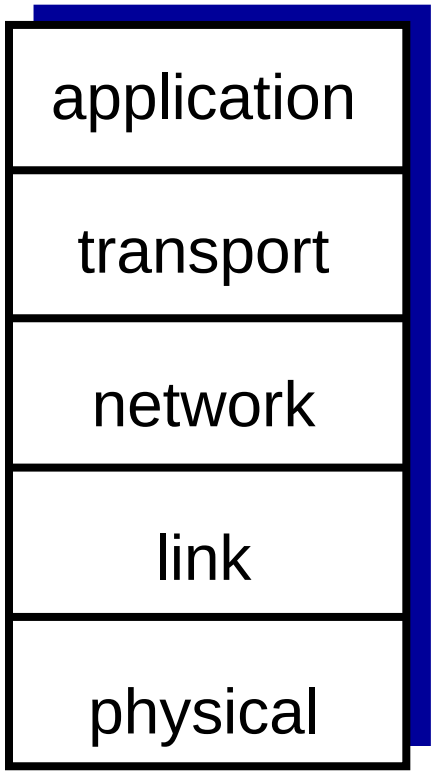
Network Programming

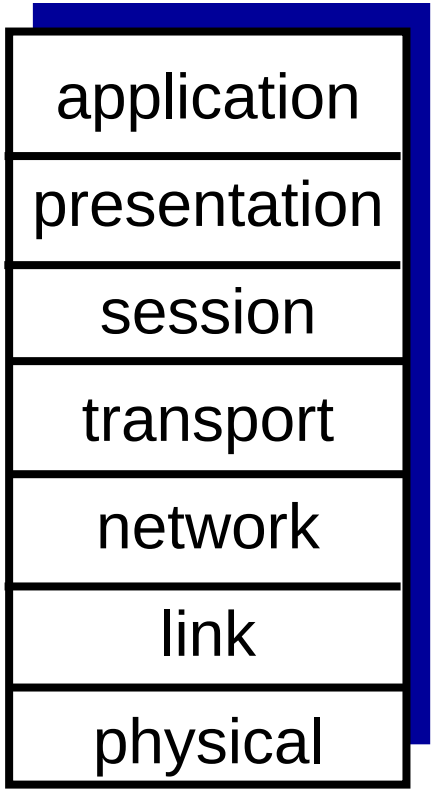
Internet protocol stack

- *application:* supporting network applications
  - FTP, SMTP, HTTP
- *transport:* process-process data transfer
  - TCP, UDP
- *network:* routing of datagrams from source to destination
  - IP, routing protocols
- *link:* data transfer between neighboring network elements
  - Ethernet, 802.111 (WiFi), PPP
- *physical:* bits “on the wire”

