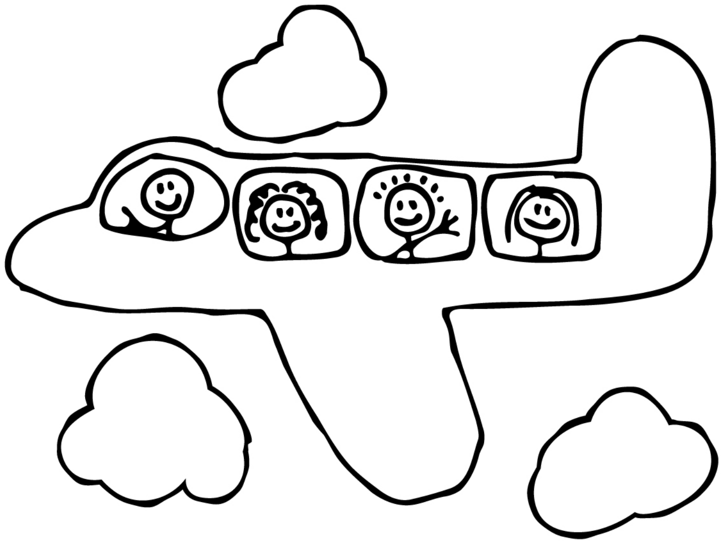
Advanced Networking

Daniele Lacamera

Section 0

Training overview



Welcome

- Requirements to follow this course
  - Some C programming skills
  - Some user-level experience with GNU/Linux
  - A GNU/Linux distribution

Welcome

- Goals of the training
  - Refresh your knowledge on networking
  - Learn how to write distributed applications
  - Become familiar with many software tools
  - Expand your knowledge of protocols implementation
  - Learn how to cope with more and less common problems related to designing distributed solutions
  - Improve your design skills

Welcome

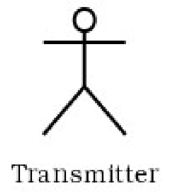
- Content of this course
  - Some theory of computer networks
  - Socket interfaces (IPv4 and IPv6)
  - The “Internet Protocol” (IP) complete analysis
  - Packet management
  - The test tools
  - Protocol reverse and forward engineering
  - Security
  - Performance optimization

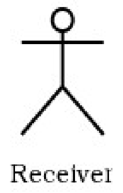
Welcome

- Training path
  - Lectures
  - Man pages
  - Source code of various projects
  - Learn by examples
  - Software tools
  - Design/code a distributed solution

Section 1
OSI layers

OSI Layers

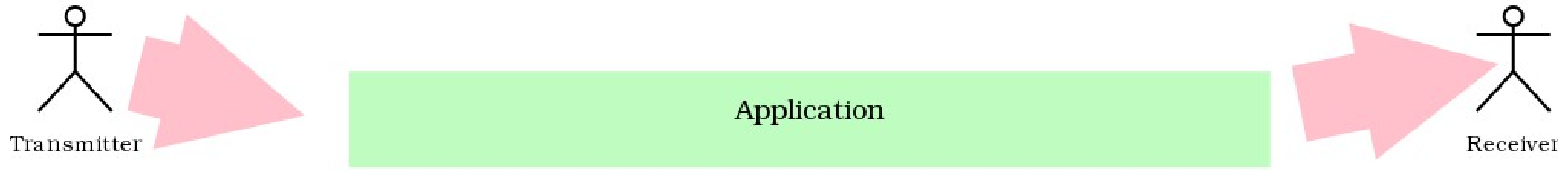




OSI Layers

Transmitter
Receiver

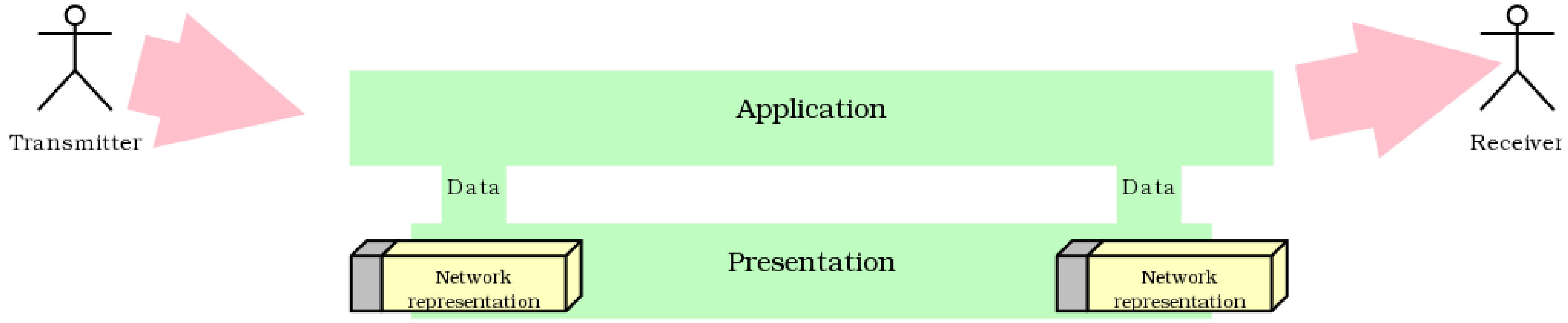
OSI Layers



OSI Layers

TSS tass TECHNOLOGY SOLUTIONS

Dedicated to make it work



OSI Layers

tass TSS TECHNOLOGY SOLUTIONS
Dedicated to make it work

- The Layer “6”: Presentation
  - Protocols for data conversion from/to human (application) formats, e.g.:
  - From application data into specific structures, types and formats for transmission over the network
  - From objects received (datagrams, streams), back to application data

OSI Layers

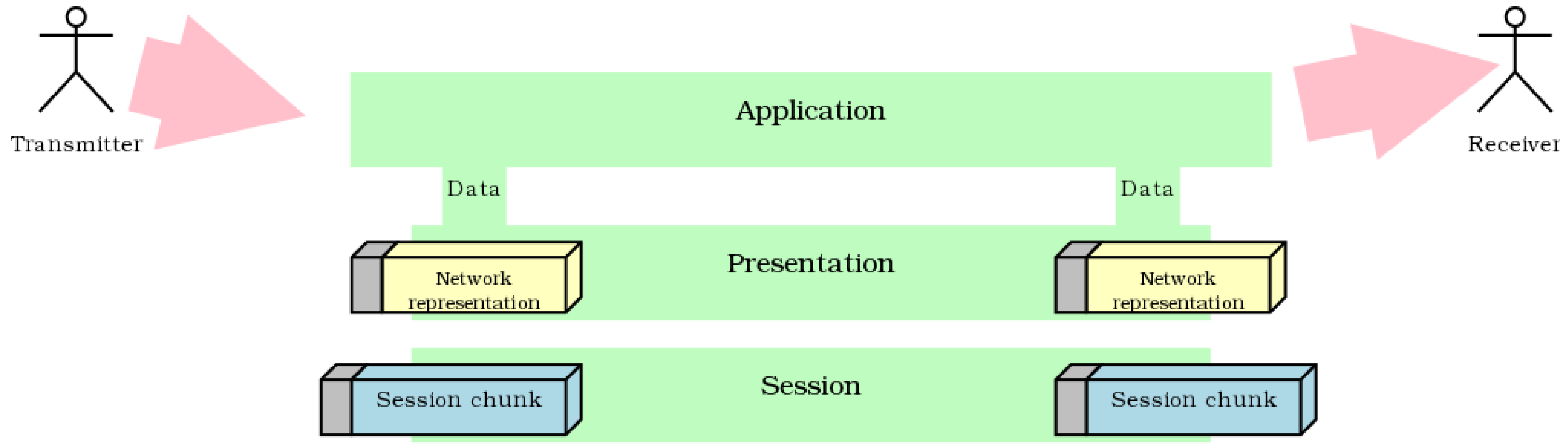
- Example: distributed random number generator
  - Keep it platform independent

```
ssize_t send_random_number(int dst, unsigned long rnd)
{
   uint32_t x = htonl(rnd);
   return send(dst, &x, sizeof(x), 0);
}

ssize_t recv_random_number(int src, unsigned long *rnd)
{
   uint32_t x;
   if (sizeof(x) != recv(src, &x, sizeof(x), 0))
      return -1;
   *rnd = ntohl(x);
   return sizeof(x);
}
```

