Concurrent Computing (Computer Networks)

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
⟨Daniel.Page@bristol.ac.uk⟩

March 14, 2016

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
   - the associated notes page may be pre-populated with extra, written explaination of material covered in lecture(s), plus
   - anything with a "grey'ed out" header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

Notes:

Continued from last lecture ...

Notes:

- ... so far so good, *but*, for example
  - Question: can we improve the efficiency of stop-and-wait?
  - Question: are other improvements/optimisations possible?
  - Question: how do we select τ, the time-out threshold?

Notes:

- ... so far so good, *but*, for example
  - Question: can we improve the efficiency of stop-and-wait?
  - Answer: yes, via **sliding-window** based on either
    - **go-back-n**, or
    - **selective-repeat**

    which *also* offer a neat solution for flow control.
  - Question: are other improvements/optimisations possible?
  - Question: how do we select τ, the time-out threshold?

Notes:

- ... so far so good, *but*, for example
  - Question: can we improve the efficiency of stop-and-wait?
  - Answer: yes, via **sliding-window** based on either
    - **go-back-n**, or
    - **selective-repeat**

    which *also* offer a neat solution for flow control.
  - Question: are other improvements/optimisations possible?
  - Answer: yes, lots, e.g.,
    - **cumulative ACKs**,
    - **selective ACKs**,
    - **delayed ACKs**,
    - ...
  - Question: how do we select τ, the time-out threshold?

Notes:

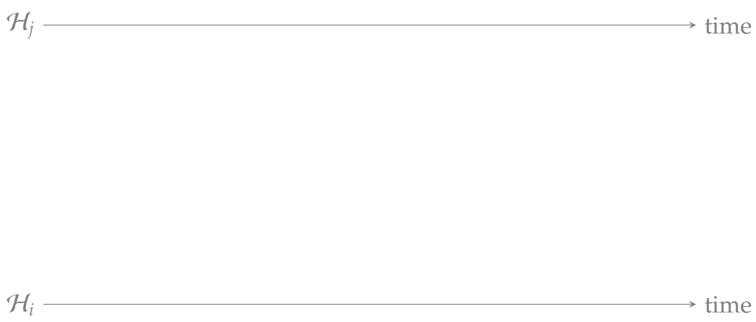
COMS20001 lecture: week #23

- ... so far so good, *but*, for example
  - Question: can we improve the efficiency of stop-and-wait?
  - Answer: yes, via **sliding-window** based on either
    - **go-back-n**, or
    - **selective-repeat**

    which *also* offer a neat solution for flow control.
  - Question: are other improvements/optimisations possible?
  - Answer: yes, lots, e.g.,
    - **cumulative ACKs**,
    - **selective ACKs**,
    - **delayed ACKs**,
    - ...
  - Question: how do we select τ, the time-out threshold?
  - Answer: using a moving average of measured RTT.

Notes:

TCP (1) – Sliding-window ARQ

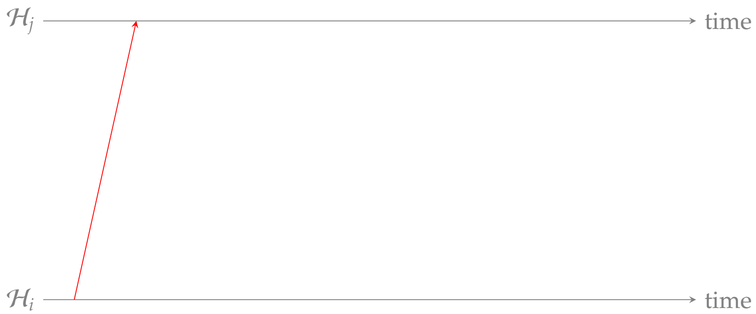


- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.

Notes:

- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

TCP (1) – Sliding-window ARQ

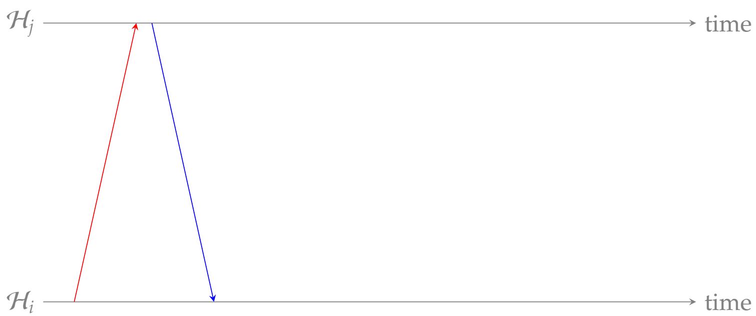


- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.

Notes:

- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

TCP (1) – Sliding-window ARQ

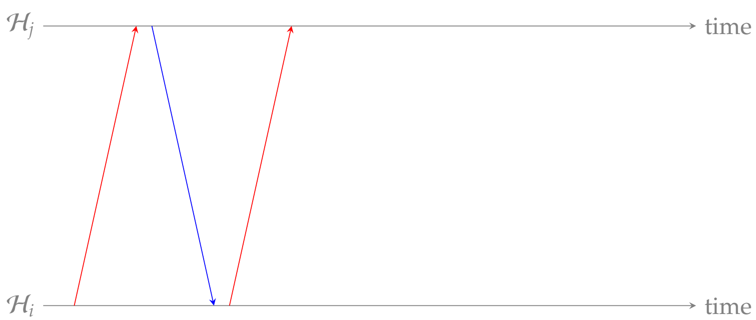


- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.

Notes:

- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

TCP (1) – Sliding-window ARQ

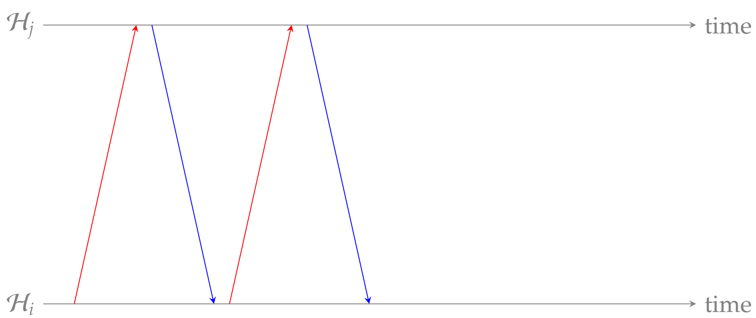


- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.

Notes:

- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

TCP (1) – Sliding-window ARQ

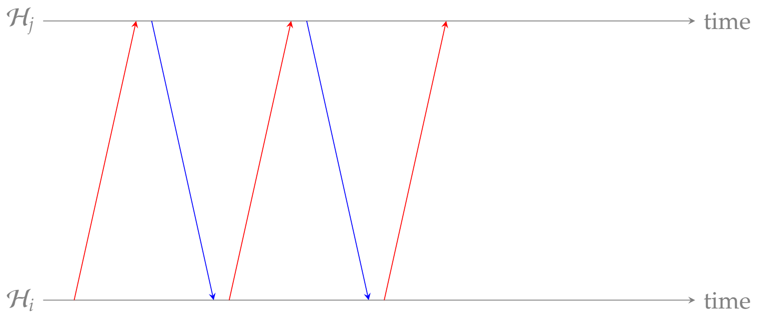


- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.

Notes:

- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

TCP (1) – Sliding-window ARQ

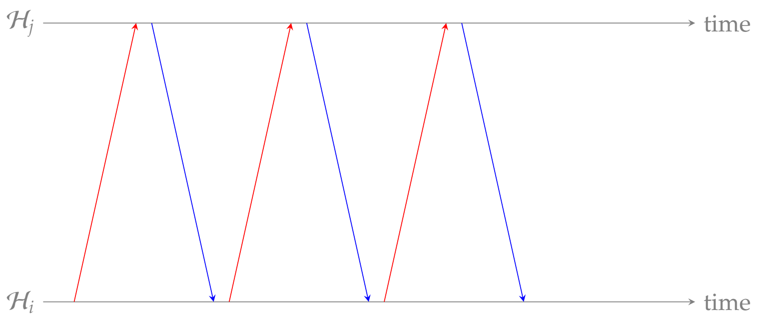


- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.

Notes:

- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

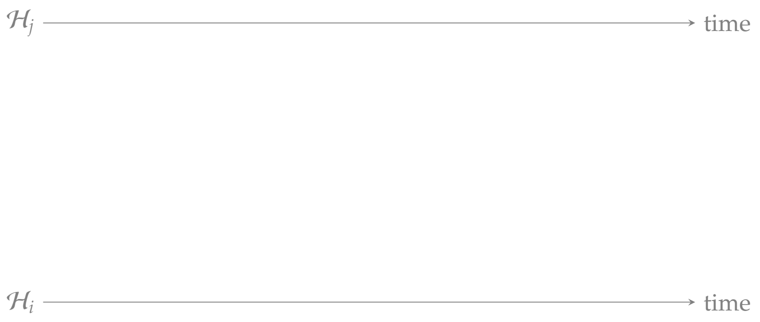
TCP (1) – Sliding-window ARQ



- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.

Notes:

- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

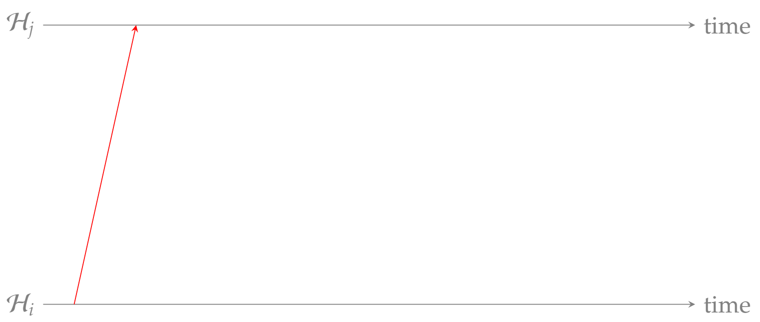


- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.
- Solution: generalise to $w > 1$ via **sliding-window**.

Notes:

- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

TCP (1) – Sliding-window ARQ

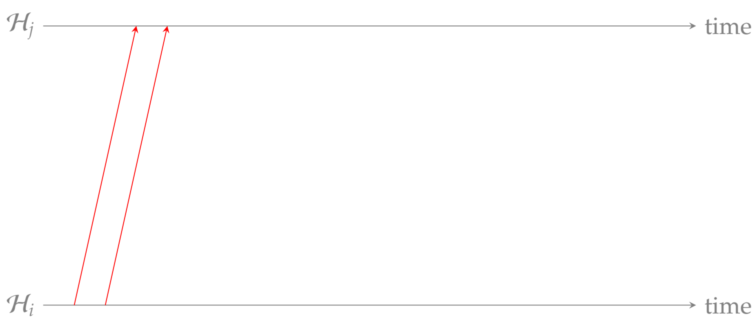


- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.
- Solution: generalise to $w > 1$ via **sliding-window**.

Notes:

- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

TCP (1) – Sliding-window ARQ

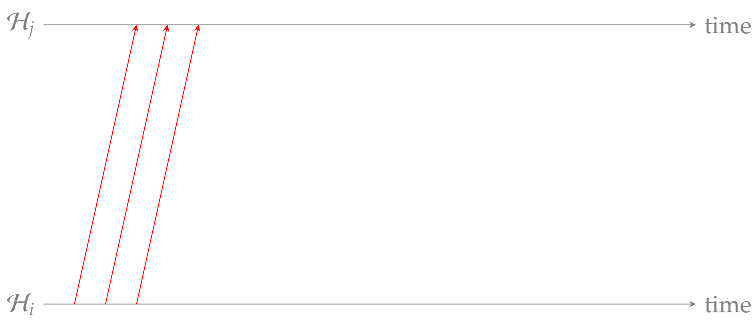


- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.
- Solution: generalise to $w > 1$ via **sliding-window**.

Notes:

- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

TCP (1) – Sliding-window ARQ



- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.
- Solution: generalise to $w > 1$ via **sliding-window**.

Notes:

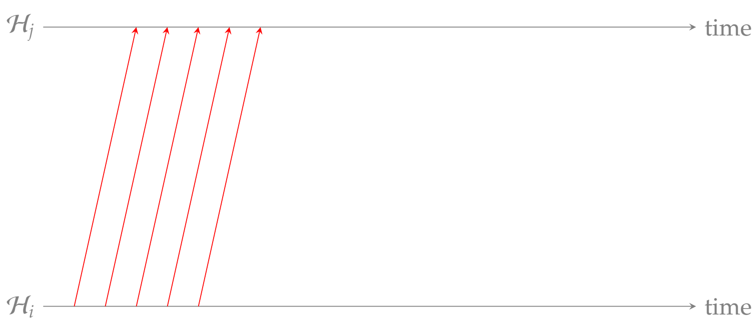
- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.
- Solution: generalise to $w > 1$ via **sliding-window**.

Notes:

- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

TCP (1) – Sliding-window ARQ



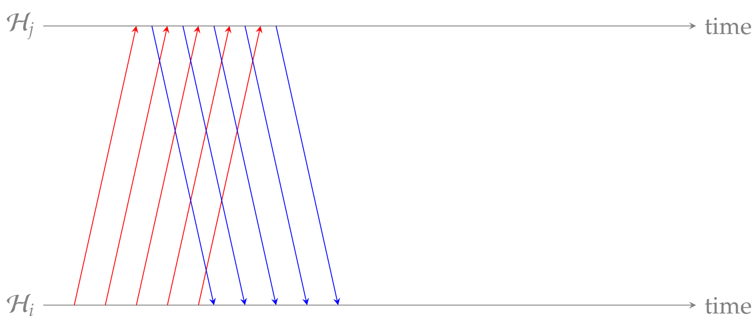
- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.
- Solution: generalise to $w > 1$ via **sliding-window**.

Daniel Page (Daniel.Page@bristol.ac.uk)
Concurrent Computing (Computer Networks) git # 3627080 @ 2016-03-11 University of BRISTOL

Notes:

- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

TCP (1) – Sliding-window ARQ

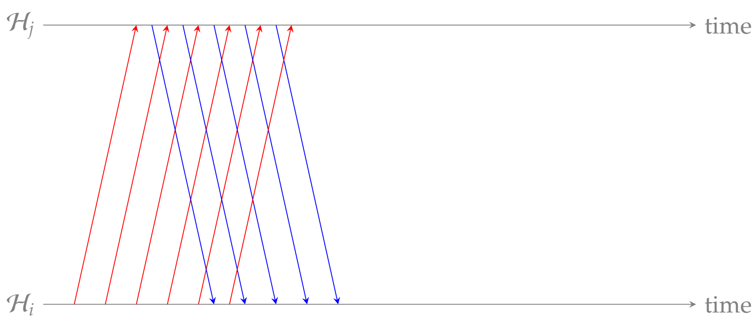


- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.
- Solution: generalise to $w > 1$ via **sliding-window**.

Notes:

- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

TCP (1) – Sliding-window ARQ



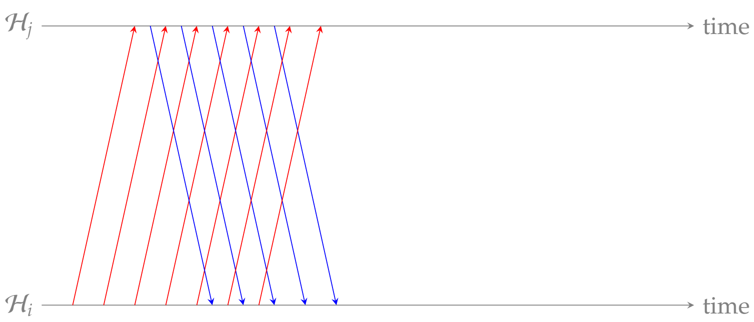
- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.
- Solution: generalise to $w > 1$ via **sliding-window**.

Notes:

- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

Daniel Page ⟨Daniel.Page@bristol.ac.uk⟩
Concurrent Computing (Computer Networks) git # 3627080 @ 2016-03-11
University of BRISTOL

TCP (1) – Sliding-window ARQ

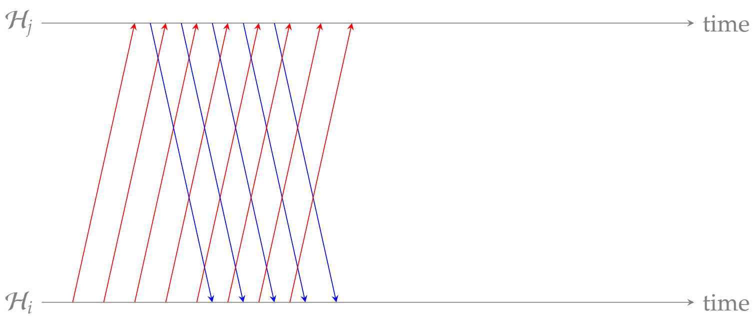


- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.
- Solution: generalise to $w > 1$ via **sliding-window**.

Notes:

- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

TCP (1) – Sliding-window ARQ

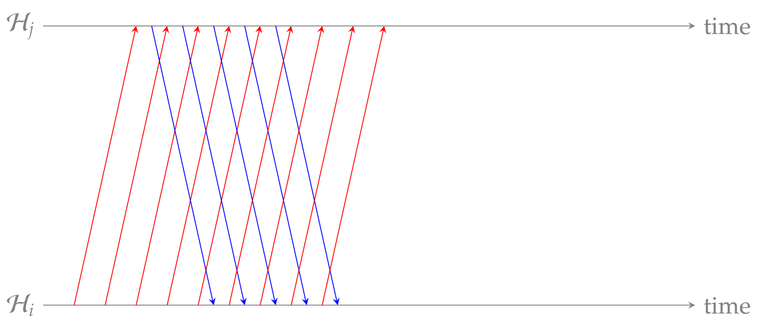


- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.
- Solution: generalise to $w > 1$ via **sliding-window**.

Notes:

- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

TCP (1) – Sliding-window ARQ

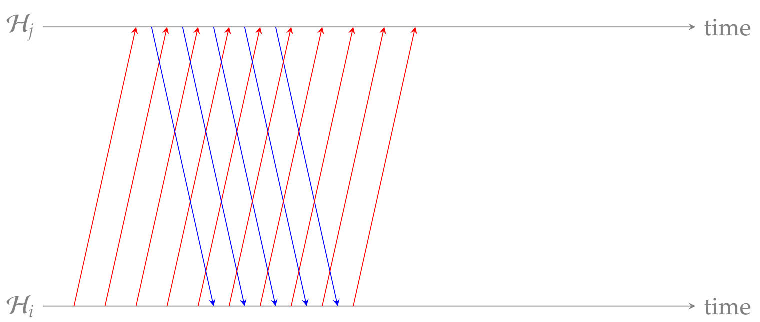


- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.
- Solution: generalise to $w > 1$ via **sliding-window**.

Notes:

- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

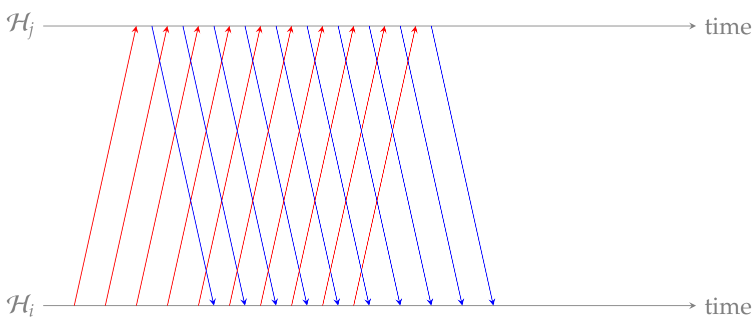
TCP (1) – Sliding-window ARQ



- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.
- Solution: generalise to $w > 1$ via **sliding-window**.

