TCP

TCP State

The various stages a TCP connection can be in (setting up, tearing down, transmitting data, etc.) are complicated

TCP

TCP State

The various stages a TCP connection can be in (setting up, tearing down, transmitting data, etc.) are complicated

There is a standard TCP state diagram that describes how TCP should act in most cases

TCP

TCP State

The various stages a TCP connection can be in (setting up, tearing down, transmitting data, etc.) are complicated

There is a standard TCP state diagram that describes how TCP should act in most cases

Though it only covers non-error cases: it does not say what to do if, say, a SYNFIN segment arrives

TCP

TCP State

The various stages a TCP connection can be in (setting up, tearing down, transmitting data, etc.) are complicated

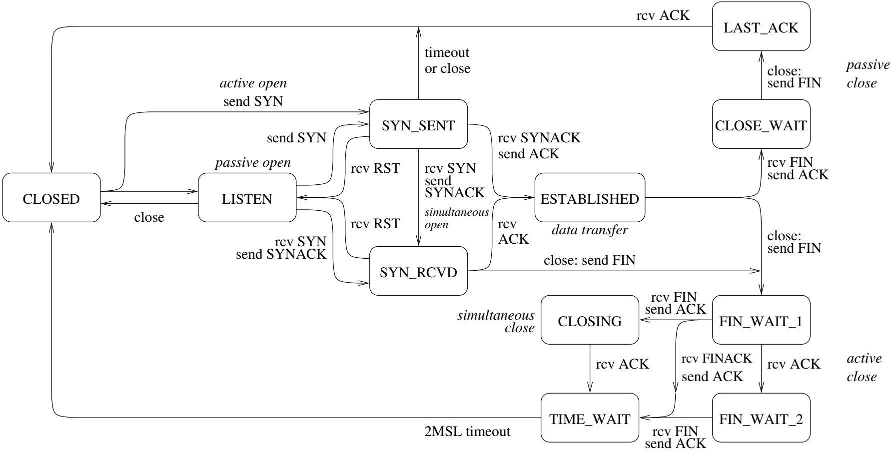
There is a standard TCP state diagram that describes how TCP should act in most cases

Though it only covers non-error cases: it does not say what to do if, say, a SYNFIN segment arrives

And it shows little about timeouts and retransmissions

TCP

TCP State



TCP State Diagram

We start (and end) in CLOSED

TCP

TCP State

We start (and end) in CLOSED

There are the two opens: active and passive

We start (and end) in CLOSED

There are the two opens: active and passive

LISTEN is a server waiting for a connection

TCP

TCP State

We start (and end) in CLOSED

There are the two opens: active and passive

LISTEN is a server waiting for a connection

ESTABLISHED is the normal data transfer state

We start (and end) in CLOSED

There are the two opens: active and passive

LISTEN is a server waiting for a connection

ESTABLISHED is the normal data transfer state

And the two closes: active and passive

TCP

TCP State

We start (and end) in CLOSED

There are the two opens: active and passive

LISTEN is a server waiting for a connection

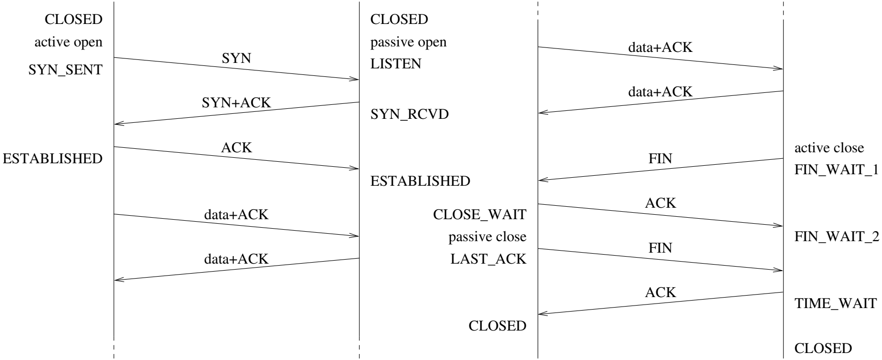
ESTABLISHED is the normal data transfer state

