- Last time: “best effort” delivery as the service abstraction
    - Not delivered -> **timeout + retransmit**
    - Delivered n> 1 times -> **transform operations to be idempotent**
    - Delivered altered -> **checksum or crypto**
    - Delivered out of order -> **sequence number**
    - On top of this service abstraction, we can build:
        - VoIP
        - User Datagrams
        - VPN (IP-in-UDP/IP-in-IP/IPsec)
            - Q: How does Netflix determine where an IP address is actually from?
            - A: Netflix would look at the IP addresses provided by VPN services and ban those IP addresses.
- Short get: get(key) -> value
    - E.g. `host`: what is the IP address that corresponds to a host?
        - With packet loss, it takes a longer time to reply, but would still give an answer
        - This service is “reliable” despite the fact that it is built on a unreliable “best effort” service abstraction

```cpp
C/C++
// Server
void recv ( const string& service ) {
	UDPSocket sock;
	sock.bind ( Address (“0”, service) );
	Address source (“0”);
	string payload;
	while (true) {
		sock.recv( source, payload);
		cout << “Message from” << source.to_string() << “: “
	<< payload << endl;
		if (payload == "best_class_ever" ) {
			sock.sendto( source, "EE180");
		}
	}
}
```

```C/C++
// Sender
void run( const string& host, const string& service, const
string& query) {
      UDPSocket sock;
      sock.set_blocking( false );
      Address source ("0");
      string answer;

      // retransmit the query (with a small timeout), until there
is a reply
      do {
            sock.sendto(Address(host, service), query);
            this_thread::sleep_for(seconds(1));
            sock.recv(source, answer);
            if (answer.empty()) {
                  cerr << "No reply, retransmitting" << endl;
            }
      } while (answer.empty())

      cout << "Got reply to " << query << ": " << answer << endl;
}
```

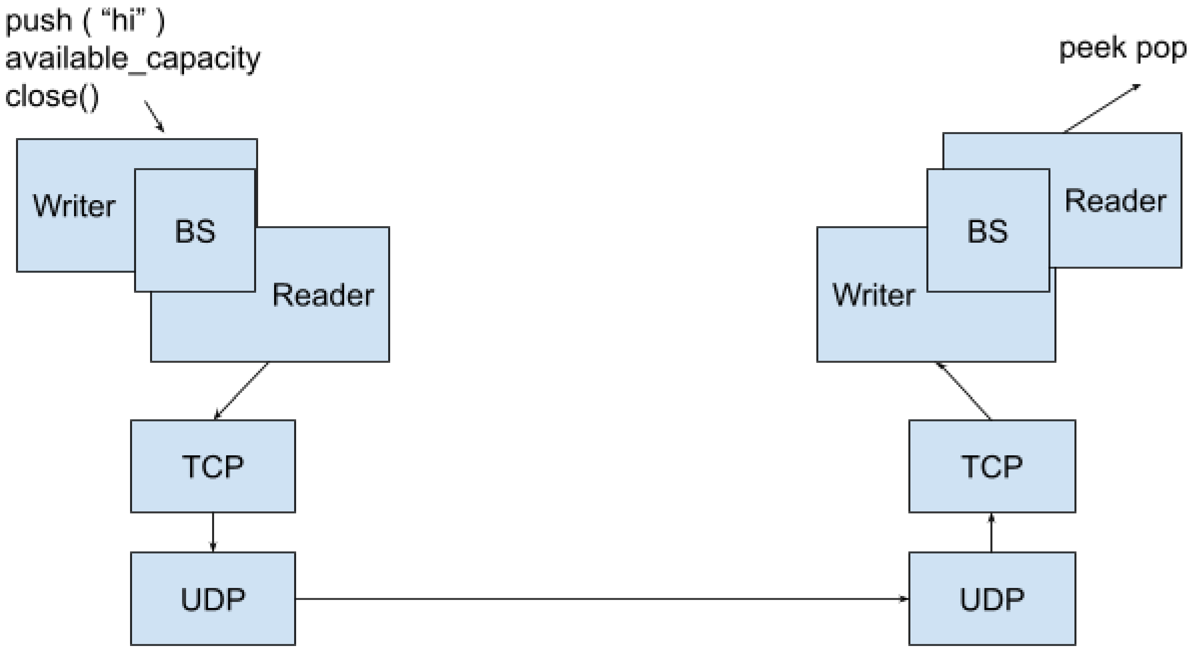
  - By doing this, we implement a “reliable” service on top of an “unreliable” service abstraction, and this is also how many real-word reliable services are built (e.g. `host`).
    - And also: Domain Name System (DNS): what is the IP address of an internet domain name?
    - DHCP (Dynamic Host Configuration Protocol): what is the IP address I am supposed to use?
- Set: (e.g. set the back door open)
  - Both short get and set (the back door open), you could say how ever many times you want and it does not change the ending state
  - But for `` `pop(7)` ``, `` `push(“hi”)` ``, it matters how many times you say it.
  - Idempotent: doing one time or more than one time does not change the ending state (GET PUT). The strategy we used above works for something idempotent, but not for non-idempotent action
- Do a non-idempotent operation (POST):

