

# Modèle Client-Serveur MQTT

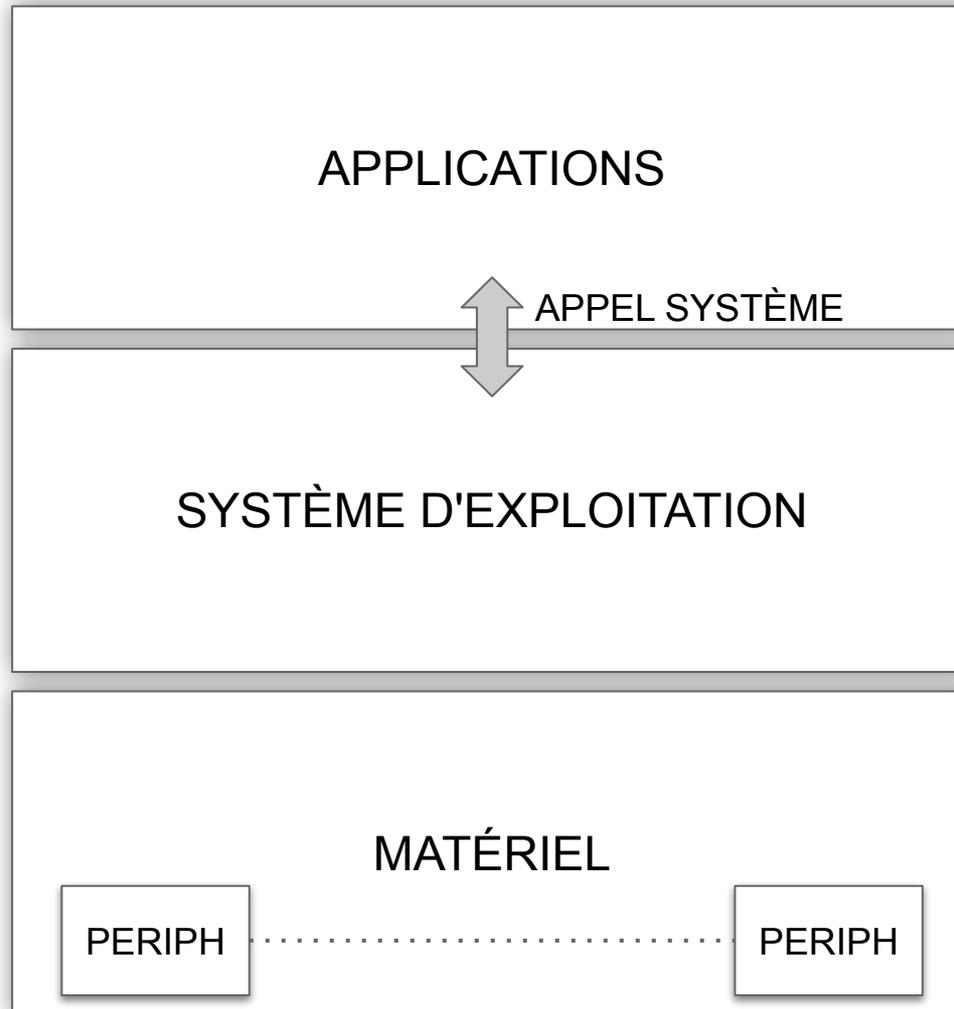
Module IOC — MU4IN109

TCP/IP Eleftherios Kosmas

<https://www.csd.uoc.gr/~hy556/material/tutorials/cs556-3rd-tutorial.pdf>

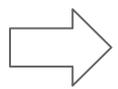
(extrait du tutoriel de 100 slides...)