And the two closes: active and passive

This state diagram is followed for each end of a connection, i.e., each socket in the socketpair

TCP

TCP State



Typical TCP Timeline

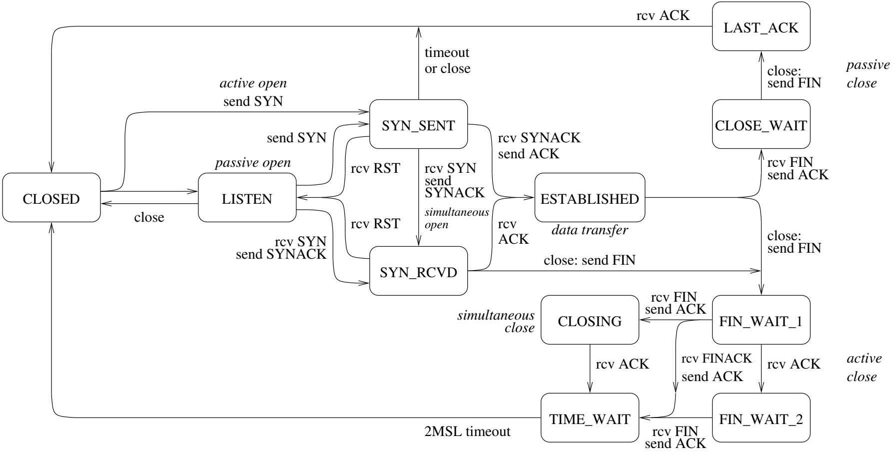
The active close is somewhat complicated by the need for reliability

The active close is somewhat complicated by the need for reliability

The TIME_WAIT state (also called 2MSL state) appears before the final close: the active-close end of the connection must remain non-closed until a time period has passed

TCP

TCP State



TCP State Diagram

TCP

TCP State

At this point this end of the connection has received a final ACK and sent its final ACK

TCP

TCP State

At this point this end of the connection has received a final ACK and sent its final ACK

In a perfect world this would be enough to close the connection

TCP

TCP State

At this point this end of the connection has received a final ACK and sent its final ACK

In a perfect world this would be enough to close the connection

But we have to deal with the case of the final ACK being lost

TCP

TCP State

At this point this end of the connection has received a final ACK and sent its final ACK

In a perfect world this would be enough to close the connection

But we have to deal with the case of the final ACK being lost

And resent if it didn't get to the other end

TCP

TCP State

Just because the application is done with the connection, it doesn’t mean the connection is finished and the OS can discard all the connection state

TCP

TCP State

Just because the application is done with the connection, it doesn't mean the connection is finished and the OS can discard all the connection state

The *maximum segment lifetime* (MSL) is a value that represents the longest time a segment can live in the network before being discarded (probably through TTL expiry)

TCP

TCP State

Just because the application is done with the connection, it doesn’t mean the connection is finished and the OS can discard all the connection state

The *maximum segment lifetime* (MSL) is a value that represents the longest time a segment can live in the network before being discarded (probably through TTL expiry)

This was originally defined to be 2 minutes, but implementations often choose smaller values, like 60 seconds

TCP

TCP State

Just because the application is done with the connection, it doesn't mean the connection is finished and the OS can discard all the connection state

The *maximum segment lifetime* (MSL) is a value that represents the longest time a segment can live in the network before being discarded (probably through TTL expiry)

This was originally defined to be 2 minutes, but implementations often choose smaller values, like 60 seconds

A TCP connection is required stay in TIME_WAIT for twice the MSL

This is in case the final ACK (of the final FIN) was lost and needs to be retransmitted

TCP

TCP State

This is in case the final ACK (of the final FIN) was lost and needs to be retransmitted

The OS has to keep the connection hanging around for a little to cover this case

TCP

TCP State

This is in case the final ACK (of the final FIN) was lost and needs to be retransmitted