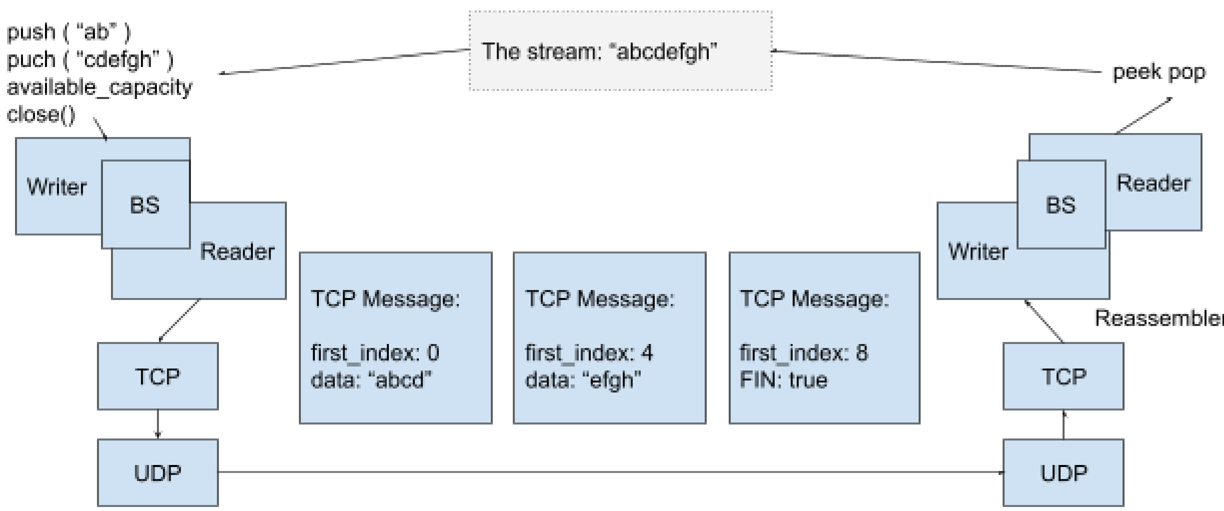
  By having a set of launched missiles, we make launch_missle idempotent

```
C/C++
// Server
void launch_missle() {
     cout << "Launching one missle" << endl;
}

void recv ( const string& service ) {
     unordered_set<uint64_t> launched_missle;
     UDPSocket sock;
     sock.bind ( Address (“0”, service) );
     Address source (“0”);
     string payload;
     while (true) {
          sock.recv( source, payload);
          cout << “Message from” << source.to_string() << “: “
          << payload << endl;
          if (payload == "best_class_ever" ) {
               sock.sendto(source, "EE180");
          } else if (payload == "launch_one_missle" + missle_id
          ) {
               if (missle_id not in launched_missle ) {
                    launch_missle();
                    launched_missle.insert(missle_id);
               }
               sock.sendto(source, "ack");
          }
     }
}
```

- ByteStream: push, pop, peek needs to be transformed into idempotent operations, and this is achieved by **TCP**

- 

- What should be in the TCP Sender message to make these operations idempotent?
    - `push (“abcd”)` works iff each message is delivered exactly once
    - `push(“abcd”) + message unique id`, but the sender needs to keep a set of any message sent
    - Create a reassembler, `first_index: 0, data: “abcd”` `first_index: 4, data: “efgh”, `first_index = 8, FIN=true`
- 

