Network Programming

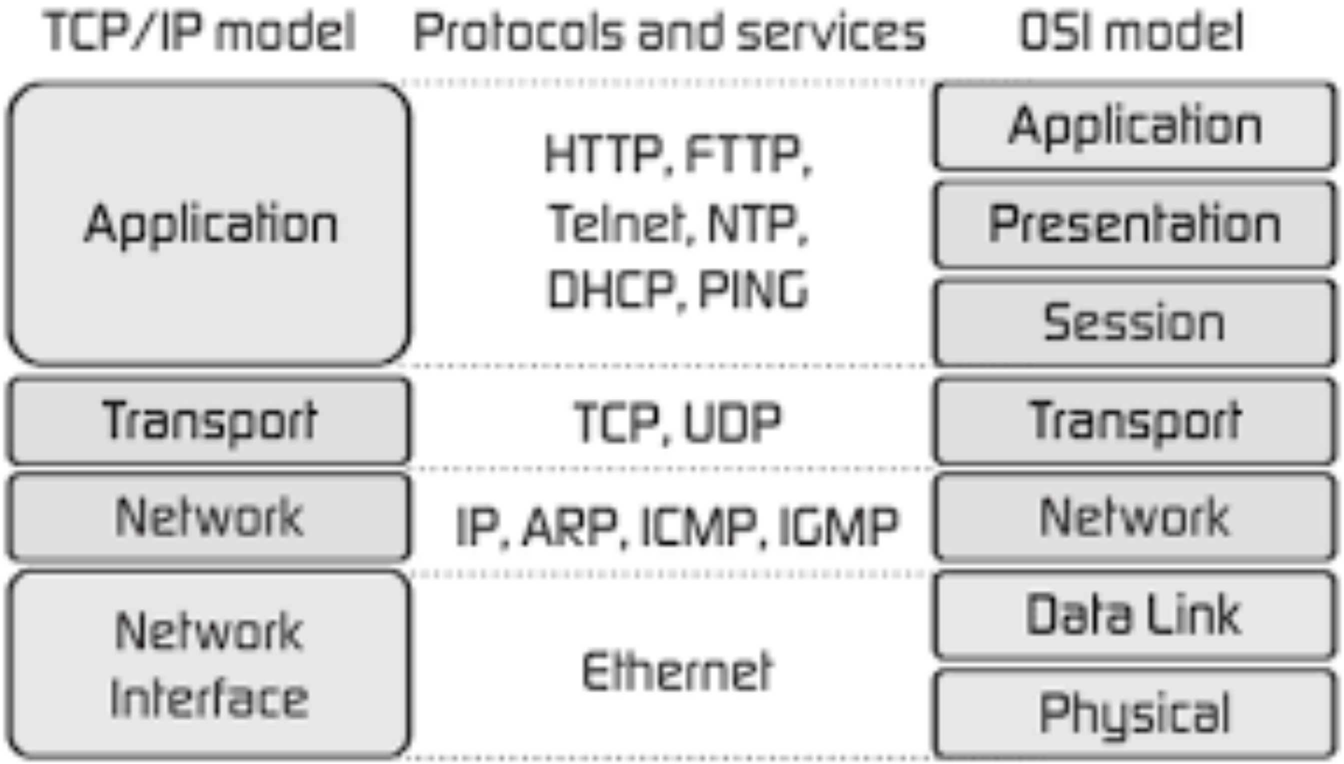
L. Pautet

Version EN, 2.0

IP, UDP et TCP

- IETF has defined a series of protocols:
  - IPv4 and IPv6 for addressing machines;
  - TCP and UDP for exchanging messages over IP:
  - These are basic standards from a user point of view
  - Other protocols concern physical or application aspects
  - The IETF only defines a message format and the controllers for establishing connections, exchanging messages, handling errors, fragmentation, etc.

OSI Model vs TCP/IP Model



OSI Model – Presentation layer Heterogenous systems

- Endianness designates the byte ordering in memory
- A big-endian system stores the most significant byte of an integer at the smallest memory address
- For internet protocols, the network order is big-endianness.
- Functions convert 16-bit and 32-bit integers between network byte order and host byte order
  - htonl(net_long host_long)
  - htons(net_short host_short)
  - ntohl(host_long net_long)
  - ntohs(host_short net_short)

IPv4 address space

- IPv4 uses 32_bit addresses with a quad-dotted decomposition:
  - 137.194.2.34, netid = 137.194, hostid = 2.34
- An IPv4 address is divided into two parts: netid et hostid
  - Netid : network identifier
  - Hostid : host identifier
- Used for routing and network interface identification
- IPv6 was developed by the IETF to deal with IPv4 address exhaustion. Supposed to replace IPv4 but the move is complex.