The OS has to keep the connection hanging around for a little to cover this case

Even if the process that used the connection has exited

TCP

TCP State

This is in case the final ACK (of the final FIN) was lost and needs to be retransmitted

The OS has to keep the connection hanging around for a little to cover this case

Even if the process that used the connection has exited

And while in this wait state if a new process tries to make a connection using the same ports it will be denied: the old connection is still active. We don’t want to deliver late packets to the new process

TCP

TCP State

This is in case the final ACK (of the final FIN) was lost and needs to be retransmitted

The OS has to keep the connection hanging around for a little to cover this case

Even if the process that used the connection has exited

And while in this wait state if a new process tries to make a connection using the same ports it will be denied: the old connection is still active. We don't want to deliver late packets to the new process

In this sense the TCP connection and the process using it are quite separate entities

TCP

Teardown

When an application exits, the OS sends FINs on behalf of the application for all currently open connections. This makes sure everything is tidied up nicely (even if the programmer didn't)

TCP

Teardown

When an application exits, the OS sends FINs on behalf of the application for all currently open connections. This makes sure everything is tidied up nicely (even if the programmer didn't)

And if it was an active close, OS needs to hold the connection in the `2MSL` state for a while: the connection definitely outlives the application!

TCP

Teardown

When an application exits, the OS sends FINs on behalf of the application for all currently open connections. This makes sure everything is tidied up nicely (even if the programmer didn't)

And if it was an active close, OS needs to hold the connection in the `2MSL` state for a while: the connection definitely outlives the application!

If a host is shut down normally, rather than crashing, the operating system will (should!) send FINs for all currently open connections

TCP

Teardown

When an application exits, the OS sends FINs on behalf of the application for all currently open connections. This makes sure everything is tidied up nicely (even if the programmer didn't)

And if it was an active close, OS needs to hold the connection in the `2MSL` state for a while: the connection definitely outlives the application!

If a host is shut down normally, rather than crashing, the operating system will (should!) send FINs for all currently open connections

It really should do the `TIME_WAIT`, but often implementations don't bother as this would hold up the shutdown

TCP Strategies

We now take a look at how TCP manages to get the best out of a connection

TCP Strategies

We now take a look at how TCP manages to get the best out of a connection

For example: TCP gets reliability by acknowledging every byte sent. Does this mean two segments for every data packet: one data packet out, one ACK packet back?

TCP Strategies

We now take a look at how TCP manages to get the best out of a connection

For example: TCP gets reliability by acknowledging every byte sent. Does this mean two segments for every data packet: one data packet out, one ACK packet back?

It is possible to implement TCP like this, but performance would be poor

TCP Strategies

We now take a look at how TCP manages to get the best out of a connection

For example: TCP gets reliability by acknowledging every byte sent. Does this mean two segments for every data packet: one data packet out, one ACK packet back?

It is possible to implement TCP like this, but performance would be poor

So a typical TCP implementation will be a bit more smart on its use of ACKs: we have already mentioned delaying an ACK to let it piggyback on a returning data segment

TCP Strategies

That is just first of many strategies a TCP implementation can employ while still following the TCP protocol

TCP Strategies

That is just first of many strategies a TCP implementation can employ while still following the TCP protocol

We shall look at a few basic strategies, starting with more detail on the advertised window

TCP Strategies

Advertised Window

As data arrives at its destination the OS puts it into a buffer, ready for the receiving application to read it. We have already seen the TCP *advertised window* in a returning segment which indicates how much of this buffer space is left

TCP Strategies

Advertised Window

As data arrives at its destination the OS puts it into a buffer, ready for the receiving application to read it. We have already seen the TCP *advertised window* in a returning segment which indicates how much of this buffer space is left

The space left depends on

- how fast the sender is sending the data
- how fast the application is reading the data

TCP Strategies

Advertised Window

As data arrives at its destination the OS puts it into a buffer, ready for the receiving application to read it. We have already seen the TCP *advertised window* in a returning segment which indicates how much of this buffer space is left

