Network Programming

David Bouchet

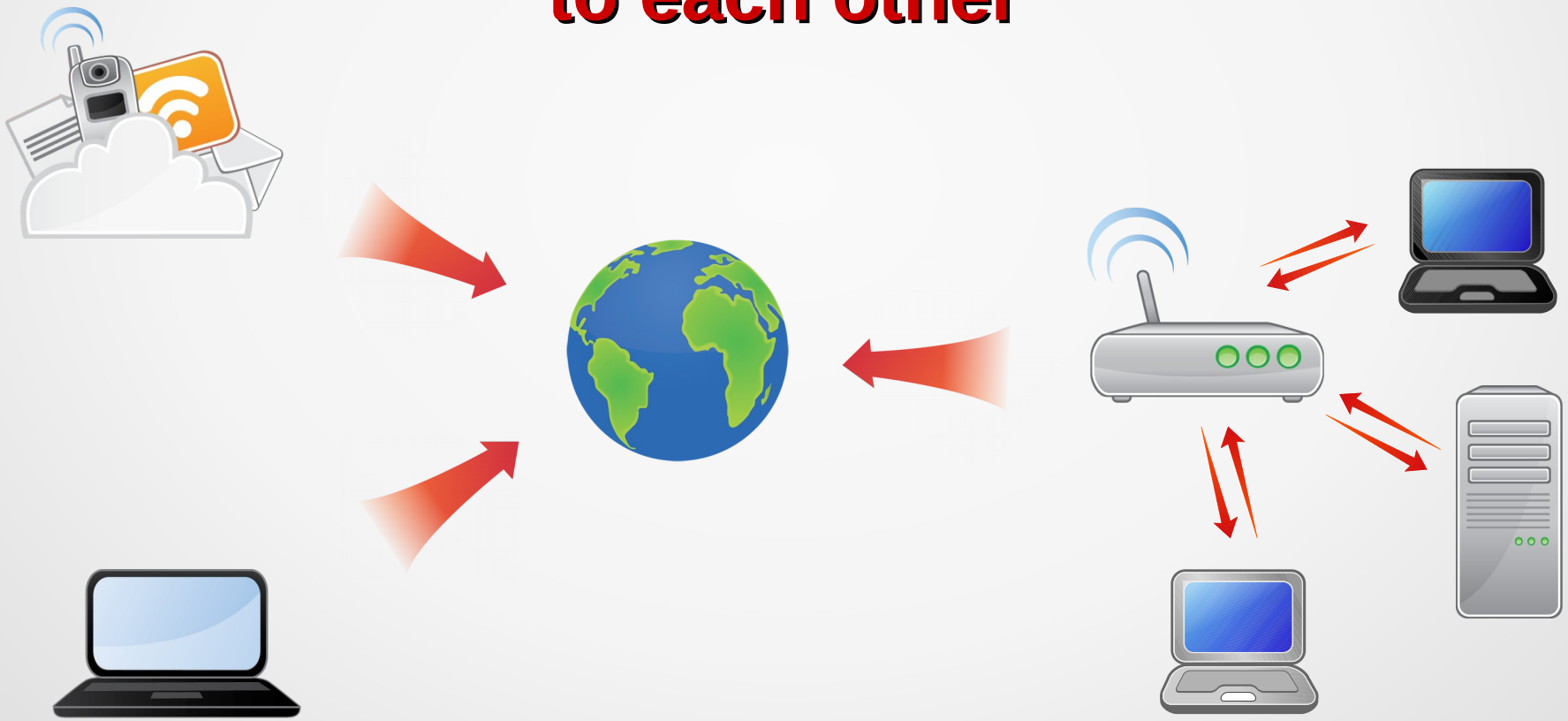
david.bouchet.epita@gmail.com

Quick Overview

1. IP and Protocol Stack
2. TCP Concepts
3. Client / Server Concepts
4. Socket API