IPv4 address space

- The netid is partitioned into network classes:
  - Class A, netid coded on 1 byte (leading bits 0):
    - Addresses from 1.0.0.0 to 126.0.0.0 (127 : specific to localhost)
  - Class B, netid coded on 2 bytes (leading bits 10) :
    - Addresses from 128.0.0.0 to 191.255.0.0
  - Class C, netid coded on 3 bytes (leading bits 110) :
    - Addresses from 192.0.0.0 to 223.255.255.0
  - Class D (multicast), netid on 3 bytes (leading bits 1110)
    - Addresses from 224.0.0.0 to 239. 255.255.0

IPv4 address space

- Specific addresses
  - 127.0.0.1 : « localhost », loopback address
  - 0.0.0.0 : invalid address

- Reserved private IPv4 addresses :
  - Class A : netid 10, hostid 0.0.1 to 255.255.255
  - Class B : netid 172.16.0 to 172.31, hostid 0.0 to 255.255
  - Class C : netid 192.168 to 192.168, hostid0.0 to 255.255

TCP vs UDP

- TCP: connection oriented protocol above IP (phone)
  - Reliable protocol (with error handling)
    - Max packet size (MTU), segmentation mechanism
    - Guaranteed opening, routing and closing of the connection
  - Flow management mechanism to avoid saturating the network (Nagle algo)
- UDP: message oriented protocol above IP (postal mail)
  - No guarantee of routing or reception order
  - If "everything is fine » (LAN), we avoid the complexity of TCP
  - Useful for light applications, soft real time (multimedia)