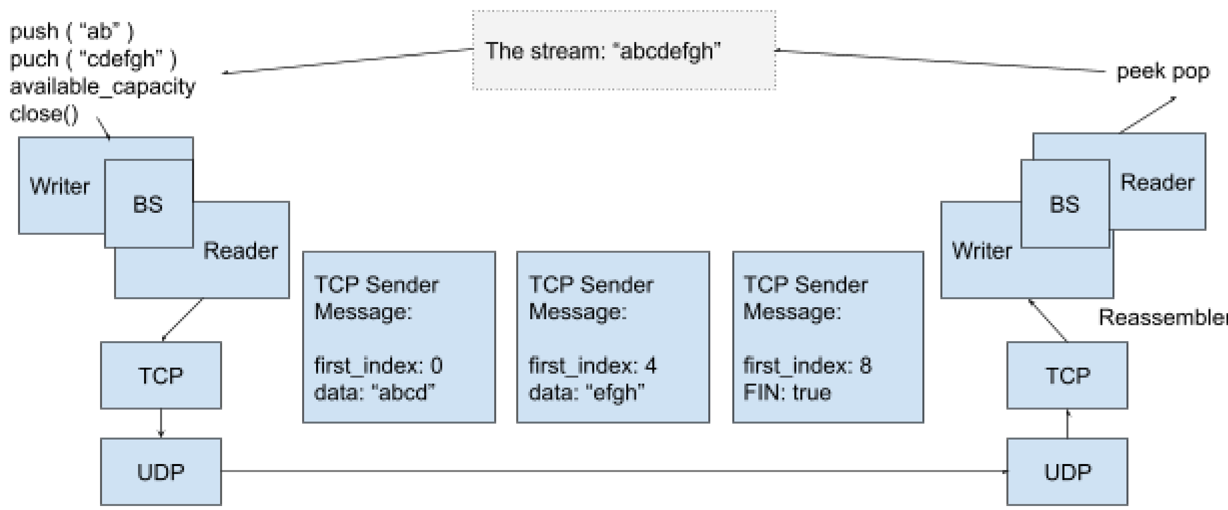
- Stacks of service abstraction
    - Short gets (DNS, DHCP) -> User datagrams -> Internet Datagrams
    - Byte Stream –(TCP)--> User datagrams -> Internet Datagrams
        - Web requests/responses (`HTTP`) -> Byte Stream
            - Youtube/Wikipedia -> Web requests/responses
        - Email sending (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

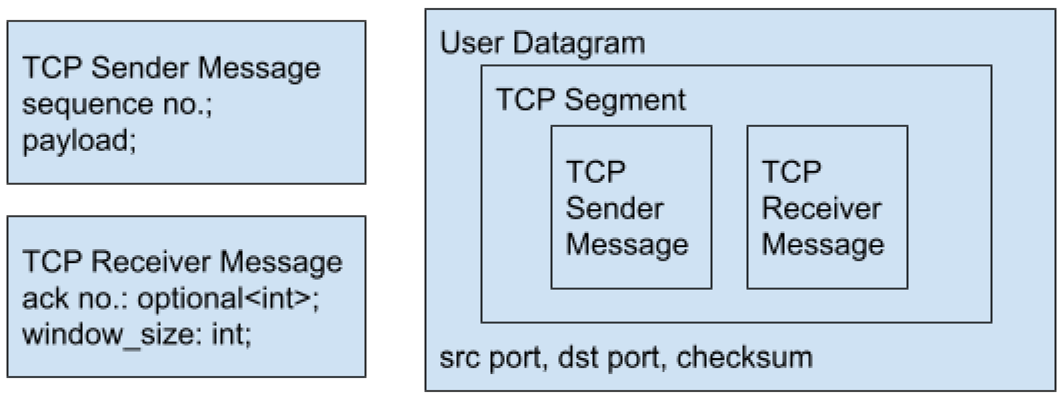
    FIN



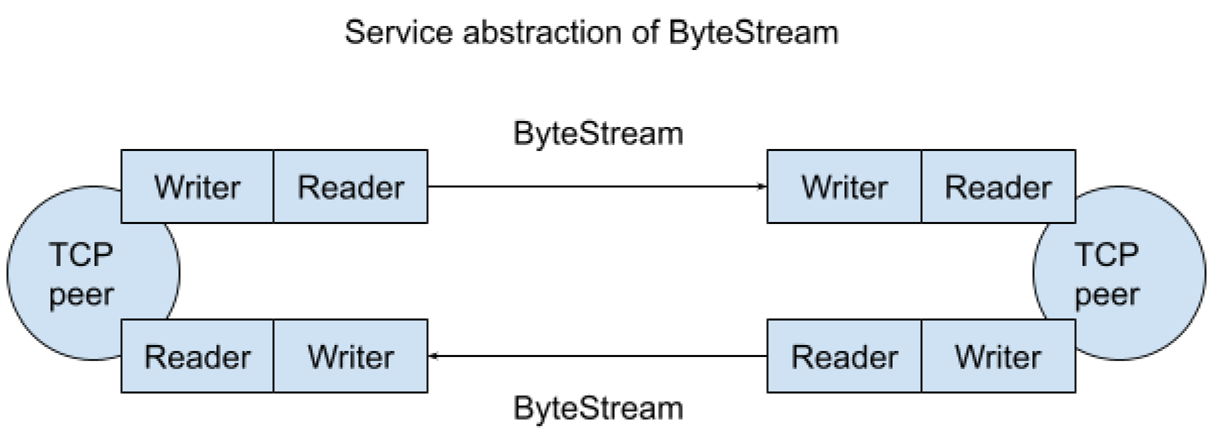
  - This works for out-of-order or multiple deliveries. Since UDP has a checksum field, altered TCP Message would be ignored on the UDP layer.
  - What if datagrams are missing?
    - How does the sender know that a datagram needs to be sent multiple times?
    - DNS/DHCP: if we don't receive an answer, 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}**

