University of Glasgow | School of Computing Science

Networks & Operating Systems Essentials

Dr Angelos Marnerides

<angelos.marnerides@glasgow.ac.uk>

School of Computing Science, Room: S122

Today, on NOSE2…

I'd tell you a UDP joke
but you might not get it…

What is the best part about TCP jokes?

I get to keep telling them until you get them…

Based on slides © 2017 Colin Perkins

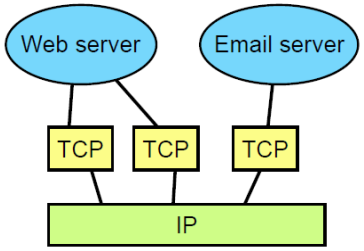
TRANSPORT LAYER (L4)

The Transport Layer

- Role
  - Isolate upper layers from the network layer
    - Hide network complexity
    - Make unreliable network appear reliable
    - Enhance QoS of network layer
  - Provide a useful, convenient, easy to use service
    - An easy to understand service model
    - An easy to use programming API
      - The Berkeley sockets API – very widely used by application programmers
      - Compare to network layer API – usually hidden in operating system internals
- Functions:
  - Addressing and multiplexing
  - Reliability
  - Framing
  - Congestion control
- Operates process-to-process, not host-to-host

The Transport Layer

- Addressing & multiplexing:
  - The network layer address identifies a host
  - The transport layer address identifies a user process – a service – running on a host
- Reliability
  - Network layer is unreliable
    - *Best effort* packet switching in the Internet
    - But even nominally reliable circuits may fail
  - Transport layer enhances the quality of service provided by the network, to match application needs
    - Appropriate *end-to-end* reliability
  - Different applications need different reliability
    - Email and file transfer → all data must arrive, in the order sent, but no strict timeliness requirement
    - Voice or streaming video → can tolerate a small amount of data loss, but requires timely delivery



The Transport Layer

- Framing
  - Applications may wish to send structured data
    - Transport layer responsible for maintaining the boundaries
    - Transport must frame the original data, **if** this is part of the service model
- Congestion and Flow Control
  - Transport layer controls the application **sending** rate
    - To match rate at which network layer can deliver data – congestion control
    - To match rate at which receiver can process the data – flow control
  - Must be performed end-to-end, since only end points know characteristics of entire path
  - Different applications have different needs for congestion control
    - Email and file transfer → elastic applications; faster is better, but don't care about actual sending rate
    - Voice or streaming video → inelastic applications; have minimum and maximum sending rates, and care about the actual sending rate

The End-to-End Principle

- Is it better to place functionality within the network or at the end points?
  - Only put functions that are absolutely necessary within the network, leave everything else to end systems
    - Example: put reliability in the transport layer, rather than the network
    - If the network is not guaranteed 100% reliable, the application will have to check the data anyway
    - ➢ Don't check in the network, leave to the end-to-end transport protocol, where the check is visible to the application
  - One of the defining principles of the Internet

End-To-End Arguments in System Design

J. H. SALTZER, D. P. REED, and D. D. CLARK
Massachusetts Institute of Technology Laboratory for Computer Science

1. INTRODUCTION

Choosing the proper boundaries between functions is perhaps the primary activity of the computer system designer. Design principles that provide guidance in this choice of function placement are among the most important tools of a system designer. This paper discusses one class of function placement argument that has been used for many years with neither explicit recognition nor much conviction. However, the emergence of the data communication network as a computer system component has sharpened this line of function placement argument by making more apparent the situations in which and the reasons why it applies. This paper articulates the argument explicitly, so as to examine its nature and to see how general it really is. The argument appeals to application requirements and provides a rationale for moving a function upward in a layered system closer to the application that uses the function. We begin by considering the communication network version of the argument.

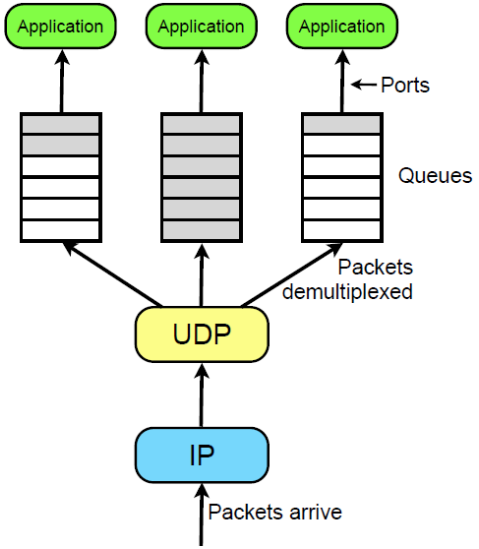
J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. ACM TOCS, 2(4):277–288, November 1984. http://dx.doi.org/10.1145/357401.357402