Notes:

- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

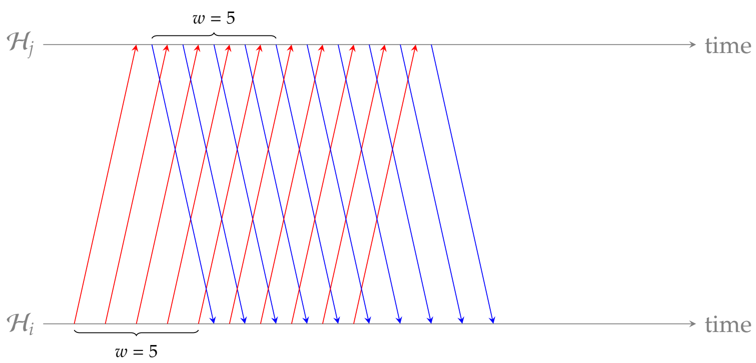


- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.
- Solution: generalise to $w > 1$ via **sliding-window**.

Notes:

- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

TCP (1) – Sliding-window ARQ

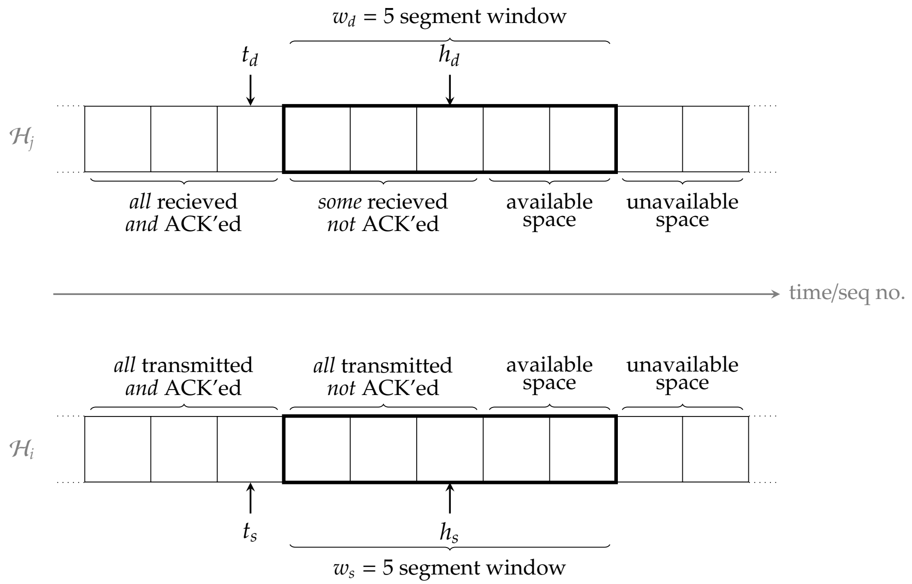


- Problem: stop-and-wait allows $w = 1$ un-ACK'ed segment
  - for a LAN this is fine, but
  - for an inter-network this is not fine, due to the high(er) bandwidth-latency product.
- Solution: generalise to $w > 1$ via **sliding-window**.

Notes:

- It might be obvious just by looking at the illustration, but what we've done here is basically applied the concept of pipelining. Just like a pipelined processor allows more than one instruction to be in-flight (e.g., one might be being executed, another decoded and a third fetched), we're operating the communication channel st. more than one un-ACK'ed segment can be in-flight.
- [9] gives a good overview of sliding window in the context of TCP.

TCP (2) – Sliding-window ARQ

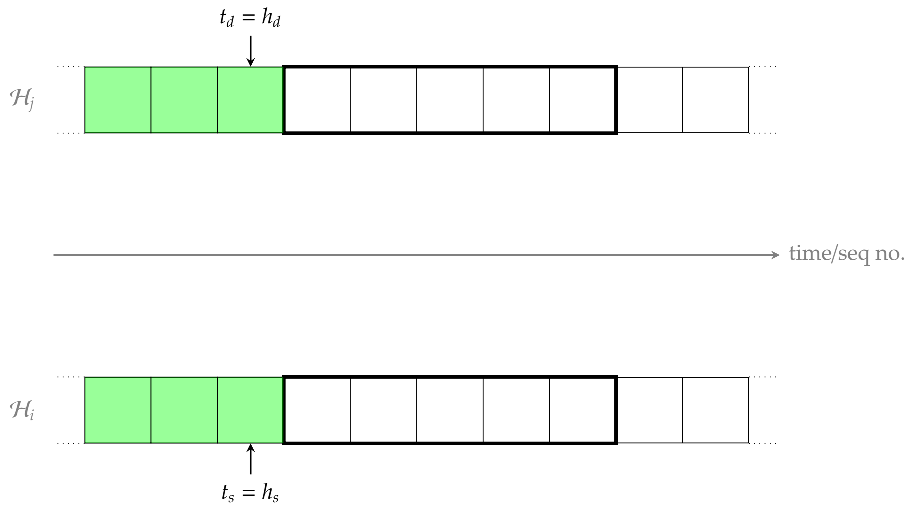


Notes:

- There are various formulations of the protocol: keep in mind that this is just *one* option. If you read a specific textbook, for example, the notation etc. may differ (a common case is where pointers are off-by-one in the sense they point at the first free segment rather than last used segment, the later of which is the approach here), and it might refer to transmitter and receiver rather than source and destination. Either way, the algorithm *should* basically still work the same way.
- Notice that **ACK clocking** is essentially still evident, in the sense that ACKs from the destination *still* control when transmission by the source can occur.
- The variables used, and the relationships between them, are as follows:
  - w_s and w_d are the window sizes used by source and destination respectively,
  - h_s is maintained by the source, and points at the last segment (i.e., highest sequence number) transmitted,
  - h_d is maintained by the destination, and points at the last segment (i.e., highest sequence number) received,
  - t_s is maintained by the source, and points at the first segment (i.e., lowest sequence number) for which an ACK has been received, and
  - t_d is maintained by the destination, and points at the first segment (i.e., lowest sequence number) for which an ACK has been transmitted.
- It isn't a *requirement* that $w_s = w_d$: they can differ, and in some cases this is in fact appropriate. However, there isn't much value in $w_d > w_s$ since in this case the receive buffer can never be filled.
- Each of the pointers (i.e., h_s etc.) is really a sequence number, so they will increase monotonically (until they wrap-around wrt. the number of bits allocated, e.g., at $2^{32} - 1$ back to 0).
- The source is certain all segments upto the t_s-th have been received (since an ACK was received). However, it cannot be certain about any of the $(t_s + 1)$-th upto the h_s-th segments (since no ACK was received yet).
- At the destination, a given segment between the $(t_d + 1)$-th and h_d-th will obviously have been received. However, not *all* may have been so: if the content is received out-of-order, this part of the buffer will have been filled non-contiguously (meaning there could be "gaps").
- More formally, we know the relation
  $$t_s \le t_d \le h_d \le h_s \le t_s + w_s$$
  must hold:
  1. $t_s \le t_d$ because the source cannot have received a higher ACK than transmitted by the destination,
  2. $t_d \le h_d$ because segments ACK'ed by the destination clearly cannot be beyond those it has actually received,
  3. $h_d \le h_s$ because the destination cannot have received a higher sequence number than transmitted by the source, and
  4. $h_s \le t_s + w_s$ because the source cannot have transmitted more than w_s segments (if none are ACK'ed).

TCP (2) – Sliding-window ARQ

- Example:

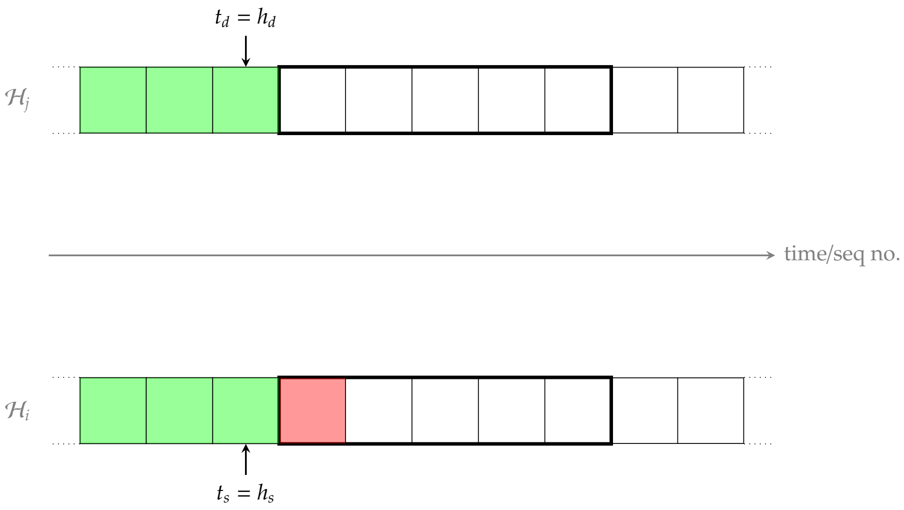


Notes:

- There are various formulations of the protocol: keep in mind that this is just *one* option. If you read a specific textbook, for example, the notation etc. may differ (a common case is where pointers are off-by-one in the sense they point at the first free segment rather than last used segment, the later of which is the approach here), and it might refer to transmitter and receiver rather than source and destination. Either way, the algorithm *should* basically still work the same way.
- Notice that **ACK clocking** is essentially still evident, in the sense that ACKs from the destination *still* control when transmission by the source can occur.
- The variables used, and the relationships between them, are as follows:
  - w_s and w_d are the window sizes used by source and destination respectively,
  - h_s is maintained by the source, and points at the last segment (i.e., highest sequence number) transmitted,
  - h_d is maintained by the destination, and points at the last segment (i.e., highest sequence number) received,
  - t_s is maintained by the source, and points at the first segment (i.e., lowest sequence number) for which an ACK has been received, and
  - t_d is maintained by the destination, and points at the first segment (i.e., lowest sequence number) for which an ACK has been transmitted.
- It isn't a *requirement* that $w_s = w_d$: they can differ, and in some cases this is in fact appropriate. However, there isn't much value in $w_d > w_s$ since in this case the receive buffer can never be filled.
- Each of the pointers (i.e., h_s etc.) is really a sequence number, so they will increase monotonically (until they wrap-around wrt. the number of bits allocated, e.g., at $2^{32} - 1$ back to 0).
- The source is certain all segments upto the t_s-th have been received (since an ACK was received). However, it cannot be certain about any of the $(t_s + 1)$-th upto the h_s-th segments (since no ACK was received yet).
- At the destination, a given segment between the $(t_d + 1)$-th and h_d-th will obviously have been received. However, not *all* may have been so: if the content is received out-of-order, this part of the buffer will have been filled non-contiguously (meaning there could be "gaps").
- More formally, we know the relation
  $$t_s \le t_d \le h_d \le h_s \le t_s + w_s$$
  must hold:
  1. $t_s \le t_d$ because the source cannot have received a higher ACK than transmitted by the destination,
  2. $t_d \le h_d$ because segments ACK'ed by the destination clearly cannot be beyond those it has actually received,
  3. $h_d \le h_s$ because the destination cannot have received a higher sequence number than transmitted by the source, and
  4. $h_s \le t_s + w_s$ because the source cannot have transmitted more than w_s segments (if none are ACK'ed).

TCP (2) – Sliding-window ARQ

- **Example**: application layer invokes `send` on source.

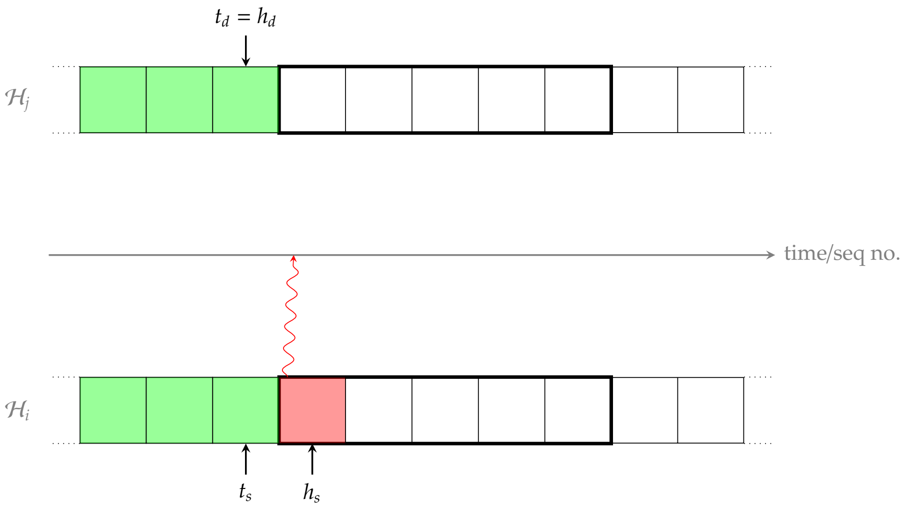


Notes:

- There are various formulations of the protocol: keep in mind that this is just *one* option. If you read a specific textbook, for example, the notation etc. may differ (a common case is where pointers are off-by-one in the sense they point at the first free segment rather than last used segment, the later of which is the approach here), and it might refer to transmitter and receiver rather than source and destination. Either way, the algorithm *should* basically still work the same way.
- Notice that **ACK clocking** is essentially still evident, in the sense that ACKs from the destination *still* control when transmission by the source can occur.
- The variables used, and the relationships between them, are as follows:
  - w_s and w_d are the window sizes used by source and destination respectively,
  - h_s is maintained by the source, and points at the last segment (i.e., highest sequence number) transmitted,
  - h_d is maintained by the destination, and points at the last segment (i.e., highest sequence number) received,
  - t_s is maintained by the source, and points at the first segment (i.e., lowest sequence number) for which an ACK has been received, and
  - t_d is maintained by the destination, and points at the first segment (i.e., lowest sequence number) for which an ACK has been transmitted.
- It isn't a *requirement* that $w_s = w_d$: they can differ, and in some cases this is in fact appropriate. However, there isn't much value in $w_d > w_s$ since in this case the receive buffer can never be filled.
- Each of the pointers (i.e., h_s etc.) is really a sequence number, so they will increase monotonically (until they wrap-around wrt. the number of bits allocated, e.g., at $2^{32} - 1$ back to 0).
- The source is certain all segments upto the t_s-th have been received (since an ACK was received). However, it cannot be certain about any of the $(t_s + 1)$-th upto the h_s-th segments (since no ACK was received yet).
- At the destination, a given segment between the $(t_d + 1)$-th and h_d-th will obviously have been received. However, not *all* may have been so: if the content is received out-of-order, this part of the buffer will have been filled non-contiguously (meaning there could be "gaps").
- More formally, we know the relation
$$t_s \le t_d \le h_d \le h_s \le t_s + w_s$$
must hold:
  1. $t_s \le t_d$ because the source cannot have received a higher ACK than transmitted by the destination,
  2. $t_d \le h_d$ because segments ACK'ed by the destination clearly cannot be beyond those it has actually received,
  3. $h_d \le h_s$ because the destination cannot have received a higher sequence number than transmitted by the source, and
  4. $h_s \le t_s + w_s$ because the source cannot have transmitted more than w_s segments (if none are ACK'ed).

TCP (2) – Sliding-window ARQ

- **Example**: update pointer and transmit segment.

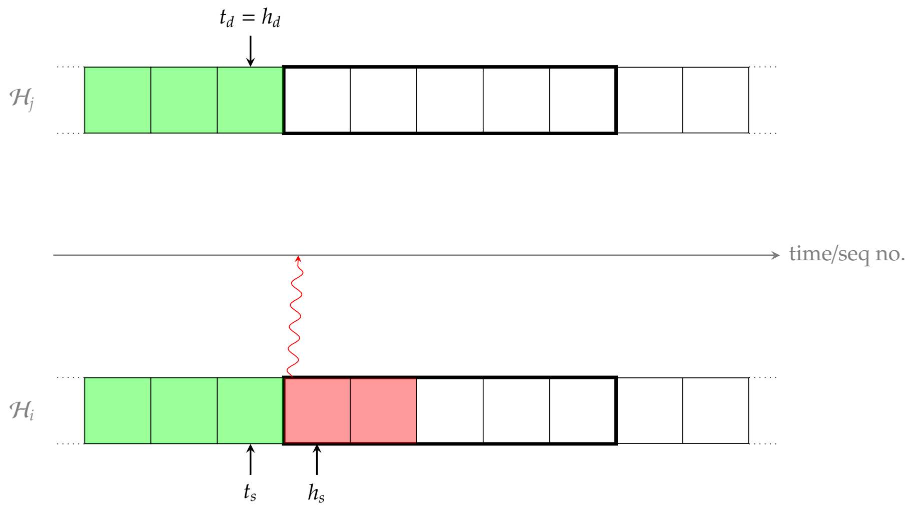


Notes:

- There are various formulations of the protocol: keep in mind that this is just *one* option. If you read a specific textbook, for example, the notation etc. may differ (a common case is where pointers are off-by-one in the sense they point at the first free segment rather than last used segment, the later of which is the approach here), and it might refer to transmitter and receiver rather than source and destination. Either way, the algorithm *should* basically still work the same way.
- Notice that **ACK clocking** is essentially still evident, in the sense that ACKs from the destination *still* control when transmission by the source can occur.
- The variables used, and the relationships between them, are as follows:
  - w_s and w_d are the window sizes used by source and destination respectively,
  - h_s is maintained by the source, and points at the last segment (i.e., highest sequence number) transmitted,
  - h_d is maintained by the destination, and points at the last segment (i.e., highest sequence number) received,
  - t_s is maintained by the source, and points at the first segment (i.e., lowest sequence number) for which an ACK has been received, and
  - t_d is maintained by the destination, and points at the first segment (i.e., lowest sequence number) for which an ACK has been transmitted.
- It isn't a *requirement* that $w_s = w_d$: they can differ, and in some cases this is in fact appropriate. However, there isn't much value in $w_d > w_s$ since in this case the receive buffer can never be filled.
- Each of the pointers (i.e., h_s etc.) is really a sequence number, so they will increase monotonically (until they wrap-around wrt. the number of bits allocated, e.g., at $2^{32} - 1$ back to 0).
- The source is certain all segments upto the t_s-th have been received (since an ACK was received). However, it cannot be certain about any of the $(t_s + 1)$-th upto the h_s-th segments (since no ACK was received yet).
- At the destination, a given segment between the $(t_d + 1)$-th and h_d-th will obviously have been received. However, not *all* may have been so: if the content is received out-of-order, this part of the buffer will have been filled non-contiguously (meaning there could be "gaps").
- More formally, we know the relation
$$t_s \le t_d \le h_d \le h_s \le t_s + w_s$$
must hold:
  1. $t_s \le t_d$ because the source cannot have received a higher ACK than transmitted by the destination,
  2. $t_d \le h_d$ because segments ACK'ed by the destination clearly cannot be beyond those it has actually received,
  3. $h_d \le h_s$ because the destination cannot have received a higher sequence number than transmitted by the source, and
  4. $h_s \le t_s + w_s$ because the source cannot have transmitted more than w_s segments (if none are ACK'ed).

TCP (2) – Sliding-window ARQ

- **Example**: application layer invokes `send` on source.