The space left depends on

- how fast the sender is sending the data
- how fast the application is reading the data

If the data arrives faster than it is read, the buffer will fill up

TCP Strategies

Advertised Window

The advertised window is how TCP tells the source to slow down or speed up

TCP Strategies

Advertised Window

The advertised window is how TCP tells the source to slow down or speed up

It is a *sliding window* mechanism, used as a form of flow control

TCP Strategies

Advertised Window

The advertised window is how TCP tells the source to slow down or speed up

It is a *sliding window* mechanism, used as a form of flow control

Imagine the bytes being sent as a long stream, starting at byte 0 (actually byte n, given by the initial sequence number) and going up

TCP Strategies

Advertised Window

The advertised window is how TCP tells the source to slow down or speed up

It is a *sliding window* mechanism, used as a form of flow control

Imagine the bytes being sent as a long stream, starting at byte 0 (actually byte n, given by the initial sequence number) and going up

A sliding window describes the range of bytes in the stream the sender can transmit next

TCP Strategies

Advertised Window

As the window gets smaller, the sender should send more slowly

TCP Strategies

Advertised Window

As the window gets smaller, the sender should send more slowly

As the window gets bigger, the sender can send more quickly

TCP Strategies

Advertised Window

As the window gets smaller, the sender should send more slowly

As the window gets bigger, the sender can send more quickly

The sender recomputes the space available in the receiver every time it receives an ACK

TCP Strategies

Advertised Window

The left hand edge of the window is defined by the acknowledgement number in the latest ACK

TCP Strategies

Advertised Window

The left hand edge of the window is defined by the acknowledgement number in the latest ACK

The right hand edge is then given by adding on the size of the advertised window

TCP Strategies

Advertised Window

The left hand edge of the window is defined by the acknowledgement number in the latest ACK

The right hand edge is then given by adding on the size of the advertised window

The window size is sent in every ACK segment

TCP Strategies

Advertised Window

The left hand edge of the window is defined by the acknowledgement number in the latest ACK

The right hand edge is then given by adding on the size of the advertised window

The window size is sent in every ACK segment

As more ACKs are received, the window *closes* as the left edge advances

TCP Strategies

Advertised Window

As the application reads data, the window *opens* as the right edge advances

TCP Strategies

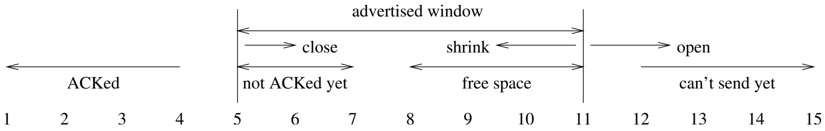
Advertised Window

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

Advertised Window

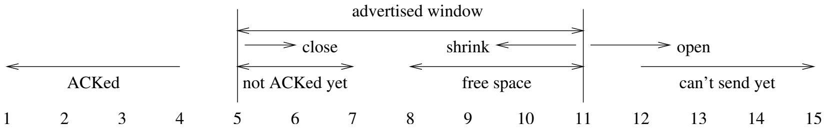


TCP sliding window

This is from the point of view of the sending end of a connection;

TCP Strategies

Advertised Window

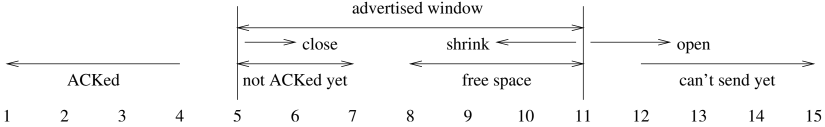


TCP sliding window

The situation is that we have just sent a segment with bytes 5-7; then received an ACK of 5 with a window of 7;

TCP Strategies

Advertised Window

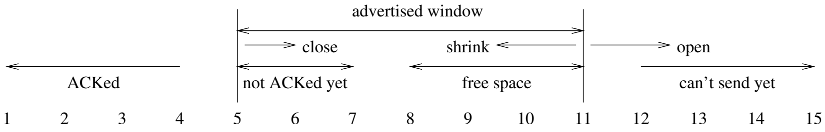


TCP sliding window

Bytes to the left of the window (1-4) have been ACKed and are safe in the destination;