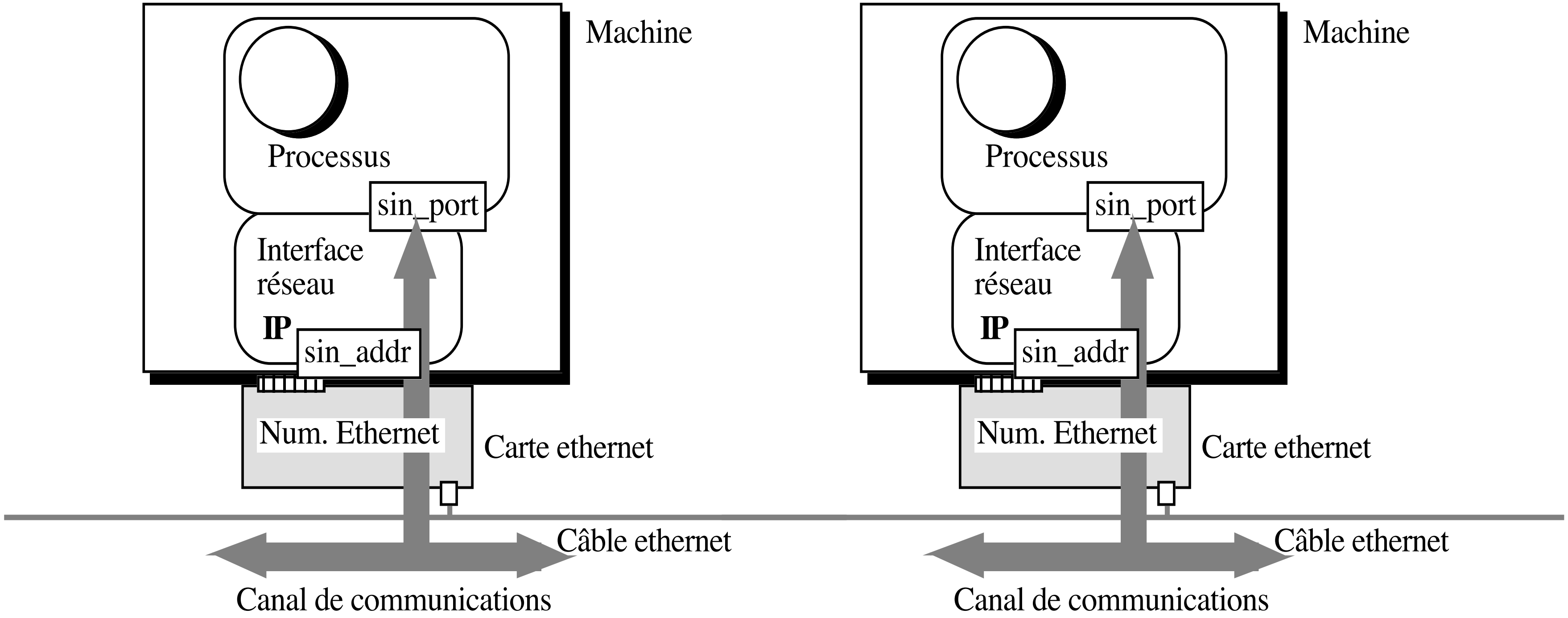
BSD Sockets
An API for IP, TCP and UDP

- No API defined by the IETF
- BSD Sockets: model and API used in Unix
  - Adapted for other platforms, including Windows
- Inspired by the Unix resource model:
  - Every resource is a file (same for sockets)
  - Connection oriented sockets follow producer / consumer semantics (pipe) and are used with open / read / write / close traditional file operations
  - Message oriented sockets are used with open / send / recv / close operations (still close to file operations)

What is a socket ?

- Socket
  - An IP address and a port number
  - When created, comes with sending and receiving buffers
- Socket pair
  - Specify the two end points
    - TCP : same end points (connection)
    - UDP : specify receiver or sender end point for each call
  - 4-tuple: (client IP addr + port, server IP addr + port)

Overview



Machine
Processus
sin_port
Interface
réseau
IP
sin_addr
Num. Ethernet
Carte ethernet
Câble ethernet
Canal de communications
Machine
Processus
sin_port
Interface
réseau
IP
sin_addr
Num. Ethernet
Carte ethernet
Câble ethernet
Canal de communications

Connection-Oriented Protocol TCP on client side

- The caller (client calling a service of a server):
  - Create a socket and allocate buffers
  - Build the network address (IP address + port)
    - Server identified by its IP address (or its name) and its port
    - Use predefined IP addresses (INADDR_LOOPBACK)
    - Get IP address from name with gethostbyname (DNS)
    - No name directory for ports, only reserved ports (IPPORT_RESERVED)
  - Connect to the server (three-way handshake)
  - Read from or write to the socket
  - Close the socket

Connection-Oriented Protocol Canvas of a TCP client

1. Build the server address
2. Request to a directory for hosts (but not for ports)

   gethostbyname is blocking (request to a predefined name server)
3. Request a connection with server

```
sock = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr server_addr;
server_addr.sin_addr = gethostbyname(« www.enst.fr »);
server_addr.sin_port = server_port;
connect(sock, &server_addr, sizeof(struct sockaddr));
```

Connection-Oriented Protocol
Ping Pong TCP client

```
int main() {
  int sock = socket(AF_INET, SOCK_STREAM, 0);
  sockaddr_in addr = {.sin_family = AF_INET,
                      .sin_port = htons(8080),
                      .sin_addr.s_addr = INADDR_LOOPBACK};
  connect(sock, (struct sockaddr*)&addr, sizeof(addr));
  int msg;
  while(1) {
    msg = 1; // Ping
    write(sock, &msg, sizeof(msg));
    read(sock, &msg, sizeof(msg));
    printf("Received: %s\n", msg == 2 ? "Pong" : "?");
    sleep(1);
  }
}
```