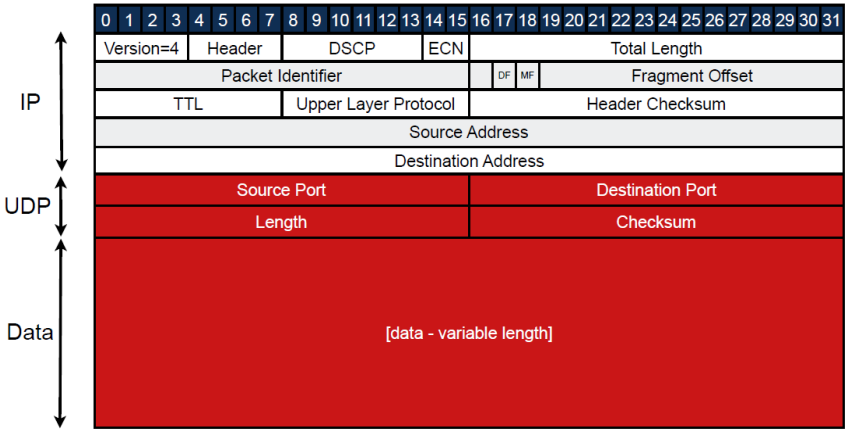
Internet Transport Protocols

- The Internet Protocol provides a common base for various transports
  - User Datagram Protocol (UDP)
  - Transmission Control Protocol (TCP)
  - *Datagram Congestion Control Protocol (DCCP)*
  - *Stream Control Transmission Protocol (SCTP)*
  - *QUIC (pronounced "quick")*
- Each makes different design choices

Internet Transport Protocols: UDP

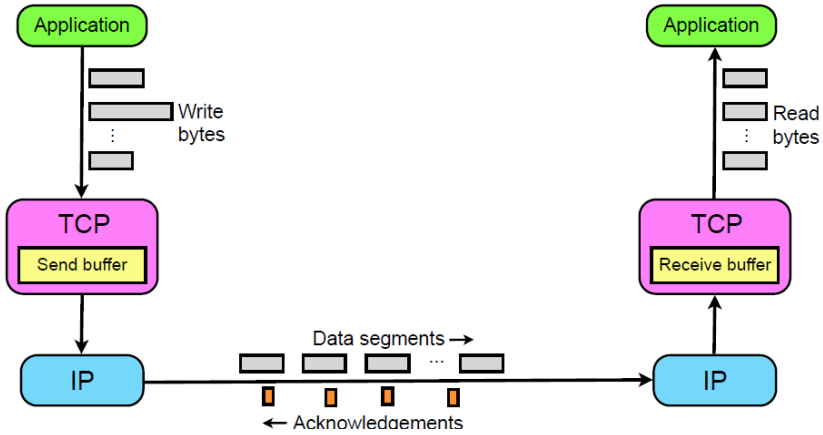
- Simplest transport protocol
- Exposes raw IP service to applications
  - Connectionless, best effort packet delivery: framed, but unreliable
  - No congestion control
- Adds a 16 bit port number as a service identifier
  - Port numbers in the range [0, 65535]

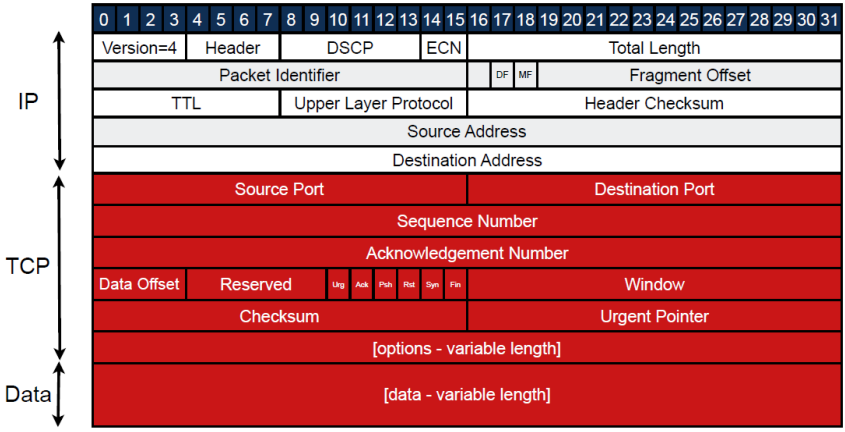




Internet Transport Protocols: TCP

- Reliable byte stream protocol over IP
  - Adds reliability
  - Adds congestion control
- Doesn't provide framing
  - Delivers an ordered byte stream, the application must impose structure
- Adds a 16 bit port number as a service identifier
  - Port numbers in the range [0, 65535]





