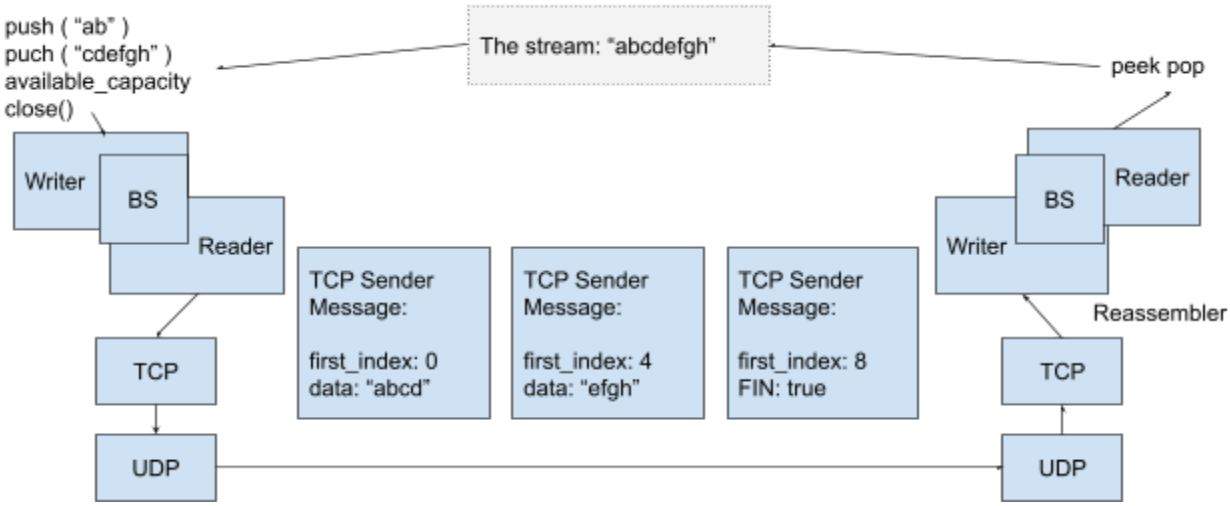
- Stacks of service abstraction
    - Short gets (DNS, DHCP) -> User datagrams -> Internet Datagrams
    - Byte Stream –(TCP)--> User datagrams -> Internet Datagrams
        - Web requests/responses (`HTTP`) -> Byte Stream
            - Youtube/Wikipedia -> Web requests/responses
        - Email sending (SMTP) -> Byte Stream
        - Email receiving (IMAP) -> Byte Stream
- Multiplexing ByteStream
    - “u8 u8” (Which byte stream; what is the byte)
        - Any reading and writing of one byte would be actually two bytes, the first byte for which byte stream and the second for the actual byte
    - “u8 u8 {u8 u8 … u8}” (which byte stream; size of payload; sequence of characters of the string chunk)
        - Any reading and writing of n bytes would be actually n + 2 bytes
        - Tagged byte stream: HTTP/2 | SPDY
- How to make ByteStream push idempotent?
    - TCP Sender Message