Notes:

- There are various formulations of the protocol: keep in mind that this is just *one* option. If you read a specific textbook, for example, the notation etc. may differ (a common case is where pointers are off-by-one in the sense they point at the first free segment rather than last used segment, the later of which is the approach here), and it might refer to transmitter and receiver rather than source and destination. Either way, the algorithm *should* basically still work the same way.
- Notice that **ACK clocking** is essentially still evident, in the sense that ACKs from the destination *still* control when transmission by the source can occur.
- The variables used, and the relationships between them, are as follows:
  - w_s and w_d are the window sizes used by source and destination respectively,
  - h_s is maintained by the source, and points at the last segment (i.e., highest sequence number) transmitted,
  - h_d is maintained by the destination, and points at the last segment (i.e., highest sequence number) received,
  - t_s is maintained by the source, and points at the first segment (i.e., lowest sequence number) for which an ACK has been received, and
  - t_d is maintained by the destination, and points at the first segment (i.e., lowest sequence number) for which an ACK has been transmitted.
- It isn't a *requirement* that $w_s = w_d$: they can differ, and in some cases this is in fact appropriate. However, there isn't much value in $w_d > w_s$ since in this case the receive buffer can never be filled.
- Each of the pointers (i.e., h_s etc.) is really a sequence number, so they will increase monotonically (until they wrap-around wrt. the number of bits allocated, e.g., at $2^{32} - 1$ back to 0).
- The source is certain all segments upto the t_s-th have been received (since an ACK was received). However, it cannot be certain about any of the $(t_s + 1)$-th upto the h_s-th segments (since no ACK was received yet).
- At the destination, a given segment between the $(t_d + 1)$-th and h_d-th will obviously have been received. However, not *all* may have been so: if the content is received out-of-order, this part of the buffer will have been filled non-contiguously (meaning there could be "gaps").
- More formally, we know the relation

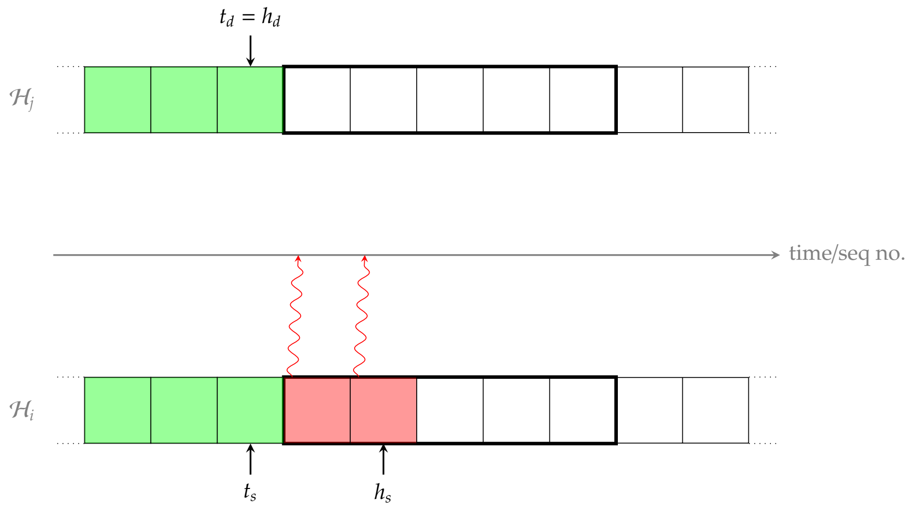
$$t_s \le t_d \le h_d \le h_s \le t_s + w_s$$

must hold:

1. $t_s \le t_d$ because the source cannot have received a higher ACK than transmitted by the destination,
2. $t_d \le h_d$ because segments ACK'ed by the destination clearly cannot be beyond those it has actually received,
3. $h_d \le h_s$ because the destination cannot have received a higher sequence number than transmitted by the source, and
4. $h_s \le t_s + w_s$ because the source cannot have transmitted more than w_s segments (if none are ACK'ed).

TCP (2) – Sliding-window ARQ

- **Example**: update pointer and transmit segment.



Notes:

- There are various formulations of the protocol: keep in mind that this is just *one* option. If you read a specific textbook, for example, the notation etc. may differ (a common case is where pointers are off-by-one in the sense they point at the first free segment rather than last used segment, the later of which is the approach here), and it might refer to transmitter and receiver rather than source and destination. Either way, the algorithm *should* basically still work the same way.
- Notice that **ACK clocking** is essentially still evident, in the sense that ACKs from the destination *still* control when transmission by the source can occur.
- The variables used, and the relationships between them, are as follows:
  - w_s and w_d are the window sizes used by source and destination respectively,
  - h_s is maintained by the source, and points at the last segment (i.e., highest sequence number) transmitted,
  - h_d is maintained by the destination, and points at the last segment (i.e., highest sequence number) received,
  - t_s is maintained by the source, and points at the first segment (i.e., lowest sequence number) for which an ACK has been received, and
  - t_d is maintained by the destination, and points at the first segment (i.e., lowest sequence number) for which an ACK has been transmitted.
- It isn't a *requirement* that $w_s = w_d$: they can differ, and in some cases this is in fact appropriate. However, there isn't much value in $w_d > w_s$ since in this case the receive buffer can never be filled.
- Each of the pointers (i.e., h_s etc.) is really a sequence number, so they will increase monotonically (until they wrap-around wrt. the number of bits allocated, e.g., at $2^{32} - 1$ back to 0).
- The source is certain all segments upto the t_s-th have been received (since an ACK was received). However, it cannot be certain about any of the $(t_s + 1)$-th upto the h_s-th segments (since no ACK was received yet).
- At the destination, a given segment between the $(t_d + 1)$-th and h_d-th will obviously have been received. However, not *all* may have been so: if the content is received out-of-order, this part of the buffer will have been filled non-contiguously (meaning there could be "gaps").
- More formally, we know the relation

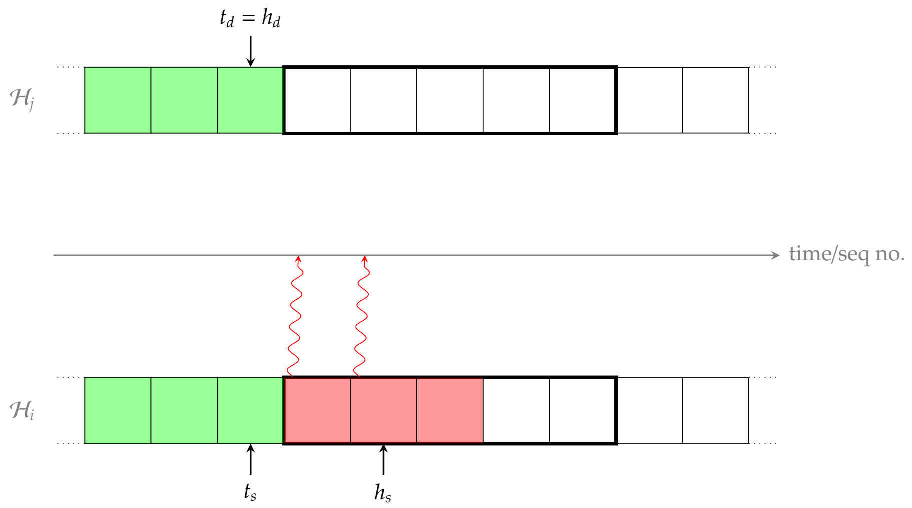
$$t_s \le t_d \le h_d \le h_s \le t_s + w_s$$

must hold:

1. $t_s \le t_d$ because the source cannot have received a higher ACK than transmitted by the destination,
2. $t_d \le h_d$ because segments ACK'ed by the destination clearly cannot be beyond those it has actually received,
3. $h_d \le h_s$ because the destination cannot have received a higher sequence number than transmitted by the source, and
4. $h_s \le t_s + w_s$ because the source cannot have transmitted more than w_s segments (if none are ACK'ed).

TCP (2) – Sliding-window ARQ

- Example: application layer invokes `send` on source.



Notes:

- There are various formulations of the protocol: keep in mind that this is just *one* option. If you read a specific textbook, for example, the notation etc. may differ (a common case is where pointers are off-by-one in the sense they point at the first free segment rather than last used segment, the later of which is the approach here), and it might refer to transmitter and receiver rather than source and destination. Either way, the algorithm *should* basically still work the same way.
- Notice that **ACK clocking** is essentially still evident, in the sense that ACKs from the destination *still* control when transmission by the source can occur.
- The variables used, and the relationships between them, are as follows:
  - w_s and w_d are the window sizes used by source and destination respectively,
  - h_s is maintained by the source, and points at the last segment (i.e., highest sequence number) transmitted,
  - h_d is maintained by the destination, and points at the last segment (i.e., highest sequence number) received,
  - t_s is maintained by the source, and points at the first segment (i.e., lowest sequence number) for which an ACK has been received, and
  - t_d is maintained by the destination, and points at the first segment (i.e., lowest sequence number) for which an ACK has been transmitted.
- It isn't a *requirement* that $w_s = w_d$: they can differ, and in some cases this is in fact appropriate. However, there isn't much value in $w_d > w_s$ since in this case the receive buffer can never be filled.
- Each of the pointers (i.e., h_s etc.) is really a sequence number, so they will increase monotonically (until they wrap-around wrt. the number of bits allocated, e.g., at $2^{32} - 1$ back to 0).
- The source is certain all segments upto the t_s-th have been received (since an ACK was received). However, it cannot be certain about any of the $(t_s + 1)$-th upto the h_s-th segments (since no ACK was received yet).
- At the destination, a given segment between the $(t_d + 1)$-th and h_d-th will obviously have been received. However, not *all* may have been so: if the content is received out-of-order, this part of the buffer will have been filled non-contiguously (meaning there could be "gaps").
- More formally, we know the relation

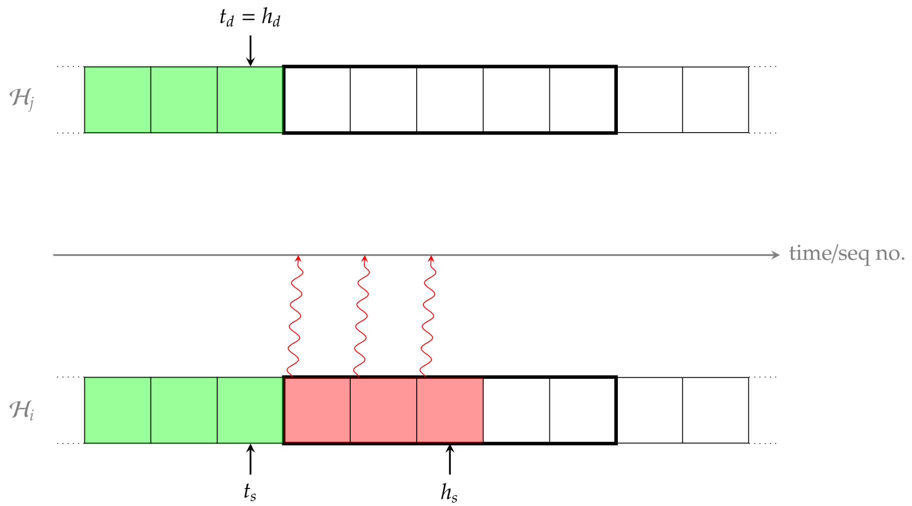
$$t_s \le t_d \le h_d \le h_s \le t_s + w_s$$

  must hold:

  1. $t_s \le t_d$ because the source cannot have received a higher ACK than transmitted by the destination,
  2. $t_d \le h_d$ because segments ACK'ed by the destination clearly cannot be beyond those it has actually received,
  3. $h_d \le h_s$ because the destination cannot have received a higher sequence number than transmitted by the source, and
  4. $h_s \le t_s + w_s$ because the source cannot have transmitted more than w_s segments (if none are ACK'ed).

TCP (2) – Sliding-window ARQ

- Example: update pointer and transmit segment.



Notes:

- There are various formulations of the protocol: keep in mind that this is just *one* option. If you read a specific textbook, for example, the notation etc. may differ (a common case is where pointers are off-by-one in the sense they point at the first free segment rather than last used segment, the later of which is the approach here), and it might refer to transmitter and receiver rather than source and destination. Either way, the algorithm *should* basically still work the same way.
- Notice that **ACK clocking** is essentially still evident, in the sense that ACKs from the destination *still* control when transmission by the source can occur.
- The variables used, and the relationships between them, are as follows:
  - w_s and w_d are the window sizes used by source and destination respectively,
  - h_s is maintained by the source, and points at the last segment (i.e., highest sequence number) transmitted,
  - h_d is maintained by the destination, and points at the last segment (i.e., highest sequence number) received,
  - t_s is maintained by the source, and points at the first segment (i.e., lowest sequence number) for which an ACK has been received, and
  - t_d is maintained by the destination, and points at the first segment (i.e., lowest sequence number) for which an ACK has been transmitted.
- It isn't a *requirement* that $w_s = w_d$: they can differ, and in some cases this is in fact appropriate. However, there isn't much value in $w_d > w_s$ since in this case the receive buffer can never be filled.
- Each of the pointers (i.e., h_s etc.) is really a sequence number, so they will increase monotonically (until they wrap-around wrt. the number of bits allocated, e.g., at $2^{32} - 1$ back to 0).
- The source is certain all segments upto the t_s-th have been received (since an ACK was received). However, it cannot be certain about any of the $(t_s + 1)$-th upto the h_s-th segments (since no ACK was received yet).
- At the destination, a given segment between the $(t_d + 1)$-th and h_d-th will obviously have been received. However, not *all* may have been so: if the content is received out-of-order, this part of the buffer will have been filled non-contiguously (meaning there could be "gaps").
- More formally, we know the relation

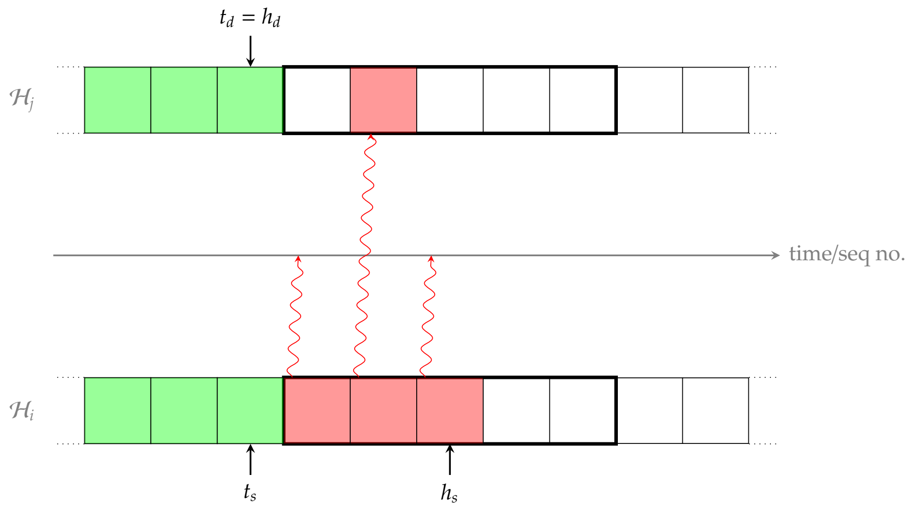
$$t_s \le t_d \le h_d \le h_s \le t_s + w_s$$

  must hold:

  1. $t_s \le t_d$ because the source cannot have received a higher ACK than transmitted by the destination,
  2. $t_d \le h_d$ because segments ACK'ed by the destination clearly cannot be beyond those it has actually received,
  3. $h_d \le h_s$ because the destination cannot have received a higher sequence number than transmitted by the source, and
  4. $h_s \le t_s + w_s$ because the source cannot have transmitted more than w_s segments (if none are ACK'ed).

TCP (2) – Sliding-window ARQ

- **Example**: segment received.



Notes:

- There are various formulations of the protocol: keep in mind that this is just *one* option. If you read a specific textbook, for example, the notation etc. may differ (a common case is where pointers are off-by-one in the sense they point at the first free segment rather than last used segment, the later of which is the approach here), and it might refer to transmitter and receiver rather than source and destination. Either way, the algorithm *should* basically still work the same way.
- Notice that **ACK clocking** is essentially still evident, in the sense that ACKs from the destination *still* control when transmission by the source can occur.
- The variables used, and the relationships between them, are as follows:
  - w_s and w_d are the window sizes used by source and destination respectively,
  - h_s is maintained by the source, and points at the last segment (i.e., highest sequence number) transmitted,
  - h_d is maintained by the destination, and points at the last segment (i.e., highest sequence number) received,
  - t_s is maintained by the source, and points at the first segment (i.e., lowest sequence number) for which an ACK has been received, and
  - t_d is maintained by the destination, and points at the first segment (i.e., lowest sequence number) for which an ACK has been transmitted.
- It isn't a *requirement* that $w_s = w_d$: they can differ, and in some cases this is in fact appropriate. However, there isn't much value in $w_d > w_s$ since in this case the receive buffer can never be filled.
- Each of the pointers (i.e., h_s etc.) is really a sequence number, so they will increase monotonically (until they wrap-around wrt. the number of bits allocated, e.g., at $2^{32} - 1$ back to 0).
- The source is certain all segments upto the t_s-th have been received (since an ACK was received). However, it cannot be certain about any of the $(t_s + 1)$-th upto the h_s-th segments (since no ACK was received yet).
- At the destination, a given segment between the $(t_d + 1)$-th and h_d-th will obviously have been received. However, not *all* may have been so: if the content is received out-of-order, this part of the buffer will have been filled non-contiguously (meaning there could be "gaps").
- More formally, we know the relation

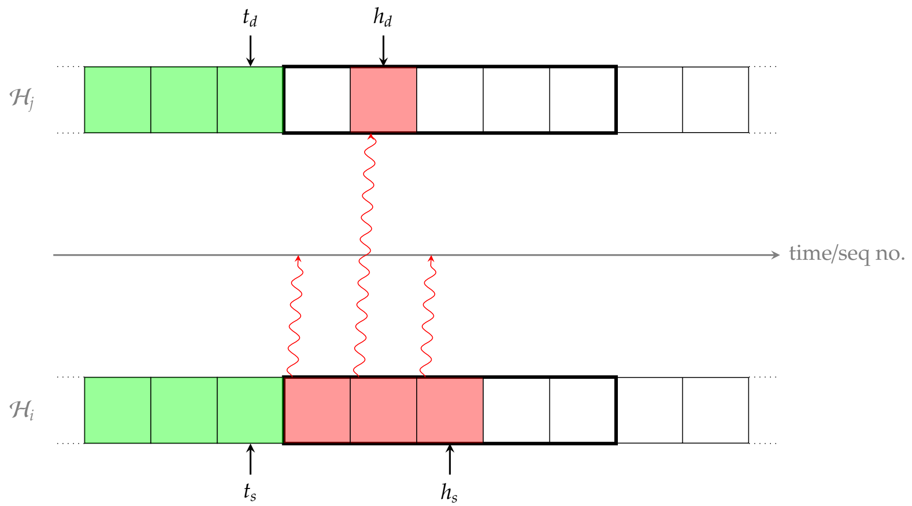
$$t_s \leq t_d \leq h_d \leq h_s \leq t_s + w_s$$

  must hold:

  1. $t_s \leq t_d$ because the source cannot have received a higher ACK than transmitted by the destination,
  2. $t_d \leq h_d$ because segments ACK'ed by the destination clearly cannot be beyond those it has actually received,
  3. $h_d \leq h_s$ because the destination cannot have received a higher sequence number than transmitted by the source, and
  4. $h_s \leq t_s + w_s$ because the source cannot have transmitted more than w_s segments (if none are ACK'ed).

TCP (2) – Sliding-window ARQ

- **Example**: update pointer.