Extra: Internet Transport Protocols: SCTP/DCCP

- Datagram Congestion Control Protocol (DCCP)
  - Unreliable, connection oriented, congestion controlled datagram service
    - "TCP without reliability" or "UDP with connections and congestion control"
    - Potentially easier for NAT boxes and firewalls than UDP
    - Congestion control algorithm ("CCID") negotiated at connection setup – range of algorithms supported
  - Adds 32 bit service code in addition to port number
  - Use case: streaming multimedia and IPTV
- Stream Control Transmission Protocol (SCTP)
  - Reliable datagram service, ordered per stream
  - Multiple streams within a single association
  - Multiple connection management
    - Fail-over from one IP address to another, for reliable multi-homing
  - TCP-like congestion control
  - Use case: telephony signalling; "a better TCP"
- Neither of these widely used at this time, but potentially useful in future

Extra: Internet Transport Protocols: QUIC

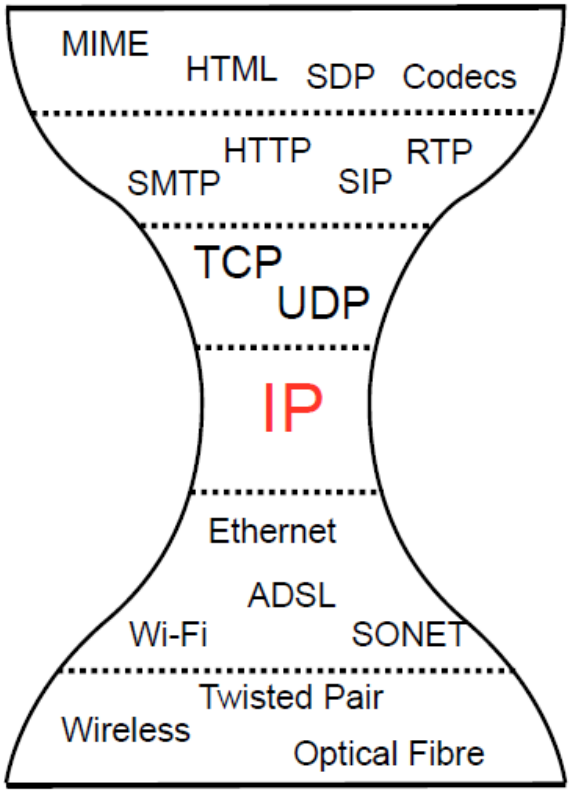
- QUIC
  - Extremely reliable,
  - Connectionless,
  - TCP-like congestion control but on the userspace
  - Reduced connection times,
  - Better convergence when packets are lost
  - More browser-friendly
  - ***Only one with encryption by default***
- *Widely used (e.g., Chrome, Firefox, Chrome2, Microsoft Edge), not standardized yet*

Internet Transport Protocols: Summary

Protocol	Addressing	Reliable?	Framed?	Congestion Controlled?
UDP	16 bit port number	Unreliable packet delivery	Yes – uses explicit datagrams	No – handled by application layer
TCP	16 bit port number	Reliable ordered byte stream	No – handled by application layer	Yes – suitable for elastic applications

Aside: Deployment Considerations

- IP is agnostic of the transport layer protocol
- But, firewalls perform "deep packet inspection" and look beyond the IP header to make policy decisions
  - The only secure policy is to disallow anything not understood
  - Implication: very difficult to deploy new transport protocols (e.g., DCCP and SCTP) in the Internet
  - Implication: limits future evolution of the network
- If protocols cannot be deployed natively, they can be tunnelled
  - WebRTC data channel → SCTP over DTLS over UDP
    - Peer-to-peer data for web applications
  - QUIC → multiplexed stream transport protocol running over UDP
  - UDP passes through NATs and firewalls, that native transport protocols do not
    - So tunnel new transport inside UDP packets



The Berkeley Sockets API

- Widely used low-level C networking API
  - First introduced in 4.3BSD Unix
  - Now available on most platforms: Linux, *BSD, MacOS X, Windows, etc.
  - Largely compatible cross-platform
- Sockets provide a standard interface between network and application
- Two types of socket:
  - Stream – provides a virtual circuit service
  - Datagram – delivers individual packets
- Independent of network type
  - Commonly used with IP sockets but not specific to the Internet protocols
    - Stream sockets map onto TCP/IP connections
    - Datagram sockets map onto UDP/IP