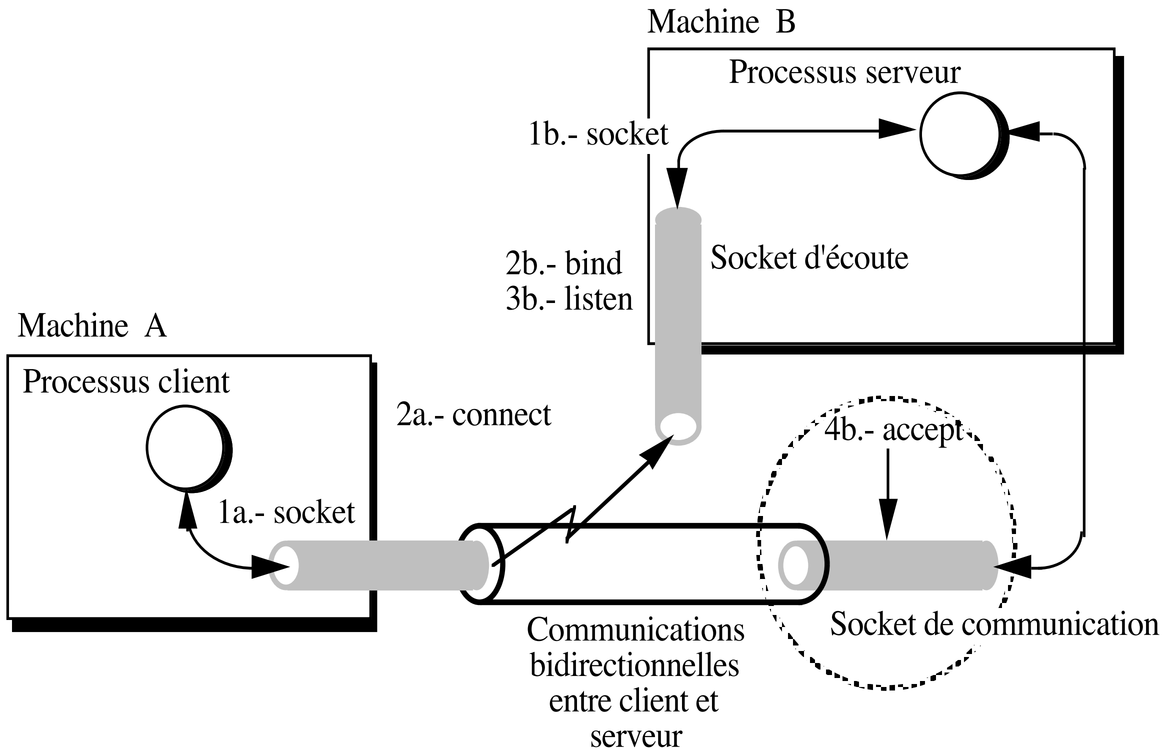
Connection-Oriented Protocol TCP on server side

- The callee (or server and its service):
  - Create a server socket
  - Associate address (IP address + port) to socket
  - Limit number of pending connections
  - Wait for incoming connections
  - For each incoming connection:
    - Accept the connection (a new socket is created);
    - Read from or write to the new socket
    - Close the new socket

Connection-Oriented Protocol Ping Pong TCP server

```c
int main() {
  int server_fd = socket(AF_INET, SOCK_STREAM, 0);
  struct sockaddr_in addr = {.sin_family = AF_INET,
                             .sin_port = htons(8080),
                             .sin_addr.s_addr = INADDR_ANY};
  bind(server_fd, (struct sockaddr *)&addr, sizeof(addr));
  listen(server_fd, 1);
  int client_fd = accept(server_fd, 0, 0);
  int msg;
  while (1) {
    read(client_fd, &msg, sizeof(msg));
    printf("Received: %s\n", msg == 1 ? "Ping" : "?");
    msg = 2; // Pong
    write(client_fd, &msg, sizeof(msg));
  }
}
```