      first_index

      data

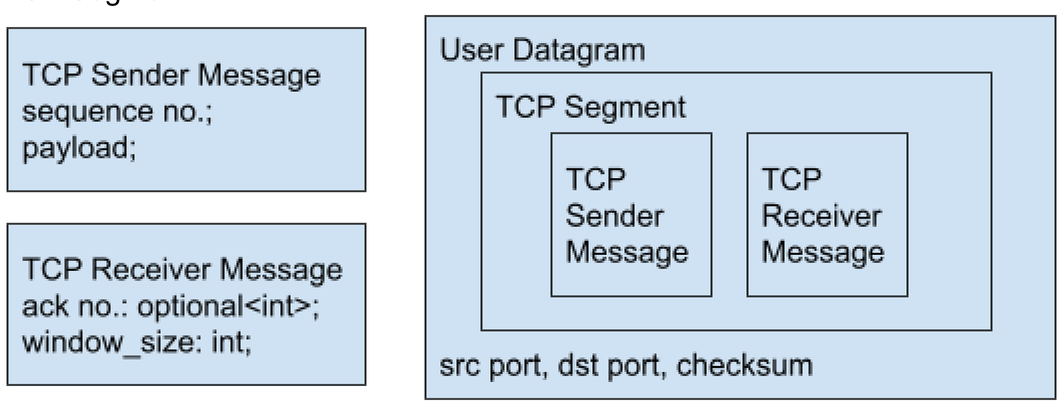
      FIN



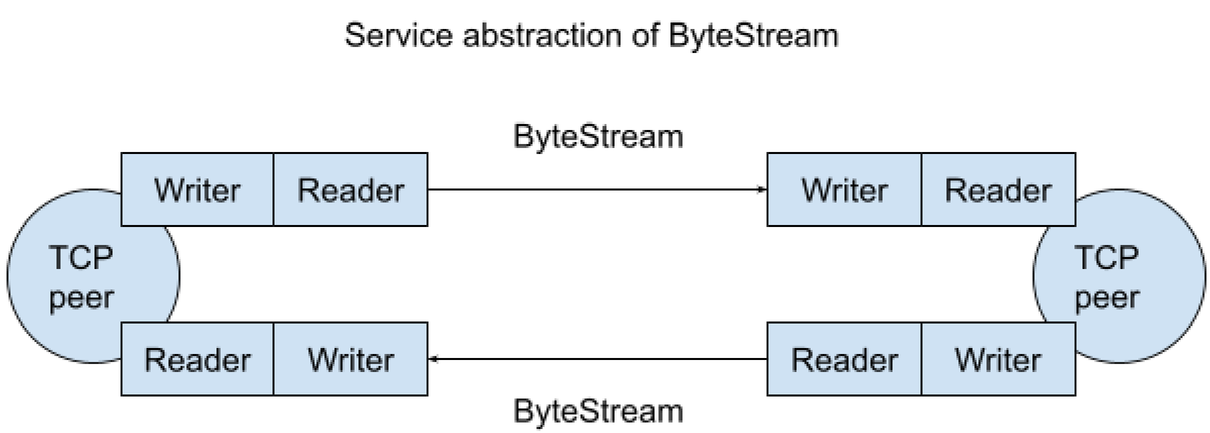
    - This works for out-of-order or multiple deliveries. Since UDP has a checksum field, altered TCP Message would be ignored on the UDP layer.
    - What if datagrams are missing?
        - How does the sender know that a datagram needs to be sent multiple times?
        - DNS/DHCP: if we don’t receive an answer, then we retransmit. But such response/answer does not exist in `pushing` (void push())
    - Acknowledgement
        - TCP Receiver Message
            - “A B C D E F G” each byte sent as a separate TCP sender message, and “D” is not received.

        - "I got the sender message with first-index = 2, length = 1."
            - Valid but more work. There will be one receiver message for each sender message.
        - "Got anything? Y/N. Next needed: #3."
            - Acknowledgements are accumulative, and that make life simpler.
        - Give FIN flag a number: "A B C D E F G FIN".
          TCP Sender Message: {sequence number, data}
          TCP Receiver Message: {Next needed: optional<int>}
          and TCP Receiver Message {Next needed: optional(8)} would mean FIN is received.
        - TCP Receiver Message: {Next needed: optional<int>; available capacity:int}
            - {Next needed: optional(3); available capacity: 2} === Receiver wants to here about "DE".
            - "DE" is the **window**. (Red area in that picture of lab 1)
    - **TCP Receiver Message: {Ack no: optional<int>; window size: int}**

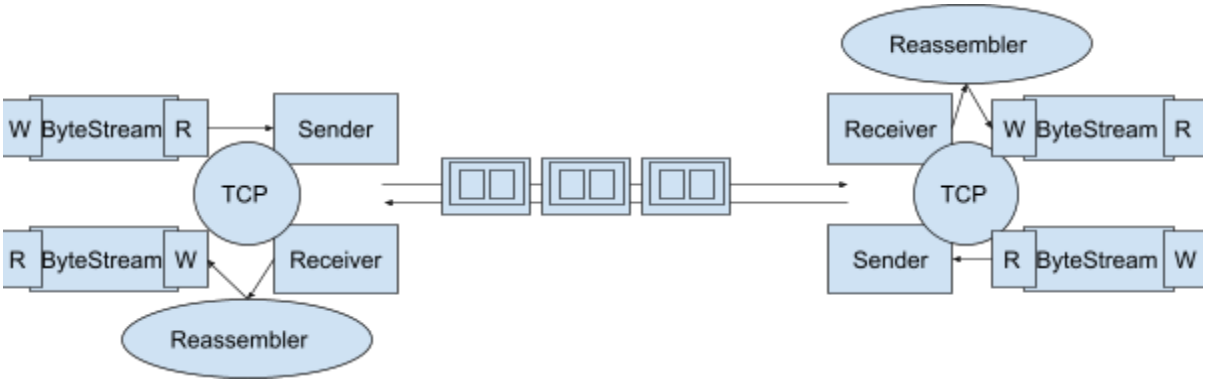
- TCP Segment



- This is the service abstraction that TCP is providing:



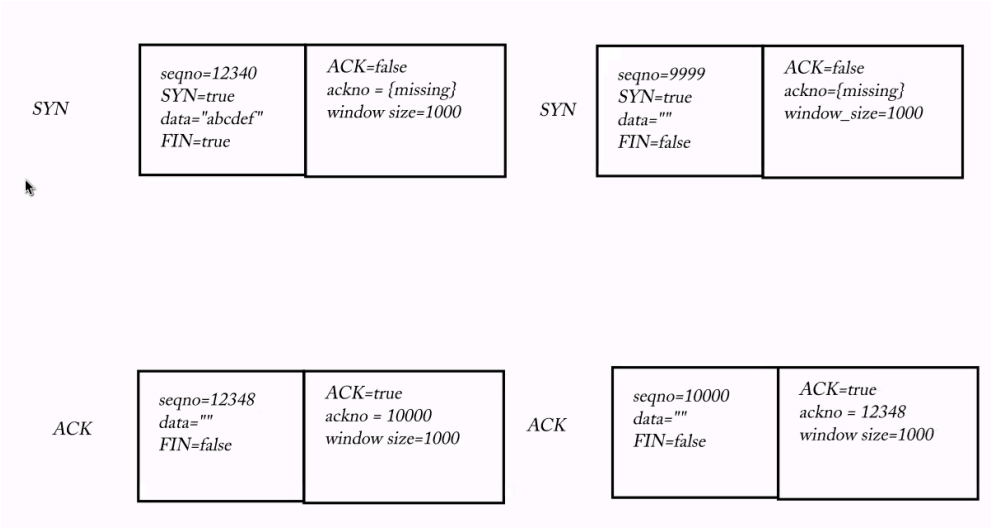
- And this is what happens under the hood (and also what you will be implementing in the labs)



- Rules of TCP
    - Reply to any nonempty TCP Server Message

- What happens when a stream ends?
    - My sender has ended its outgoing bytestream, but the incoming bytestream from the peer may not be ended.
    - When a stream ends, can the same pair of ports be used? Reusing the same pair of ports makes it not clear to tell whether a datagram belongs to the old stream or the new stream.
    - We want a new INCARNATION of the connection (new connection on the same pair of ports)
    - **Sequence number**: start from a random big number + **SYN**: this sequence number should be viewed as the beginning of a stream
        - If the sequence number doesn't make sense on the old stream, and the SYN flag is true, the receiver knows this is a new incarnation of the connection.
        - e.g. {sequnce_no=12345, data="abcdef", SYN=true, FIN=true}, and {sequence_no=99999, data="xyz", SYN=true, FIN=true}
    - First seqno belongs to SYN flag, next seqnos belong to each byte of stream, final seqno belongs to FIN flag.
        - It is very important to have SYN flag and FIN flag delivered reliably, so therefore receiver need to acknowledge SYN seqno and FIN seqno
- What happens to TCP receiver message's next_needed_idx field before receiving the SYN flag from the peer?
    - Without seqno:
        - I: {{first_index=0, data="abcdef", FIN=true}, {next_needed=0, window_size=1000}}
        - Peer: {{first_index=0, data="", FIN=true}, {next_needed=7, window_size=1000}}
        - I: {{first_index=7, data="", FIN=false}, {next_needed=1, window_size=1000}}
    - With seqno and SYN:

