9. Network Interface

Outline

- Quick introduction
- Sockets and ports
- Primitive operations
- Examples
- Threaded server model

Sockets

What?

- Bidirectional communication channel across systems (called *hosts*)
- Common interface to multiple layers of multiple networking protocol stacks

Sockets

What?

- Bidirectional communication channel across systems (called *hosts*)
- Common interface to multiple layers of multiple networking protocol stacks

Networking Domains

- *INET*: Internet Protocol (IP)
- *UNIX*: efficient host-local communication
- And many others (IPv6, X.25, etc.)

- `$ man 7 socket`
- `$ man 7 ip` (for INET sockets)
- `$ man 7 unix`

Sockets

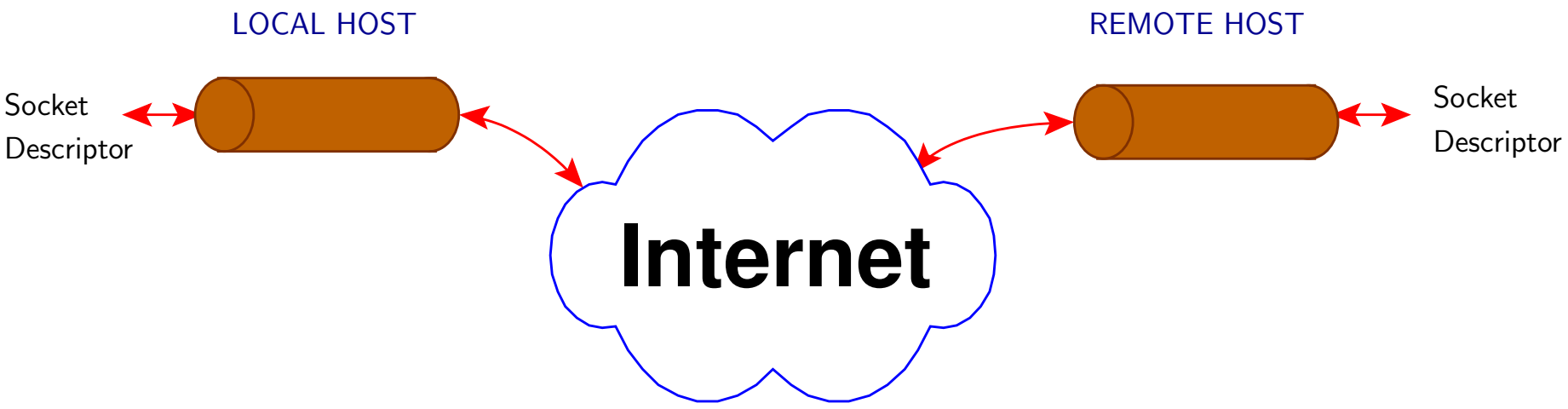
What?

- Bidirectional communication channel across systems (called *hosts*)
- Common interface to multiple layers of multiple networking protocol stacks