Connection-Oriented Protocol TCP Sequential Management



Connection-Oriented Protocol TCP on a Multi-Threaded Server

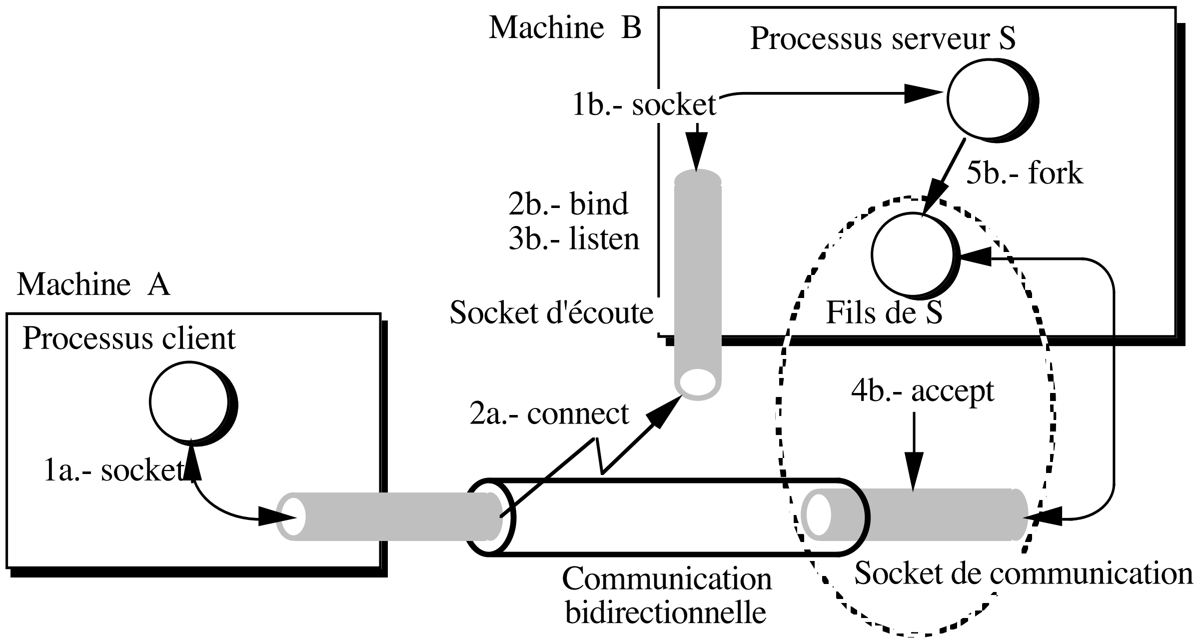
- Create a server socket (for incoming connection)
- Wait for a connection request (from a client)
- Create a new socket to handle client connection
- Concurrent management
  - 2 sockets (server socket + new socket)
  - 2 threads (or processes) => Use of patterns
    - Leader / Followers (leader accepts & delegates to followers)
    - Half/Sync – Half/Async (accept, connect, read, write block)
    - Executor service

Connection-Oriented Protocol Ping Pong Multi-Threaded Server

```
void* handle(void* fd) {
    int cfd = *(int*)fd;
    while (1) {
        int msg;
        read(cfd, &msg, sizeof(msg));
        write(cfd, &(int){2}, sizeof(int));
    }
}
int main() {
    int sfd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr = {AF_INET, htons(8080), INADDR_ANY};
    bind(sfd, (struct sockaddr*)&addr, sizeof(addr));
    listen(sfd, 5);
    while (1) {
        int* cfd = malloc(sizeof(int));
        *cfd = accept(sfd, 0, 0);
        pthread_t t;
        pthread_create(&t, 0, handle, cfd);
    }
}
```

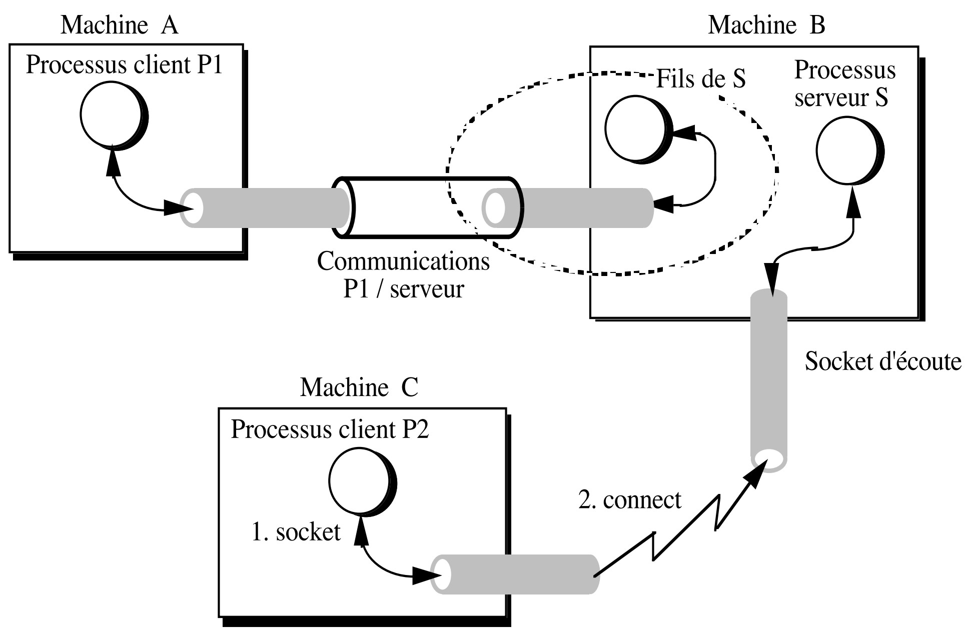
Connection-Oriented Protocol TCP Concurrent Management 1