TCP Strategies

Advertised Window

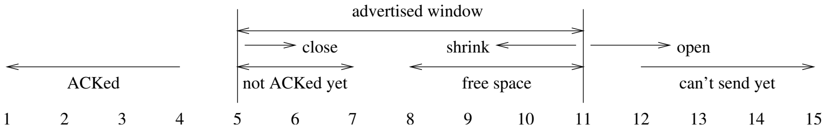


TCP sliding window

The advertised window tells us there is space for 7 bytes in the destination: bytes to the right (12 onwards) cannot be sent yet as the destination has nowhere to put them;

TCP Strategies

Advertised Window

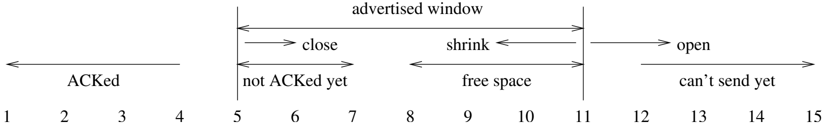


TCP sliding window

Bytes within the window are either not ACKed yet, or represent free space;

TCP Strategies

Advertised Window

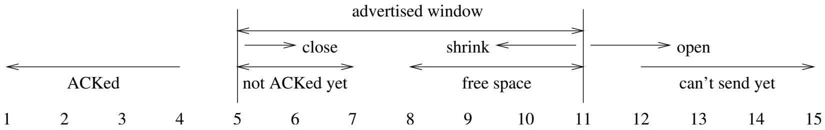


TCP sliding window

unACKed bytes (5-7) are those that have been sent by the sender, possibly received by the destination, and an ACK not yet received by the sender and possibly not yet sent by the receiver;

TCP Strategies

Advertised Window

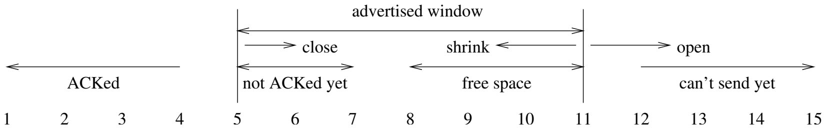


TCP sliding window

The free space (8-11) is the actual number of bytes that the sender can be sure that can be buffered;

TCP Strategies

Advertised Window

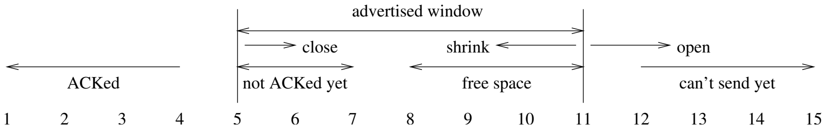


TCP sliding window

The sender can compute this free space as the latest window value minus the number of bytes sent but as yet unACKed;

TCP Strategies

Advertised Window



TCP sliding window

Thus the sender knows the limit on how much more data it can currently send

TCP Strategies

Advertised Window

It is not unusual for the window to reduce to 0, for example when the destination application is reading its data slowly

TCP Strategies

Advertised Window

It is not unusual for the window to reduce to 0, for example when the destination application is reading its data slowly

The sender will have to wait before sending more data

TCP Strategies

Advertised Window

It is not unusual for the window to reduce to 0, for example when the destination application is reading its data slowly

The sender will have to wait before sending more data

When the receiver is ready to receive more data it will send a duplicate ACK with the same ACK number as the ACK with window 0, but now with a non-zero window: this is a *window update segment*

TCP Strategies

Advertised Window

It is not unusual for the window to reduce to 0, for example when the destination application is reading its data slowly

The sender will have to wait before sending more data

When the receiver is ready to receive more data it will send a duplicate ACK with the same ACK number as the ACK with window 0, but now with a non-zero window: this is a *window update segment*