OSI Layers

Dedicated to make it work

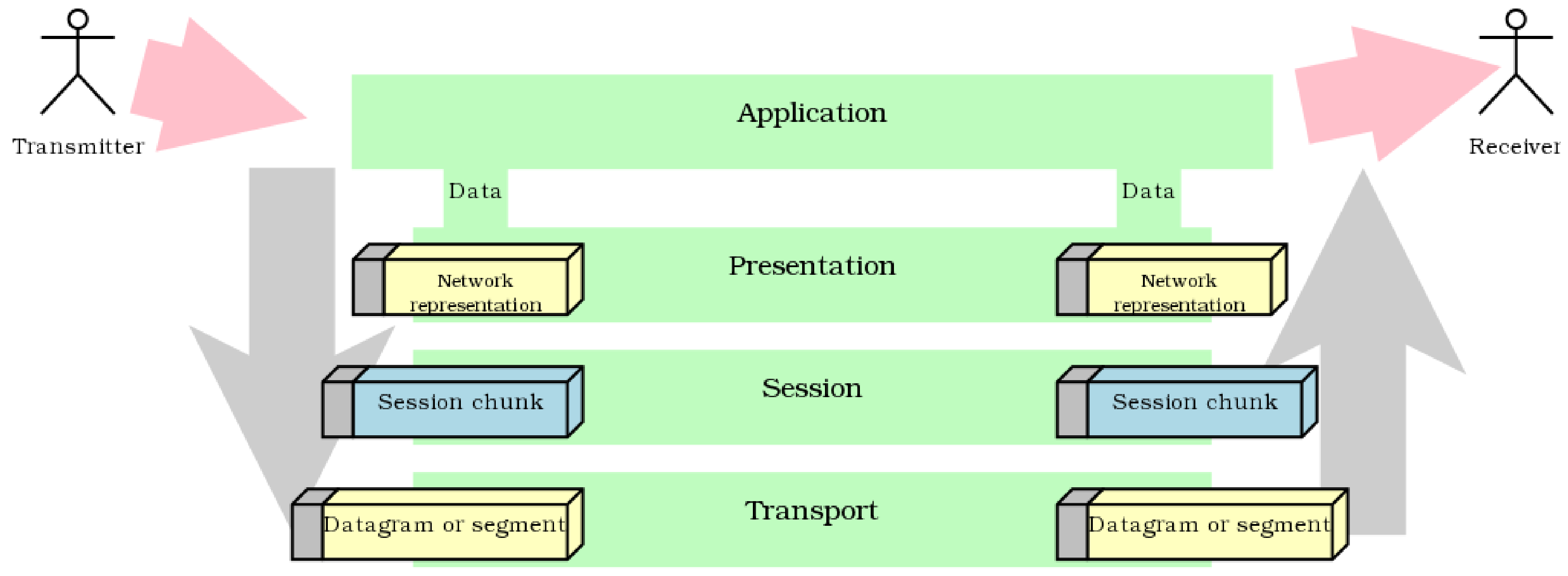


OSI Layers

- The Layer “5”: Session
  - Protocols for applications connections
  - Establish a connection between two remote applications
  - Can implement authentication/authorization
  - Deal with data synchronization and multiplexing
  - Many (stateful) protocols define their own sessions: HTTP, telnet, FTP, ssh, ...

OSI Layers

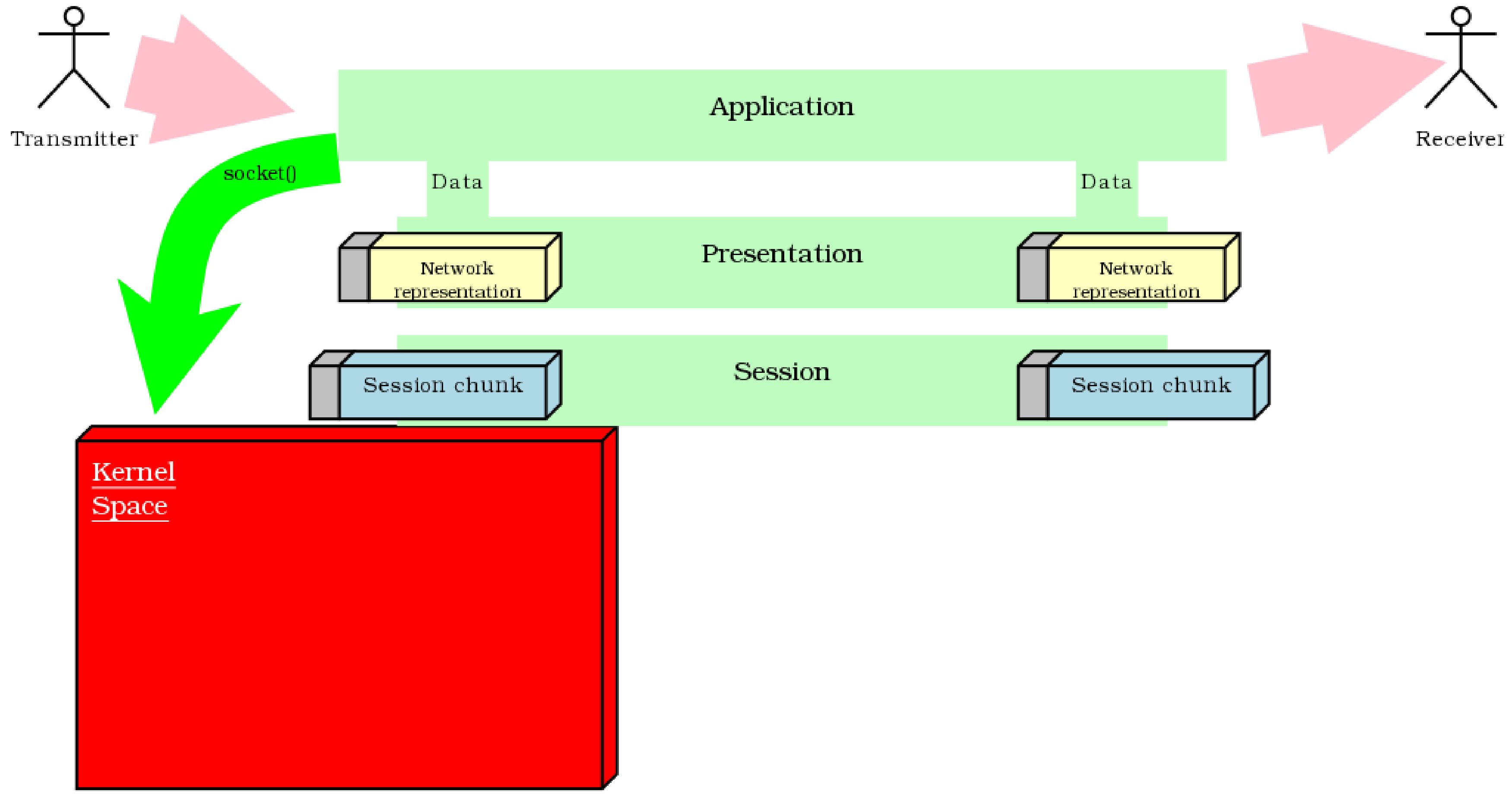
Dedicated to make it work



OSI Layers

- Protocols of layers 7,6,5 belong in userspace
  - Applications MAY implement strategies related to those three layers only
  - Layer separation MUST be kept
  - An API to the Operating System must exist to access lower layers
    - Lower layers have system-wide shared parameters and structures
    - Multiple applications need to access the lower layers simultaneously

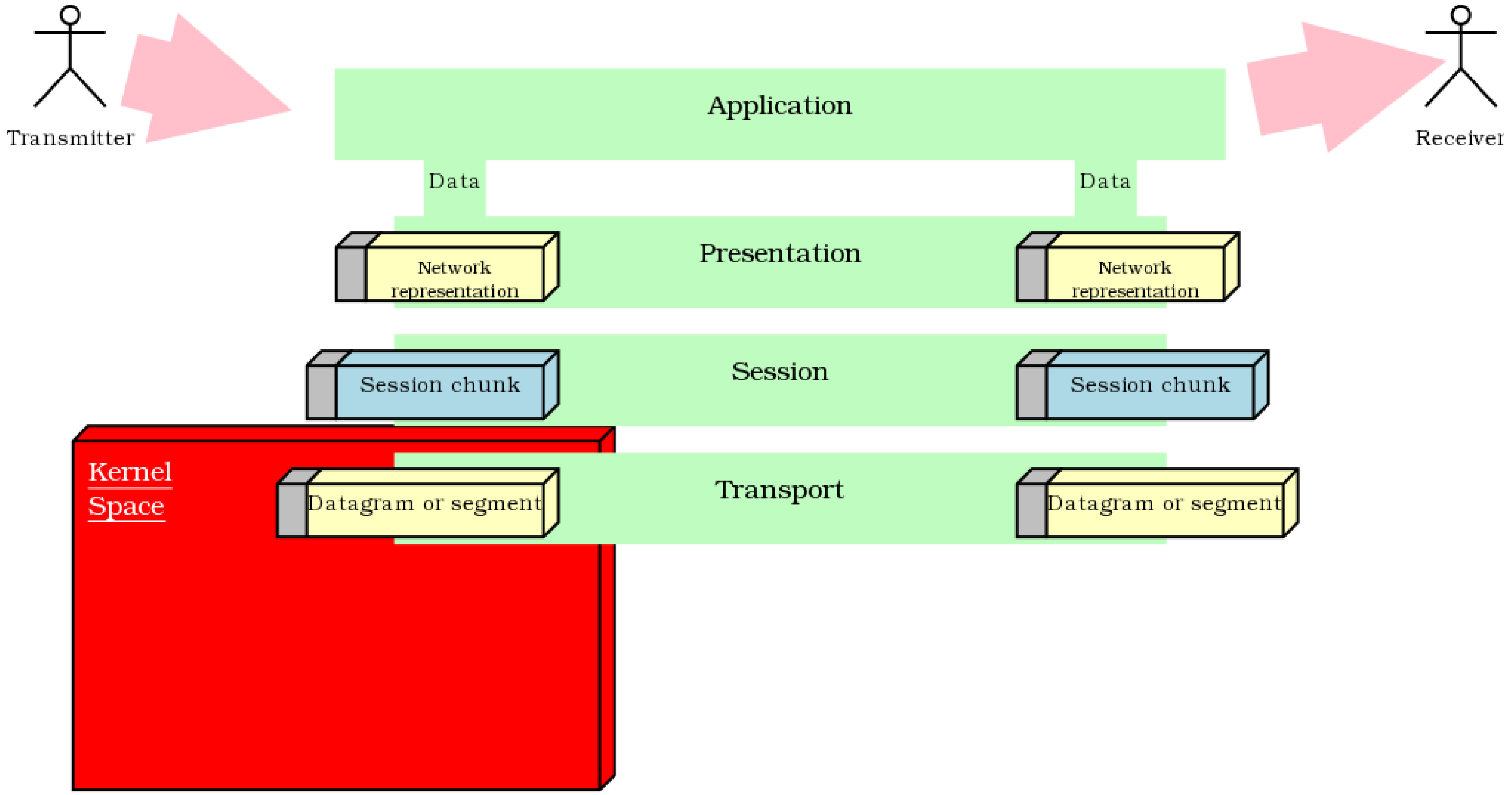
OSI Layers


Transmitter
Receiver
Application
socket()
Data
Data
Presentation
Network representation
Network representation
Session
Session chunk
Session chunk
Kernel Space