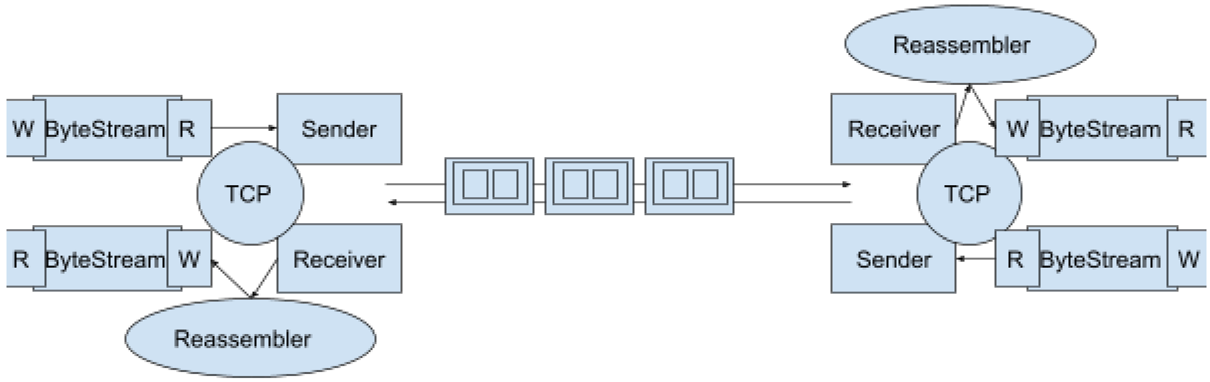
- TCP Segment



- This is the service abstraction that TCP is providing:

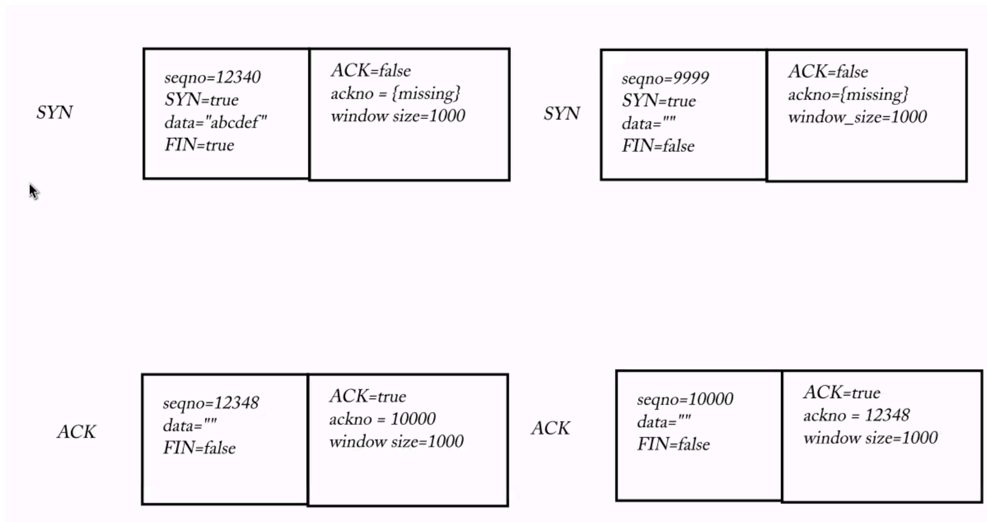


- And this is what happens under the hood (and also what you will be implementing in the labs)



- 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        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |       |C|E|U|A|P|R|S|F|                               |
| Offset| Rsrvd |W|C|R|C|S|S|Y|I|            Window             |
|       |       |R|E|G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           [Options]                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               :
:                             Data                              :
:                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      Note that one tick mark represents one bit position.
```

- Wireshark tool