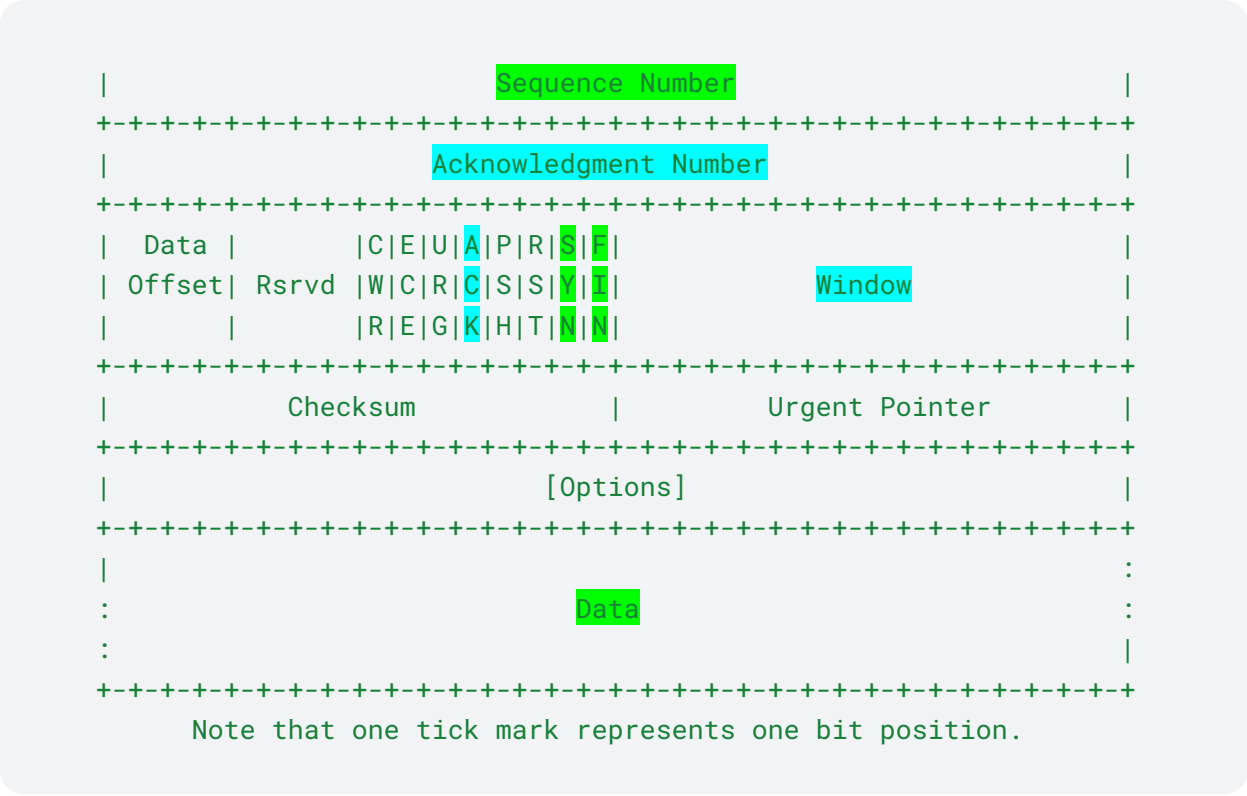
        - I: {{seqno=12340, SYN=true, data="abcdef", FIN=true}, {**What should this be? (before seeing 9999 from the Peer**)}}
        - Peer: {{seqno=9999, SYN=true, data="", FIN=true}, {next_needed=12348, window_size=1000}}
        - I: {{next_needed=10001, window size =1000}}
    - ackno = optional<int> (a pair of ACK flag and ackno int)
        - I: {{seqno=12340, SYN=true, data="abcdef", FIN=true}, {ACK=false, ackno={missing}, window_size=1000}} (SYN)
        - Peer: {{seqno=9999, SYN=true, data="", FIN=true}, {ACK=true, ackno=12348, window_size=1000}} (SYN+ACK)
        - I: {{ACK=true, ackno=10001, window size =1000}} (ACK)
    - (SYN) + (SYN+ACK) + (ACK) = "the three-way handshake"
    - What if the two SYN messages are sent at the same time?



    -
        - Not a classic "three-way handshake" but still a valid way of starting a TCP connection.
- Standardized TCP Message:
    - Sender: {sequence number, SYN, data, FIN}
    - Receiver: {ackno: optional<int>, window_size}
    - "User Datagram" info

https://www.rfc-editor.org/rfc/rfc9293.html#name-header-format

```
Unset
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |          Source Port          |       Destination Port        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```



- Wireshark tool