5. Code

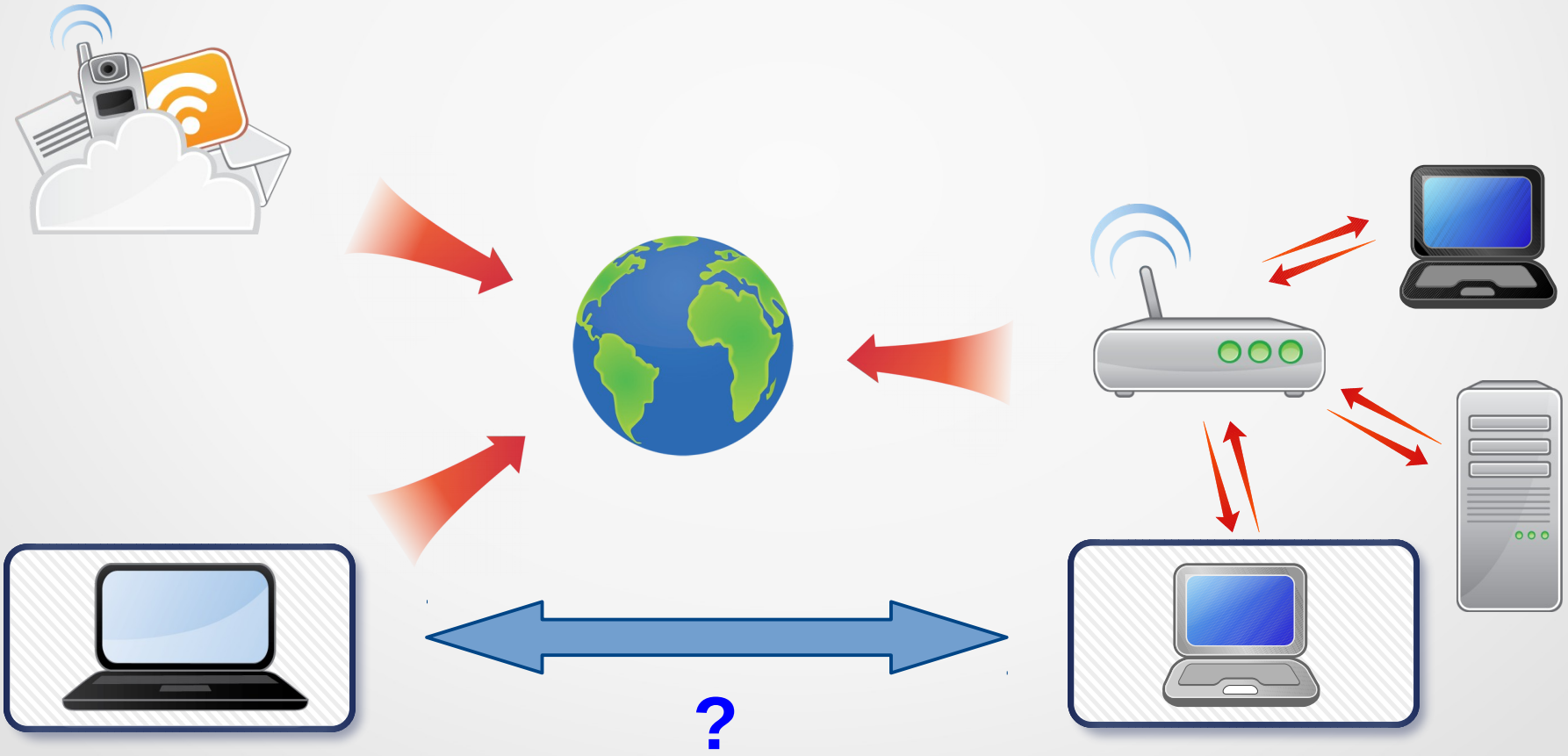
Computer Network Programming

Connecting computers and networks to each other



Computer Network Programming

How can we connect these two computers?