It may or may not contain data itself

TCP Strategies

Advertised Window

It is not unusual for the window to reduce to 0, for example when the destination application is reading its data slowly

The sender will have to wait before sending more data

When the receiver is ready to receive more data it will send a duplicate ACK with the same ACK number as the ACK with window 0, but now with a non-zero window: this is a *window update segment*

It may or may not contain data itself

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

TCP Strategies

Delayed ACKs

The next strategy we have mentioned before

TCP Strategies

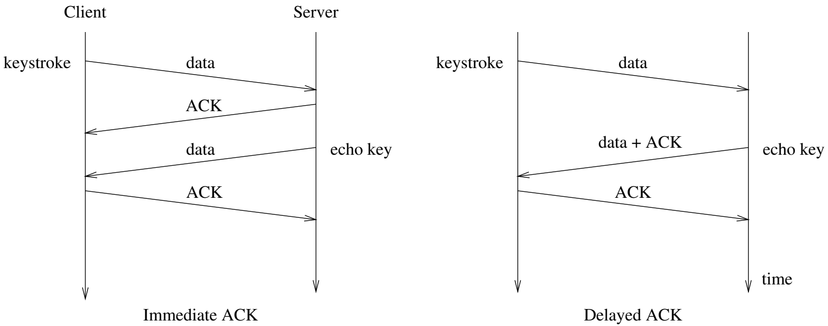
Delayed ACKs

The next strategy we have mentioned before

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

TCP Strategies

Delayed ACKs

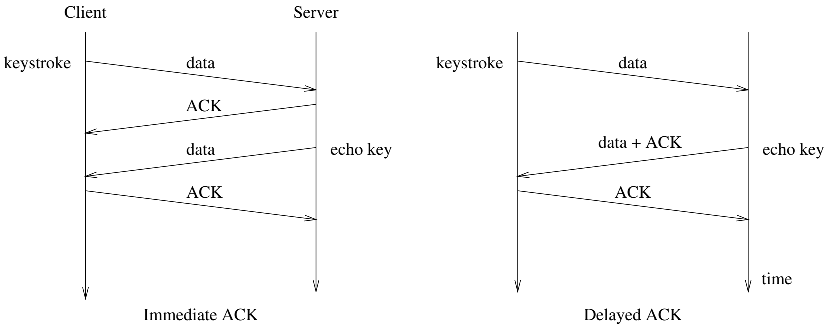


Immediate vs. delayed ACK

For example, when logged in to a remote terminal each keystroke is echoed back to your screen;

TCP Strategies

Delayed ACKs

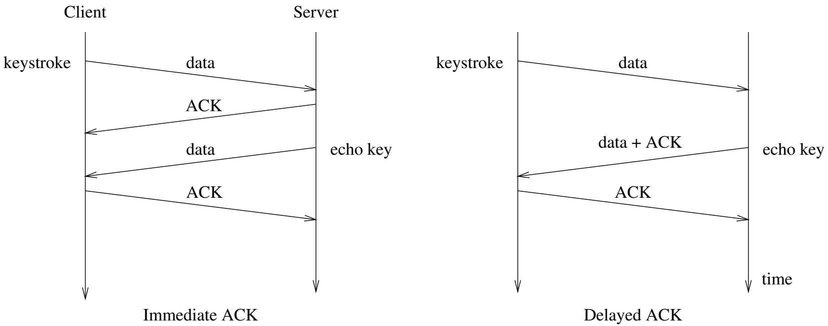


Immediate vs. delayed ACK

An immediate ACK would use four segments;

TCP Strategies

Delayed ACKs



Immediate vs. delayed ACK

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

TCP Strategies

Delayed ACKs

As far as the user is concerned, they see the keystroke echo in the same way, with no extra delay, but fewer segments are sent

TCP Strategies

Delayed ACKs

As far as the user is concerned, they see the keystroke echo in the same way, with no extra delay, but fewer segments are sent

It is important to reduce the traffic on a heavily loaded network

TCP Strategies

Delayed ACKs

As far as the user is concerned, they see the keystroke echo in the same way, with no extra delay, but fewer segments are sent