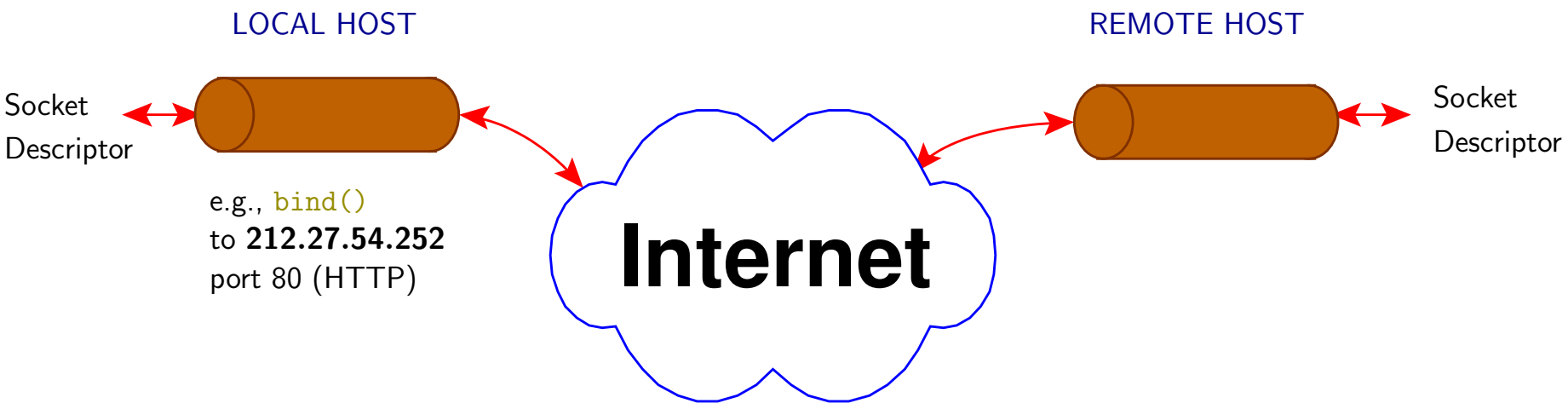
Socket Types

- STREAM: *connected* FIFO streams, reliable (error detection and replay), without message boundaries, much like `pipe`s across hosts
- DGRAM: *connection-less*, unreliable (duplication, reorder, loss) exchange of messages of fixed length (datagrams)
- RAW: direct access to the raw protocol (not for UNIX sockets)
- Each type is associated with a specific mechanism to *address* remote sockets
  - `$ man 7 tcp` (for STREAM sockets)
  - `$ man 7 udp` (for DGRAM sockets)
  - `$ man 7 raw` (for RAW sockets)
- Typical example
  32-bit IPv4 address and 16-bit port for STREAM/DGRAM INET sockets

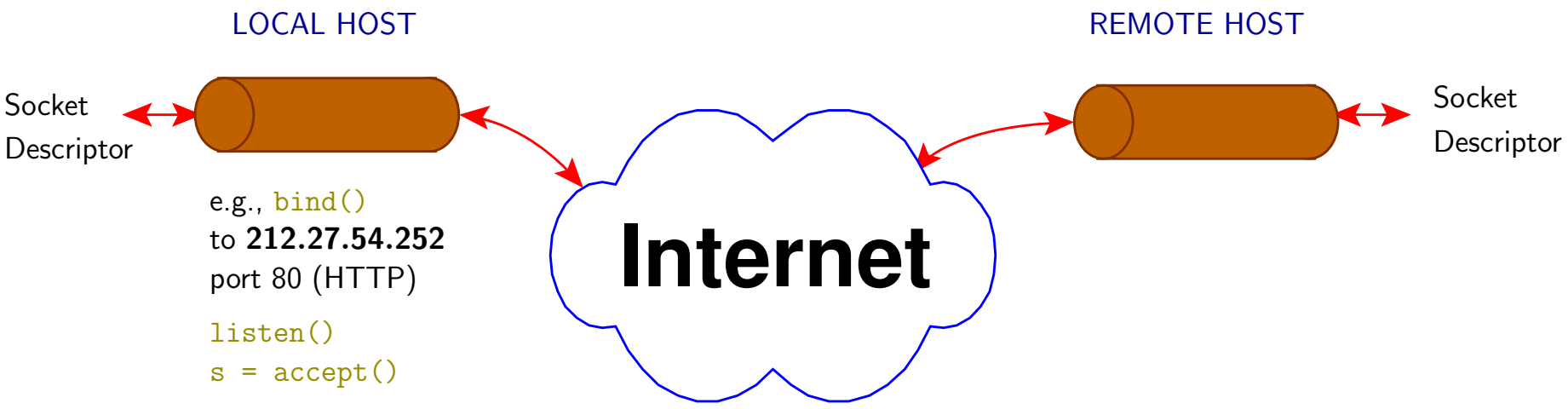
Establishing a Socket-to-Socket Connection