The Internet Protocol – IP

Goals: Abstracting heterogeneous networks

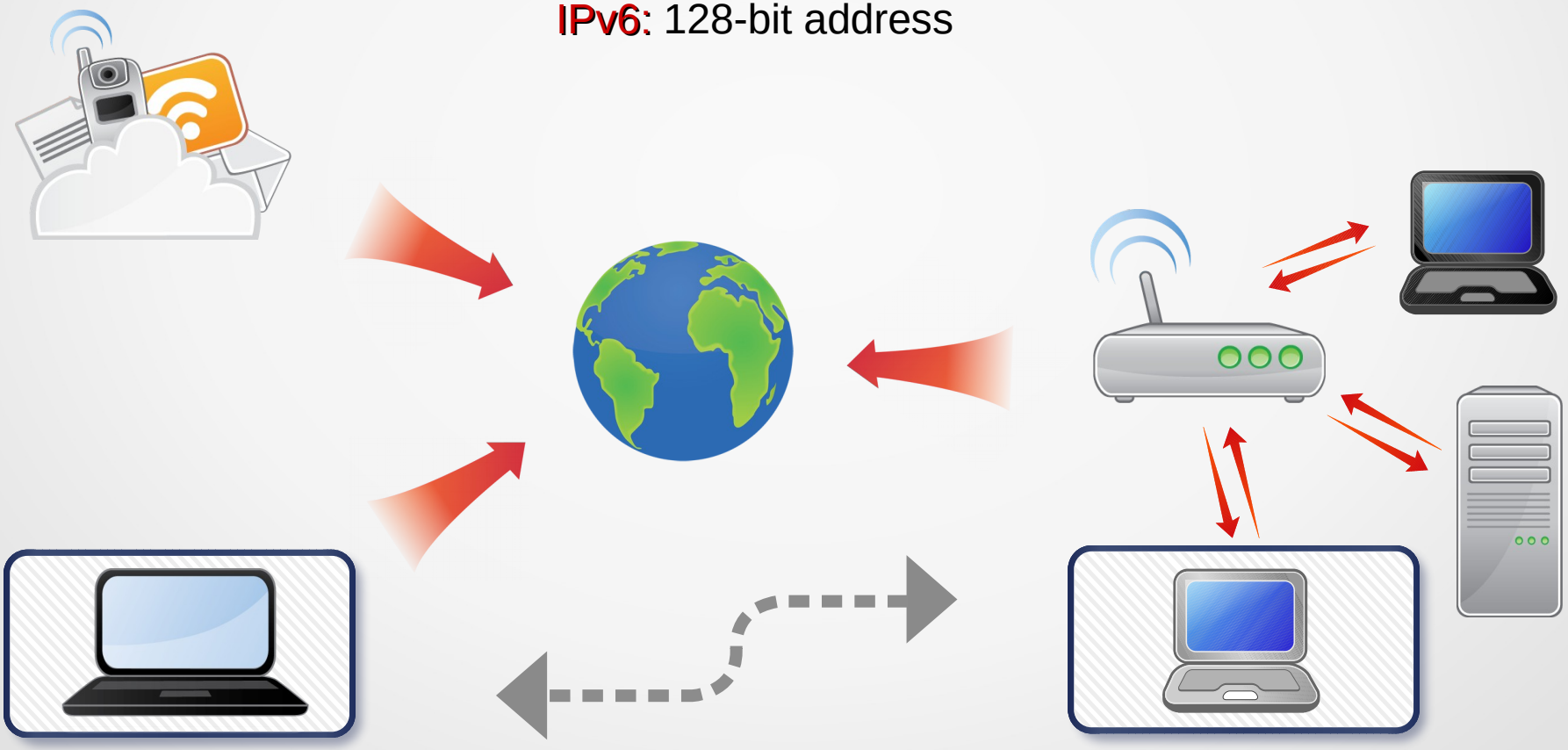
- **Inter**connected **Net**works
- Unified address space over the whole network
- Provides a global logical net over physical ones
- **Routing:** Selecting paths in networks