It is important to reduce the traffic on a heavily loaded network

It also reduces the chance of a lost segment

TCP Strategies

Delayed ACKs

By delaying, we might also be able to ACK more than one segment at a time

TCP Strategies

Delayed ACKs

By delaying, we might also be able to ACK more than one segment at a time

If we receive, say, two segments in a period we are delaying, we can simply ACK the last segment: this implicitly ACKs the previous two segments

TCP Strategies

Delayed ACKs

By delaying, we might also be able to ACK more than one segment at a time

If we receive, say, two segments in a period we are delaying, we can simply ACK the last segment: this implicitly ACKs the previous two segments

An ACK is actually about acknowledging bytes, not acknowledging segments, but will usually align with segments

TCP Strategies

Delayed ACKs

By delaying, we might also be able to ACK more than one segment at a time

If we receive, say, two segments in a period we are delaying, we can simply ACK the last segment: this implicitly ACKs the previous two segments

An ACK is actually about acknowledging bytes, not acknowledging segments, but will usually align with segments

So an ACK indicates which byte we are expecting next and says all previous bytes have been safely received

TCP Strategies

Delayed ACKs

By delaying, we might also be able to ACK more than one segment at a time

If we receive, say, two segments in a period we are delaying, we can simply ACK the last segment: this implicitly ACKs the previous two segments

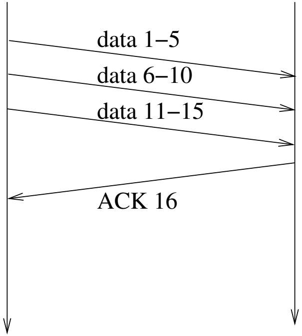
An ACK is actually about acknowledging bytes, not acknowledging segments, but will usually align with segments

So an ACK indicates which byte we are expecting next and says all previous bytes have been safely received

This reduces traffic again

TCP Strategies

Delayed ACKs



ACKing bytes received

ACKs acknowledge bytes received, not segments

TCP Strategies

Delayed ACKs

So how long to delay an ACK?

TCP Strategies

Delayed ACKs

So how long to delay an ACK?

If too long, the sender might think the segment was lost and resend

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 or multiple ACKed segments

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 or multiple ACKed segments

A typical implementation will delay for up to 200ms

TCP Strategies

Delayed ACKs

The TCP specification says you should send an ACK for at least every second full-sized segment and you *must not* delay for more than 500ms

TCP Strategies

Delayed ACKs

The TCP specification says you should send an ACK for at least every second full-sized segment and you *must not* delay for more than 500ms

This one of the many timers associated with TCP

TCP Strategies

Delayed ACKs

The TCP specification says you should send an ACK for at least every second full-sized segment and you *must not* delay for more than 500ms

This one of the many timers associated with TCP

Each time you receive a data segment the TCP software should 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

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 operating systems have a single global timer that fires every 200ms rather than a timer per segment received

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 operating systems have a single global timer that fires every 200ms rather than a timer per segment received

When the timer goes off, all unACKed segments are ACKed

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 operating systems have a single global timer that fires every 200ms rather than a timer per segment received

When the timer goes off, all unACKed segments are ACKed

Not so accurate as per-segment timers, but much easier to implement

TCP Strategies

Delayed ACKs

There is another rule concerning delayed ACKs

TCP Strategies

Delayed ACKs

There is another rule concerning delayed ACKs

If you get an out-of-order segment (its sequence number is not the one you are expecting next, e.g., a segment was possibly lost), you *must not* delay, but send an ACK immediately

TCP Strategies

Delayed ACKs

There is another rule concerning delayed ACKs

If you get an out-of-order segment (its sequence number is not the one you are expecting next, e.g., a segment was possibly lost), you *must not* delay, but send an ACK immediately

This might well be a duplicate ACK of one you sent earlier. This is to inform the sender as soon as possible that something might have gone wrong