Notes:

- There are various formulations of the protocol: keep in mind that this is just *one* option. If you read a specific textbook, for example, the notation etc. may differ (a common case is where pointers are off-by-one in the sense they point at the first free segment rather than last used segment, the later of which is the approach here), and it might refer to transmitter and receiver rather than source and destination. Either way, the algorithm *should* basically still work the same way.
- Notice that **ACK clocking** is essentially still evident, in the sense that ACKs from the destination *still* control when transmission by the source can occur.
- The variables used, and the relationships between them, are as follows:
  - w_s and w_d are the window sizes used by source and destination respectively,
  - h_s is maintained by the source, and points at the last segment (i.e., highest sequence number) transmitted,
  - h_d is maintained by the destination, and points at the last segment (i.e., highest sequence number) received,
  - t_s is maintained by the source, and points at the first segment (i.e., lowest sequence number) for which an ACK has been received, and
  - t_d is maintained by the destination, and points at the first segment (i.e., lowest sequence number) for which an ACK has been transmitted.
- It isn't a *requirement* that $w_s = w_d$: they can differ, and in some cases this is in fact appropriate. However, there isn't much value in $w_d > w_s$ since in this case the receive buffer can never be filled.
- Each of the pointers (i.e., h_s etc.) is really a sequence number, so they will increase monotonically (until they wrap-around wrt. the number of bits allocated, e.g., at $2^{32} - 1$ back to 0).
- The source is certain all segments upto the t_s-th have been received (since an ACK was received). However, it cannot be certain about any of the $(t_s + 1)$-th upto the h_s-th segments (since no ACK was received yet).
- At the destination, a given segment between the $(t_d + 1)$-th and h_d-th will obviously have been received. However, not *all* may have been so: if the content is received out-of-order, this part of the buffer will have been filled non-contiguously (meaning there could be "gaps").
- More formally, we know the relation

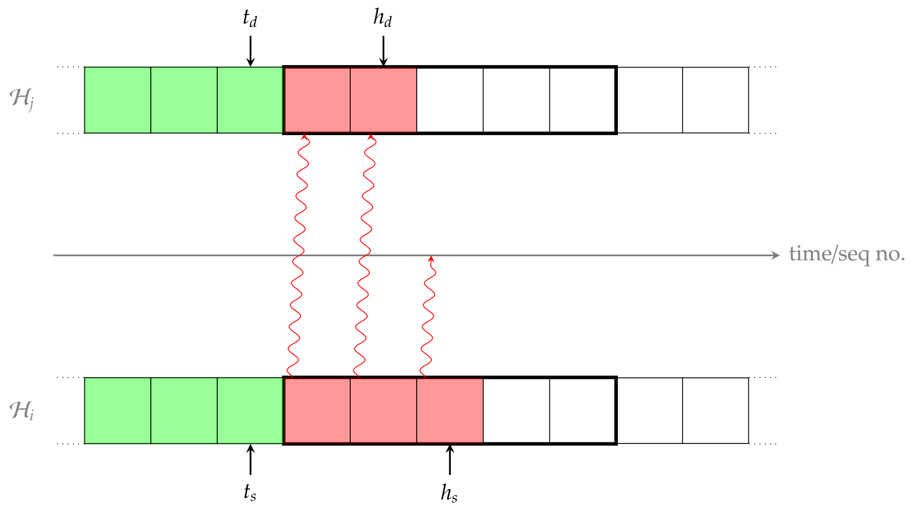
$$t_s \leq t_d \leq h_d \leq h_s \leq t_s + w_s$$

  must hold:

  1. $t_s \leq t_d$ because the source cannot have received a higher ACK than transmitted by the destination,
  2. $t_d \leq h_d$ because segments ACK'ed by the destination clearly cannot be beyond those it has actually received,
  3. $h_d \leq h_s$ because the destination cannot have received a higher sequence number than transmitted by the source, and
  4. $h_s \leq t_s + w_s$ because the source cannot have transmitted more than w_s segments (if none are ACK'ed).

TCP (2) – Sliding-window ARQ

- Example: segment received.



Notes:

- There are various formulations of the protocol: keep in mind that this is just *one* option. If you read a specific textbook, for example, the notation etc. may differ (a common case is where pointers are off-by-one in the sense they point at the first free segment rather than last used segment, the later of which is the approach here), and it might refer to transmitter and receiver rather than source and destination. Either way, the algorithm *should* basically still work the same way.
- Notice that **ACK clocking** is essentially still evident, in the sense that ACKs from the destination *still* control when transmission by the source can occur.
- The variables used, and the relationships between them, are as follows:
  - w_s and w_d are the window sizes used by source and destination respectively,
  - h_s is maintained by the source, and points at the last segment (i.e., highest sequence number) transmitted,
  - h_d is maintained by the destination, and points at the last segment (i.e., highest sequence number) received,
  - t_s is maintained by the source, and points at the first segment (i.e., lowest sequence number) for which an ACK has been received, and
  - t_d is maintained by the destination, and points at the first segment (i.e., lowest sequence number) for which an ACK has been transmitted.
- It isn't a *requirement* that $w_s = w_d$: they can differ, and in some cases this is in fact appropriate. However, there isn't much value in $w_d > w_s$ since in this case the receive buffer can never be filled.
- Each of the pointers (i.e., h_s etc.) is really a sequence number, so they will increase monotonically (until they wrap-around wrt. the number of bits allocated, e.g., at $2^{32} - 1$ back to 0).
- The source is certain all segments upto the t_s-th have been received (since an ACK was received). However, it cannot be certain about any of the $(t_s + 1)$-th upto the h_s-th segments (since no ACK was received yet).
- At the destination, a given segment between the $(t_d + 1)$-th and h_d-th will obviously have been received. However, not *all* may have been so: if the content is received out-of-order, this part of the buffer will have been filled non-contiguously (meaning there could be "gaps").
- More formally, we know the relation

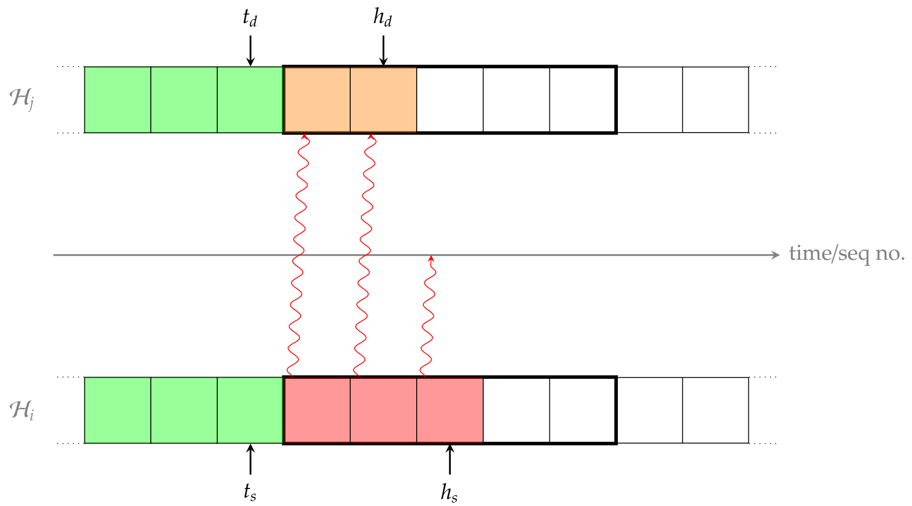
  $$t_s \le t_d \le h_d \le h_s \le t_s + w_s$$

  must hold:

  1. $t_s \le t_d$ because the source cannot have received a higher ACK than transmitted by the destination,
  2. $t_d \le h_d$ because segments ACK'ed by the destination clearly cannot be beyond those it has actually received,
  3. $h_d \le h_s$ because the destination cannot have received a higher sequence number than transmitted by the source, and
  4. $h_s \le t_s + w_s$ because the source cannot have transmitted more than w_s segments (if none are ACK'ed).

TCP (2) – Sliding-window ARQ

- Example: application layer invokes `recv` on destination.



Notes:

- There are various formulations of the protocol: keep in mind that this is just *one* option. If you read a specific textbook, for example, the notation etc. may differ (a common case is where pointers are off-by-one in the sense they point at the first free segment rather than last used segment, the later of which is the approach here), and it might refer to transmitter and receiver rather than source and destination. Either way, the algorithm *should* basically still work the same way.
- Notice that **ACK clocking** is essentially still evident, in the sense that ACKs from the destination *still* control when transmission by the source can occur.
- The variables used, and the relationships between them, are as follows:
  - w_s and w_d are the window sizes used by source and destination respectively,
  - h_s is maintained by the source, and points at the last segment (i.e., highest sequence number) transmitted,
  - h_d is maintained by the destination, and points at the last segment (i.e., highest sequence number) received,
  - t_s is maintained by the source, and points at the first segment (i.e., lowest sequence number) for which an ACK has been received, and
  - t_d is maintained by the destination, and points at the first segment (i.e., lowest sequence number) for which an ACK has been transmitted.
- It isn't a *requirement* that $w_s = w_d$: they can differ, and in some cases this is in fact appropriate. However, there isn't much value in $w_d > w_s$ since in this case the receive buffer can never be filled.
- Each of the pointers (i.e., h_s etc.) is really a sequence number, so they will increase monotonically (until they wrap-around wrt. the number of bits allocated, e.g., at $2^{32} - 1$ back to 0).
- The source is certain all segments upto the t_s-th have been received (since an ACK was received). However, it cannot be certain about any of the $(t_s + 1)$-th upto the h_s-th segments (since no ACK was received yet).
- At the destination, a given segment between the $(t_d + 1)$-th and h_d-th will obviously have been received. However, not *all* may have been so: if the content is received out-of-order, this part of the buffer will have been filled non-contiguously (meaning there could be "gaps").
- More formally, we know the relation

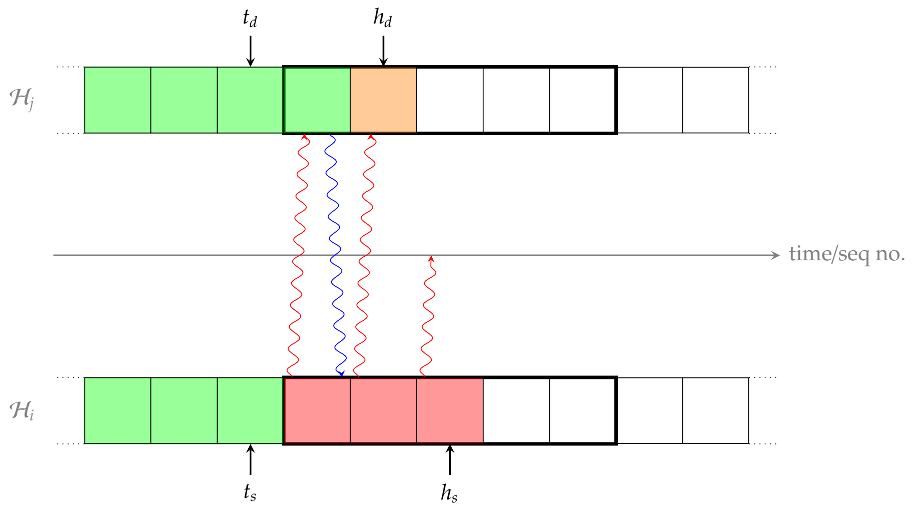
  $$t_s \le t_d \le h_d \le h_s \le t_s + w_s$$

  must hold:

  1. $t_s \le t_d$ because the source cannot have received a higher ACK than transmitted by the destination,
  2. $t_d \le h_d$ because segments ACK'ed by the destination clearly cannot be beyond those it has actually received,
  3. $h_d \le h_s$ because the destination cannot have received a higher sequence number than transmitted by the source, and
  4. $h_s \le t_s + w_s$ because the source cannot have transmitted more than w_s segments (if none are ACK'ed).

TCP (2) – Sliding-window ARQ

- **Example**: transmit ACK.

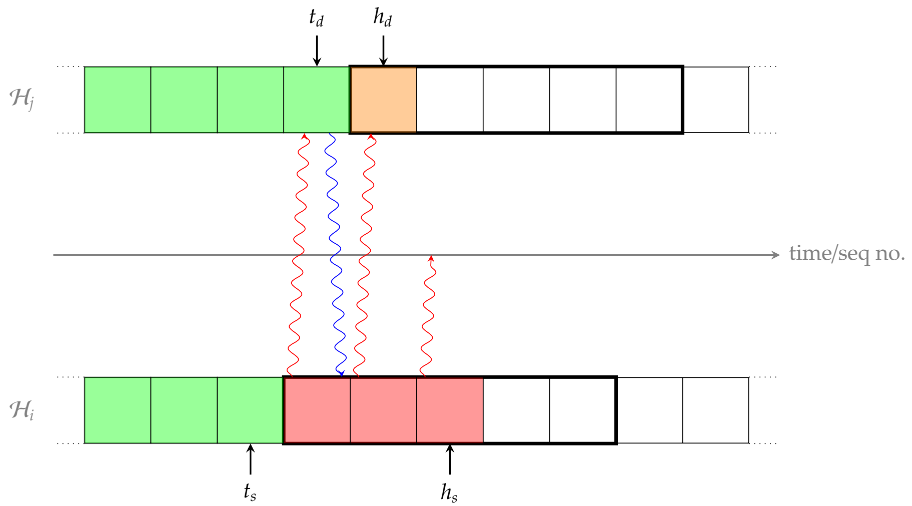


Notes:

- There are various formulations of the protocol: keep in mind that this is just *one* option. If you read a specific textbook, for example, the notation etc. may differ (a common case is where pointers are off-by-one in the sense they point at the first free segment rather than last used segment, the later of which is the approach here), and it might refer to transmitter and receiver rather than source and destination. Either way, the algorithm *should* basically still work the same way.
- Notice that **ACK clocking** is essentially still evident, in the sense that ACKs from the destination *still* control when transmission by the source can occur.
- The variables used, and the relationships between them, are as follows:
  - w_s and w_d are the window sizes used by source and destination respectively,
  - h_s is maintained by the source, and points at the last segment (i.e., highest sequence number) transmitted,
  - h_d is maintained by the destination, and points at the last segment (i.e., highest sequence number) received,
  - t_s is maintained by the source, and points at the first segment (i.e., lowest sequence number) for which an ACK has been received, and
  - t_d is maintained by the destination, and points at the first segment (i.e., lowest sequence number) for which an ACK has been transmitted.
- It isn't a *requirement* that $w_S = w_d$: they can differ, and in some cases this is in fact appropriate. However, there isn't much value in $w_d > w_S$ since in this case the receive buffer can never be filled.
- Each of the pointers (i.e., h_S etc.) is really a sequence number, so they will increase monotonically (until they wrap-around wrt. the number of bits allocated, e.g., at $2^{32} - 1$ back to 0).
- The source is certain all segments upto the t_S-th have been received (since an ACK was received). However, it cannot be certain about any of the $(t_S + 1)$-th upto the h_S-th segments (since no ACK was received yet).
- At the destination, a given segment between the $(t_d + 1)$-th and h_d-th will obviously have been received. However, not *all* may have been so: if the content is received out-of-order, this part of the buffer will have been filled non-contiguously (meaning there could be "gaps").
- More formally, we know the relation
$$t_S \leq t_d \leq h_d \leq h_S \leq t_S + w_S$$
must hold:
  1. $t_s \leq t_d$ because the source cannot have received a higher ACK than transmitted by the destination,
  2. $t_d \leq h_d$ because segments ACK'ed by the destination clearly cannot be beyond those it has actually received,
  3. $h_d \leq h_s$ because the destination cannot have received a higher sequence number than transmitted by the source, and
  4. $h_s \leq t_s + w_s$ because the source cannot have transmitted more than w_s segments (if none are ACK'ed).

TCP (2) – Sliding-window ARQ

- **Example**: update pointer (which shifts destination window).

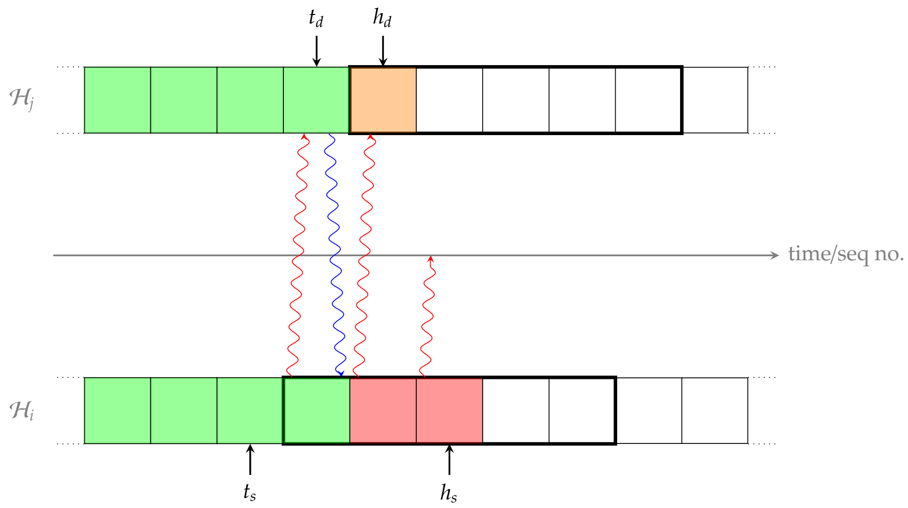


Notes:

- There are various formulations of the protocol: keep in mind that this is just *one* option. If you read a specific textbook, for example, the notation etc. may differ (a common case is where pointers are off-by-one in the sense they point at the first free segment rather than last used segment, the later of which is the approach here), and it might refer to transmitter and receiver rather than source and destination. Either way, the algorithm *should* basically still work the same way.
- Notice that **ACK clocking** is essentially still evident, in the sense that ACKs from the destination *still* control when transmission by the source can occur.
- The variables used, and the relationships between them, are as follows:
  - w_s and w_d are the window sizes used by source and destination respectively,
  - h_s is maintained by the source, and points at the last segment (i.e., highest sequence number) transmitted,
  - h_d is maintained by the destination, and points at the last segment (i.e., highest sequence number) received,
  - t_s is maintained by the source, and points at the first segment (i.e., lowest sequence number) for which an ACK has been received, and
  - t_d is maintained by the destination, and points at the first segment (i.e., lowest sequence number) for which an ACK has been transmitted.
- It isn't a *requirement* that $w_S = w_d$: they can differ, and in some cases this is in fact appropriate. However, there isn't much value in $w_d > w_S$ since in this case the receive buffer can never be filled.
- Each of the pointers (i.e., h_S etc.) is really a sequence number, so they will increase monotonically (until they wrap-around wrt. the number of bits allocated, e.g., at $2^{32} - 1$ back to 0).
- The source is certain all segments upto the t_S-th have been received (since an ACK was received). However, it cannot be certain about any of the $(t_S + 1)$-th upto the h_S-th segments (since no ACK was received yet).
- At the destination, a given segment between the $(t_d + 1)$-th and h_d-th will obviously have been received. However, not *all* may have been so: if the content is received out-of-order, this part of the buffer will have been filled non-contiguously (meaning there could be "gaps").
- More formally, we know the relation
$$t_S \leq t_d \leq h_d \leq h_S \leq t_S + w_S$$
must hold:
  1. $t_s \leq t_d$ because the source cannot have received a higher ACK than transmitted by the destination,
  2. $t_d \leq h_d$ because segments ACK'ed by the destination clearly cannot be beyond those it has actually received,
  3. $h_d \leq h_s$ because the destination cannot have received a higher sequence number than transmitted by the source, and
  4. $h_s \leq t_s + w_s$ because the source cannot have transmitted more than w_s segments (if none are ACK'ed).