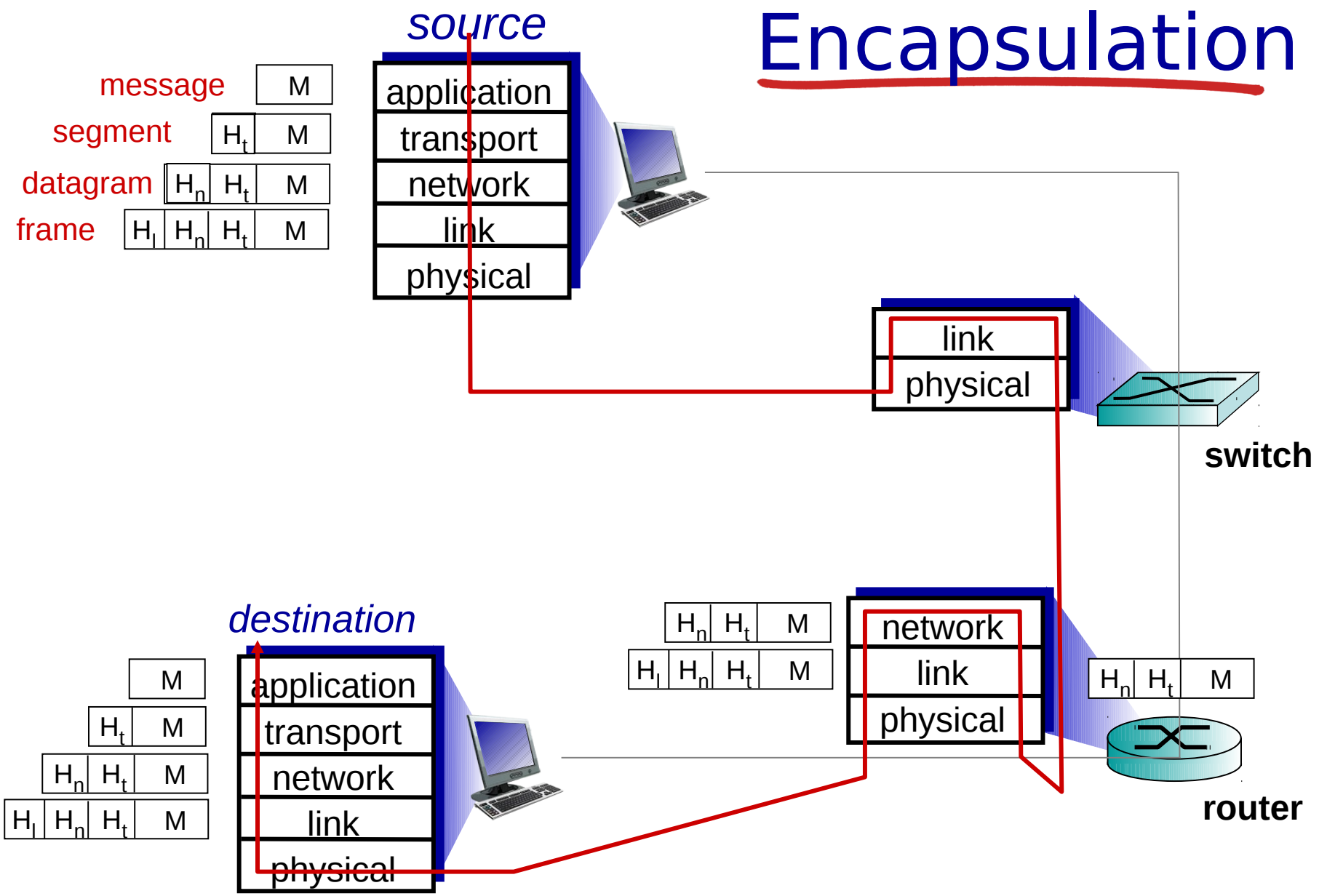


ISO/OSI reference model

- *presentation:* allow applications to interpret meaning of data, e.g., encryption, compression, machine-specific conventions
- *session:* synchronization, checkpointing, recovery of data exchange
- Internet stack "missing" these layers!
  - these services, *if needed,* must be implemented in application?



Encapsulation



Common Transport Protocols

- UDP (User Datagram Protocol)
  - “Connectionless”
  - No guarantee that datagram will reach destination
  - No guarantee that order will be preserved
  - No guarantee that there won’t be repeats.
- TCP (Transmission Control Protocol)
  - “Connection-oriented”
  - Reliable
  - Sequence
  - Full duplex

Common Network Layer Protocols

- IPv4
  - 32 bit host addresses
  - Running out! (Ran out?)
- IPv6
  - 128 bit host addresses

API

- **socket**: create a socket and return a file descriptor
- Server
  - **bind**: establish the socket's "interface" and port.
  - **listen**: put the socket in "passive" mode.
  - **accept**: wait for a connection
- Client
  - **connect**: request a connection to a server.

Getting a Socket Descriptor

- **int socket(int domain, int type, int protocol)**
  - <sys/socket.h>
  - Returns socket descriptor, or -1
- Domain: (address family)
  - **AF_INET: IPv4**
  - AF_INET6: IPv6
  - AF_UNIX: UNIX
    - AF_LOCAL
- Type
  - **SOCK_STREAM: sequenced, reliable, bidirectional, connection-oriented (TCP)**
  - SOCK_DGRAM: fixed-length, connectionless, unreliable (UDP)
  - SOCK_RAW: direct access to network level, skipping transport level
  - SOCK_SEQPACKET (SCTP)
- Protocol: zero specifies “use the default”.

Associating Addresses with Sockets

- int **bind**(int socfd,
  struct sockaddr *addr,
  socklent_t len)
- Why pass a length?
- Because there are different structs for different types of socket addresses.

Addressing Formats

- sockaddr {
  sa_family_t sa_family;
  char sa_data[14];
  }
- sockaddr_in {
  sa_family_t sin_family;
  in_port_t sin_port; // uint16_t
  struct in_addr sin_addr;
  }
- in_addr {
  in_addr_t s_addr; // uint32_t
  }
  - Reason for the struct in_addr is historical.
  - You can safely treat sin_addr as an unsigned int
  - s_addr can be INADDR_ANY for a server socket
- “Well-known” ports are listed in: /etc/services
  - ftp: 21, ssh: 22, smtp: 25, http: 80
- Only root can use port numbers < 1024

Addressing Issues

- Byte Ordering Issue
  - Big-endian vs. little-endian
  - Conversion functions: <arpa/inet.h>
    - htonl, htons, ntohl, ntohs
- Human readable:
  - Old: inet_addr, inet_ntoa // only for IPv4 (deprecated)
  - New: inet_ntop, inet_pton // works on both IPv4 and IPv6
  - #include <arpa/inet.h>
    const char *inet_ntop(int af, const void *restrict src,
    char *restrict dst, socklen_t size);
    - af: AF_INET, src points to in_addr,
      dst will be dotted-decimal
    - af: AF_INET6, src points to in6_addr,
      dst will be in an “IPv6 appropriate format”
  - int inet_pton(int domain, const char *restrict str,
    void *restrict addr);

Address Lookup

- int gethostname(char *_name_, int _size_);
  - Name of current host
- struct hostent *gethostbyname(char *_name_);

- NB: IPv6 uses different functions…

Connection Establishment

- int listen(int *sockfd*, int *backlog*)
  - “Server announces that it is willing to accept connection requests”
  - backlog provides “hint to the system”
- int accept(int *sockfd*,
  struct sockaddr **addr*,
  socklen_t **len*)
  - Wait for a connection using file descriptor returned by socket
  - If you want to know who connected to you provide an address to fill in.
  - Return value is a file descriptor that the server can use to communicate with the client (for both reading and writing).

Client?

- Get a socket
  - Same as with the server
- Set up a sockaddr to the server
  - Use the *server's* address, not your own.
  - Otherwise the setup is the same
- Connect the sockaddr
  - int connect(int sockfd,
    struct sockaddr *serv_addr,
    socklen_t addrlen);
  - The return value just indicates whether you succeeded.
  - The sockfd can now be used to read from / write to the server.