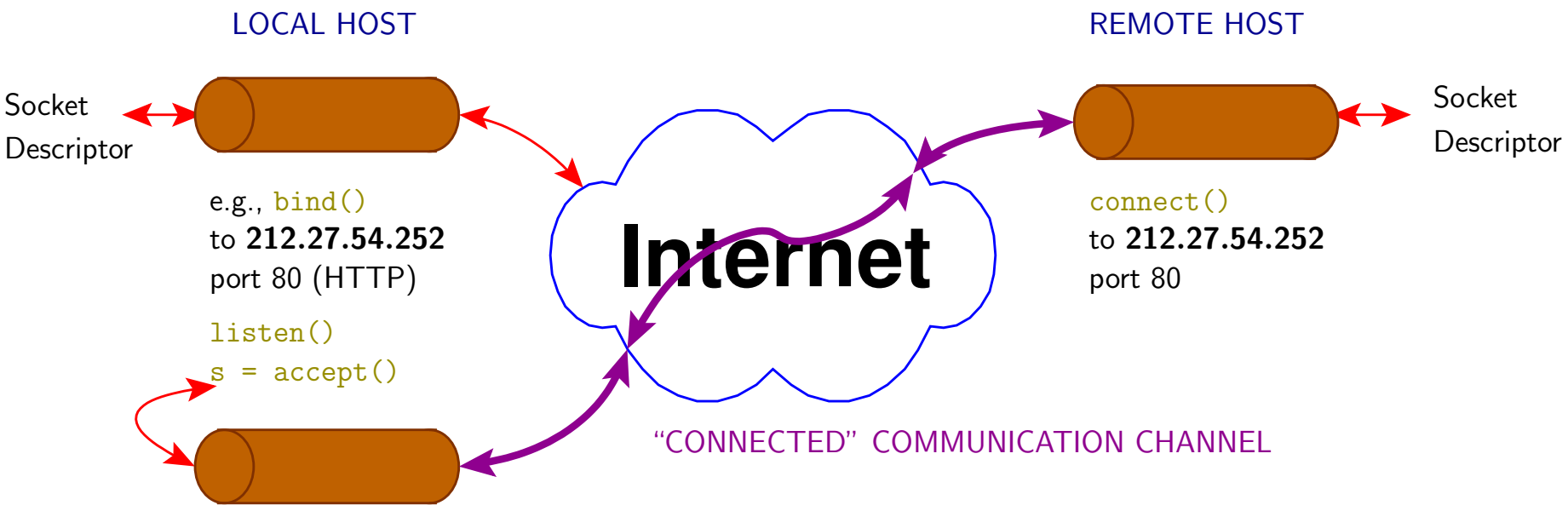
Establishing a Socket-to-Socket Connection



Establishing a Socket-to-Socket Connection



Establishing a Socket-to-Socket Connection



Scenarios for Socket-to-Socket Connection

Direct Communication Scenario

- Create a socket with `socket()`
- Bind to a local address with `bind()`
- Call `listen()` to tell the socket that new connections shall be accepted
- In the remote host, go through the first **2** steps exchanging the roles of local and remote addresses, and calling `connect()` instead of `bind()`
- Only DGRAM (UDP) sockets can be operated that way
- In addition to the unreliability of UDP, this approach leads to painful problems
  1. Port numbers provide only a partial support for a rendez-vous protocol: no synchronization is enforced
  2. Reading or writing from an unconnected socket raises `SIGPIPE` (like writing to a pipe without readers)
  3. The socket cannot be reused for another connection

Scenarios for Socket-to-Socket Connection

TCP Abstraction: Creation of a Private Channel

- Create a socket with `socket()`
- Bind to a local address with `bind()`
- Call `listen()` to tell the socket that new connections shall be accepted
- Call `accept()` to wait for an incoming connection, returning a new socket associated with a private channel for this connection
- In the remote host, go through the first **2** steps exchanging the roles of local and remote addresses, and calling `connect()` instead of `bind()`
- The original pair of sockets can be reused to create more connections