OSI Layers

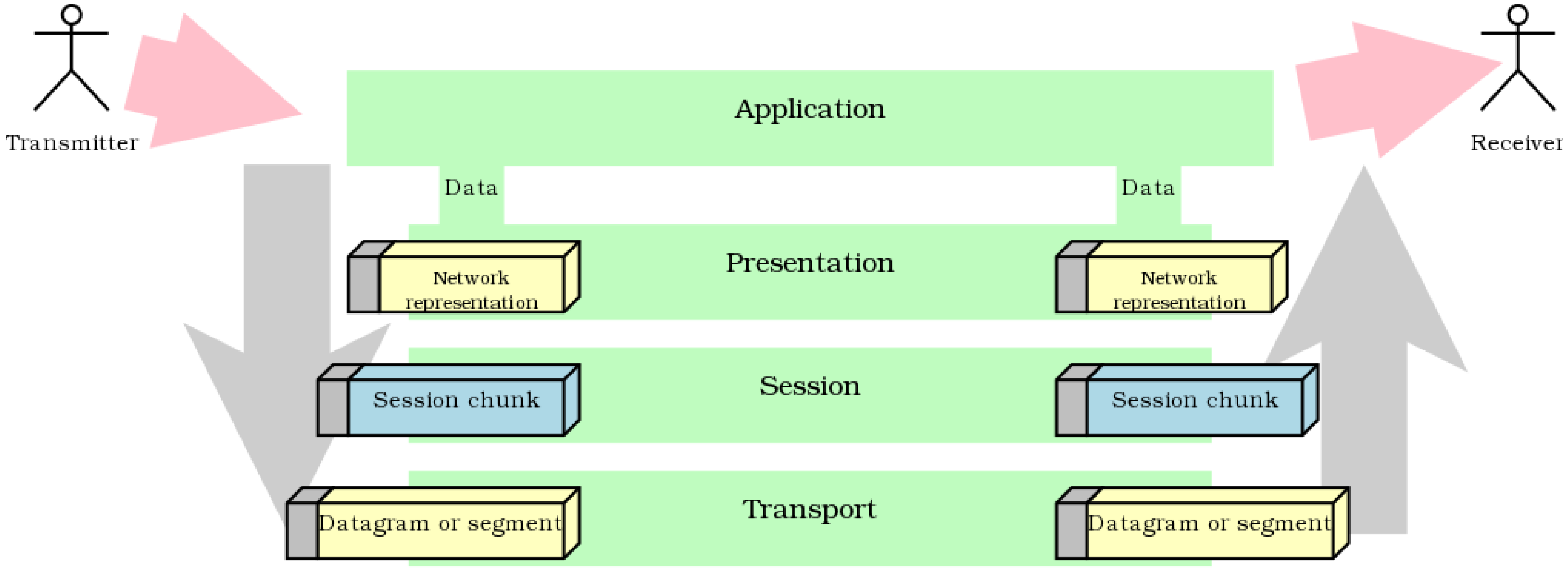
TASS TSS TECHNOLOGY SOLUTIONS

Dedicated to make it work



OSI Layers

Dedicated to make it work



OSI Layers

- The Layer “4”: Transport
  - Transport protocols provide generic abstractions for data transfer
  - Divide the resources of the system, [de-]multiplexing data from/to different applications (“ports” abstraction)
  - May implement built-in features such as reliability, data order, integrity
  - May implement flow control and congestion control
  - Introduce the concept of Datagram (connectionless) or Stream+segments (connection)
  - Prepare the split the information into packets

OSI Layers

- UDP
  - Multiplexing (ports)
  - Integrity of **datagram**
  - 1-to-1 and 1-to-many unreliable communication
- TCP
  - Multiplexing (ports)
  - Integrity of **flow**
  - Bandwidth estimation, flow control
  - 1-to-1 reliable communication

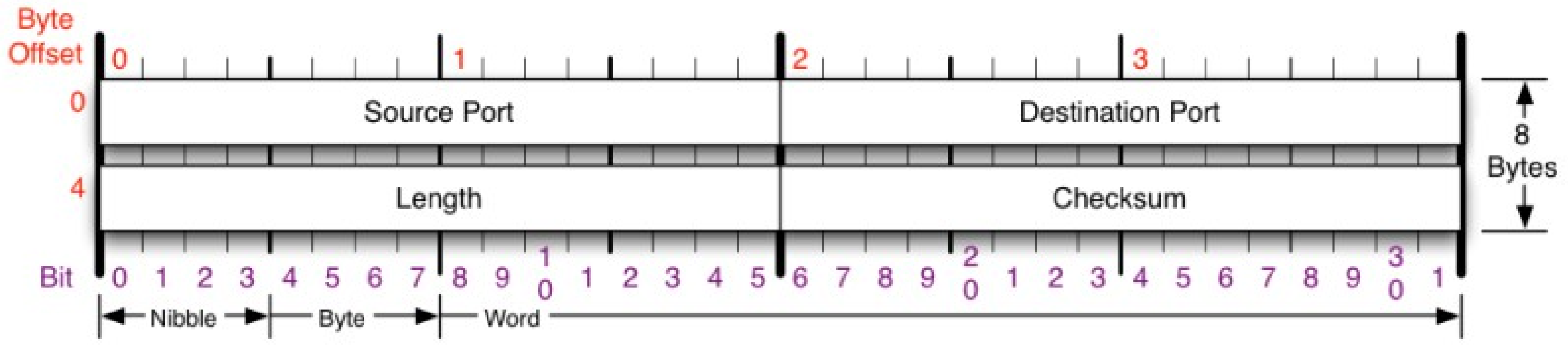
OSI Layers

- UDP vs TCP
- UDP: simple, lightweight, efficient
  - Allows one-to-many communication
  - No reliability
  - No guarantee of in-order delivery
- TCP: complex, reliable, network-friendly
  - Uses only the available bandwidth
  - Keep data integrity on each side of the “pipe”

UDP Header

tass TECHNOLOGY SOLUTIONS

Dedicated to make it work



Checksum

Checksum of entire UDP segment and pseudo header (parts of IP header)

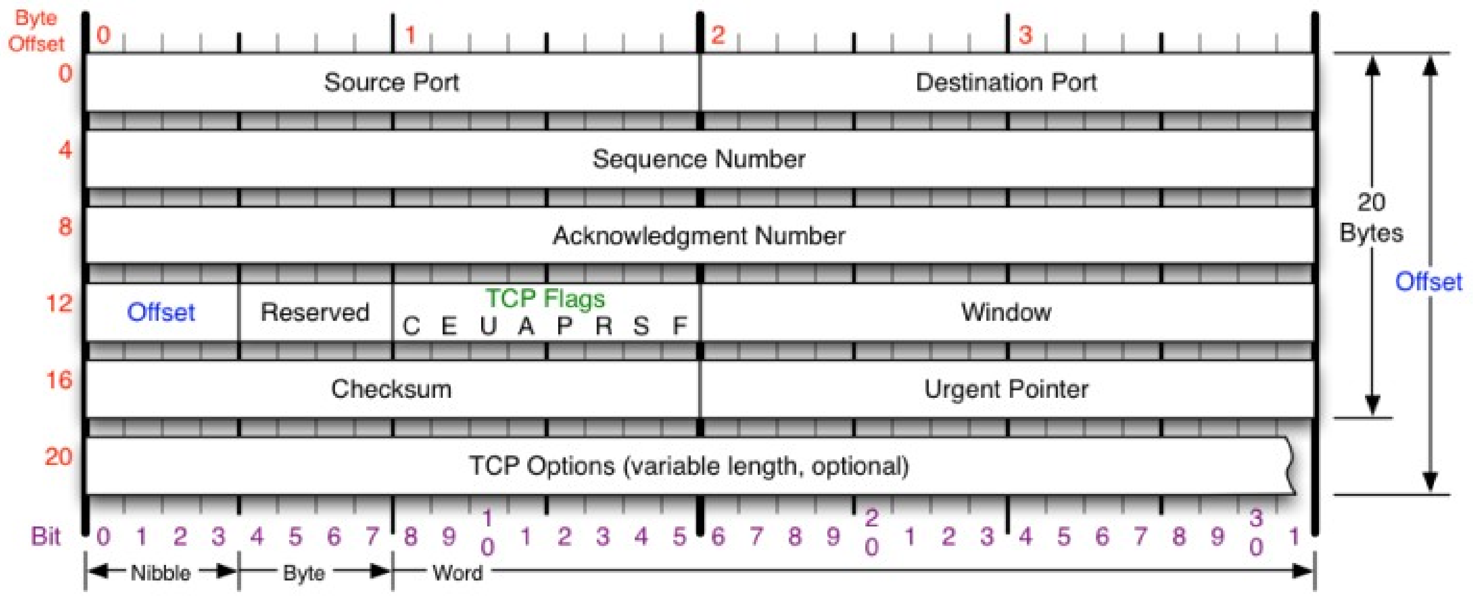
RFC 768

Please refer to RFC 768 for the complete User Datagram Protocol (UDP) Specification.