Based on slides © 2017 Colin Perkins

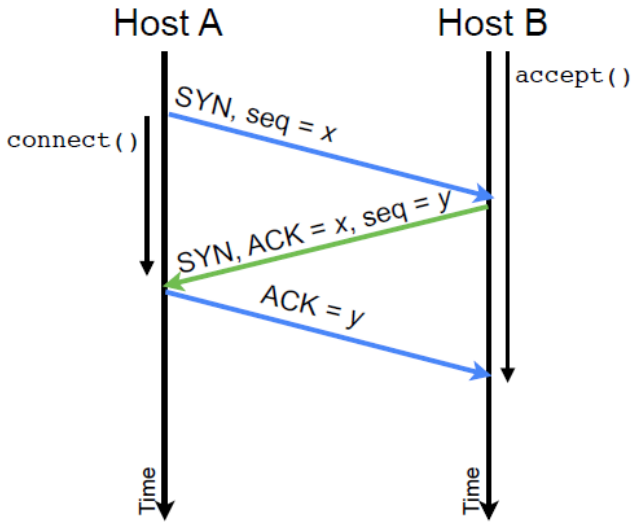
TCP

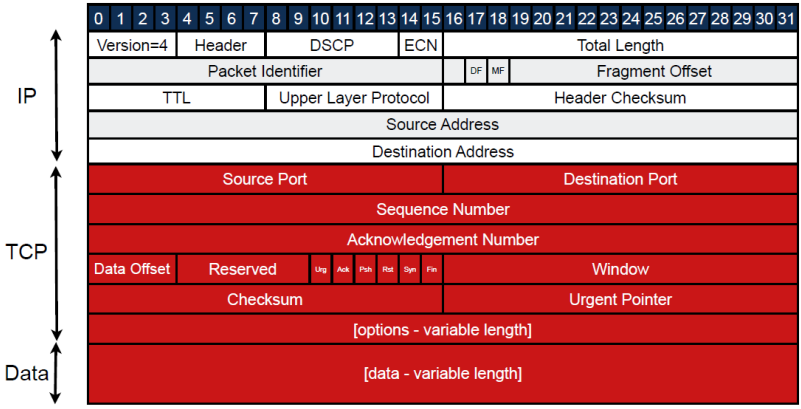
Internet Transport Protocols: TCP

- Reliable byte stream protocol over IP
  - Adds reliability
  - Adds congestion control
- But how can we add reliability?
  - When a connection is initiated/terminated?
  - During communication?
- Gist of it:
  - Sequence numbers
  - Acknowledgements
  - Multi-round processes

TCP Connection Setup: The 3-way Handshake

- TCP connections use 3-way handshake
  - The SYN and ACK flags in the TCP header signal connection progress
  - Initial packet has SYN bit set, includes randomly chosen initial sequence number
  - Reply also has SYN bit set and randomly chosen sequence number, ACKnowledges initial packet
  - Handshake completed by ACKnowledgement of second packet
- Combination ensures robustness
  - Randomly chosen initial sequence numbers give robustness to delayed packets or restarted hosts
  - Acknowledgements ensure reliability
- Similar handshake ends connection, with FIN bits signalling the teardown

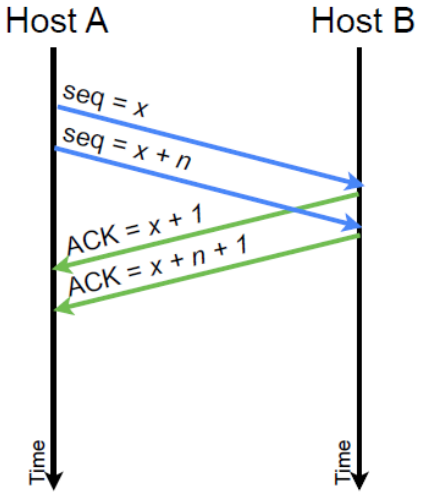


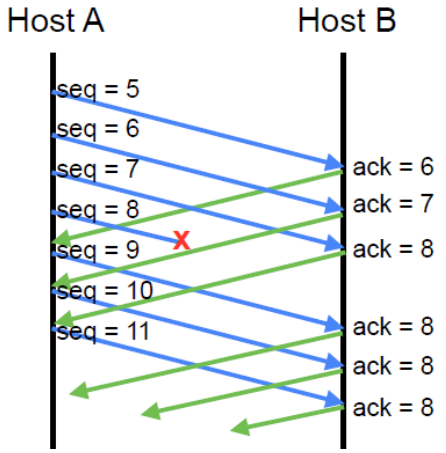


TCP Reliability

- TCP connections are reliable
  - Each TCP packet has a sequence number and an acknowledgement number
    - Sequence number counts how many bytes are sent (this example is unrealistic, since it shows one byte being sent per packet)
  - Acknowledgement number specifies the next byte expected to be received
    - Cumulative positive acknowledgement
    - Only acknowledge contiguous data packets (sliding window protocol, so several data packets in flight)
    - If a packet is lost, receipt of subsequent packets will trigger duplicate acknowledgements
  - TCP layer retransmits lost packets – this is invisible to the application