The Internet Protocol – IP

Each computer has an IP address

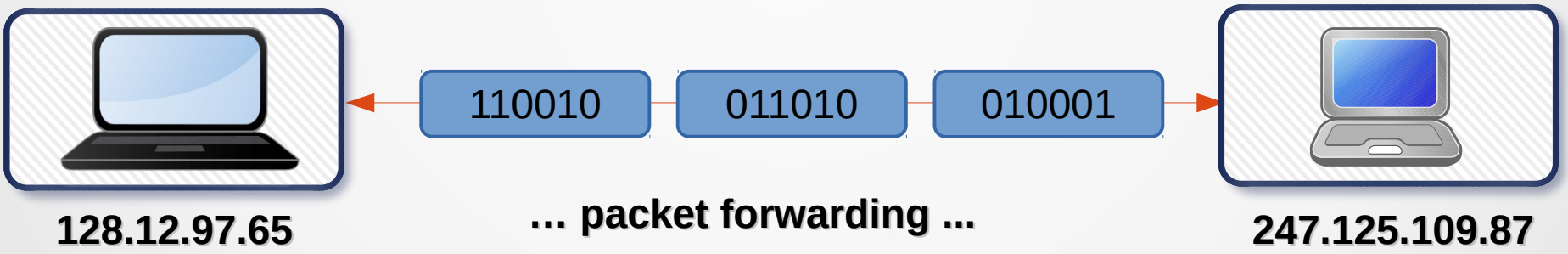
IPv4: 32-bit address

IPv6: 128-bit address



Network Packets

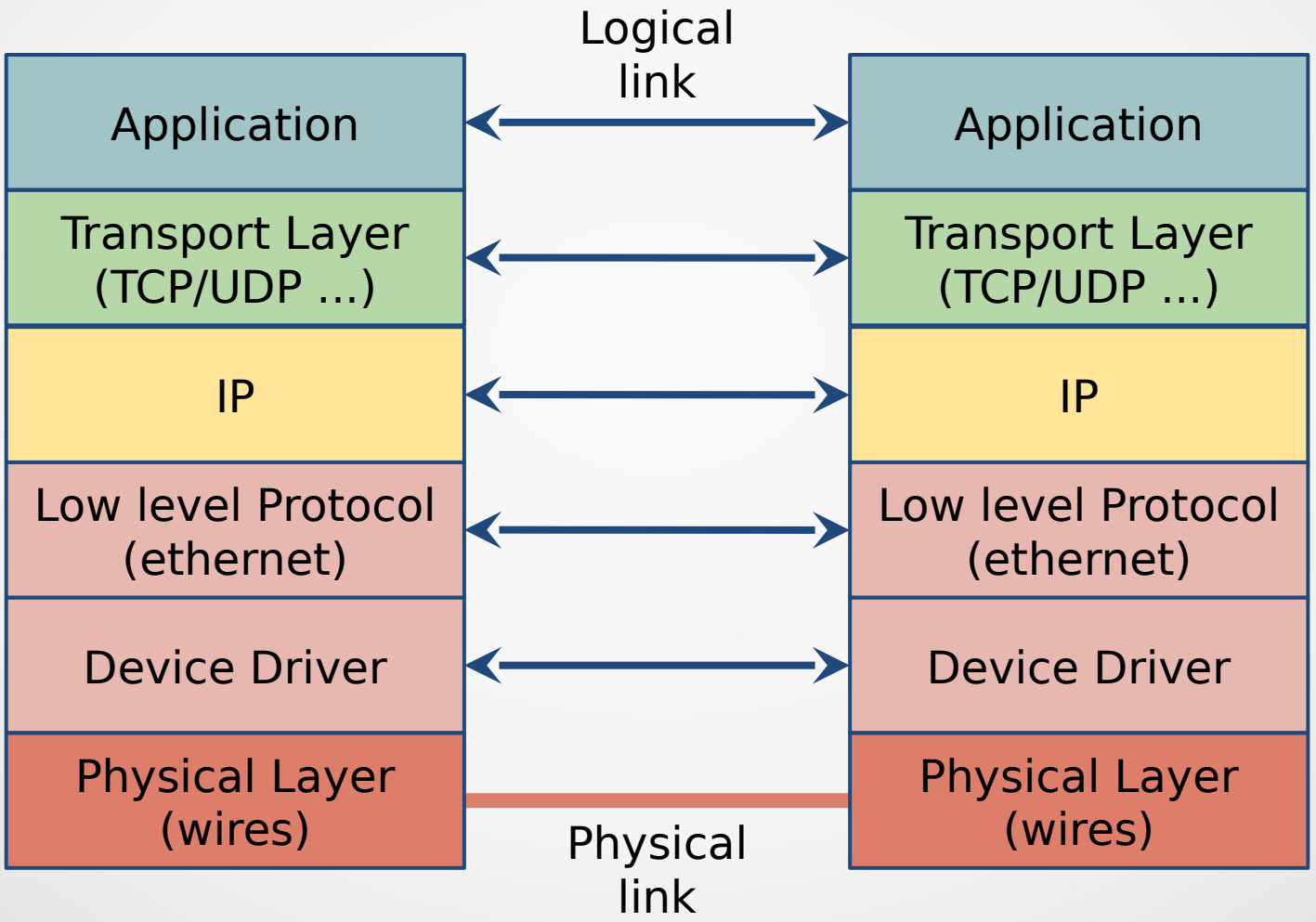
Data is split into packets



Protocol Stack

- IP relies on a whole stack of protocols
- Horizontal logical connection
- Vertical concrete communication

Protocol Stack



Protocol Stack

- **Physical Layer:** Made up of electronic circuits
- **Device Driver:** Software interface to drive the physical layer
- **Low Level Protocol:** Transfers data between adjacent or local networks
- **Internet Protocol:** Routing and packet forwarding.