TCP Header

tass TECHNOLOGY SOLUTIONS TSS

Dedicated to make it work



TCP Flags

C E U A P R S F

	Congestion Window
C	0x80 Reduced (CWR)
E	0x40 ECN Echo (ECE)
U	0x20 Urgent
A	0x10 Ack
P	0x08 Push
R	0x04 Reset
S	0x02 Syn
F	0x01 Fin

Congestion Notification

ECN (Explicit Congestion Notification). See RFC 3168 for full details, valid states below.

Packet State	DSB	ECN bits
Syn	0 0	1 1
Syn-Ack	0 0	0 1
Ack	0 1	0 0
No Congestion	0 1	0 0
No Congestion	1 0	0 0
Congestion	1 1	0 0
Receiver Response	1 1	0 1
Sender Response	1 1	1 1

TCP Options

0 End of Options List
1 No Operation (NOP, Pad)
2 Maximum segment size
3 Window Scale
4 Selective ACK ok
8 Timestamp

Checksum

Checksum of entire TCP segment and pseudo header (parts of IP header)

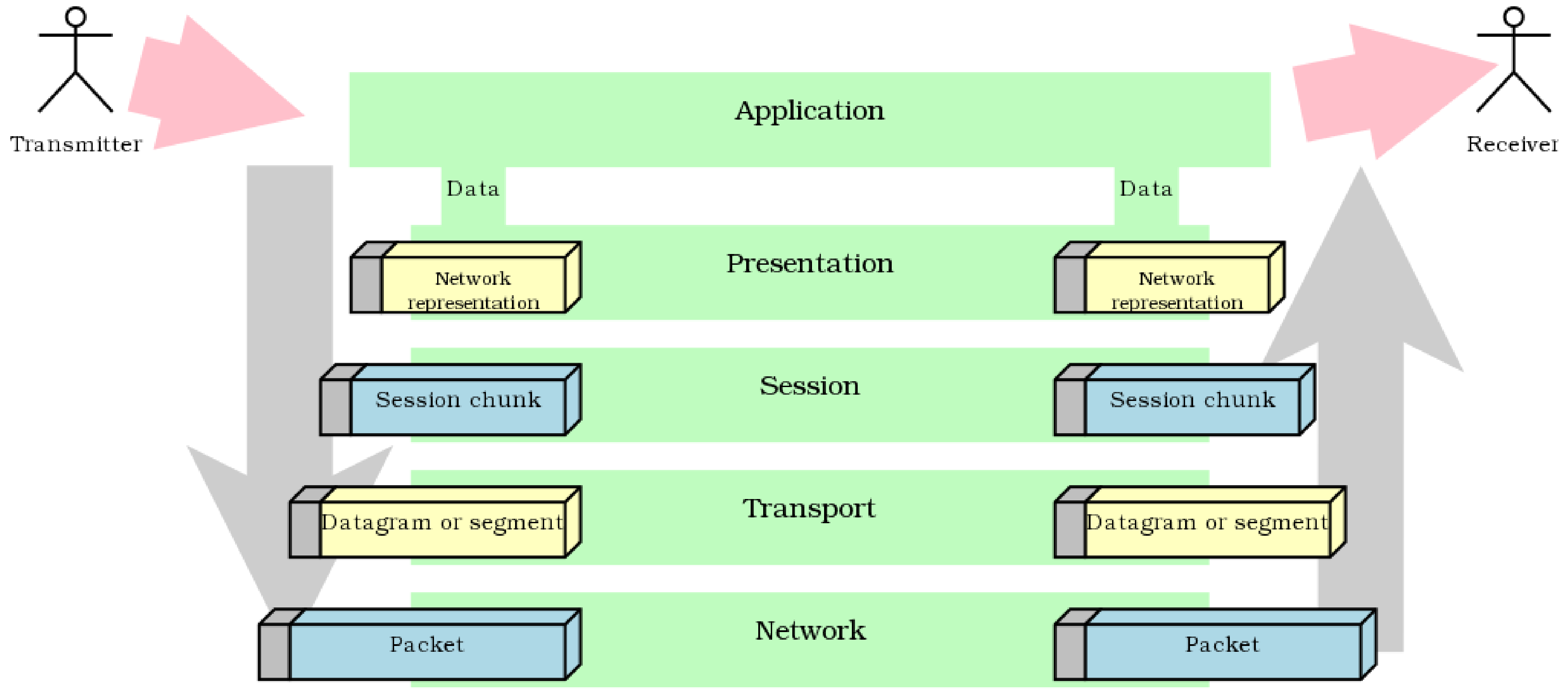
Offset

Number of 32-bit words in TCP header, minimum value of 5. Multiply by 4 to get byte count.

RFC 793

Please refer to RFC 793 for the complete Transmission Control Protocol (TCP) Specification.

OSI Layers


Transmitter
Receiver
Application
Data
Data
Presentation
Network representation
Network representation
Session
Session chunk
Session chunk
Transport
Datagram or segment
Datagram or segment
Network
Packet
Packet

OSI Layers

- The Layer “3”: Network
  - Provides packet management, delivery, forward
  - Responsible to creating paths for data transfer among endpoints/peers, acting on each intermediate node
  - Manages lifetime and integrity of the single packets
  - Routing, NAT, firewalling belong (mostly) to this layer
  - No reliability or sequencing provided: bad packets are discarded. Routes along the path might change unexpectedly

OSI Layers

- Internet Protocol (IP)
  - Designed to interconnect a global network of hosts
  - Subnetting
  - Static and dynamic routing paths
- Current versions
  - IPv4
    - Less than 4.3 billions of different addresses, not efficiently distributed
    - Requires additional protocols (ARP, NAT)
  - IPv6

OSI Layers

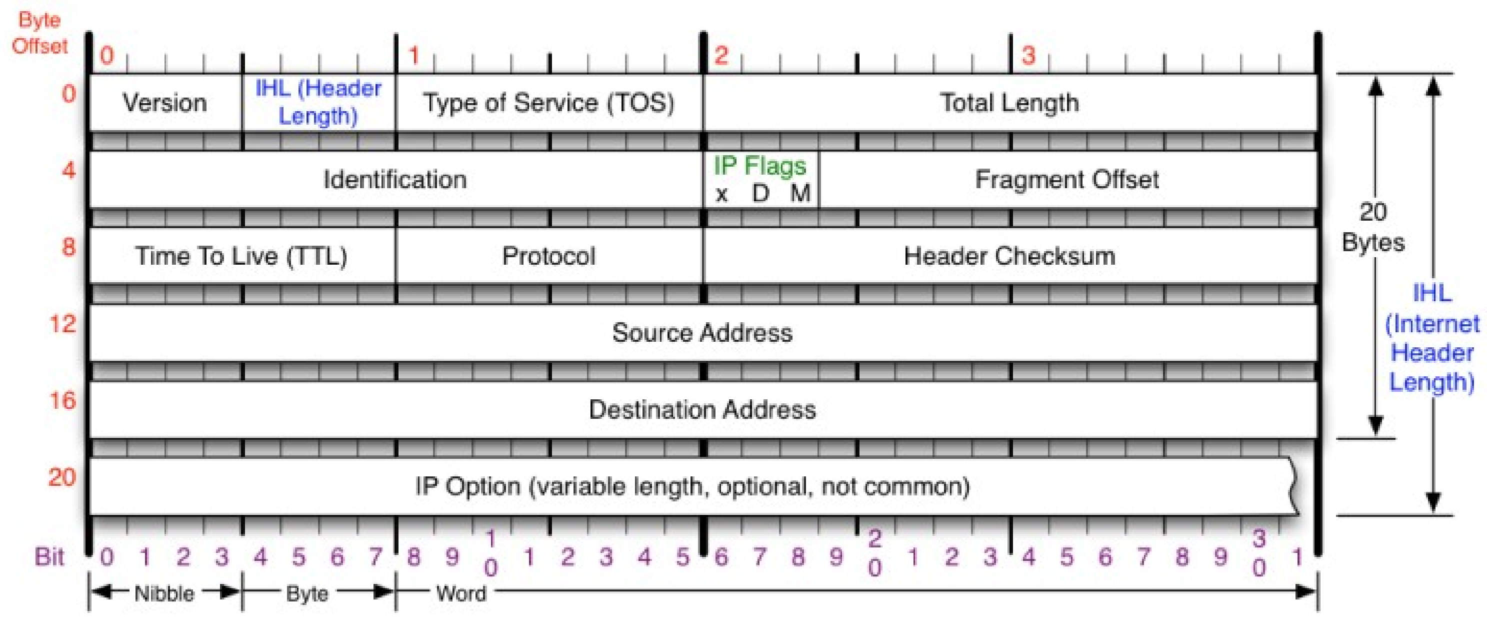
- IP: Implementation
  - Host side:
    - Ensure incoming packets are delivered to the upper layers
    - Route the outgoing traffic towards the correct network interface
  - Router side:
    - Compute next-hop using packet information and routing tables
    - Update routing information

OSI Layers

- IP: lower layers integration
  - PPP channels
    - do not require link-level addressing (next hop is fixed)
    - Examples: modems, SLIP, ...
  - HW requiring Link layer addressing
    - ARP is used to discover IPv4 ↔ HW addr associations
    - Datalink layer will use the information in the ARP table to populate the next hop's HW address field
  - One-to-many HW addresses
    - FF:FF:FF:FF:FF:FF – Broadcast address (send to all)
    - 01:00:5E:xx:xx:xx – IPv4 Multicast
    - 33:33:xx:xx:xx:xx – IPv6 Multicast

IPv4 Header

Dedicated to make it work



Version

Version of IP Protocol. 4 and 6 are valid. This diagram represents version 4 structure only.