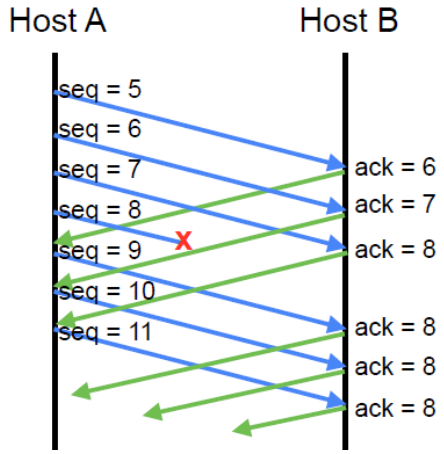
- How is Loss Detected?
  - Triple duplicate ACK → some packets lost, but later packets arriving
    - Triple duplicate = Four identical ACKs in a row
  - Timeout → send data but acknowledgements stop returning
    - Either the receiver or the network path has failed



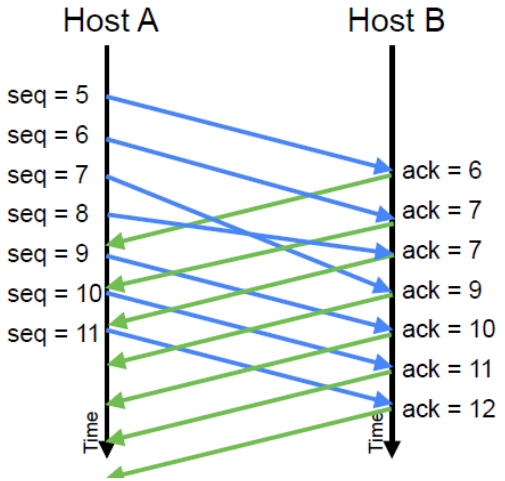


TCP Reliability

- How about packet reordering?
  - Duplicate ACKs can also be caused by packet delay leading to reordering
  - Gives appearance of loss, when the data was merely delayed
  - TCP uses triple duplicate ACK as indication of packet loss to prevent reordered packets causing retransmissions
    - Assumption: packets will only be delayed a little; if delayed enough that a triple duplicate ACK is generated, TCP will treat the packet as lost and send a retransmission



- End result: data delivered in order, even after loss occurs
  - TCP will retransmit the missing data, transparently to the application
  - A recv() for missing data will block until it arrives; TCP always delivers data in an in-order contiguous sequence



Congestion Control

- Definition: Adapting speed of transmission to match available end-to-end network capacity
  - Prevents congestion collapse of a network
- Can implement congestion control at either the network or the transport layer
  - Network layer
    - \+ Safe
    - \+ Ensures all transport protocols are congestion controlled
    - – Requires all applications to use the same congestion control scheme
  - Transport layer
    - \+ Flexible
    - \+ Transports protocols can optimise congestion control for applications
    - – A misbehaving transport can congest the network
- Which would you prefer?
- Key principles:
  - Conservation of packets
  - Additive increase and multiplicative decrease (AIMD) of the sending rate
  - Together, ensure stability of the network

