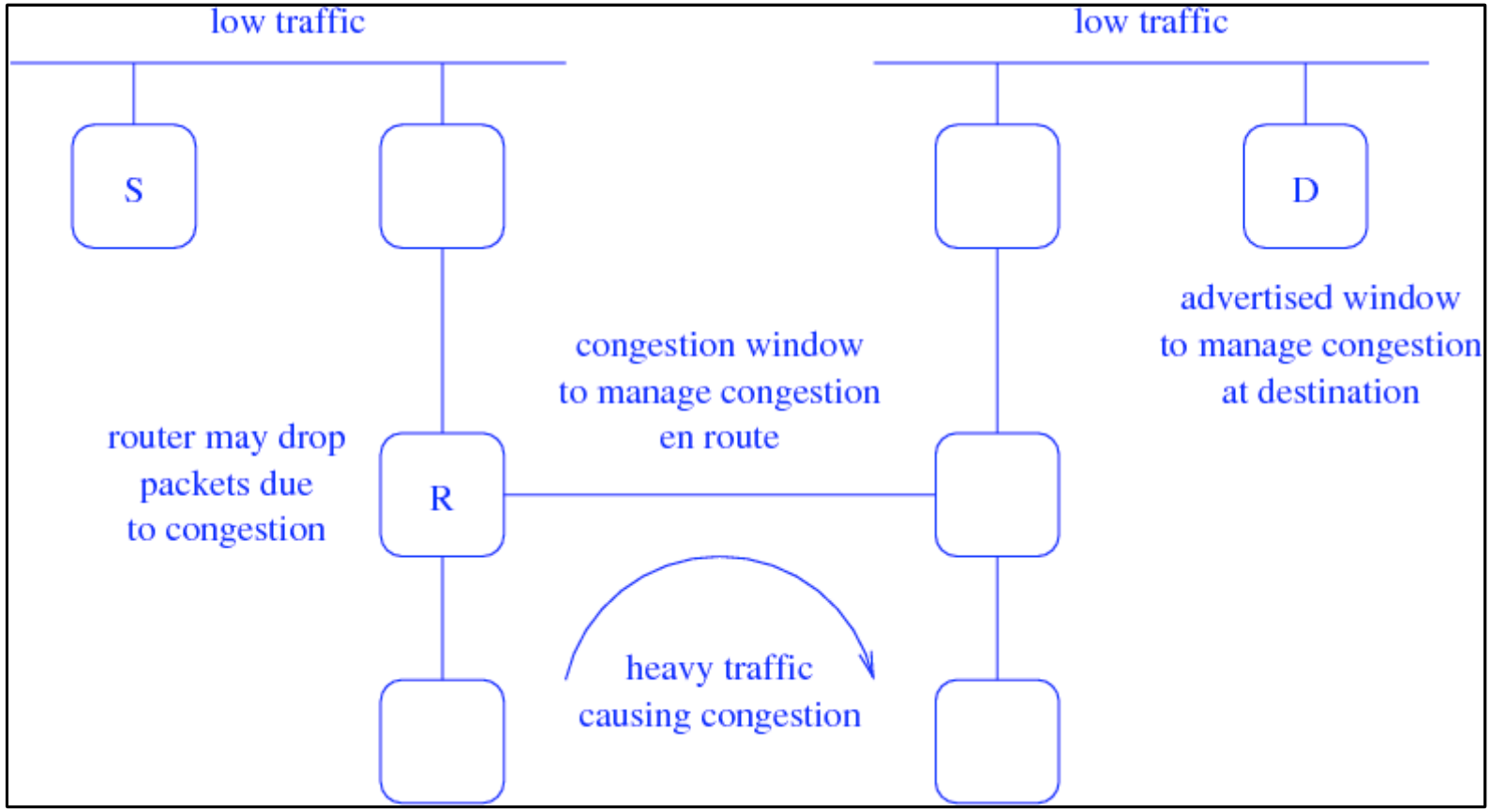
- Also in the destination if it is too slow to receive the data at the rate the network can supply it

TCP Strategies



- The advertised window deals with congestion at the destination

TCP Strategies



- Path congestion needs a different mechanism: the congestion window

TCP Strategies

Congestion Control

- If we have a lot of data to send we do not want to wait for every ACK before sending the next segment
- Better is to send several segments and then wait to see from the ACKs which were safely received
- Time spent waiting is time we might have been sending segments

TCP Strategies

Congestion Control

- But we can't send *too* many segments at once in case the network is congested: we'll just make things worse
- So we need a way to estimate the capacity of the network to send as many segments at once as we can, but not too many
- If we get it right, we will have a continual stream of segments going out and ACKs coming back