TCP Strategies

Delayed ACKs

There is another rule concerning delayed ACKs

If you get an out-of-order segment (its sequence number is not the one you are expecting next, e.g., a segment was possibly lost), you *must not* delay, but send an ACK immediately

This might well be a duplicate ACK of one you sent earlier. This is to inform the sender as soon as possible that something might have gone wrong

Though the other end will wait for three duplicate ACKs just to be sure before resending

TCP Strategies

Nagle

Next strategy: when sending keystrokes (or other small data) over a network there is a lot of wasted bandwidth

TCP Strategies

Nagle

Next strategy: when sending keystrokes (or other small data) over a network there is a lot of wasted bandwidth

A keystroke could be 1 byte

TCP Strategies

Nagle

Next strategy: when sending keystrokes (or other small data) over a network there is a lot of wasted bandwidth

A keystroke could be 1 byte

This is sent in a TCP segment that has 20 bytes of header

TCP Strategies

Nagle

Next strategy: when sending keystrokes (or other small data) over a network there is a lot of wasted bandwidth

A keystroke could be 1 byte

This is sent in a TCP segment that has 20 bytes of header

This is contained in a IP datagram with 20 bytes of header

TCP Strategies

Nagle

Next strategy: when sending keystrokes (or other small data) over a network there is a lot of wasted bandwidth

A keystroke could be 1 byte

This is sent in a TCP segment that has 20 bytes of header

This is contained in a IP datagram with 20 bytes of header

And so on down the layers

TCP Strategies

Nagle

So we are sending (for the sake of argument) a 41 byte packet for each byte of data

TCP Strategies

Nagle

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*

TCP Strategies

Nagle

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

TCP Strategies

Nagle

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

TCP Strategies

Nagle

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 of the tinygram (client)

TCP Strategies

Nagle

Nagle's Algorithm:

> *a TCP connection can have only one outstanding un-ACKed small segment: no additional small segments can be sent until that ACK has been received*

TCP Strategies

Nagle

Nagle's Algorithm:

> *a TCP connection can have only one outstanding un-ACKed small segment: no additional small segments can be sent until that ACK has been received*

If you are sending tinygrams, only send one and wait until you get its ACK before sending any more

TCP Strategies

Nagle

Nagle's Algorithm:

> *a TCP connection can have only one outstanding un-ACKed small segment: no additional small segments can be sent until that ACK has been received*

If you are sending tinygrams, only send one and wait until you get its ACK before sending any more

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

TCP Strategies

Nagle

This segment can also be sent if either (a) you collect enough small segments to fill a MSS segment, or (b) they have collectively exceeded half the destination's advertised window size

TCP Strategies

Nagle

This segment can also be sent if either (a) you collect enough small segments to fill a MSS segment, or (b) they have collectively exceeded half the destination's advertised window size

This leaves open the definition of "small"

TCP Strategies

Nagle

This segment can also be sent if either (a) you collect enough small segments to fill a MSS segment, or (b) they have collectively exceeded half the destination's advertised 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

Note that when window scaling is in effect, “small” must be at least the size of the window scale factor, as we can’t advertise a window smaller than that

TCP Strategies

Nagle

Note that when window scaling is in effect, “small” must be at least the size of the window scale factor, as we can’t advertise a window smaller than that

But that won’t be a constraint until the scale is bigger than a segment, e.g., $2^{10} = 1024$, but $2^{11} = 2048 > 1500$

TCP Strategies

Nagle

This is a very simple strategy and reduces the number of tinygrams without introducing extra perceived delay (over that delay there is there already)

TCP Strategies

Nagle

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

TCP Strategies

Nagle

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 ACKs return more slowly, fewer tinygrams are sent

TCP Strategies

Nagle

Nagle can reduce the number of segments significantly when the network is heavily loaded

TCP Strategies

Nagle

Nagle can reduce the number of segments significantly when the network is heavily loaded

On the other hand, 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

TCP Strategies

Nagle

Nagle can reduce the number of segments significantly when the network is heavily loaded

On the other hand, 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