Example: Establishing a Socket for Incoming Connections

```
#include <netdb.h>
#include <sys/socket.h>
int establish(unsigned short portnum) {
  char myname[MAXHOSTNAME+1];
  int s;
  struct sockaddr_in sa;

  bzero(&sa, sizeof(struct sockaddr_in));          // clear our address
  gethostname(myname, MAXHOSTNAME);                // who are we?
  struct hostent *hp = gethostbyname(myname);      // get our address info
  if (hp == NULL)                                  // we don't exist !?
    return -1;
  sa.sin_family= hp->h_addrtype;                   // this is our host address
  sa.sin_port= htons(portnum);                     // and our big-endian port
  if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) // create socket
    return -1;
  if (bind(s, &sa,sizeof(sa), 0) < 0) {
    close(s);
    return -1;                                     // bind address to socket
  }
  listen(s, 3);                                    // max # of queued connections
  return s;
}
```

Example: Waiting for Incoming Connections

```
int wait_for_connections(int s) {     // socket created with establish()
  struct sockaddr_in isa;             // address of socket
  int i;                              // size of address
  int t;                              // socket of connection

  if ((t = accept(s, &isa, &i)) < 0)  // accept connection if there is one
    return -1;
  return t;
}
```

Example: Opening an Outgoing Connection

```
int call_socket(char *hostname, unsigned short portnum) {
  struct sockaddr_in sa;
  struct hostent *hp;
  int a, s;

  if ((hp = gethostbyname(hostname)) == NULL) {          // do we know
    errno = ECONNREFUSED;                                // the host's address?
    return -1;                                           // no
  }

  bzero(&sa, sizeof(sa));
  bcopy(hp->h_addr, (char *)&sa.sin_addr, hp->h_length); // set address
  sa.sin_family = hp->h_addrtype;
  sa.sin_port = htons(portnum);

  if ((s = socket(hp->h_addrtype, SOCK_STREAM, 0)) < 0)  // get socket
    return -1;
  if (connect(s, &sa, sizeof(sa)) < 0) {                 // connect
    close(s);
    return -1;
  }
  return s;
}
```

Communicating Through a Pair of Sockets

Connected Socket I/O

- System calls `read()` and `write()` work as usual on *connected* sockets (otherwise raise `SIGPIPE`)
- System calls `recv()` and `send()` refine the semantics of `read()` and `write()` with additional flags to control socket-specific I/O (out-of-band, message boundaries, etc.)

Connection-Less Socket I/O

- A *single* DGRAM (UDP) socket can be used to communicate
- System calls: `recvfrom()` and `sendto()`

Application: Threaded Server Model

Dynamic Thread Creation

1. A *main thread* `listens` for a `connection` request on a *predefined port*
2. After `accepting` the request, the server creates a thread to handle the request and resumes `listening` for another request
3. The thread `detaches` itself, performs the request, closes the `socket` in response to the client's closing and returns

⟶ Key system call: `accept()`: the thread function takes the `socket` returned from the system call as a parameter

Application: Threaded Server Model

Dynamic Thread Creation

1. A *main thread* `listens` for a `connection` request on a *predefined port*
2. After `accepting` the request, the server creates a thread to handle the request and resumes `listening` for another request
3. The thread `detaches` itself, performs the request, closes the `socket` in response to the client's closing and returns

⟶ Key system call: `accept()`: the thread function takes the `socket` returned from the system call as a parameter

Worker Pool

1. A *main thread* plays the role of a *producer*
2. A *bounded* number of *worker threads* play the role of *consumers*
3. The main thread `listens` for `connection` requests and asks the workers to process them (e.g., with a signal)

Application: Threaded Server Model

Dynamic Thread Creation

1. A *main thread* `listen`s for a `connect`ion request on a *predefined port*
2. After `accept`ing the request, the server creates a thread to handle the request and resumes `listen`ing for another request
3. The thread `detach`es itself, performs the request, closes the `socket` in response to the client's closing and returns

⟶ Key system call: `accept()`: the thread function takes the `socket` returned from the system call as a parameter

Worker Pool

1. A *main thread* plays the role of a *producer*
2. A *bounded* number of *worker threads* play the role of *consumers*
3. The main thread `listen`s for `connect`ion requests and asks the workers to process them (e.g., with a signal)

More Information and Hard Core Optimizations

`http://www.kegel.com/c10k.html`