Congestion Control

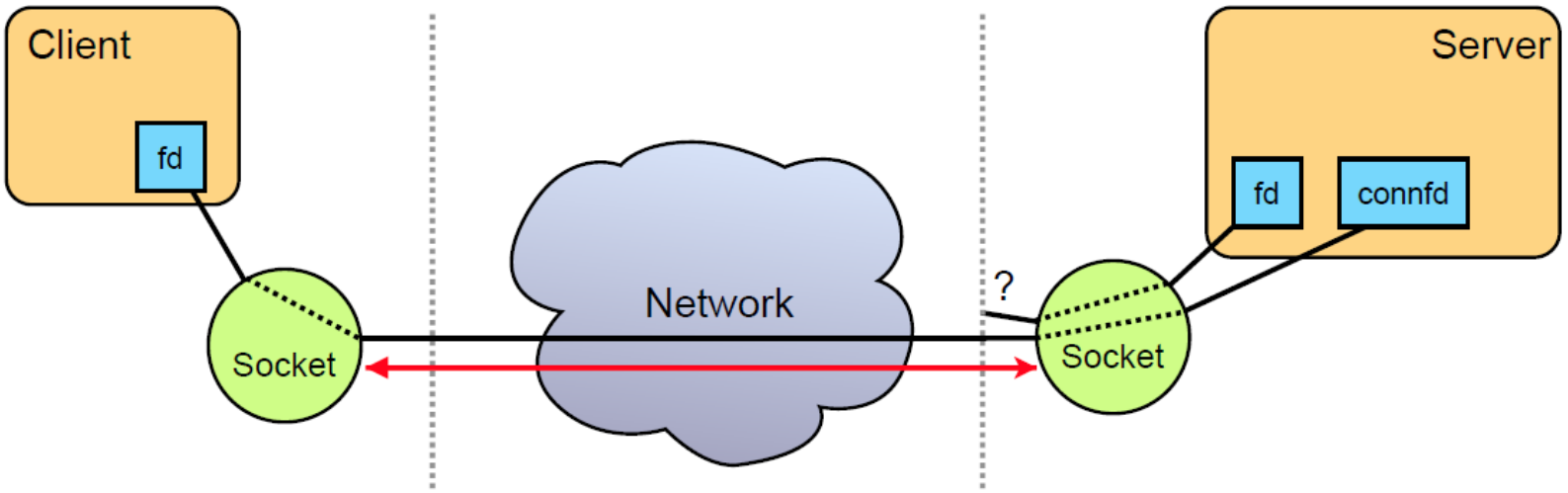
- Conservation of Packets:
  - The network has a certain capacity
    - The *bandwidth* x *delay* product of the path (volume/time x time -> volume)
  - When in equilibrium at that capacity, send one packet for each acknowledgement received
    - Total number of packets in transit is constant
    - Automatically reduces sending rate as network gets congested and delivers packets more slowly

- Adjust sending rate according to an Additive Increase Multiplicative Decrease (AIMD) algorithm:
  - Start slowly, increase gradually to find equilibrium
    - Add a small amount to the sending speed each time interval without loss
  - Respond to congestion rapidly
    - Multiply sending window by some factor $\beta < 1$ each interval where loss is seen
  - Faster reduction than increase → stability

Extra: Congestion Control

- TCP uses a window-based congestion control algorithm
  - Maintains a sliding window onto the available data that determines how much can be sent according to the AIMD algorithm
- TCP congestion control highly effective at keeping bottleneck link fully utilised
  - Provided sufficient buffering in the network: buffer size = bandwidth × delay
  - Packets queued in buffer → delay
  - TCP trades some extra delay to ensure high throughput
- TCP assumes loss is due to congestion (unless explicit congestion notification is enabled)
  - Too much traffic queued at an intermediate link → some packets dropped
  - This is not always true:
    - Wireless networks
    - High-speed long-distance optical networks
  - Much research into improved versions of TCP for wireless links

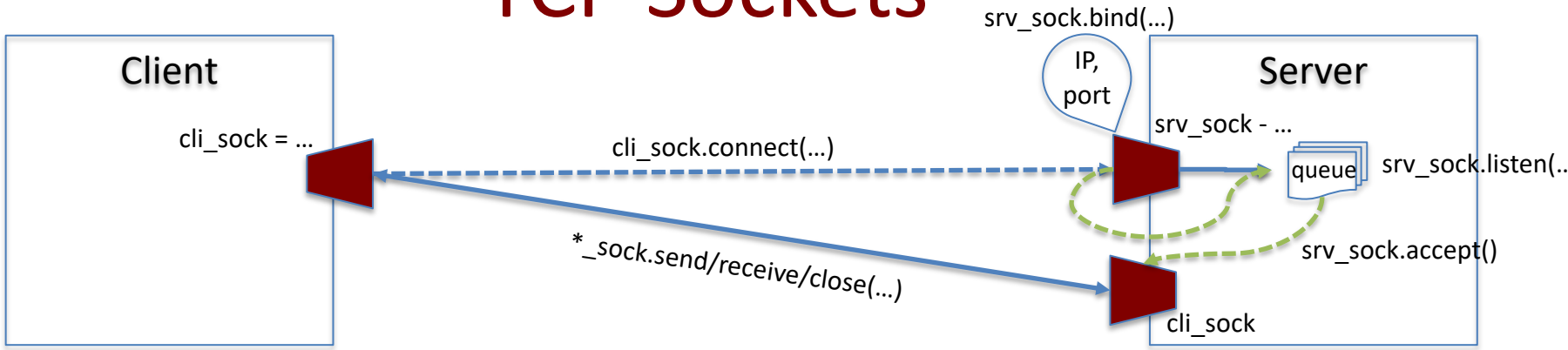
TCP Sockets




```
cli_sock = socket.socket(socket.AF_INET,
     socket.SOCK_STREAM)



cli_sock.connect(('somehost', port num))
request = …
cli_sock.send(request)
response = cli_sock.recv(num bytes)
cli_sock.close()
```

```
srv_sock = socket.socket(socket.AF_INET,
     socket.SOCK_STREAM)
srv_sock.bind(('somehost', port num))
srv_sock.listen(size of queue)

cli_sock, cli_addr = srv_sock.accept()
request = cli_sock.recv(num bytes)
response = …
cli_sock.send(response)
cli_sock.close()
```