Protocol Stack

- **Transport Layer:** Protocols (TCP, UDP, etc.) that provide services:
  - delivery to applications,
  - connected or not,
  - same or different order delivery,
  - reliability or unreliability,
  - flow control,
  - etc.
- **Application Layer:** Specifies communication protocols and interface methods (FTP, SMTP, HTTP, Telnet, IMAP, etc.)

Transport Layer

TCP

- Stream oriented
- Connected
- Acknowledgement
- Retransmissions
- Packets ordering
- Timeouts

UDP

- Datagram oriented
- Not connected
- No reception check
- Fire and forget
- No ordering

Client / Server Model

Client

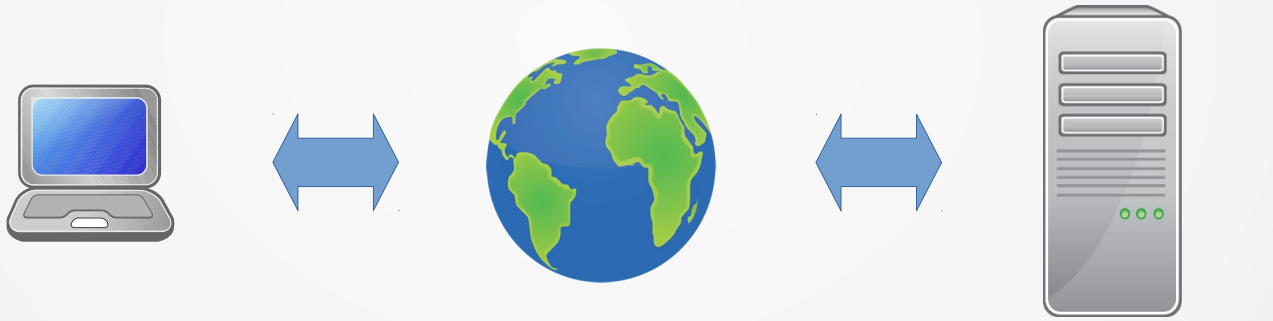
- User of the service
- Initiates connection
- Ends connection (TCP)
- Uses dynamic ports

Server

- Service provider
- Waits for connection
- Serves multiple clients
- Uses fixed ports

Client / Server Model

Example



Client
(The Web Browser)

The Internet

Server
(Apache, IIS)

Port Numbers

Port Number: **16-bit unsigned integer** (from 0 to 65,535)

- IP addresses identify hosts
- Applications identified by port numbers
- All TCP (or UDP) communications use ports
- A connection is identified by the double pair:

((IP1, port1) , (IP2, port2))

Socket API

- Most used API for TCP / UDP connections
- Compatible with classic C Input / Output
- Once established, a socket is just an FD
- Use recv(2) / send(2)
- Or read(2) / write(2)

Network Address and Service Translation

```
int getaddrinfo(const char *node,
                const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

```
void freeaddrinfo(struct addrinfo *res);
```

```
const char *gai_strerror(int errcode);
```

→ getaddrinfo(3)

Using getaddrinfo(3)

getaddrinfo() returns one or more addrinfo structures, each of which contains an Internet address that can be specified in a call to bind(2) or connect(2).

```
struct addrinfo {
    int              ai_flags;
    int              ai_family;
    int              ai_socktype;
    int              ai_protocol;
    socklen_t        ai_addrlen;
    struct sockaddr *ai_addr;
    char            *ai_canonname;
    struct addrinfo *ai_next;
};
```

→ getaddrinfo(3)

Using `getaddrinfo(3)`

If ***hints*** is not NULL it points to an ***addrinfo*** structure whose ***ai_flags***, ***ai_family***, ***ai_socktype***, and ***ai_protocol*** specify some criteria that limit the set of socket addresses returned by getaddrinfo().

All other fields in the structure pointed to by ***hints*** must contain either 0 or a null pointer.

getaddrinfo(3)

TCP Client Connection

- Use `getaddrinfo()`.
- For each *addrinfo* struct, try to:
  - Create the socket: socket(2)
  - Connect to the socket: connect(2)
  - Stop when a connection is established.
- Read from / write to the socket.
- Close the connection.

TCP Client Connection