TCP (2) – Sliding-window ARQ

- **Example**: receive ACK.



Notes:

- There are various formulations of the protocol: keep in mind that this is just *one* option. If you read a specific textbook, for example, the notation etc. may differ (a common case is where pointers are off-by-one in the sense they point at the first free segment rather than last used segment, the later of which is the approach here), and it might refer to transmitter and receiver rather than source and destination. Either way, the algorithm *should* basically still work the same way.
- Notice that **ACK clocking** is essentially still evident, in the sense that ACKs from the destination *still* control when transmission by the source can occur.
- The variables used, and the relationships between them, are as follows:
  - w_s and w_d are the window sizes used by source and destination respectively,
  - h_s is maintained by the source, and points at the last segment (i.e., highest sequence number) transmitted,
  - h_d is maintained by the destination, and points at the last segment (i.e., highest sequence number) received,
  - t_s is maintained by the source, and points at the first segment (i.e., lowest sequence number) for which an ACK has been received, and
  - t_d is maintained by the destination, and points at the first segment (i.e., lowest sequence number) for which an ACK has been transmitted.
- It isn't a *requirement* that $w_s = w_d$: they can differ, and in some cases this is in fact appropriate. However, there isn't much value in $w_d > w_s$ since in this case the receive buffer can never be filled.
- Each of the pointers (i.e., h_s etc.) is really a sequence number, so they will increase monotonically (until they wrap-around wrt. the number of bits allocated, e.g., at $2^{32} - 1$ back to 0).
- The source is certain all segments upto the t_s-th have been received (since an ACK was received). However, it cannot be certain about any of the $(t_s + 1)$-th upto the h_s-th segments (since no ACK was received yet).
- At the destination, a given segment between the $(t_d + 1)$-th and h_d-th will obviously have been received. However, not *all* may have been so: if the content is received out-of-order, this part of the buffer will have been filled non-contiguously (meaning there could be "gaps").
- More formally, we know the relation

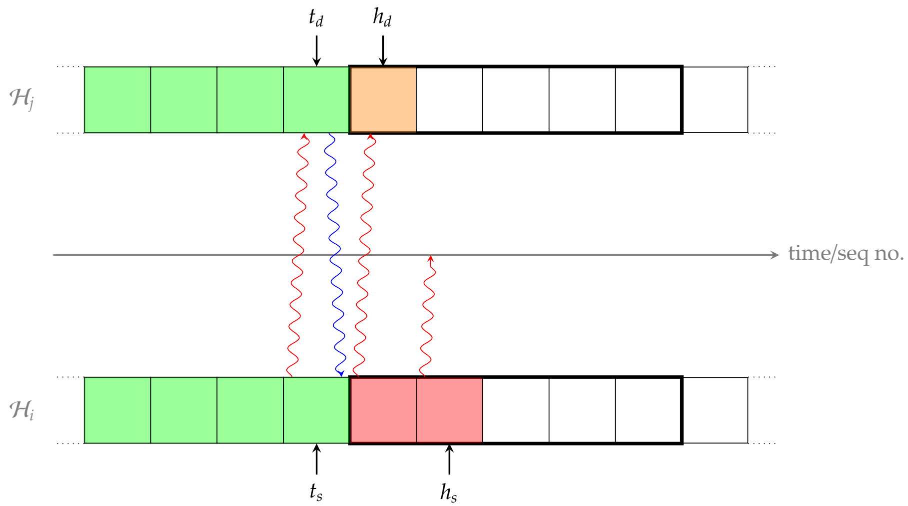
  $$t_s \leq t_d \leq h_d \leq h_s \leq t_s + w_s$$

  must hold:

  1. $t_s \leq t_d$ because the source cannot have received a higher ACK than transmitted by the destination,
  2. $t_d \leq h_d$ because segments ACK'ed by the destination clearly cannot be beyond those it has actually received,
  3. $h_d \leq h_s$ because the destination cannot have received a higher sequence number than transmitted by the source, and
  4. $h_s \leq t_s + w_s$ because the source cannot have transmitted more than w_s segments (if none are ACK'ed).

TCP (2) – Sliding-window ARQ

- **Example**: update pointer (which shifts source window).



Notes:

- There are various formulations of the protocol: keep in mind that this is just *one* option. If you read a specific textbook, for example, the notation etc. may differ (a common case is where pointers are off-by-one in the sense they point at the first free segment rather than last used segment, the later of which is the approach here), and it might refer to transmitter and receiver rather than source and destination. Either way, the algorithm *should* basically still work the same way.
- Notice that **ACK clocking** is essentially still evident, in the sense that ACKs from the destination *still* control when transmission by the source can occur.
- The variables used, and the relationships between them, are as follows:
  - w_s and w_d are the window sizes used by source and destination respectively,
  - h_s is maintained by the source, and points at the last segment (i.e., highest sequence number) transmitted,
  - h_d is maintained by the destination, and points at the last segment (i.e., highest sequence number) received,
  - t_s is maintained by the source, and points at the first segment (i.e., lowest sequence number) for which an ACK has been received, and
  - t_d is maintained by the destination, and points at the first segment (i.e., lowest sequence number) for which an ACK has been transmitted.
- It isn't a *requirement* that $w_s = w_d$: they can differ, and in some cases this is in fact appropriate. However, there isn't much value in $w_d > w_s$ since in this case the receive buffer can never be filled.
- Each of the pointers (i.e., h_s etc.) is really a sequence number, so they will increase monotonically (until they wrap-around wrt. the number of bits allocated, e.g., at $2^{32} - 1$ back to 0).
- The source is certain all segments upto the t_s-th have been received (since an ACK was received). However, it cannot be certain about any of the $(t_s + 1)$-th upto the h_s-th segments (since no ACK was received yet).
- At the destination, a given segment between the $(t_d + 1)$-th and h_d-th will obviously have been received. However, not *all* may have been so: if the content is received out-of-order, this part of the buffer will have been filled non-contiguously (meaning there could be "gaps").
- More formally, we know the relation

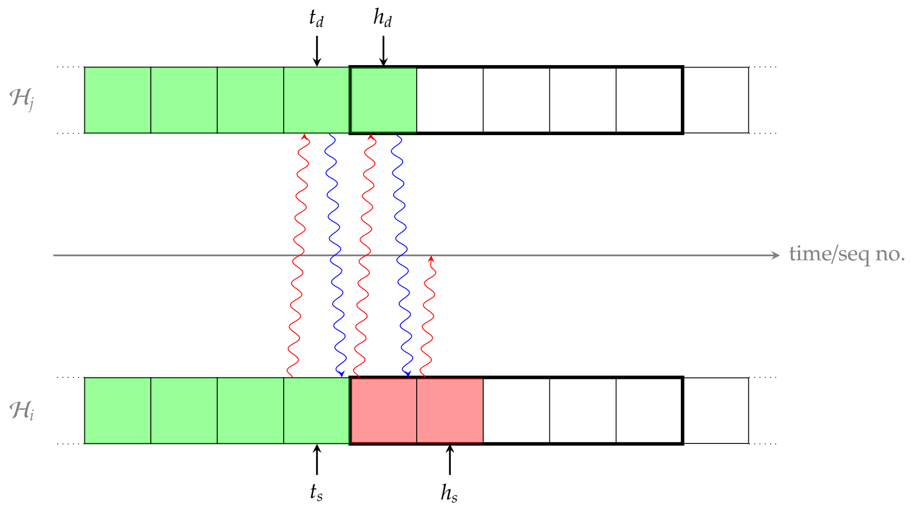
  $$t_s \leq t_d \leq h_d \leq h_s \leq t_s + w_s$$

  must hold:

  1. $t_s \leq t_d$ because the source cannot have received a higher ACK than transmitted by the destination,
  2. $t_d \leq h_d$ because segments ACK'ed by the destination clearly cannot be beyond those it has actually received,
  3. $h_d \leq h_s$ because the destination cannot have received a higher sequence number than transmitted by the source, and
  4. $h_s \leq t_s + w_s$ because the source cannot have transmitted more than w_s segments (if none are ACK'ed).

TCP (2) – Sliding-window ARQ

- **Example**: transmit ACK.



Notes:

- There are various formulations of the protocol: keep in mind that this is just *one* option. If you read a specific textbook, for example, the notation etc. may differ (a common case is where pointers are off-by-one in the sense they point at the first free segment rather than last used segment, the later of which is the approach here), and it might refer to transmitter and receiver rather than source and destination. Either way, the algorithm *should* basically still work the same way.
- Notice that **ACK clocking** is essentially still evident, in the sense that ACKs from the destination *still* control when transmission by the source can occur.
- The variables used, and the relationships between them, are as follows:
  - w_s and w_d are the window sizes used by source and destination respectively,
  - h_s is maintained by the source, and points at the last segment (i.e., highest sequence number) transmitted,
  - h_d is maintained by the destination, and points at the last segment (i.e., highest sequence number) received,
  - t_s is maintained by the source, and points at the first segment (i.e., lowest sequence number) for which an ACK has been received, and
  - t_d is maintained by the destination, and points at the first segment (i.e., lowest sequence number) for which an ACK has been transmitted.
- It isn't a *requirement* that $w_s = w_d$: they can differ, and in some cases this is in fact appropriate. However, there isn't much value in $w_d > w_s$ since in this case the receive buffer can never be filled.
- Each of the pointers (i.e., h_s etc.) is really a sequence number, so they will increase monotonically (until they wrap-around wrt. the number of bits allocated, e.g., at $2^{32} - 1$ back to 0).
- The source is certain all segments upto the t_s-th have been received (since an ACK was received). However, it cannot be certain about any of the $(t_s + 1)$-th upto the h_s-th segments (since no ACK was received yet).
- At the destination, a given segment between the $(t_d + 1)$-th and h_d-th will obviously have been received. However, not *all* may have been so: if the content is received out-of-order, this part of the buffer will have been filled non-contiguously (meaning there could be "gaps").
- More formally, we know the relation

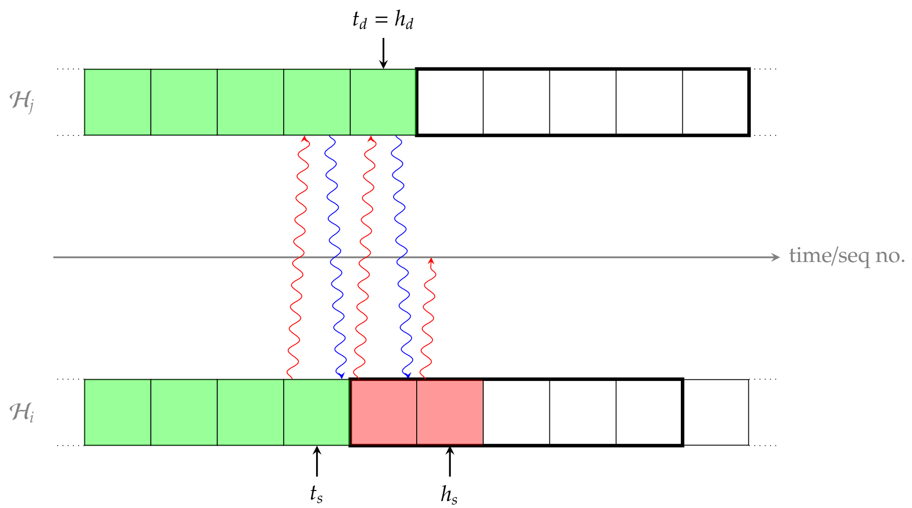
$$t_s \leq t_d \leq h_d \leq h_s \leq t_s + w_s$$

  must hold:

  1. $t_s \leq t_d$ because the source cannot have received a higher ACK than transmitted by the destination,
  2. $t_d \leq h_d$ because segments ACK'ed by the destination clearly cannot be beyond those it has actually received,
  3. $h_d \leq h_s$ because the destination cannot have received a higher sequence number than transmitted by the source, and
  4. $h_s \leq t_s + w_s$ because the source cannot have transmitted more than w_s segments (if none are ACK'ed).

TCP (2) – Sliding-window ARQ

- **Example**: update pointer (which shifts destination window).



Notes:

- There are various formulations of the protocol: keep in mind that this is just *one* option. If you read a specific textbook, for example, the notation etc. may differ (a common case is where pointers are off-by-one in the sense they point at the first free segment rather than last used segment, the later of which is the approach here), and it might refer to transmitter and receiver rather than source and destination. Either way, the algorithm *should* basically still work the same way.
- Notice that **ACK clocking** is essentially still evident, in the sense that ACKs from the destination *still* control when transmission by the source can occur.
- The variables used, and the relationships between them, are as follows:
  - w_s and w_d are the window sizes used by source and destination respectively,
  - h_s is maintained by the source, and points at the last segment (i.e., highest sequence number) transmitted,
  - h_d is maintained by the destination, and points at the last segment (i.e., highest sequence number) received,
  - t_s is maintained by the source, and points at the first segment (i.e., lowest sequence number) for which an ACK has been received, and
  - t_d is maintained by the destination, and points at the first segment (i.e., lowest sequence number) for which an ACK has been transmitted.
- It isn't a *requirement* that $w_s = w_d$: they can differ, and in some cases this is in fact appropriate. However, there isn't much value in $w_d > w_s$ since in this case the receive buffer can never be filled.
- Each of the pointers (i.e., h_s etc.) is really a sequence number, so they will increase monotonically (until they wrap-around wrt. the number of bits allocated, e.g., at $2^{32} - 1$ back to 0).
- The source is certain all segments upto the t_s-th have been received (since an ACK was received). However, it cannot be certain about any of the $(t_s + 1)$-th upto the h_s-th segments (since no ACK was received yet).
- At the destination, a given segment between the $(t_d + 1)$-th and h_d-th will obviously have been received. However, not *all* may have been so: if the content is received out-of-order, this part of the buffer will have been filled non-contiguously (meaning there could be "gaps").
- More formally, we know the relation

$$t_s \leq t_d \leq h_d \leq h_s \leq t_s + w_s$$

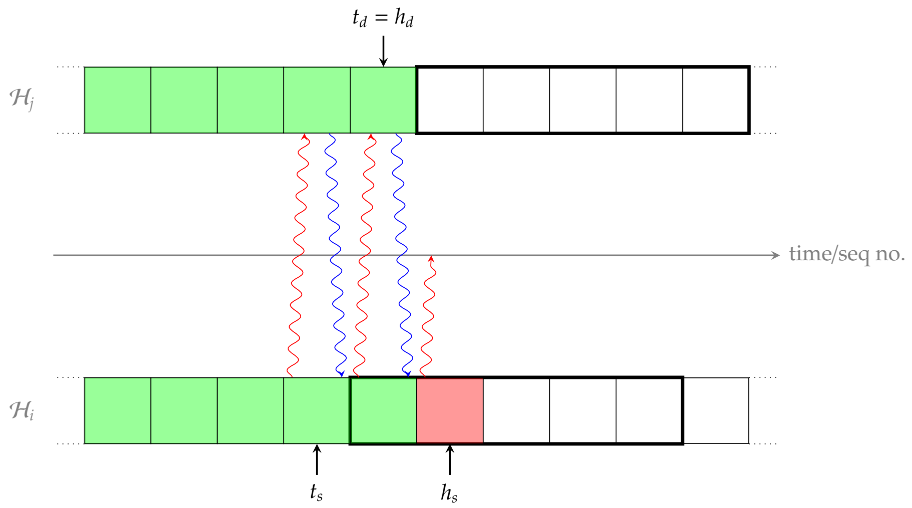
  must hold:

  1. $t_s \leq t_d$ because the source cannot have received a higher ACK than transmitted by the destination,
  2. $t_d \leq h_d$ because segments ACK'ed by the destination clearly cannot be beyond those it has actually received,
  3. $h_d \leq h_s$ because the destination cannot have received a higher sequence number than transmitted by the source, and
  4. $h_s \leq t_s + w_s$ because the source cannot have transmitted more than w_s segments (if none are ACK'ed).

Daniel Page ⟨Daniel.Page@bristol.ac.uk⟩
Concurrent Computing (Computer Networks) git # 3627080 @ 2016-03-11
University of BRISTOL

TCP (2) – Sliding-window ARQ

- **Example**: receive ACK.



Notes:

- There are various formulations of the protocol: keep in mind that this is just *one* option. If you read a specific textbook, for example, the notation etc. may differ (a common case is where pointers are off-by-one in the sense they point at the first free segment rather than last used segment, the later of which is the approach here), and it might refer to transmitter and receiver rather than source and destination. Either way, the algorithm *should* basically still work the same way.
- Notice that **ACK clocking** is essentially still evident, in the sense that ACKs from the destination *still* control when transmission by the source can occur.
- The variables used, and the relationships between them, are as follows:
  - w_s and w_d are the window sizes used by source and destination respectively,
  - h_s is maintained by the source, and points at the last segment (i.e., highest sequence number) transmitted,
  - h_d is maintained by the destination, and points at the last segment (i.e., highest sequence number) received,
  - t_s is maintained by the source, and points at the first segment (i.e., lowest sequence number) for which an ACK has been received, and
  - t_d is maintained by the destination, and points at the first segment (i.e., lowest sequence number) for which an ACK has been transmitted.
- It isn't a *requirement* that $w_S = w_d$: they can differ, and in some cases this is in fact appropriate. However, there isn't much value in $w_d > w_S$ since in this case the receive buffer can never be filled.
- Each of the pointers (i.e., h_S etc.) is really a sequence number, so they will increase monotonically (until they wrap-around wrt. the number of bits allocated, e.g., at $2^{32} - 1$ back to 0).
- The source is certain all segments upto the t_S-th have been received (since an ACK was received). However, it cannot be certain about any of the $(t_S + 1)$-th upto the h_S-th segments (since no ACK was received yet).
- At the destination, a given segment between the $(t_d + 1)$-th and h_d-th will obviously have been received. However, not *all* may have been so: if the content is received out-of-order, this part of the buffer will have been filled non-contiguously (meaning there could be "gaps").
- More formally, we know the relation

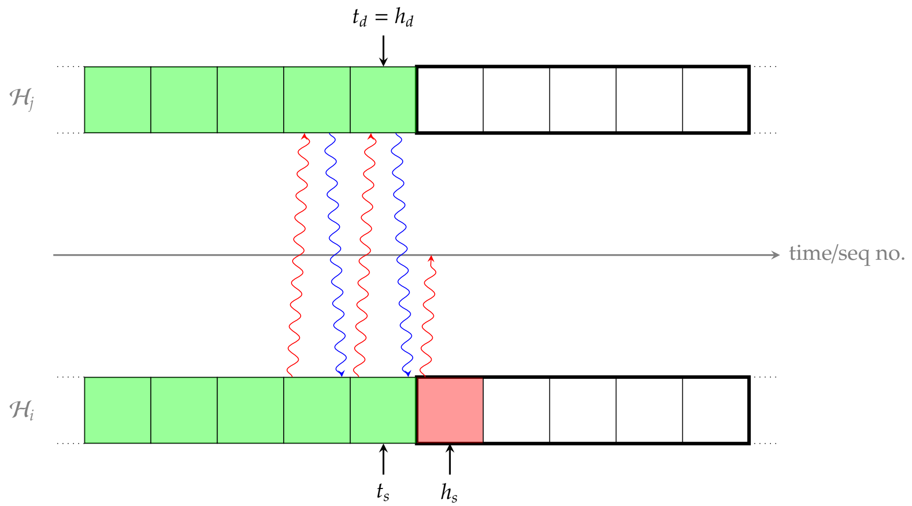
  $$t_S \le t_d \le h_d \le h_S \le t_S + w_S$$

  must hold:

  1. $t_s \le t_d$ because the source cannot have received a higher ACK than transmitted by the destination,
  2. $t_d \le h_d$ because segments ACK'ed by the destination clearly cannot be beyond those it has actually received,
  3. $h_d \le h_s$ because the destination cannot have received a higher sequence number than transmitted by the source, and
  4. $h_s \le t_s + w_s$ because the source cannot have transmitted more than w_s segments (if none are ACK'ed).