Header Length

Number of 32-bit words in TCP header, minimum value of 5. Multiply by 4 to get byte count.

Protocol

IP Protocol ID. Including (but not limited to):

1 ICMP	17 UDP	57 SKIP
2 IGMP	47 GRE	88 EIGRP
6 TCP	50 ESP	89 OSPF
9 IGRP	51 AH	115 L2TP

Total Length

Total length of IP datagram, or IP fragment if fragmented. Measured in Bytes.

Fragment Offset

Fragment offset from start of IP datagram. Measured in 8 byte (2 words, 64 bits) increments. If IP datagram is fragmented, fragment size (Total Length) must be a multiple of 8 bytes.

Header Checksum

Checksum of entire IP header

IP Flags

x D M

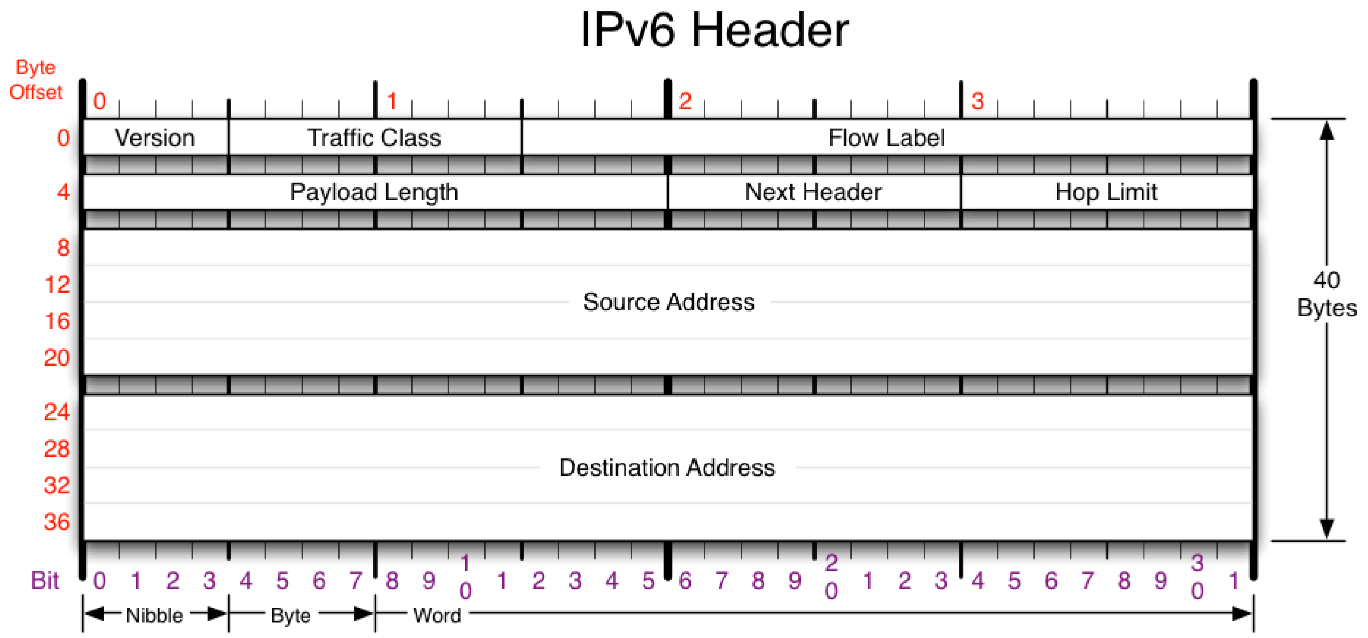
x 0x80 reserved (evil bit)
D 0x40 Do Not Fragment
M 0x20 More Fragments follow

RFC 791

Please refer to RFC 791 for the complete Internet Protocol (IP) Specification.

IPv6 Header

tass TECHNOLOGY SOLUTIONS
Dedicated to make it work



Version

Version of IP Protocol. 4 and 6 are valid. This diagram represents version 6 structure only.

Traffic Class

8 bit traffic class field.

Flow Label

20 bit flow label.

Payload Length

16-bit unsigned integer. Length of the IPv6 payload, i.e., the rest of the packet following this IPv6 header, in octets. Any extension headers are considered part of the payload.

Source Address

128-bit address of the originator of the packet.

Next Header

8-bit selector. Identifies the type of header immediately following the IPv6 header. Uses the same values as the IPv4 Protocol field.

Destination Address

128-bit address of the intended recipient of the packet (possibly not the ultimate recipient, if a Routing header is present).

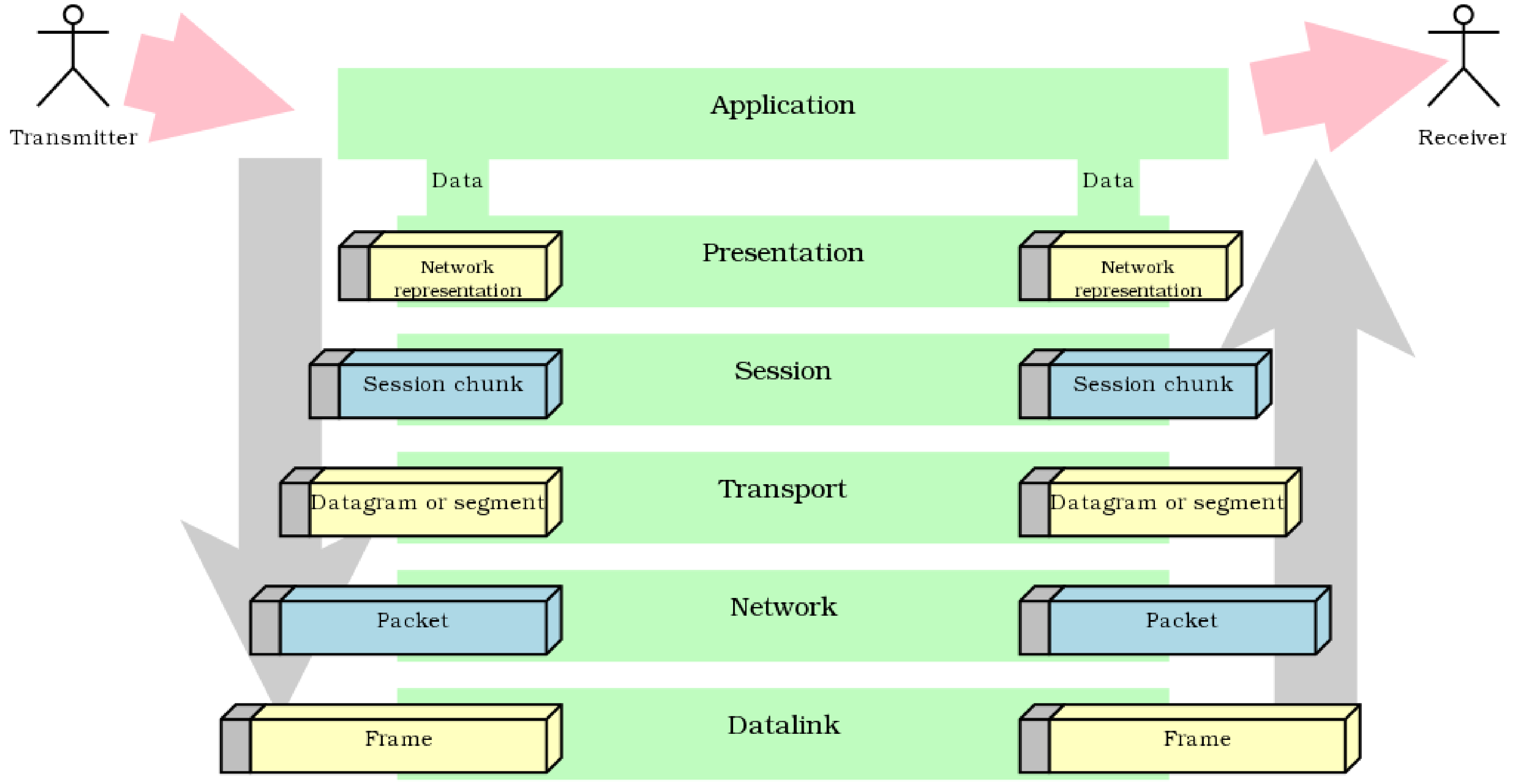
Hop Limit

8-bit unsigned integer. Decremented by 1 by each node that forwards the packet. The packet is discarded if Hop Limit is decremented to zero.