```c
struct addrinfo hints;
struct addrinfo *result;
int addrinfo_error;

memset(&hints, 0, sizeof (struct addrinfo));
hints.ai_family = AF_INET;       // IPv4 only
hints.ai_socktype = SOCK_STREAM; // TCP

// Get your info
addrinfo_error = getaddrinfo(name, port, &hints, &result);

// Error management
if (addrinfo_error != 0)
{
   errx(EXIT_FAILURE, "Fail getting address for %s on port %s: %s",
   name, port, gai_strerror(addrinfo_error));
}
```

TCP Client Connection

```
// result points to a linked list
// try to connect for each result
for (rp = result; rp != NULL; rp = rp->ai_next)
{
  cnx = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
  if (cnx == -1) continue;
  if (connect(cnx, rp->ai_addr, rp->ai_addrlen) != -1) break;
  close(cnx);
}

freeaddrinfo(result);

if (rp == NULL)
  errx(EXIT_FAILURE, "Couldn't connect");
```

TCP Client Connection (deprecated)

```
void client() {
  int                   sockfd;
  struct sockaddr_in    addr;
  struct hostent       *server;
  // Socket creation
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  // Get server address
  server = gethostbyname("example.com");
  // Init sockaddr struct
  memset(&addr, 0, sizeof (struct sockaddr_in));
  addr.sin_family = AF_INET;
  memcpy(&addr.sin_addr.s_addr, server->h_addr_list[0], server->h_length);
  addr.sin_port = htons(80);
  // Connection
  connect(sockfd, (struct sockaddr*)&addr, sizeof (struct sockaddr_in));
  // read/write on sockfd
  // Done
  close(sockfd);
}
```

TCP Server Connection

- Use `getaddrinfo()` with:
  hints.ai_flags = AI_PASSIVE;
- For each *addrinfo* struct, try to:
  - Create the socket: socket(2)
  - Bind the socket: bind(2)
- Use listen(2) to listen for connection.
- In an infinite loop:
  - Use accept() to accept incoming connection.
  - Read from / write to the socket.
  - Close the connection.

Getting and Setting Options on Sockets

The getsockopt(2) and setsockopt(2) functions allow you to get and set different options on sockets.

The options are listed in socket(7).

Getting and Setting Options on Sockets

Example

When you close a socket, the connection is not necessarily closed immediately by the system. Therefore, you cannot bind again to the server right away. You have to wait.

When the `SO_REUSEADDR` option is enabled, you can bind again to the server immediately.

Getting and Setting Options on Sockets

Example

To enable the SO_REUSEADDR option, you have to set its value to 1.

```
int value = 1;

int err = setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR, &value, sizeof(int));

if (err == -1)
{
    // Error handling
}
```

More Than One Connection

Idea: Handling a connection while waiting for others.

```
// after init ...
listen(fd_accept, 5);
for (;;) {
  // Accept a cnx
  fdcnx = accept(fd_accept, (struct sockaddr*)&remote, &rlen);
  if (fork()) {
    // father
    close(fdcnx);
    continue;
  }
  // child
  close(fd_accept);
  // Read/Write ... Wait for EOF from client side
  close(fdcnx);
}
close(fd_accept);
```

Managing Zombies

- After each connection, the handling process becomes a zombie.
- We shall catch SIGCHLD to clear that.

```
void chldhandler(int sig)
{
  wait(NULL);
}

void server(uint16_t portno)
{
  // ...
  signal(SIGCHLD, chldhandler);
  // ...
}
```

or

```
void server(uint16_t portno)
{
  // ...
  signal(SIGCHLD, SIG_IGN);
  // ...
}
```

Managing Zombies

man 2 sigaction

```
[...]
POSIX.1-1990 disallowed setting the action for
SIGCHLD to SIG_IGN. POSIX.1-2001 and later allow
this possibility, so that ignoring SIGCHLD can be
used to prevent the creation of zombies (see
wait(2)). Nevertheless, the historical BSD and
System V behaviors for ignoring SIGCHLD differ, so
that the only completely portable method of
ensuring that terminated children do not become
zombies is to catch the SIGCHLD signal and perform
a wait(2) or similar.
[...]
```