TCP (2) – Sliding-window ARQ

- **Example**: update pointer (which shifts source window).



Notes:

- There are various formulations of the protocol: keep in mind that this is just *one* option. If you read a specific textbook, for example, the notation etc. may differ (a common case is where pointers are off-by-one in the sense they point at the first free segment rather than last used segment, the later of which is the approach here), and it might refer to transmitter and receiver rather than source and destination. Either way, the algorithm *should* basically still work the same way.
- Notice that **ACK clocking** is essentially still evident, in the sense that ACKs from the destination *still* control when transmission by the source can occur.
- The variables used, and the relationships between them, are as follows:
  - w_s and w_d are the window sizes used by source and destination respectively,
  - h_s is maintained by the source, and points at the last segment (i.e., highest sequence number) transmitted,
  - h_d is maintained by the destination, and points at the last segment (i.e., highest sequence number) received,
  - t_s is maintained by the source, and points at the first segment (i.e., lowest sequence number) for which an ACK has been received, and
  - t_d is maintained by the destination, and points at the first segment (i.e., lowest sequence number) for which an ACK has been transmitted.
- It isn't a *requirement* that $w_S = w_d$: they can differ, and in some cases this is in fact appropriate. However, there isn't much value in $w_d > w_S$ since in this case the receive buffer can never be filled.
- Each of the pointers (i.e., h_S etc.) is really a sequence number, so they will increase monotonically (until they wrap-around wrt. the number of bits allocated, e.g., at $2^{32} - 1$ back to 0).
- The source is certain all segments upto the t_S-th have been received (since an ACK was received). However, it cannot be certain about any of the $(t_S + 1)$-th upto the h_S-th segments (since no ACK was received yet).
- At the destination, a given segment between the $(t_d + 1)$-th and h_d-th will obviously have been received. However, not *all* may have been so: if the content is received out-of-order, this part of the buffer will have been filled non-contiguously (meaning there could be "gaps").
- More formally, we know the relation

  $$t_S \le t_d \le h_d \le h_S \le t_S + w_S$$

  must hold:

  1. $t_s \le t_d$ because the source cannot have received a higher ACK than transmitted by the destination,
  2. $t_d \le h_d$ because segments ACK'ed by the destination clearly cannot be beyond those it has actually received,
  3. $h_d \le h_s$ because the destination cannot have received a higher sequence number than transmitted by the source, and
  4. $h_s \le t_s + w_s$ because the source cannot have transmitted more than w_s segments (if none are ACK'ed).

TCP (3) – Sliding-window ARQ

Algorithm (source)

Assuming the source maintains

- a w_s-element (cyclic) buffer,
- t_s and h_s pointers into the buffer, and
- w_s time-out timers

then various events can occur:

1. Application layer invokes `send`:
   1.1 if $h_s < t_s + w_s$
      - copy segment into buffer at h_s,
      - set $h_s \leftarrow h_s + 1$, then
      - transmit segment and start time-out timer

   1.2 otherwise block transmission.
2. Transport layer receives ACK:
   - set $t_s \leftarrow t_s + 1$, then
   - unblock upto one pending transmission.
3. Transport layer times-out:
   - *re*transmit segment wrt. timer that timed-out.

Algorithm (destination)

Assuming the destination maintains

- a w_d-segment (cyclic) buffer, and
- t_d and h_s pointers into the buffer

then various events can occur:

1. Application layer invokes `recv`:
   - copy upto m in-order segments from buffer at $t_d + 1$ onward,
   - set $t_d \leftarrow t_d + m$, then
   - transmit ACK(s).
2. Transport layer receives data:
   2.1 if $t_d <$ seq no. $\leq t_d + w_d$, buffer segment,
   2.2 otherwise drop segment.

Notes:

- This is the most generalised sliding-window algorithm, namely **selective repeat**. However, it neatly captures two special cases:
  1. if $w_s = 1$ and $w_d = 1$ then we have **stop-and-wait**, and
  2. if $w_s > 1$ and $w_d = 1$ then we have **go-back-N**.

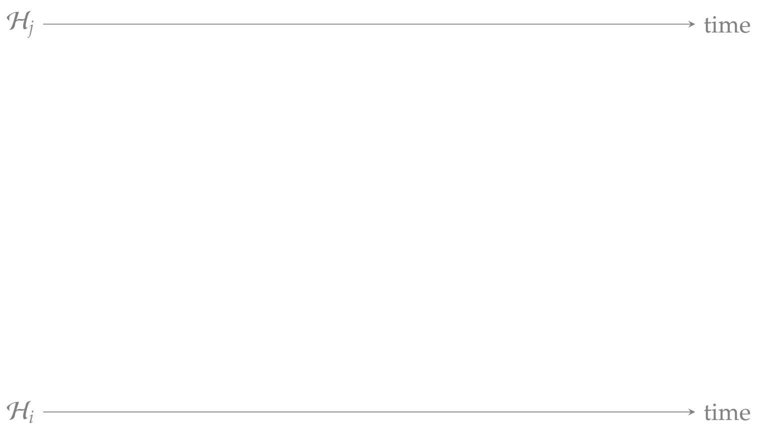
  So clearly for stop-and-wait, $w_s = 1$ tells us we cannot transmit more than one un-ACK'ed segment: it's as if we have a 1-segment buffer, so we need to wait for *that* segment to be ACK'ed before transmitting another.
  For go-back-N (which is somewhere between stop-and-wait and sliding-window based on selective-repeat in terms of complexity), the source is more capable since now $w_s > 1$. However, the receiver still has $w_d = 1$. This means a) it must receive segments in-order (unlike selective-repeat, which has a larger buffer so can buffer segments even when received out-of-order), and b) when a time-out occurs, the source must "go back N segments", i.e., start retransmitting starting with the last un-ACK'ed segment that it transmitted, rather than ACK *just* the segment that caused the time-out (as with selective-repeat).
- A choice between two retransmission policies actually exists. Here we say retransmit the segment whose time-out timer has expired; in a sense, this is a pessimistic or conservative approach because we *only* retransmit segments we can reason directly about. The alternative is that when some time-out timer expires, we retransmit *all* the subsequent un-ACK'ed segments. On one hand this is advantageous when multiple segments are lost for some reason (basically we reason that if one is lost, others will be too), because we needn't wait for their time-out timers to expire; on the other hand, if this reasoning is false then we are overly aggressive and potentially waste bandwidth due to unnecessary retransmission.

TCP (4) – Sliding-window ARQ $\rightsquigarrow$ flow control

- ... it get's even better still:
  - imagine we allow w_s grow or shrink dynamically ...
  - ... smaller w_s "throttles" the source, i.e., reduces how many un-ACK'ed segments it can transmit: if we allow the *destination* to set w_s, this yields a flow control mechanism.
- Idea:
  1. destination includes w_a the **advertised window size**
  $$w_a = w_d - (h_d - h_a)$$
  in each ACK (via the window size field), and
  2. source uses **effective window size**
  $$w_e = \min(w_s, w_a)$$
  instead of w_s.

Notes:

- Example: imagine the destination has a 4kB buffer.



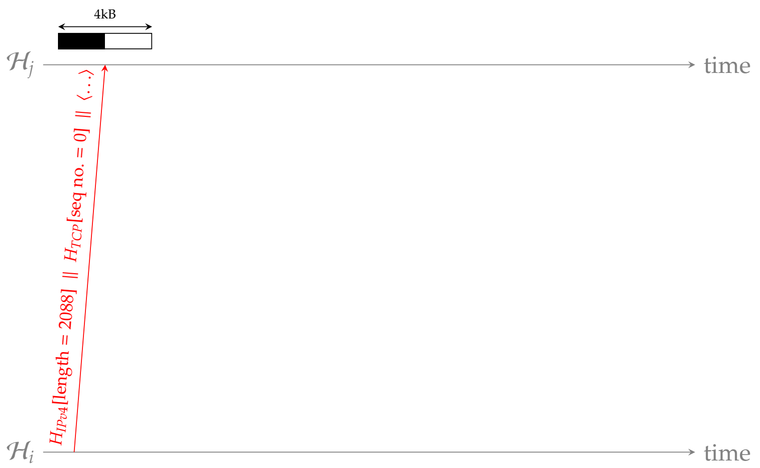
Notes:

- Why the weird notation this time?! TCP has no length field in the header: the length of the payload is simply whatever the length field in the IPv4 packet says, minus both the IPv4 and TCP header lengths. To denote a TCP payload length we therefore "cheat" and do so by showing the IPv4 header as well.
  So, if the IPv4 header says the length is xB, then the TCP payload will be

$$x - \text{IPv4 header length} - \text{TCP header length}$$

  e.g., $x - 40$ if both headers are their minimum of 20B.

- Example: imagine the destination has a 4kB buffer.



Notes:

- Why the weird notation this time?! TCP has no length field in the header: the length of the payload is simply whatever the length field in the IPv4 packet says, minus both the IPv4 and TCP header lengths. To denote a TCP payload length we therefore "cheat" and do so by showing the IPv4 header as well.
  So, if the IPv4 header says the length is xB, then the TCP payload will be

$$x - \text{IPv4 header length} - \text{TCP header length}$$

  e.g., $x - 40$ if both headers are their minimum of 20B.

TCP (5) – Sliding-window ARQ $\rightsquigarrow$ flow control

- Example: imagine the destination has a 4kB buffer.

Notes:

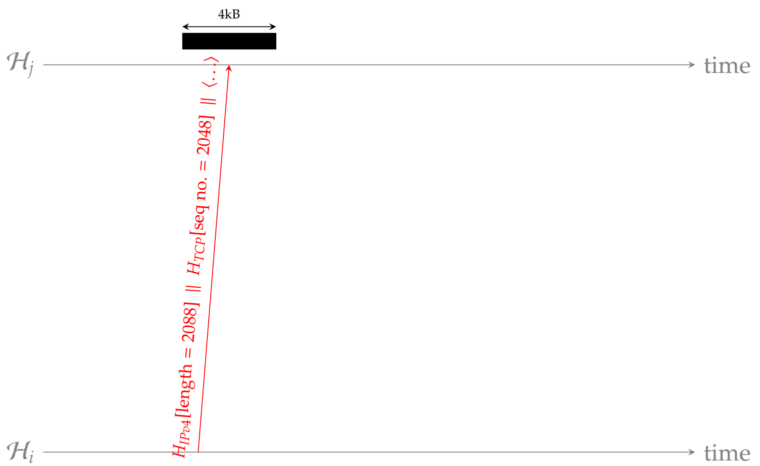
- Why the weird notation this time?! TCP has no length field in the header: the length of the payload is simply whatever the length field in the IPv4 packet says, minus both the IPv4 and TCP header lengths. To denote a TCP payload length we therefore "cheat" and do so by showing the IPv4 header as well.
  So, if the IPv4 header says the length is xB, then the TCP payload will be

  $$x - \text{IPv4 header length} - \text{TCP header length}$$

  e.g., $x - 40$ if both headers are their minimum of 20B.

TCP (5) – Sliding-window ARQ $\rightsquigarrow$ flow control

- Example: imagine the destination has a 4kB buffer.



Notes:

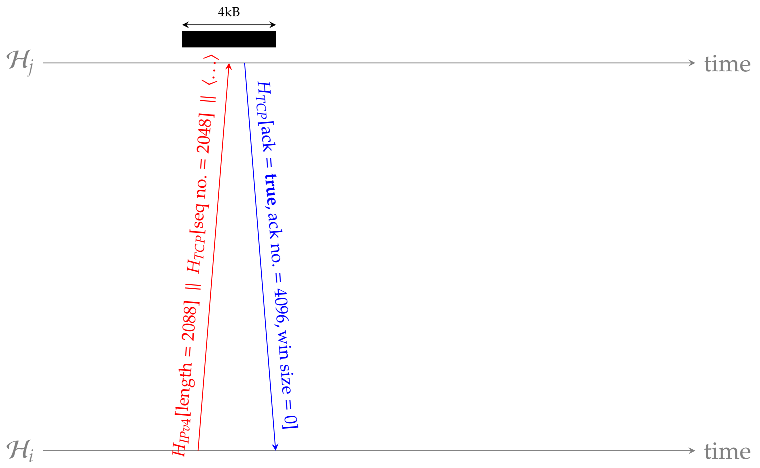
- Why the weird notation this time?! TCP has no length field in the header: the length of the payload is simply whatever the length field in the IPv4 packet says, minus both the IPv4 and TCP header lengths. To denote a TCP payload length we therefore "cheat" and do so by showing the IPv4 header as well.
  So, if the IPv4 header says the length is xB, then the TCP payload will be

  $$x - \text{IPv4 header length} - \text{TCP header length}$$

  e.g., $x - 40$ if both headers are their minimum of 20B.

TCP (5) – Sliding-window ARQ $\rightsquigarrow$ flow control

- Example: imagine the destination has a 4kB buffer.



Notes:

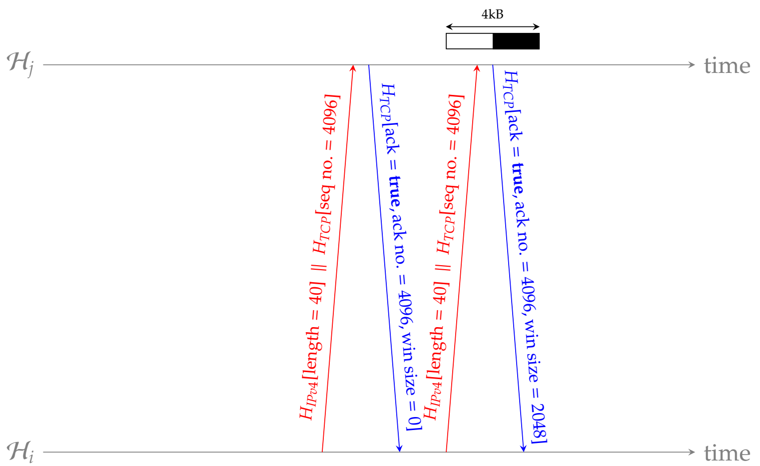
- Why the weird notation this time?! TCP has no length field in the header: the length of the payload is simply whatever the length field in the IPv4 packet says, minus both the IPv4 and TCP header lengths. To denote a TCP payload length we therefore "cheat" and do so by showing the IPv4 header as well.
  So, if the IPv4 header says the length is xB, then the TCP payload will be

$$x - \text{IPv4 header length} - \text{TCP header length}$$

  e.g., $x - 40$ if both headers are their minimum of 20B.

TCP (5) – Sliding-window ARQ $\rightsquigarrow$ flow control

- Example: imagine the destination has a 4kB buffer.



Notes:

- Why the weird notation this time?! TCP has no length field in the header: the length of the payload is simply whatever the length field in the IPv4 packet says, minus both the IPv4 and TCP header lengths. To denote a TCP payload length we therefore "cheat" and do so by showing the IPv4 header as well.
  So, if the IPv4 header says the length is xB, then the TCP payload will be

$$x - \text{IPv4 header length} - \text{TCP header length}$$

  e.g., $x - 40$ if both headers are their minimum of 20B.

TCP (5) – Sliding-window ARQ ⇝ flow control

- **Example**: imagine the destination has a 4kB buffer.

Notes:

- Why the weird notation this time?! TCP has no length field in the header: the length of the payload is simply whatever the length field in the IPv4 packet says, minus both the IPv4 and TCP header lengths. To denote a TCP payload length we therefore "cheat" and do so by showing the IPv4 header as well.
  So, if the IPv4 header says the length is xB, then the TCP payload will be

  $$x - \text{IPv4 header length} - \text{TCP header length}$$

  e.g., $x - 40$ if both headers are their minimum of 20B.

TCP (5) – Sliding-window ARQ ⇝ flow control

- **Example**: imagine the destination has a 4kB buffer.

Notes:

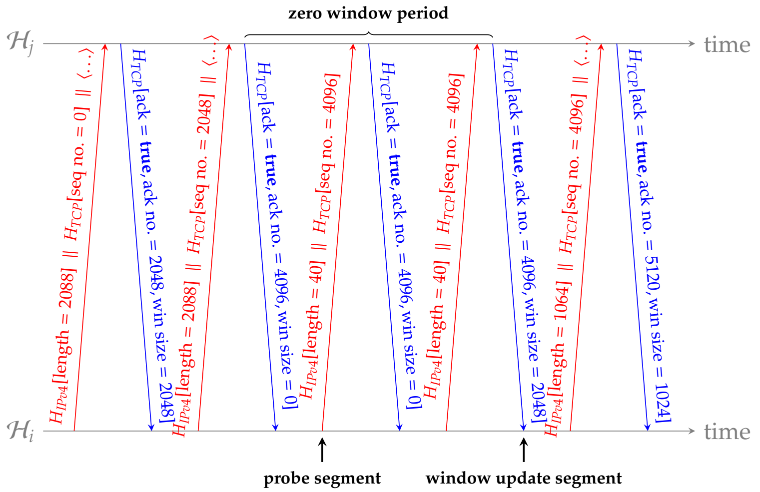
- Why the weird notation this time?! TCP has no length field in the header: the length of the payload is simply whatever the length field in the IPv4 packet says, minus both the IPv4 and TCP header lengths. To denote a TCP payload length we therefore "cheat" and do so by showing the IPv4 header as well.
  So, if the IPv4 header says the length is xB, then the TCP payload will be

  $$x - \text{IPv4 header length} - \text{TCP header length}$$

  e.g., $x - 40$ if both headers are their minimum of 20B.

TCP (5) – Sliding-window ARQ ⇝ flow control

- Example: imagine the destination has a 4kB buffer.



Notes:

- Why the weird notation this time?! TCP has no length field in the header: the length of the payload is simply whatever the length field in the IPv4 packet says, minus both the IPv4 and TCP header lengths. To denote a TCP payload length we therefore "cheat" and do so by showing the IPv4 header as well.
  So, if the IPv4 header says the length is xB, then the TCP payload will be

  $$x - \text{IPv4 header length} - \text{TCP header length}$$

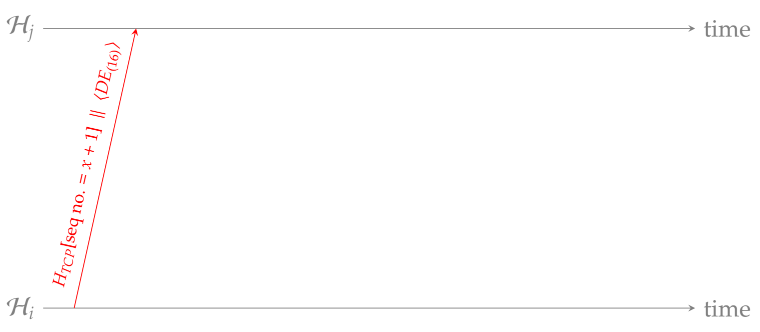
  e.g., $x - 40$ if both headers are their minimum of 20B.

TCP (6) – Sliding-window ARQ ⇝ cumulative and selective ACKs

- Problem: naively, we send one ACK per segment (whether piggybacked or not).