RFC 2460

Please refer to RFC 2460 for the complete Internet Protocol version 6 (IPv6) Specification.

OSI Layers


Transmitter
Receiver
Application
Data
Data
Presentation
Network representation
Network representation
Session
Session chunk
Session chunk
Transport
Datagram or segment
Datagram or segment
Network
Packet
Packet
Datalink
Frame
Frame

OSI Layers

- The Layer “2”: Data Link

  Link layer protocols provide network device management, packet encapsulation and delivery to the next host through a well-known physical path

  - Control/tune/supervise the physical link to the next node(s) in the network
  - Provide addressing among interfaces sharing the same physical link
  - May implement low-level error correction, sub-fragmentation and other countermeasures to deal with specific physical problems
  - Convert packets into bit-streams and vice-versa

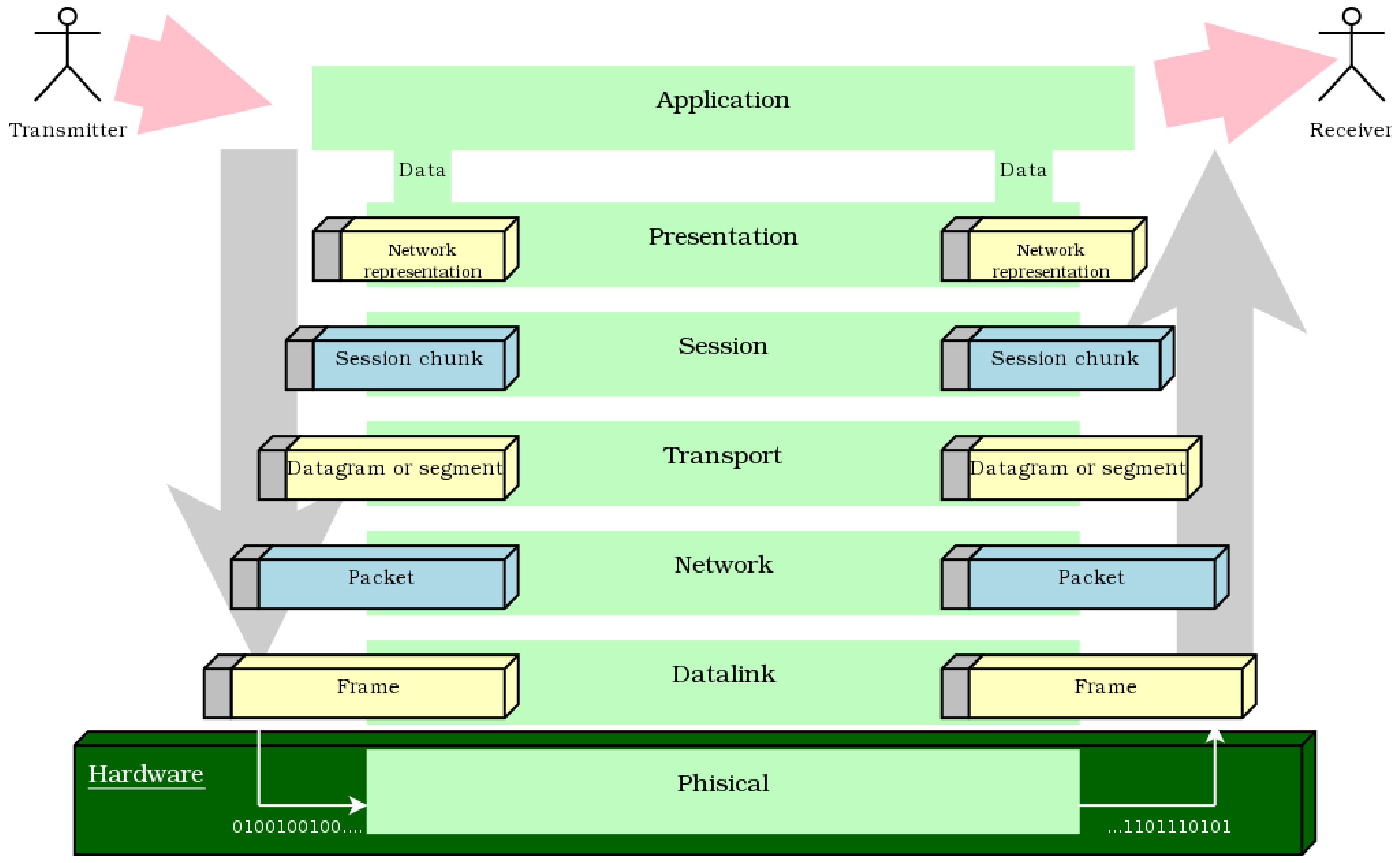
OSI Layers

- Layer 2 standalone equipment
  - Switches
    - Group multiple hosts in physical subnetworks
    - Pre-calculate the circuit-switching using src/dst HW address
  - Hubs
    - Simple HW amplifiers
    - All traffic is repeated to every port

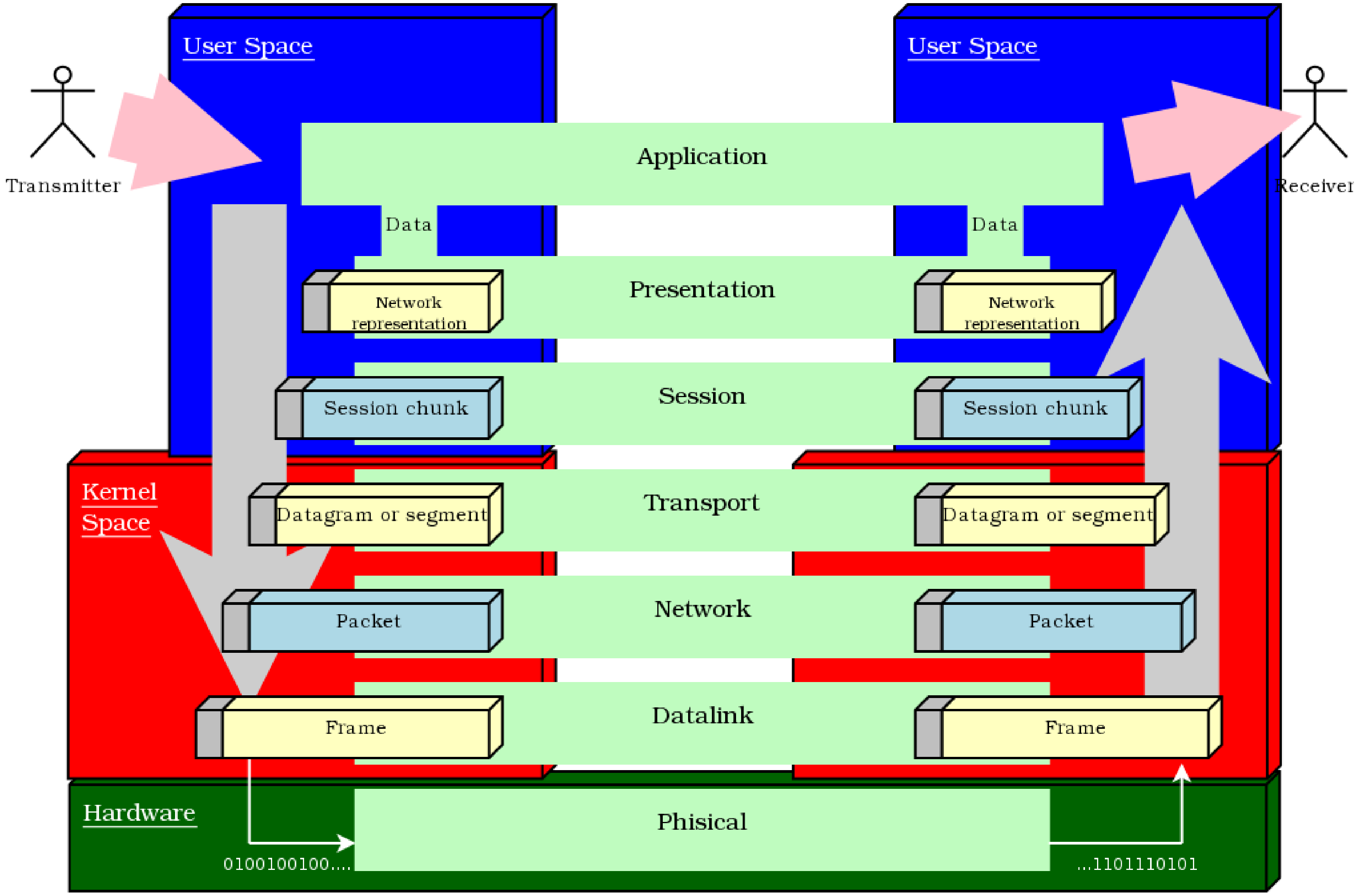
Wi-fi Access Points?