TCP Sockets




```
cli_sock = socket.socket(socket.AF_INET,
     socket.SOCK_STREAM)



cli_sock.connect((‘somehost’, port num))
request = …
cli_sock.send(request)
response = cli_sock.recv(num bytes)
cli_sock.close()
```

```
srv_sock = socket.socket(socket.AF_INET,
     socket.SOCK_STREAM)
srv_sock.bind((‘somehost’, port num))
srv_sock.listen(size of queue)

cli_sock, cli_addr = srv_sock.accept()
request = cli_sock.recv(num bytes)
response = …
cli_sock.send(response)
cli_sock.close()
```

TCP Sockets

- Sockets initially unbound, and can either accept or make a connection
- Most commonly used in a client-server fashion:
  - One host makes the socket bind(), listen() for, and accept(), connections on a well-known port, making it into a server
    - The port is a 16-bit number used to distinguish servers
    - E.g. web server listens on port 80, email server on port 25
  - The other host makes the socket connect() to that port on the server
  - Once connection is established, either side can send() data into the connection, where it becomes available for the other side to recv()

TCP Sockets: Port Number

Port Range		Name	Intended use
0	1023	Well-known (system) ports	Trusted operating system services
1024	49151	Registered (user) ports	User applications and services
49152	65535	Dynamic (ephemeral) ports	Private use, peer-to-peer applications, source ports for TCP client connections

- Servers must listen on a known port
  - IANA maintains a registry (http://www.iana.org/assignments/port-numbers)
- Distinction between system and user ports ill-advised
  - Security problems resulted
- Insufficient port space available
  - >75% of ports are registered
- TCP clients traditionally connect from a randomly chosen port in the ephemeral range

Communication over TCP

- 3-way handshake happens during the connect()/accept() calls
- Call send() to transmit data
  - Will block until the data can be written, and returns amount of data sent
  - Might not be able to send all the data, if the connection is congested
  - Returns actual number of bytes sent
  - Also see sendall()
- Call recv(X) to read up to X bytes of data from a connection
  - Will block until some data is available or the connection is closed
  - Returns a bytes object with the data that was received from the socket
  - Received data is not null terminated – potential security risk?
- All errors handled through *exceptions*
- See the Python docs for further info
  - https://docs.python.org/2/library/socket.html
  - https://docs.python.org/3/library/socket.html

Communication over TCP

- The send() call enqueues data for transmission
- This data is split into segments
- Each segment is placed in a TCP packet
- That packet is sent when allowed by the congestion control algorithm
- If the data in a send() call is too large to fit into one segment, the TCP implementation will split it into several segments
  - Similarly, several send() requests might be aggregated into a single TCP segment
  - Both are done transparently by the TCP implementation and are invisible to the application
- Implication: the data returned by recv() doesn't necessarily correspond to a single send() call
- The recv() call can return data in unpredictably sized chunks – applications must be written to cope with this