Notes:

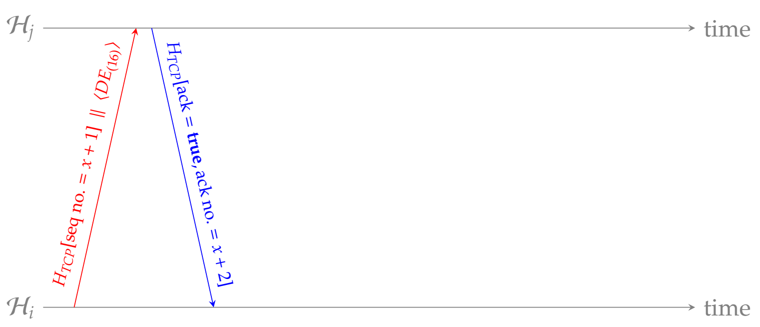
TCP (6) – Sliding-window ARQ $\rightsquigarrow$ cumulative and selective ACKs



- Problem: naively, we send one ACK per segment (whether piggybacked or not).

Notes:

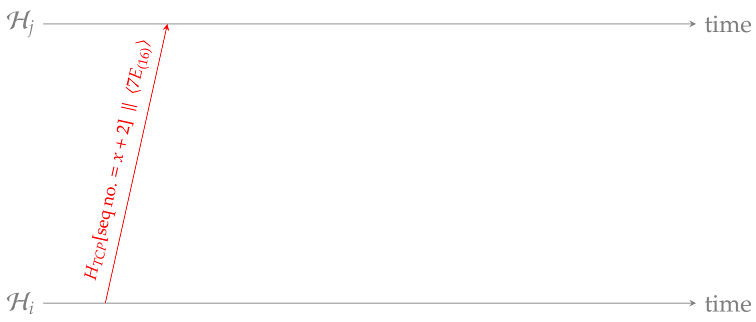
TCP (6) – Sliding-window ARQ $\rightsquigarrow$ cumulative and selective ACKs



- Problem: naively, we send one ACK per segment (whether piggybacked or not).

Notes:

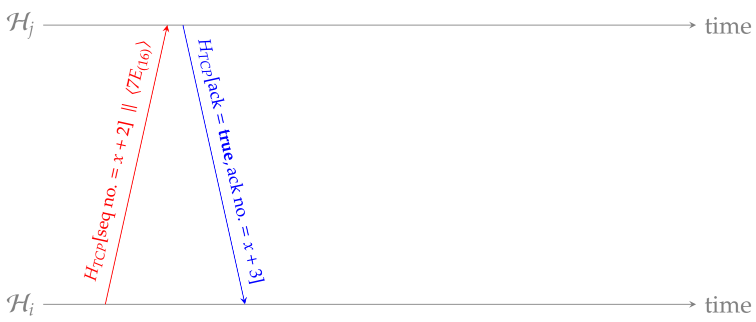
TCP (6) – Sliding-window ARQ $\rightsquigarrow$ cumulative and selective ACKs



- Problem: naively, we send one ACK per segment (whether piggybacked or not).

Notes:

TCP (6) – Sliding-window ARQ $\rightsquigarrow$ cumulative and selective ACKs



- Problem: naively, we send one ACK per segment (whether piggybacked or not).

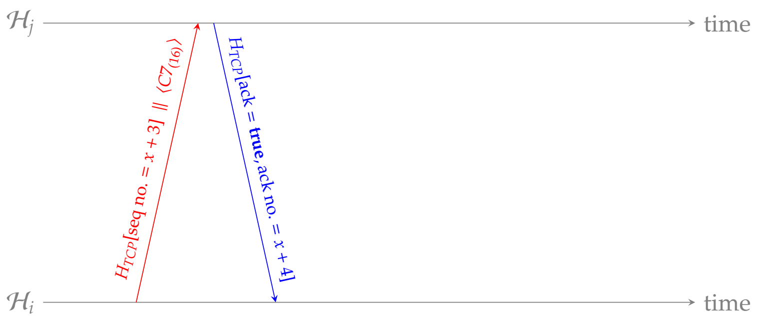
Notes:

TCP (6) – Sliding-window ARQ $\rightsquigarrow$ cumulative and selective ACKs

- Problem: naively, we send one ACK per segment (whether piggybacked or not).

Notes:

TCP (6) – Sliding-window ARQ $\rightsquigarrow$ cumulative and selective ACKs



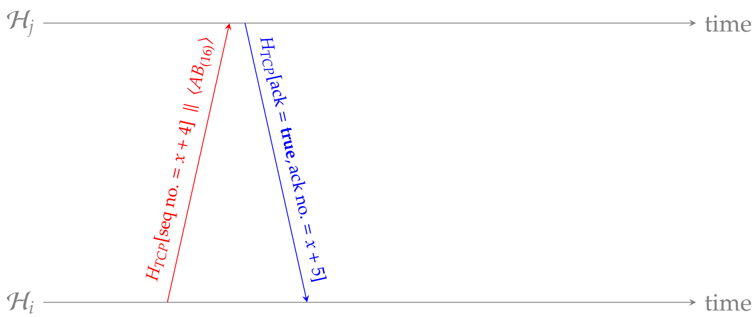
- Problem: naively, we send one ACK per segment (whether piggybacked or not).

Notes:

- Problem: naively, we send one ACK per segment (whether piggybacked or not).

Notes:

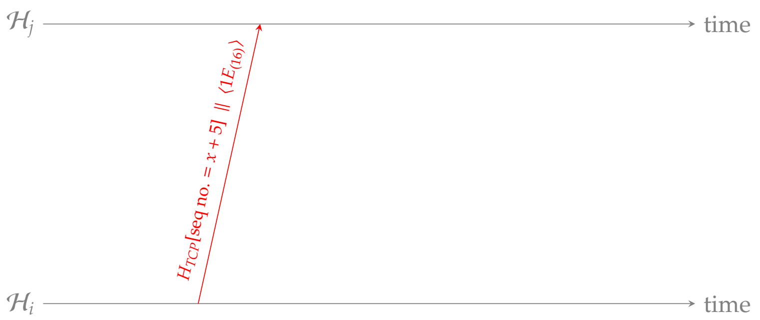
TCP (6) – Sliding-window ARQ $\rightsquigarrow$ cumulative and selective ACKs



- Problem: naively, we send one ACK per segment (whether piggybacked or not).

Notes:

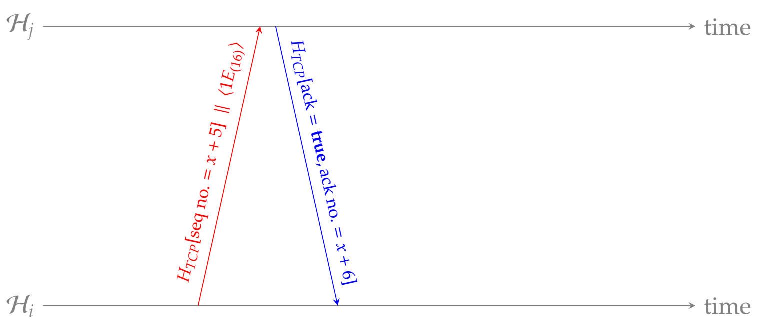
TCP (6) – Sliding-window ARQ $\leadsto$ cumulative and selective ACKs



- Problem: naively, we send one ACK per segment (whether piggybacked or not).

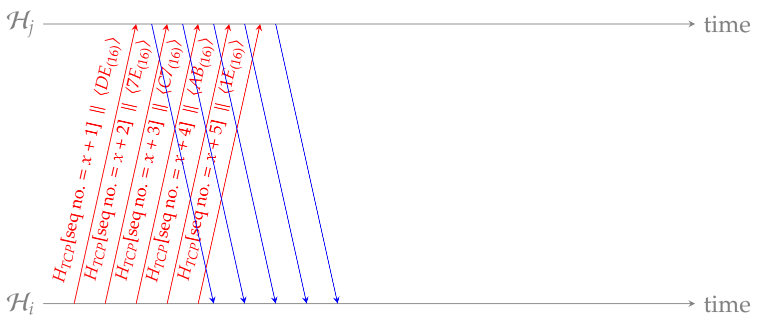
Notes:

TCP (6) – Sliding-window ARQ $\leadsto$ cumulative and selective ACKs



- Problem: naively, we send one ACK per segment (whether piggybacked or not).

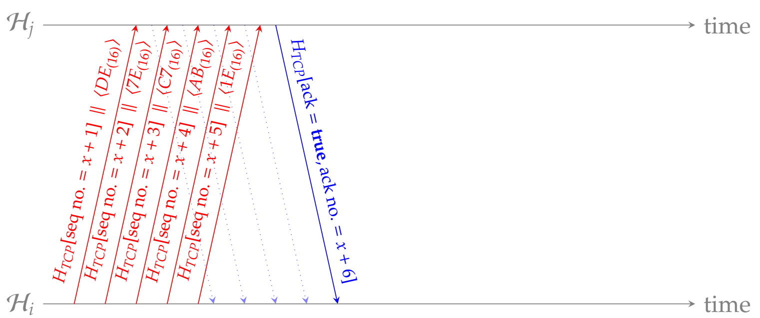
Notes:



- Problem: naively, we send one ACK per segment (whether piggybacked or not).

Notes:

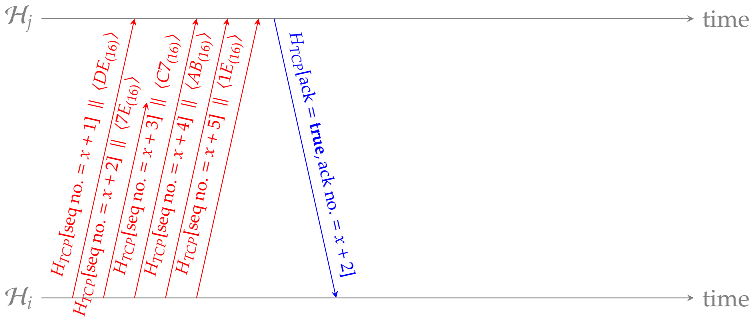
TCP (6) – Sliding-window ARQ $\rightsquigarrow$ cumulative and selective ACKs



- Problem: naively, we send one ACK per segment (whether piggybacked or not).
- Solution: allow a **cumulative ACK**, that
  - explicitly ACKs a segment, *and*
  - implicitly ACKs previous segments.

Notes:

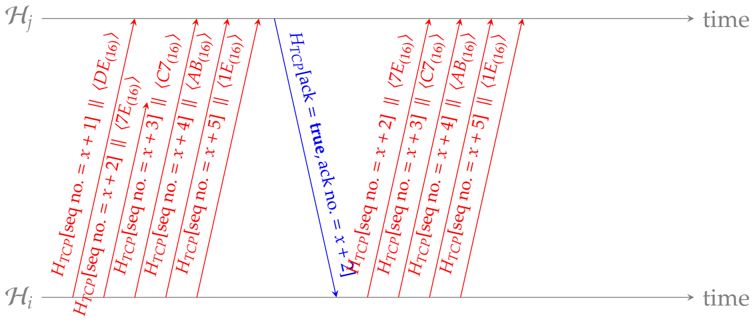
TCP (6) – Sliding-window ARQ $\rightsquigarrow$ cumulative and selective ACKs



- Problem: can't cumulatively ACK *later* segments even if valid.

Notes:

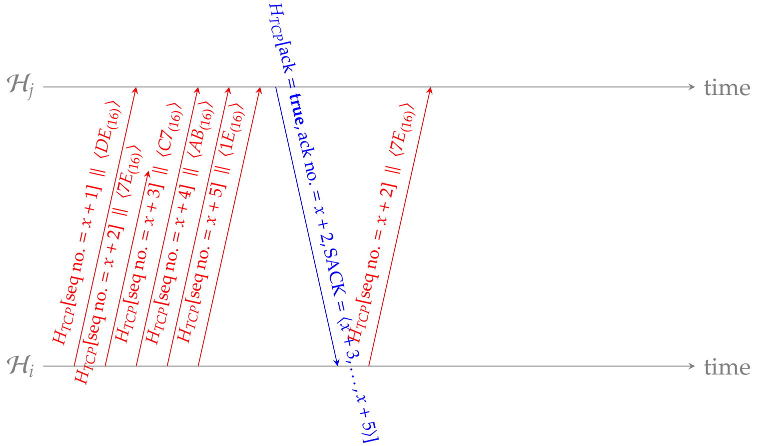
TCP (6) – Sliding-window ARQ $\rightsquigarrow$ cumulative and selective ACKs



- Problem: can't cumulatively ACK *later* segments even if valid.

Notes:

TCP (6) – Sliding-window ARQ ⇝ cumulative and selective ACKs

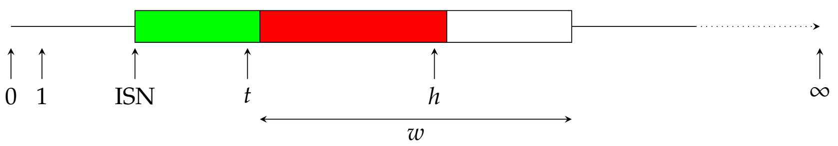


- Problem: can't cumulatively ACK *later* segments even if valid.
- Solution: allow a **selective ACK** [12], that
  - acts as (cumulative) ACK for contiguous segment(s), *and*
  - "hint" at other discontiguous segment(s).

Notes:

TCP (7) – Sliding-window ARQ ⇝ sequence number wrap-around

- In theory: sequence numbers are assumed to unbounded, e.g.,

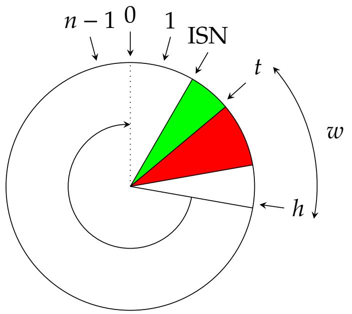


so we can just perform a simple validity check:

- if $t_d <$ seq no. $\leq t_d + w_d$ then buffer segment,
- otherwise drop segment.

Notes:

- In practice: sequence numbers are actually bounded modulo some n e.g.,



so we can encounter (at least) these problems:

- at some point $h < t$ (i.e., the interval between t_d and $t_d + w_d$ is split) making comparisons tricky(er), plus
- we *could* encounter an "old" segment whose sequence number matches a "new" segment!

Notes:

- Since sequence numbers are bounded modulo n (typically $n = 2^{n'}$ to allow an n'-bit field in the header), the maximum value is $n-1$: this means they will **wrap-around** from $n-1$ to 0. Keep in mind that the ISN *isn't* necessarily zero, so wrap-around can happen quicker than you'd think!
- Wrap-around means that if we transmit $x \text{B s}^{-1}$, we will encounter a repetition of any given sequence number every $\frac{n}{x}$ s. As a result, increasing the rate of transmission will exacerbate the problem: we'll encounter repetitions quicker.

- ... so, in summary, we need
  1. w_s and w_d to be large enough to ensure efficiency, i.e., to match the bandwidth-delay product of network, and
  2. n to be large enough st.
     - 2.1 sequence numbers cannot be ambiguous, *except*
     - 2.2 when wrap-around occurs, the likelihood of which should be minimised.
- Solution:
  1. ensure $n > w_s + w_d$, then
  2. either
     - 2.1 fix a MSL st. wrap-around shouldn't occur so fast that we can't disambiguate in-flight segments, and/or
     - 2.2 implement **Protection Against Wrapping Sequence (PAWS)** [10] to disambiguate via use of time-stamps.

Notes:

- The issue of ambiguity is easier to explain by example. Imagine we implement ARQ based on use of sliding-window, and $n = 2^3 = 8$ st. the maximum possible sequence number is $n - 1 = 7$; we set $w_s = w_d = 7$.
  Now imagine that the source and destination start communicating. The source transmits 7 segments with sequence numbers 0 through to 6, i.e.,
  $$H_{TCP}[\text{seq no.} = 0] \parallel P_0$$
  $$H_{TCP}[\text{seq no.} = 1] \parallel P_1$$
  $$\vdots$$
  $$H_{TCP}[\text{seq no.} = 6] \parallel P_2$$
  all of which reach the destination successfully. As a result, the destination transmits associated ACKs, or one cumulative ACK
  $$H_{TCP}[\text{ack no.} = 7].$$
  This indicates it expects to receive a segment numbered 7; since it is operating a sliding-window, it will actually accept segments whose sequence number is in that window, i.e., numbered 7 through to $7 + w_d - 1$. Remember that the sequence numbers wrap-around, so in reality this means it will accept segments numbered 7, 0, 1, etc. through to $7 + w_d - 1 = 7 + 8 - 1 = 6$.
  Now imagine the transmitted ACK(s) are lost. The destination state is st. it expects to receive *subsequent* segments numbered 7, 0, 1 etc. but the source retransmits the *original* segments whose sequence numbers were 0 through to 6. At this point we have a problem: the numbering is ambiguous, in that the destination (re)accepts the retransmitted original segments having incorrectly identified them as being different, subsequent segments.
- The reasoning behind the solutions is as follows:
  - The easiest case is obviously stop-and-wait: it prevents the source from ever transmitting subsequent segments until the original one has been ACK'ed; this means nothing can ever be received out-of-order. So, in order to prevent ambiguity we just need a 1-bit flag (i.e., sequence numbers st. $n = 2$).
  - The case of sliding-window is more tricky. Intuitively, we want more sequence numbers than the largest range the source and destination windows can span (at a given point in time). This is where said windows are disjoint, meaning a span of $w_s + w_d$. So put formally, we want
    $$n - 1 > w_s + w_d$$
    or, if the window sizes are equal, i.e., $w = w_s = w_d$,
    $$n > w_s + w_d = 2 \cdot w$$
    which could be written as
    $$n/2 = 2^{n'}/2 = 2^{n'-1} > w$$
    when n is a power-of-two.

TCP (10) – Adaptive retransmission

- Problem: to implement any ARQ-based scheme we need to select τ, but
  - too *small* a τ means we provoke many spurious retransmissions, and
  - too *large* a τ means we under-utilise the available bandwidth,

  i.e., τ relates to the **Goldilocks principle** [2]: we need it to be "just right" ...

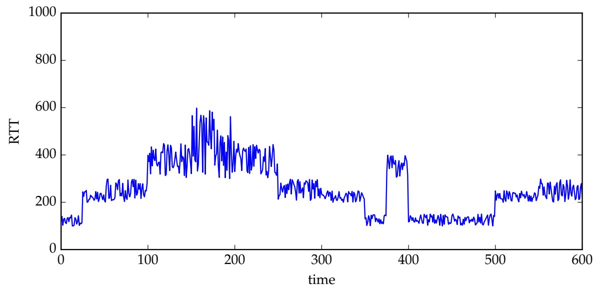
  1. for a LAN this is easy
     - RTT is typically small,
     - RTT has low variation,

     but
  2. for an inter-network this is not easy
     - RTT is typically large,
     - RTT has high variation.

Notes:

- By spurious retransmissions we mean retransmissions that could have been avoided if we'd waited a little longer: the associated segment *was* delivered and so was the acknowledgement, but we timed-out before this fact was known. Put another way, too small a τ means we guess too early (and hence often incorrectly) about the need to retransmit.

TCP (11) – Adaptive retransmission



- Note that:
  - the baseline RTT of ~ 100µs relates to the transmission and propagation delay, and
  - the noise in RTT samples comes from sources such as variation in routing and the load on hosts and the network.