OSI Layers

TASS TECHNOLOGY SOLUTIONS
Dedicated to make it work



OSI Layers


User Space
User Space
Transmitter
Receiver
Application
Data
Data
Presentation
Network representation
Network representation
Session
Session chunk
Session chunk
Kernel Space
Transport
Datagram or segment
Datagram or segment
Network
Packet
Packet
Datalink
Frame
Frame
Hardware
Phisical
0100100100....
...1101110101

OSI Layers

Layers Implementation

- Userspace layers – libraries and applications are written using the BSD socket abstraction and other libraries/tools provided by the underlying OS
- Kernelspace layers – lower layers can be accessed via specific interfaces; TCP/IP code (e.g. kernel code) might be available for modifications
- Interchangeable or reusable protocols
- Cross-layers solutions

OSI Layers

- Accessing lower layers in Linux:
  - BSD sockets interface
  - Specific interfaces
    - RAW sockets (through AF_PACKET)
    - Netlink sockets
    - Direct interaction with the kernel (setsockopt, procfs, sysfs, ioctl, ...)
  - Kernel source code available!
    - Widely known and heavily maintained code
    - Widest range of features

Section 2

Sockets

Sockets

TASS TSS TECHNOLOGY SOLUTIONS Dedicated to make it work

- Sockets are the most complete and powerful way to IPC
- They uses the file abstraction, but require some infrastructure to be set up
  - The abstraction used by UNIX is the so-called “Berkeley socket”
- The same interface can be used with several subsystem (UNIX, INET, CAN, ...)
- The abstraction is completely defined in the kernel and it depends on the family implementation

Sockets

- One interface to fit them all
  - A single interface can be used to implement all kind of Berkeley sockets
  - The only exception is addressing, which is specific for the selected address family
- Addressing is provided by the “struct sockaddr” object
  - Every family will have a specific object that inherits from sockaddr (e.g. sockaddr_un for AF_UNIX, sockaddr_in for AF_INET, …)

Sockets

- The struct sockaddr is a generic object, holding 2 bytes for the family identifier, and 14 bytes of specific addressing data
- The object can be “cast” to some more specific types, depending on the address family, each one containing specific information about the socket
- The subtype can be larger than the sockaddr object, because generic socket calls always use pointers

Sockets

- The Berkeley interface is made by the following generic functions/syscalls:
  - socket() - creates a socket with a given family and options. On success, it returns a file descriptor associated with it
  - bind() - associates the given socket to a specific socket address (a sub-type of the **sockaddr** object). This operation is required if the socket has to be reached later on using its address
- There are two types of sockets:
  - SOCK_DGRAM is the abstraction for connection-less sockets. The socket will communicate through messages, aka datagrams
  - SOCK_STREAM is an abstraction of a bidirectional flow, which gives a reliable connection between two sockets. Once the connection is established, data can be injected as for the pipe using write() and can be accessed on the other endpoint using read()

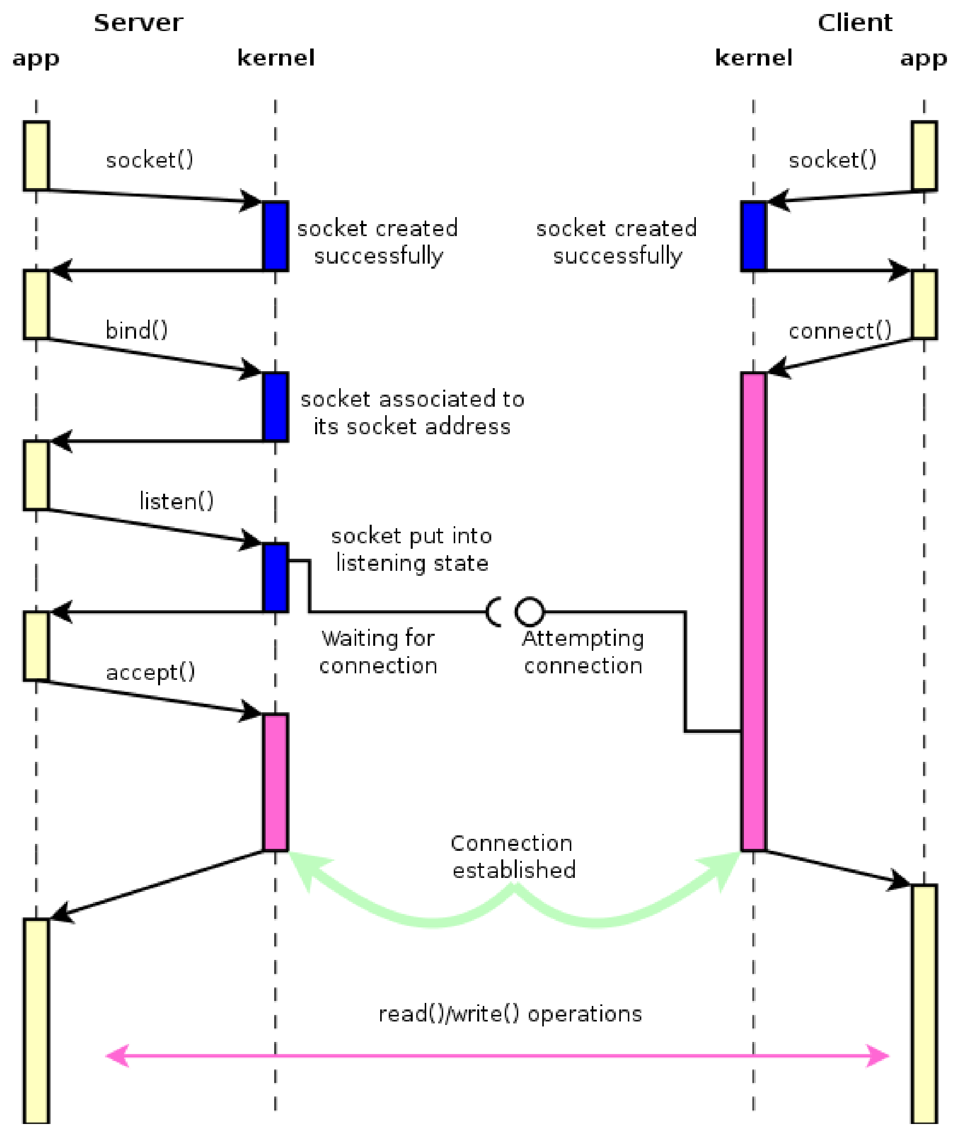
Sockets

- Send to / Receive From
  - Connectionless (DGRAM) sockets use those functions to exchange messages:
    - sendto() places the data in memory in a message to be delivered to the other endpoint of the socket communication indicated by a specific sockaddr
    - send() is used when the local socket is already bound to a remote address, i.e. bind() was called before, so no need to specify the endpoint's address
    - recvfrom() receives a message previously sent from the other endpoint and puts it into the specified area in memory. The sockaddr argument is used to return the sender's address
    - recv() is a simplified version of the call above, when the sender's address is not needed and all we want is simply receive a message from the socket

Sockets

- Connected (stream) sockets need the following infrastructure to define a connection before any data can be exchanged:
  - listen() puts the socket in a listening mode, meaning that it will be able to create new sockets upon a new incoming connection is requested from the other side
  - accept() must be called on a listening socket only, and will block until a new connection request is received. At that point, it will return a new file descriptor that is the real endpoint of the communication. Listening socket will still exist after accept() has returned, and it will be ready for the next connection request
  - connect() will try to open a new communication request to a remote socket that has been previously put into a listening state. The call will block until the connection is accepted by the other endpoint. On success, the given socket is connected and can start to exchange data with the other endpoint

Sockets


Server
app
kernel
Client
kernel
app
socket()
socket created successfully
socket()
socket created successfully
bind()
connect()
socket associated to its socket address
listen()
socket put into listening state
Waiting for connection
Attempting connection
accept()
Connection established
read()/write() operations

Sockets

- Consideration about the socket types
  - A socket without any connection (in DGRAM mode) will have the same role of its opposite endpoint, so the exact terminology for such an agent is “peer”
    - The so-called peer-to-peer communications always happens between two sockets that uses an equivalent implementation
    - Sending and receiving messages is done with the datagram concept in mind
    - In some contexts (e.g. INET) there are no warrants that the messages are received, or even in which order they will arrive…
  - A connected socket obey to the server/client paradigm:
    - A server opens a listening sockets and accepts incoming connections
    - A client tries to connect to the other endpoint
    - After the connection is established, all the data pushed into one socket will be received on the other side, reliably and in the right order.

Sockets

- INET socket
  - It is the abstraction on UNIX machines to TCP/IP communication at the application level…
  - It is used to communicate with a process on a remote machine
  - SOCK_DGRAM will use UDP
  - SOCK_STREAM will use TCP
  - The specific sockaddr subtype for this kind of socket is the sockaddr_in defined in “sys/in.h”, and it contains the following fields:
    - sin_family (Always AF_INET)
    - sin_port (A 2-bytes integer identifying the transport layer port
    - sin_addr (A structure containing the IP address of the endpoint host)
    - sin_zero (unused part of the address, filled with zeros)

Sockets

- INET6 socket
  - An IPv6 version of the classic networking socket
  - Has a structure similar to that of AF_INET
  - SOCK_DGRAM will use UDP
  - SOCK_STREAM will use TCP
  - The specific sockaddr subtype for this kind of socket is the sockaddr_in6 defined in “sys/in6.h”, and it contains the following fields:
    - sin6_family (Always AF_INET6)
    - sin6_port (A 2-bytes integer identifying the transport layer port
    - sin6_flowinfo (IPv6 flow information)
    - sin6_addr (A structure containing the IP address of the endpoint host)
    - sin6_scope_id (scope id from RFC2553)

Sockets

- The sockaddr_storage structure
  - Does not indicate a working socket
    - It is a placeholder for objects implementing sockets
    - Must be used when allocating space instead of sockaddr (it is large enough to contain any socket address information)
  - One meaningful field
    - ss_family (used to distinguish AF_INET from AF_INET6 sockets)