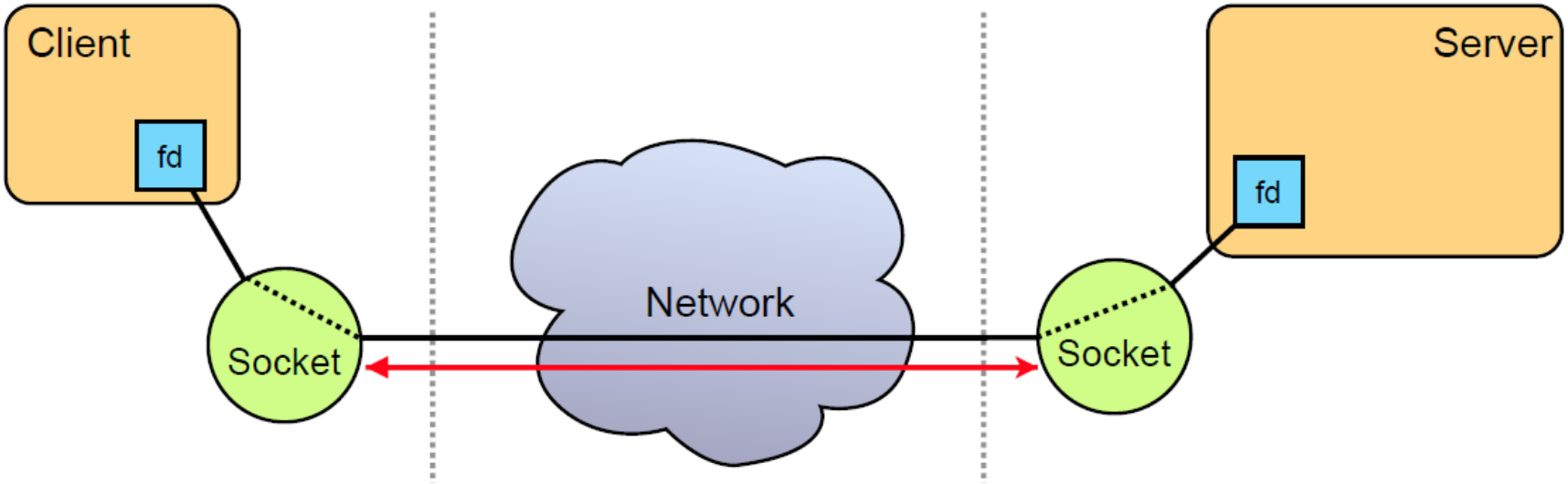
Based on slides © 2017 Colin Perkins

UDP

UDP

- UDP provides an unreliable datagram service, identifying applications via a 16 bit port number
  - UDP ports are separate from TCP ports
  - Often used peer-to-peer (e.g., for VoIP), so both peers must bind() to a known port
  - Create via socket() as usual, but specify SOCK_DGRAM as the socket type
  - No need to connect() or accept(), since no connections in UDP

UDP Sockets




```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((..., ...))
sock.sendto(..., (..., ...))
sock.recvfrom(...)
sock.close()
```

Communication over UDP

- The sendto() call sends **a single datagram**
  - Each call to sendto() can send to a different address, even though they use the same socket
  - Alternatively, connect() to an address, then use send() to send the data
    - connect() on a UDP socket merely sets a default destination address for future send() calls
- The recvfrom() call may be used to read a single datagram (also returns the sender's address)
  - recv() also usable but doesn't provide the sender's address
- Unlike TCP, each UDP datagram is sent as **exactly one** IP packet (which may however be fragmented in IPv4)
  - Each recvfrom() corresponds to a single sendto()

Communication over UDP

- But, transmission is unreliable: packets may be lost, delayed, reordered, or duplicated in transit
  - The application is responsible for correcting the order, detecting duplicates, and repairing loss, if necessary
  - Generally requires the sender to include some form of sequence number in each packet sent
- Application must organise the data so it's useful if some packets lost
  - E.g. streaming video with Intermediate and Predicted frames

Communication over UDP

- Need to provide sequencing, reliability, and timing in applications
  - Sequence numbers and acknowledgements
  - Retransmission and/or forward error correction
  - Timing recovery
- Need to implement congestion control in applications
  - To avoid congestion collapse of the network
  - Should be approximately fair to TCP
  - Difficult to do well – TCP congestion control covers many corner cases that are easy to miss
    - RFC 3448 provides a detailed specification for a well-tested algorithm
    - IETF RMCAT working group developing standard congestion control algorithms for interactive video applications running over UDP https://datatracker.ietf.org/wg/rmcat/charter/
    - Google's QUIC protocol builds on UDP to give more sophisticated service

Extra: Communication over UDP

- IETF provides guidelines for writing UDP-based applications
  - RFC 8085 – https://tools.ietf.org/html/rfc8085
  - **Read this** before trying to write UDP-based code

- For ideas on implementing congestion control in UDP, see:
  - RFC 3448: Provides a detailed specification for a well-tested algorithm
  - https://datatracker.ietf.org/wg/rmcat/charter/: IETF RMCAT working group developing standard congestion control algorithms for interactive video applications running over UDP
  - Google's QUIC protocol: Builds on UDP to give more sophisticated service

Internet Transport Protocols

- UDP Applications
  - Useful for applications that prefer timeliness to reliability
    - Voice-over-IP
    - Streaming video
  - Must be able to tolerate some loss of data
  - Must be able to adapt to congestion in the application layer

- TCP Applications
  - Useful for applications that require reliable data delivery, and can tolerate some timing variation
    - File transfer and web downloads
    - Email
    - Instant messaging
  - Default choice for most applications

Reading material

- **Peterson & Davie "Computer Networks: A systems approach"**: Chapter 5, sections 5.1, 5.2
- **Kurose & Ross "Computer Networking: A top-down approach"**: Chapter 3, sections 3.1, 3.3, 3.5, 3.7
- **Tanenbaum & Wetherall "Computer Networks" 5th edition**: Chapter 6, sections 6.1.1, 6.1.2, 6.2, 6.4, 6.5
- Bonaventure "Computer Networking" 1st edition: https://www.computer-networking.info/1st/html/transport/principles.html