- Communication with a first client

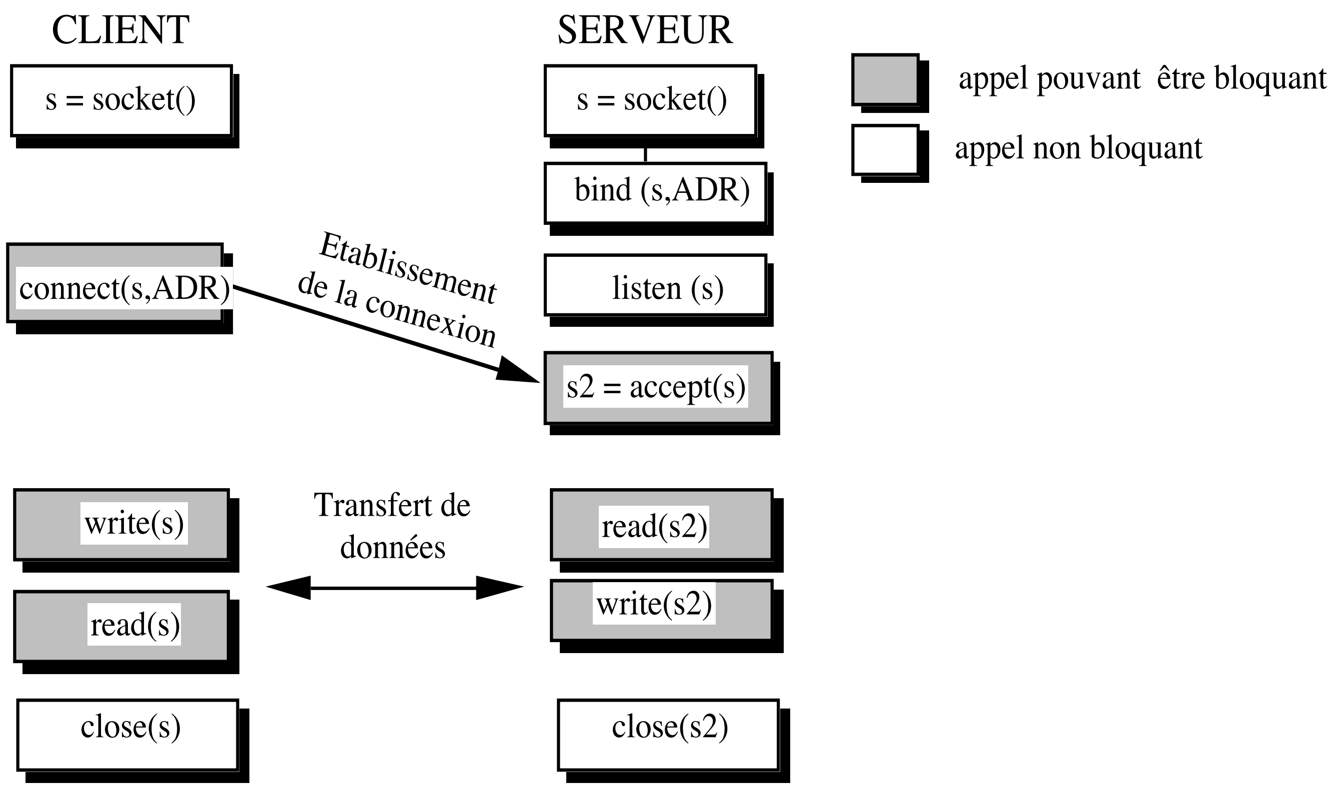


Connection-Oriented Protocol TCP Concurrent Management 2

- Communication with a second client



TCP
Blocking operations



C API TCP socket on server and client sides

- socket() = socket(domain, type, protocol)
  - Create a socket: index from open file table
  - domain = AF_INET or PF_INET
  - type = SOCK_STREAM (TCP), protocol = 0
- bind(sock, &server_addr, server_addr_len);
  - Bind socket to one of the host addresses & ports:
  - sock socket id returned by socket()
  - server_addr structure including address and port
  - server_addr_len size of structure (sizeof)

C API TCP socket on server side

- On the server side
  - listen(server_fd, nb_clients)
    - Set maximum length for the queue of pending connections
  - accept(server_fd, &client_addr, & client_addr_len)
    - Extract the first connection request on the queue of pending connections and create a new socket client_fd with same properties of server_fd
    - client_addr is filled in with the address of the client, the format is determined by the domain in which the communication is occurring.