Notes:

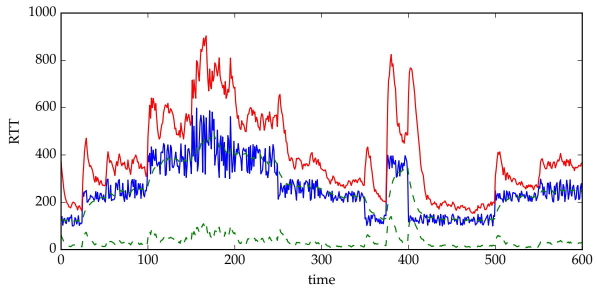
TCP (12) – Adaptive retransmission

- **Solution**: use **adaptive retransmission** [15, Section 2].
  1. Let
R_i	denote the measured RTT at time i
V_i	denote the smoothed RTT variance at time i
S_i	denote the smoothed RTT at time i
  2. Update the smoothed estimate via a moving average, i.e.,
     $$\begin{array}{lclcl} V_{i+1} & = & (1-\beta)\cdot V_i & + & \beta\cdot|R_i - S_i| \\ S_{i+1} & = & (1-\alpha)\cdot S_i & + & \alpha\cdot R_i \end{array}$$
     where $\alpha = \frac{1}{8}$ and $\beta = \frac{1}{4}$.
  3. Set τ_i (i.e., the value of τ at at time i) to
     $$\tau_i = S_i + K\cdot V_i$$
     where $K = 4$.

Notes:

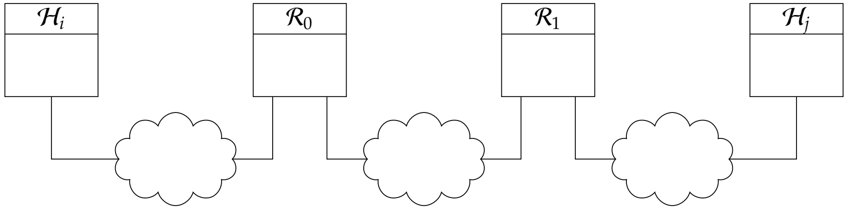
- This description omits a subtlety whereby a clock granularity *can* be considered (per [15, Section 4]) when computing τ. This just means using
  $$\tau_i = S_i + \max(G, K\cdot V_i)$$
  for some clock granularity G: the idea is that, for example, if we have computed $K\cdot V_i \simeq 0$ then this *could* be because we simply cannot measure accurately enough, so rounding up to G (i.e., how accurately we can measure) makes more sense.
- We need to initialise τ somehow: [15, Section 2] says
  1. before we have *any* RTT samples, set
     $$\tau_0 = 3s,$$
     then
  2. once we have an RTT sample, set $V_0 = R/2$, $S_0 = R$ and then
     $$\tau_1 = S_i + K\cdot V_i$$
     where $K = 4$.
- An underlying assumption is that the RTT samples accurately reflect the real RTT. In reality, there various features of TCP itself (e.g., the fact an ACK is not explicitly associated with retransmitted segments or the original transmission) and the network can result in inaccuracy; this is catered for by **Karn's Algorithm** [11], for instance.

TCP (13) – Adaptive retransmission



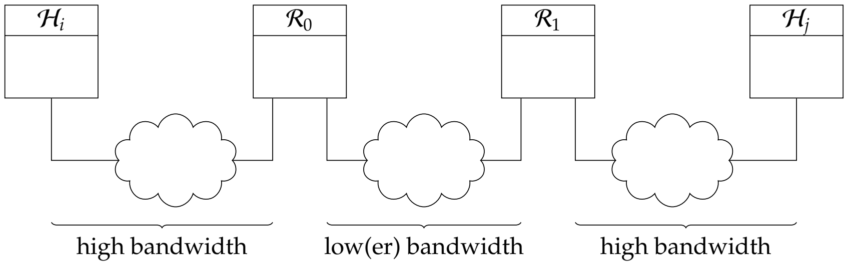
Notes:

- Fact: **ACK clocking** already "smooths" traffic somewhat to fit available bandwidth.



Notes:

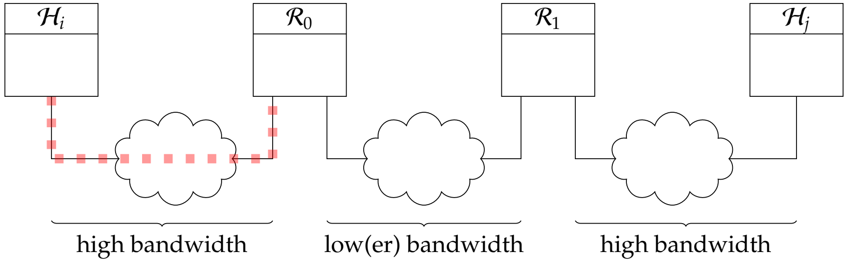
- Fact: **ACK clocking** already "smooths" traffic somewhat to fit available bandwidth.



Notes:

TCP (14) – Congestion control ⇝ ACK clocking

- **Fact**: **ACK clocking** already “smooths” traffic somewhat to fit available bandwidth.



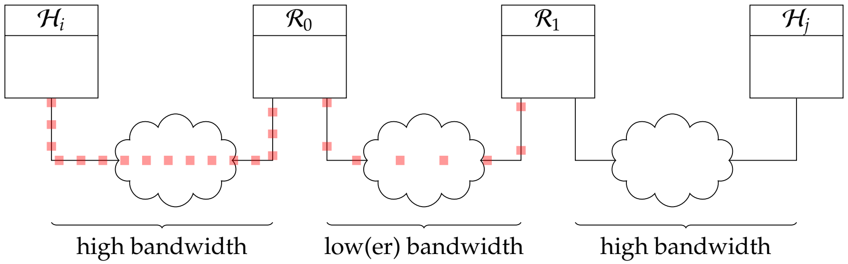
i.e.,

- $\mathcal{H}_0$ transmits segments at a high rate,

Notes:

TCP (14) – Congestion control ⇝ ACK clocking

- **Fact**: **ACK clocking** already “smooths” traffic somewhat to fit available bandwidth.

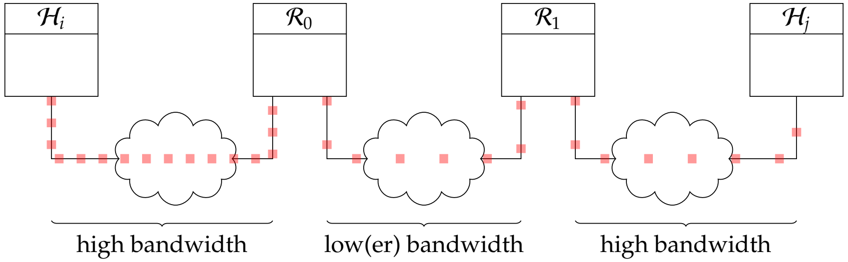


i.e.,

- $\mathcal{H}_0$ transmits segments at a high rate,
- $\mathcal{R}_0$ *must* buffer segments to cope with constrains of next hop,

Notes:

- Fact: **ACK clocking** already "smooths" traffic somewhat to fit available bandwidth.



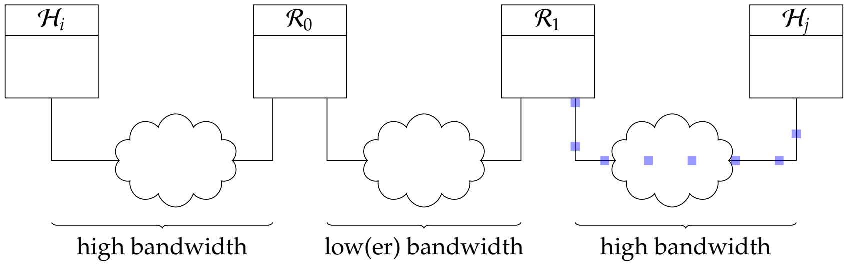
i.e.,

- $\mathcal{H}_0$ transmits segments at a high rate,
- $\mathcal{R}_0$ *must* buffer segments to cope with constrains of next hop,
- $\mathcal{H}_1$ receives segments and transmits ACKs at a low(er) rate,

Notes:

TCP (14) – Congestion control ⇝ ACK clocking

- Fact: **ACK clocking** already "smooths" traffic somewhat to fit available bandwidth.



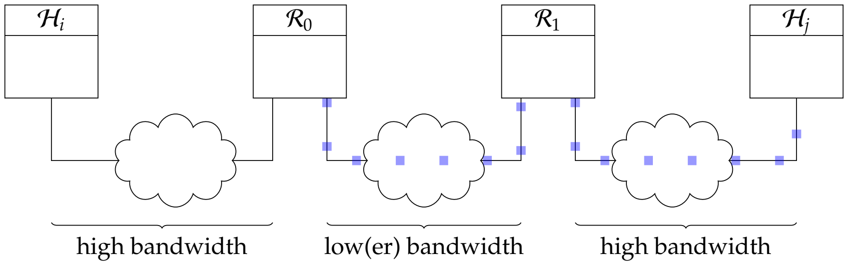
i.e.,

- $\mathcal{H}_0$ transmits segments at a high rate,
- $\mathcal{R}_0$ *must* buffer segments to cope with constrains of next hop,
- $\mathcal{H}_1$ receives segments and transmits ACKs at a low(er) rate,

Daniel Page (Daniel.Page@bristol.ac.uk)
Concurrent Computing (Computer Networks) git # 3627080 @ 2016-03-11 University of BRISTOL

Notes:

- Fact: **ACK clocking** already "smooths" traffic somewhat to fit available bandwidth.

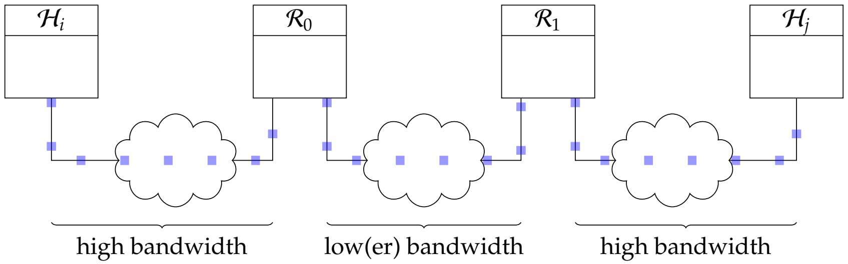


i.e.,

- $\mathcal{H}_0$ transmits segments at a high rate,
- $\mathcal{R}_0$ *must* buffer segments to cope with constrains of next hop,
- $\mathcal{H}_1$ receives segments and transmits ACKs at a low(er) rate,

Notes:

- Fact: **ACK clocking** already "smooths" traffic somewhat to fit available bandwidth.

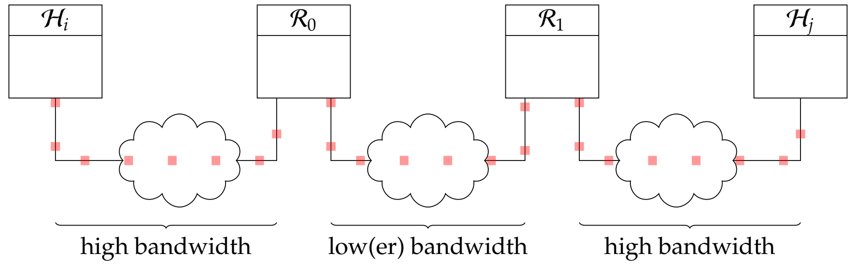


i.e.,

- $\mathcal{H}_0$ transmits segments at a high rate,
- $\mathcal{R}_0$ *must* buffer segments to cope with constrains of next hop,
- $\mathcal{H}_1$ receives segments and transmits ACKs at a low(er) rate,
- $\mathcal{H}_0$ receives ACKs and transmits segments at a low(er) rate,
- $\mathcal{R}_0$ *needn't* buffer segments to cope with constrains of next hop.

Notes:

- Fact: **ACK clocking** already "smooths" traffic somewhat to fit available bandwidth.



i.e.,

- $\mathcal{H}_0$ transmits segments at a high rate,
- $\mathcal{R}_0$ *must* buffer segments to cope with constrains of next hop,
- $\mathcal{H}_1$ receives segments and transmits ACKs at a low(er) rate,
- $\mathcal{H}_0$ receives ACKs and transmits segments at a low(er) rate,
- $\mathcal{R}_0$ *needn't* buffer segments to cope with constrains of next hop.

Notes:

- Problem: **silly window syndrome(s)**.
  1. Slow producer application on source, e.g., invokes `send` wrt. 1B messages:
     - bad: high overhead of header wrt. payload,
     - bad: more in-flight segments, meaning higher congestion.

- Solution:
  1. Implement **Nagle's Algorithm** [14], st. for "small" `send` we buffer until a) ACK arrives, or b) MSS reached.

Notes:

- The best way of thinking about Nagle's Algorithm is as a control mechanism to decide when TCP should transmit (buffered) data: it allows transmission to be decoupled with buffering (via `send`), st. a sane policy can be enforced (in this case, to cope with "small" `send`).
- [8] specifies that the delayed ACK *must* be less than 500ms, so one could view the precise choice as a tunable parameter. Either way, the net result is that the delay
  - gives time for subsequent `recv` to consume data,
  - prevents source from advancing its window, so slows down to better align with `recv` rate, and
  - reduces total number of ACKs, due to higher chance for a) larger cumulative ACK and/or b) ACK piggybacking.

- Problem: **silly window syndrome(s)**.

2. Slow consumer application on destination e.g., invokes `recv` wrt. 1B messages:
   - bad: efficiency reduced to that of stop-and-wait,
   - bad: more in-flight ACKs, meaning higher congestion.

- Solution:

2. Implement **delayed ACKs**, st. for "small" `recv` we delay upto 200ms before transmitting the associated ACK.

Notes:

- The best way of thinking about Nagle's Algorithm is as a control mechanism to decide when TCP should transmit (buffered) data: it allows transmission to be decoupled with buffering (via `send`), st. a sane policy can be enforced (in this case, to cope with "small" `send`).
- [8] specifies that the delayed ACK *must* be less than 500ms, so one could view the precise choice as a tunable parameter. Either way, the net result is that the delay
  - gives time for subsequent `recv` to consume data,
  - prevents source from advancing its window, so slows down to better align with `recv` rate, and
  - reduces total number of ACKs, due to higher chance for a) larger cumulative ACK and/or b) ACK piggybacking.

Quote

When TCP is used for the transmission of single-character messages originating at a keyboard, the typical result is that 41 byte packets (one byte of data, 40 bytes of header) are transmitted for each byte of useful data. This 4000% overhead is annoying but tolerable on lightly loaded networks. On heavily loaded networks, however, the congestion resulting from this overhead can result in lost datagrams and retransmissions, as well as excessive propagation time caused by congestion in switching nodes and gateways. In practice, throughput may drop so low that TCP connections are aborted.

– *Nagle* [14], 1984

Algorithm (Nagle's Algorithm [14])

When `send` is invoked, i.e., there is new data to transmit:

1. if available data size >= MSS and window size >= MSS, then transmit complete segment (of size MSS),
2. otherwise
   2.1 if there are un-ACK'ed segments, then buffer data until an ACK is received,
   2.2 otherwise transmit incomplete segment (of size < MSS).

Notes:

- The first decision is basically whether or not enough data (both buffered, and newly provided via `send`) exists to form a complete MSS-sized segment; there also needs to be enough space in the sliding window to accommodate it. If so, we can transmit that complete segment immediately.
  In simple terms, therefore, the algorithm attempts to capture the following rule: for as long as there is at least one un-ACK'ed segment, and the buffer isn't filled, buffer the data rather than transmitting it immediately.

Conclusions

- Take away points:
  - The transport layer interfaces applications with the network ...
  - ... *it* must (and does) cope with numerous demands, e.g.,
    - clean interface vs. diverse, complex network, and
    - efficiency vs. unreliable, unpredictable network.
  - This means TCP is complicated (UDP less so), in that
    - it supports a *lot* of functionality,
    - it evolves over time, sometimes retaining mechanisms based on assumptions or problems that no longer hold, meaning
    - interaction between mechanisms is sometimes hard to predict (and so sometimes undesirable, cf. Nagle's Algorithm vs. delayed ACKs [13]).

  and continues to
- Additional topics: a (non-exhaustive) list could include at least
  - **congestion control**,
    - (more or less) TCP-agnostic approaches, e.g., **slow start**, **fast retransmit**, **fast recovery**, **Additive Increase/Multiplicative Decrease (AIMD)**,
    - (more or less) TCP-specific approaches, e.g., **Explicit Congestion Notification (ECN)** [16],
    - TCP-specific implementations, e.g., Tahoe, Reno, Vegas, New Reno, Hybla, ...
  - **window scaling** [10] which allows larger sliding-window buffer sizes.

Notes:

References

[1] Wikipedia: Flow control.
http://en.wikipedia.org/wiki/Flow_control_(data).

[2] Wikipedia: Goldilocks principle.
http://en.wikipedia.org/wiki/Goldilocks_principle.

[3] Wikipedia: Karn's algorithm.
http://en.wikipedia.org/wiki/Karn's_algorithm.

[4] Wikipedia: Nagle's algorithm.
http://en.wikipedia.org/wiki/Nagle's_algorithm.

[5] Wikipedia: Silly window syndrome.
http://en.wikipedia.org/wiki/Silly_window_syndrome.

[6] Wikipedia: Sliding window protocol.
http://en.wikipedia.org/wiki/Sliding_window_protocol.

[7] Wikipedia: Transmission control protocol.
http://en.wikipedia.org/wiki/Transmission_Control_Protocol.

[8] R. Braden.
Requirements for Internet hosts – communication layers.
Internet Engineering Task Force (IETF) Request for Comments (RFC) 1122, 1989.
http://tools.ietf.org/html/rfc1122.

Notes:

References

[9] D.D. Clark.
Window and acknowledgement strategy in TCP.
Internet Engineering Task Force (IETF) Request for Comments (RFC) 813, 1982.
http://tools.ietf.org/html/rfc813.

[10] V. Jacobson, R. Braden, and D. Borman.
TCP extensions for high performance.
Internet Engineering Task Force (IETF) Request for Comments (RFC) 1323, 1992.
http://tools.ietf.org/html/rfc1323.

[11] P. Karn and C. Partridge.
Improving round-trip time estimates in reliable transport protocols.
ACM SIGCOMM Computer Communication Review, 17(5):2–7, 1987.

[12] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow.
TCP selective acknowledgment options.
Internet Engineering Task Force (IETF) Request for Comments (RFC) 2018, 1996.
http://tools.ietf.org/html/rfc2018.

[13] G. Minshall, Y. Saito, J.C. Mogul, and B. Verghese.
Application performance pitfalls and TCP's Nagle algorithm.
ACM SIGMETRICS Performance Evaluation Review, 27(4):36–44, 2000.

Notes:

References

[14] J. Nagle.
Congestion control in IP/TCP internetwork.
Internet Engineering Task Force (IETF) Request for Comments (RFC) 896, 1984.
http://tools.ietf.org/html/rfc896.

[15] V. Paxson and M. Allman.
Computing TCP's retransmission time.
Internet Engineering Task Force (IETF) Request for Comments (RFC) 2988, 2000.
http://tools.ietf.org/html/rfc2988.

[16] K. Ramakrishnan, S. Floyd, and D. Black.
The addition of Explicit Congestion Notification (ECN) to IP.
Internet Engineering Task Force (IETF) Request for Comments (RFC) 3168, 2001.
http://tools.ietf.org/html/rfc3168.

[17] W. Stallings.
Chapter 23: Transport protocols.
In *Data and Computer Communications*. Pearson, 9th edition, 2010.

Notes:

Daniel Page ⟨Daniel.Page@bristol.ac.uk⟩
Concurrent Computing (Computer Networks) git # 3627080 @ 2016-03-11
University of BRISTOL