Sockets

- Hostname/Address conversion
  - There are helper functions to:
    - Convert IP addresses from/to human readable format to/from in_addr_t (required by sockaddr_in):
      - inet_aton(), inet_addr(), inet_network(), inet_ntoa()
    - Convert hostnames to socket addresses:
      - getaddrinfo()
    - Convert back socket addresses to hostnames:
      - getnameinfo()

Sockets

- Network addressing
  - Socket's address family is chosen upon creation
    - AF_INET for IPv4
    - AF_INET6 for IPv6
  - Socket address must be an object of the expected type (and length)
    - sockaddr_in for IPv4
    - sockaddr_in6 for IPv6
  - getaddrinfo() entry must be picked accordingly

Sockets

TASS TECHNOLOGY SOLUTIONS

Dedicated to make it work

Designing a socket application:

- Structure your data types
  - Use only fixed length fields (no “int” or “long”, ...)
  - Use packed structures
- Make a diagram of data flows
  - DGRAM or STREAM?
  - How many endpoints are involved?
  - Do you have to prioritize receiving and/or sending?
  - Is your communication balanced?
- Separate global objects from socket specific objects
- Size your buffers according to the expected infrastructure

Sockets

Structure your data types

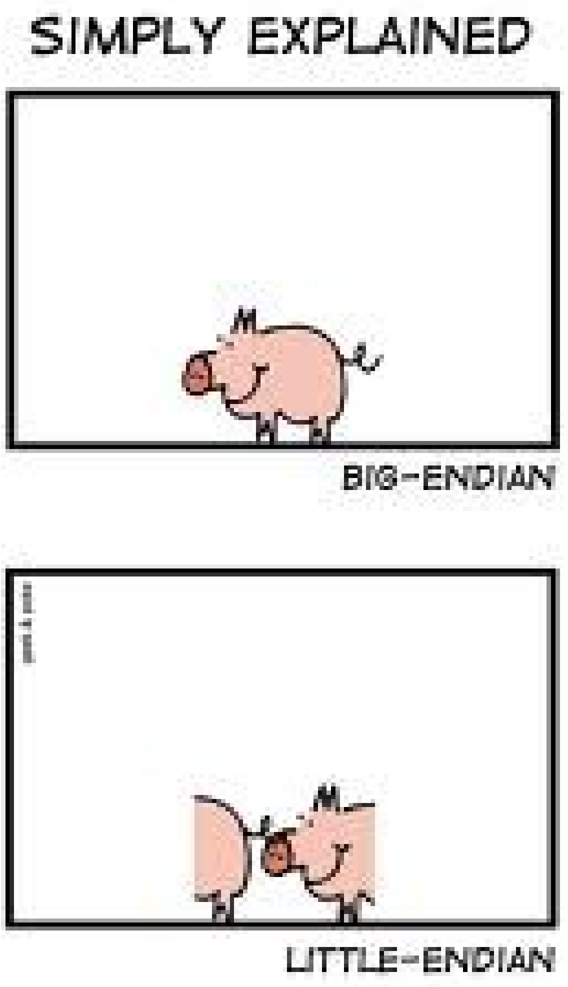
- Never make assumptions about endianness or size of your standard types (such as char, int, long, short). Those types change with the architecture/CPU, and you want your application to be portable across different platforms and Oss
- Use types from <stdint.h> (they are standard from C89, but they can be redefined in the platform support package when not present, as they are well-known)

Sockets

Structure your data types

- Use “network to host” and “host to network” functions available in most systems to convert integer represented over several bytes into BIG ENDIAN

ntohs() htons() ntohl htonl()



Structure your data types

- Never make assumption about the length of your Abstract data types
  - Enums follow the system default word length – do not use them in shared structures
  - Whenever possible, your presentation layer should convert enums into fixed-length fields
  - If you really must use enums, ensure that you add a pragma directive to pre-define the length of the enums

```
#pragma enum 2
```

Structure your data types

- Never make assumption about padding bytes in your ADT
  - struct members are always aligned to the system specific word length, unless explicitly stated otherwise

```
struct my_data_type {
   uint32_t a;
   uint16_t b;
   uint8_t  c;
   uint32_t d;
};
```

What is the length of this struct?

Sockets

Structure your data types

- Never make assumption about padding bytes in your ADT
  - The “pragma pack” preprocessor directive modifies the current alignment rule for struct members

```
#pragma pack 1

struct my_data_type {
   uint32_t a;
   uint16_t b;
   uint8_t  c;
   uint32_t d;
};
```

Sockets

tass TSS TECHNOLOGY SOLUTIONS Dedicated to make it work

Structure your data types

- Never make assumption about padding bytes in your ADT
  - The gcc-dialect attribute “packed” will introduce no padding to objects within the struct, if the struct declaration contains it

```
struct __attribute__((packed)) my_data_type {
   uint32_t a;
   uint16_t b;
   uint8_t  c;
   uint32_t d;
};
```

Sockets

Use well-known primitive types

- Never make assumption about the representation of standard types (int, char)
- Never make assumptions on float representation
  - Use fixed point algebra for decimals

```
uint32 temperature; /* Indicated in 1/128 K */

double net_to_celsius(uint32_t temp)
{
    return (double)(temp >> 7) +
(((double)(temp & 0x7F)) / 128.0) – 273.0;
}
```

Structure your data types

- Never make assumption about padding bytes in your ADT
  - The gcc-dialect attribute “packed” will introduce no padding to objects within the struct, if the struct declaration contains it

```
struct __attribute__((packed)) my_data_type {
   uint32_t a;
   uint16_t b;
   uint8_t  c;
   uint32_t d;
};
```

Sockets

- DGRAM vs STREAM
  - Depends on the use case
  - Rule of thumb: if reliability and/or order is important, use STREAM
  - Stream sockets don't support one-to-many (only end-to-end communication)
  - Never try to reinvent STREAM using datagrams
- STREAM glossary: connection, server, client
- DGRAM glossary: peer, message, datagram

Sockets

- Sequential operation vs Threading
  - How to serve multiple clients at one time?
  - The standard way:

```
int listening_sock, accepted_sock;

listen(listening_sock);
for (;;) {
   accepted_sock = accept(listening_sock, NULL, NULL);

   if (fork() == 0) {
      close(listening_sock);
      serve_client(accepted_sock);
      exit(0);
   }
}
```

Sockets

TASS TECHNOLOGY SOLUTIONS
Dedicated to make it work

- Sequential operation vs Threading
  - How to deal with the “mailman” problem?
    - One application is exchanging data in both directions between two open sockets
    - Data is asynchronous and can be received from both sides at any time
  - Possible solutions:
    - Use one thread/process for each flow path
    - Use poll() or select()

Sockets

tass TSS TECHNOLOGY SOLUTIONS
Dedicated to make it work

- Performance
  - Take into account the topology of the underlying network
  - Collect all the information about the data-link requirements (MTU, buffer size of the interfaces, buffer size of routing objects along the path)
  - Define the data size of your datagram/stream buffers
  - Minimize/avoid memcpy's in your code

Sockets

- One-to-many socket communication
  - Use UDP with a broadcast address as destination
    - “global” broadcast: all hosts (255.255.255.255)
    - Limited to a subnet: last address of the network range
  - Use a multicast UDP group
    - Allows differentiation of traffic by IP using Class D addresses: from 224.0.0.0 to 239.255.255.255 in ipv4 or the ff00::/8 range in IPv6
    - Requires explicit subscription to channel URL via IGMP

Sockets

- The mreq structure

```
struct ip_mreq
{
    /* IP multicast address of group */
    struct in_addr imr_multiaddr;

    /* local IP address of interface */
    struct in_addr imr_interface;
};
```

- Multicast socket options
  - IP_MULTICAST_LOOP
  - IP_MULTICAST_TTL
  - IP_MULTICAST_IF
  - IP_ADD_MEMBERSHIP
  - IP_DROP_MEMBERSHIP

Sockets

- Multicast sender

```
int yes = 1, no = 0, ttl = 32;

/* Create normal UDP Socket */
socket_fd = socket(AF_INET, SOCK_DGRAM, 0);

/* Loop off, to avoid receiving data being sent */
setsockopt(socket_fd, IPPROTO_IP, IP_MULTICAST_LOOP,
&no, sizeof(no));

/* Set higher TTL if needed (default is 1) */
setsockopt(socket_fd, IPPROTO_IP, IP_MULTICAST_TTL,
&ttl, sizeof(ttl));

/* Send data to group listeners */
sendto(socket_fd, data, len, (struct sockaddr*)&dest, socksize);
```

Sockets

tass TSS TECHNOLOGY SOLUTIONS

Dedicated to make it work

- Multicast receiver:

```
struct ip_mreq mreq;

/* Create normal UDP Socket */
socket_fd = socket(AF_INET, SOCK_DGRAM, 0);

/* Bind to Multicast group, to allow recv() */
bind(socket_fd, (struct sockaddr*)&mcast, sizeof(mcast));

/* Fill ip_mreq structure */
mreq.imr_multiaddr = mcast.sin_addr;
mreq.imr_interface.s_addr = htonl(INADDR_ANY);

/* This triggers the IGMP JOIN packet */
setsockopt(socket_fd, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq,
sizeof(mreq));

/* Receive Multicast traffic */
recvfrom(socket_fd, data, len, &sender, &sender_socksize);
```