C API TCP socket on client side

- On the client side
  - connect(sock, &server_addr, server_addr_len)
    - Initiate a connection on a socket
    - Attempt to make a connection to another socket on the server side. The other end point is specified by server_addr, which includes an IP address and a port.

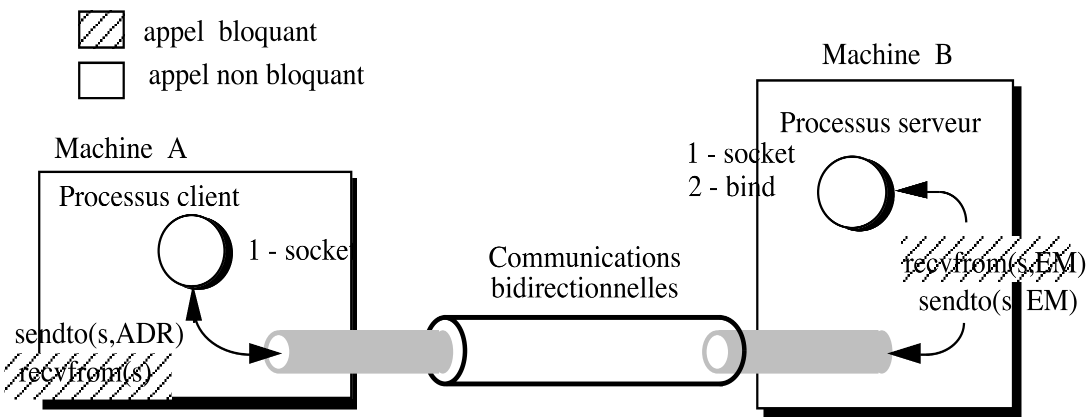
C API TCP socket on server and client sides

1. Standard functions for files:
   - read/write(sock, message, message_len)

2. Specific function (fine grain control):
   - send/recv(sock, message, message_len, option)
     - Example of option : MSG_PEEK
     - Peeks at an incoming message. The data is treated as unread and the next recv() or similar function shall still return this data

Message-Oriented Protocol UDP client or server side

- Client (caller) :
  - Create a socket ;
  - sendto or recvfrom on socket
- Server (callee) :
  - Create a socket ;
  - Bind socket to an address (IP address + port)
  - sendto or recvfrom on socket



Message-Oriented Protocol UDP client or server side

- Send a message through connectionless-mode socket (or connection-mode but address ignored)
  - sendto(sock, message, message_len, 0,
    &receiver_addr, receiver_addr_len)
- Receive a message from connectionless-mode socket (or connection-mode but useless)
  - recvfrom(sock, message, message_len, flags,
    &sender_addr, & sender_addr_len)
  - sender_addr allows the application to retrieve the source address of received data

Message-Oriented Protocol Ping Pong UDP server

```c
int main() {
  int sock = socket(AF_INET, SOCK_DGRAM, 0);
  struct sockaddr_in addr = {AF_INET, htons(8080), INADDR_ANY};
  bind(sock, (struct sockaddr*)&addr, sizeof(addr));
  while (1) {
    struct sockaddr_in client_addr;
    socklen_t len = sizeof(client_addr);
    int msg;
    recvfrom(sock, &msg, sizeof(msg), 0,
             (struct sockaddr*)&client_addr, &len);
    sendto(sock, &(int){2}, sizeof(int), 0,
              (struct sockaddr*)&client_addr, len);
  }
}
```

Message-Oriented Protocol
Ping Pong UDP client

```
int main() {
  int sock = socket(AF_INET, SOCK_DGRAM, 0);
  struct sockaddr_in addr =
    {AF_INET, htons(8080), INADDR_LOOPBACK};
  while (1) {
    sendto(sock, &(int){1}, sizeof(int), 0,
           (struct sockaddr*)&addr, sizeof(addr));
    int reply;
    recvfrom(sock, &reply, sizeof(reply), 0,
             NULL, NULL);
    sleep(1);
  }
}
```

TCP and UDP multiplexing overview

- Monitors multiple file descriptors (sockets) simultaneously to detect when they become "ready" for I/O operations (read/write/error), avoiding busy-waiting.
- **Synchronous I/O Multiplexing**
  - Checks sockets in user-space (no kernel callbacks)
  - Returns when any socket is ready or timeout occurs
- **Advantages:**
  - Dedicated threads per socket inefficiently use resources
  - Single-threaded concurrency (no threads/processes needed)
  - Portable (works on all POSIX systems)
  - Efficient for small-scale socket monitoring
  - O(n) time complexity (linearly scans all descriptors)

TCP and UDP multiplexing *select()*

```
int select(int nfds,                    // Highest fd
           fd_set *readfds,             // Check readability
           fd_set *writefds,            // Check writability
           fd_set *exceptfds,           // Check exceptions
           struct timeval *timeout);    // Max wait time (NULL = forever)
```

- **File Descriptor Sets:**
  - readfds: Sockets with incoming data (avoid read() blocking)
  - writefds: Sockets ready for sending (avoid write() blocking)
  - exceptfds: Sockets with errors (e.g., TCP out-of-band data)
- **Events:**
  - *accept* is considered as a *read* operation
  - *connect* is considered as a *write* operation

TCP and UDP multiplexing bit sets or masks

- Bitsets is a simple set data structure
  - n is in the bitset s if (s && 2^n) is true
- FD_CLR(fd, &fdset)
  - Clear the bit for the file fd in the file set fdset.
- FD_ISSET(fd, &fdset)
  - Return a non-zero value if the bit for the file fd is set in the file set pointed to by fdset, and 0 otherwise.
- FD_SET(fd, &fdset)
  - Ses the bit for the file fd in the file set fdset.
- FD_ZERO(&fdset)
  - Initialise the file set fdset to have zero bits for all files.

TCP and UDP multiplexing

Multiple Ping Pong Sequential Server

```c
int s = socket(…);
struct sockaddr_in a = {…};
bind(s, …);
listen(s, 5);
fd_set fds;
int max = s, c[16] = {0};
while (1) {
  FD_ZERO(&fds);
  FD_SET(s, &fds);
  for (int i = 0; i < 16; i++)
    if (c[i] > 0)
      FD_SET(c[i], &fds);
  select(max + 1, &fds, 0, 0, 0);
```

```c
  if (FD_ISSET(s, &fds)) {
    for (int i = 0; i < 16; i++)
      if (!c[i]) {
        c[i] = accept(s, 0, 0);
        if (c[i] > max) max = c[i];
        break;
      }
  }
  for (int i = 0; i < 16; i++)
    if (c[i] &&
      FD_ISSET(c[i], &fds)) {
      int m;
      if (read(c[i], &m, 4) <= 0)
        c[i] = 0;
      else
        write(c[i], &(int){2}, 4);
    }
}
```

TCP and UDP
Other Utilities

- Retrieve information about hosts (blocking read operations)
  - gethostbyaddr(struct sockaddr *HostAddr, int HostAdddrLen, int Type);
  - gethostbyname(char *HostName),
  - gethostent()
- Retrieve local information about ports (tcp/udp, id, name)
  - getservbyport(int Port, char *Proto)
  - getservbyname(char *Nom, char *Proto)
- Retrieve information about address or port of the specified socket
  - getsockname(Socket, &sa, &len)
  - getpeername(Socket, &sa, &len)
- Shutdown socket (all or part)
  - shutdown(Socket, Direction)
  - close(Socket)

TCP and UDP
Summary on Sockets

- Powerfull API:
  - Multicast, asynchronous behaviour (O_NONBLOCK)…
- … but requires additional tools …
  - Executor services, Design Patterns …
- … data conversions …
  - htons,
- … and a lot of programming …

Middleware vs Sockets Programming

- Middleware
  - Abstraction layer for distributed systems communication
- Pros
  - Faster development
  - Built-in scaling, fault tolerance
  - Cross-platform compatibility
- Semantics:
  - Akka (actor-based messaging)
  - MQTT (pub/sub for IoT)
  - CORBA (distributed objects)

- Sockets Programming
  - Low-level network communication
- Pros:
  - Maximum performance & control
  - No middleware dependencies
  - Ideal for custom protocols
- When to Use
  - Middleware: Complex systems, interoperability
  - Sockets: Latency-sensitive applications