# 3 couches



# Appels Système Fondamentaux (user)

Dans UNIX, « tout est fichier »



Un périphérique est un fichier

```
int fd; char * pathname ; int flags;  
char buffer[100]; int len, actuel_len;
```

- `fd = open( pathname, flags );`
- `actuel_len = read( fd, buffer, len );`
- `actuel_len = write( fd, buffer, len );`
- `close( fd);`

longueur max

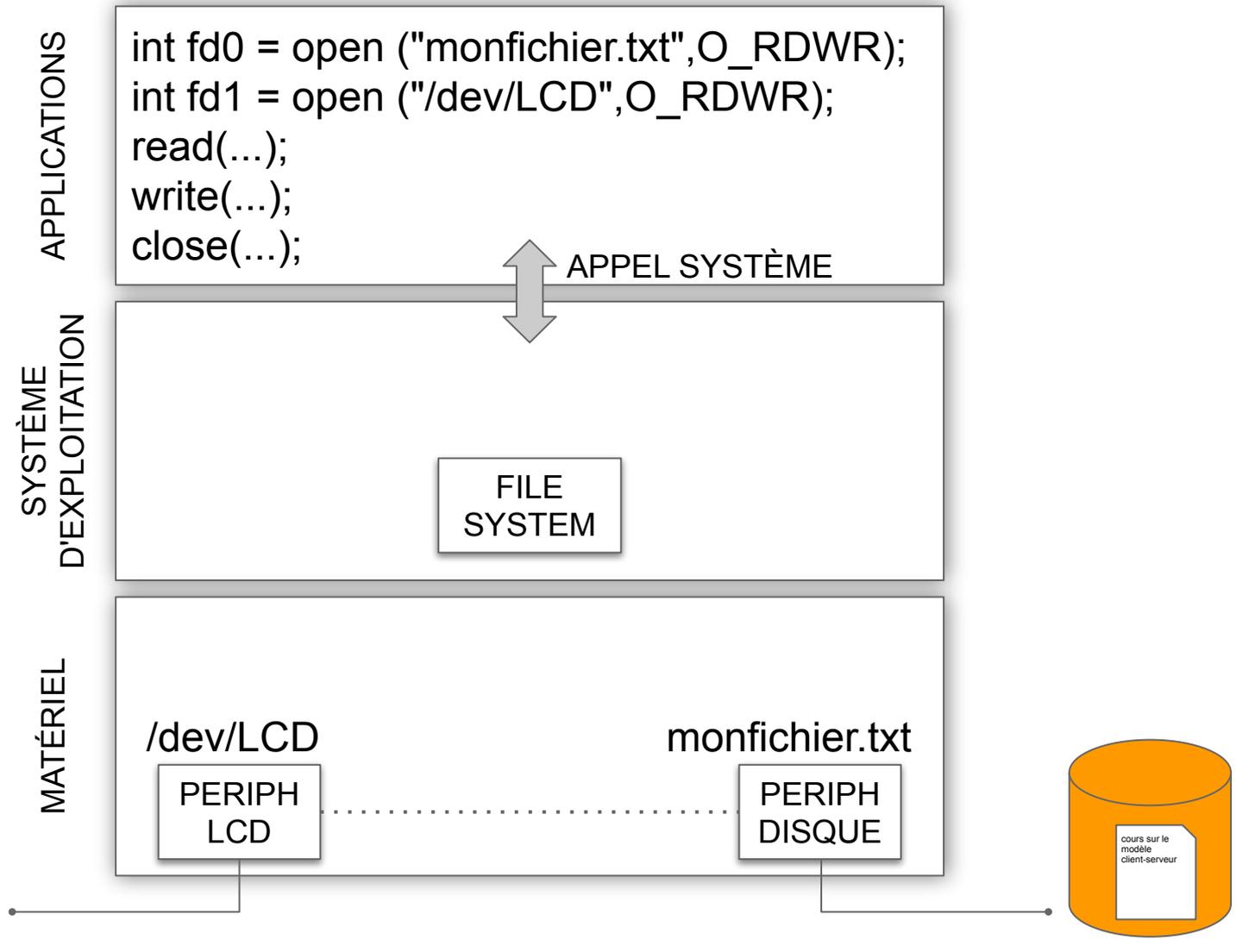
len

len

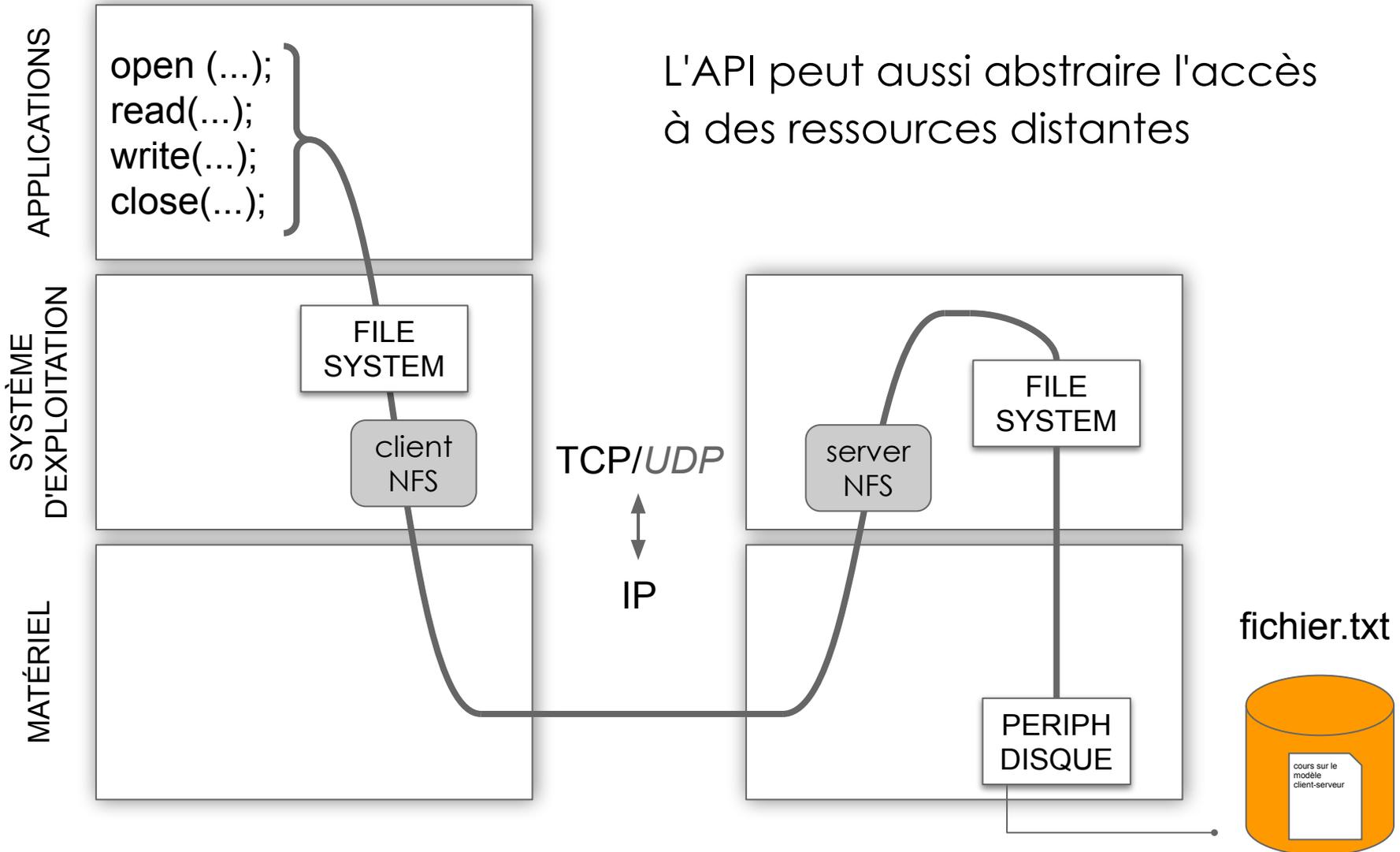
longueur exacte

Les communications réseau ont presque la même API,  
mais il y a en plus des fonctions pour établir et gérer la connexion

# Accès Fichier local

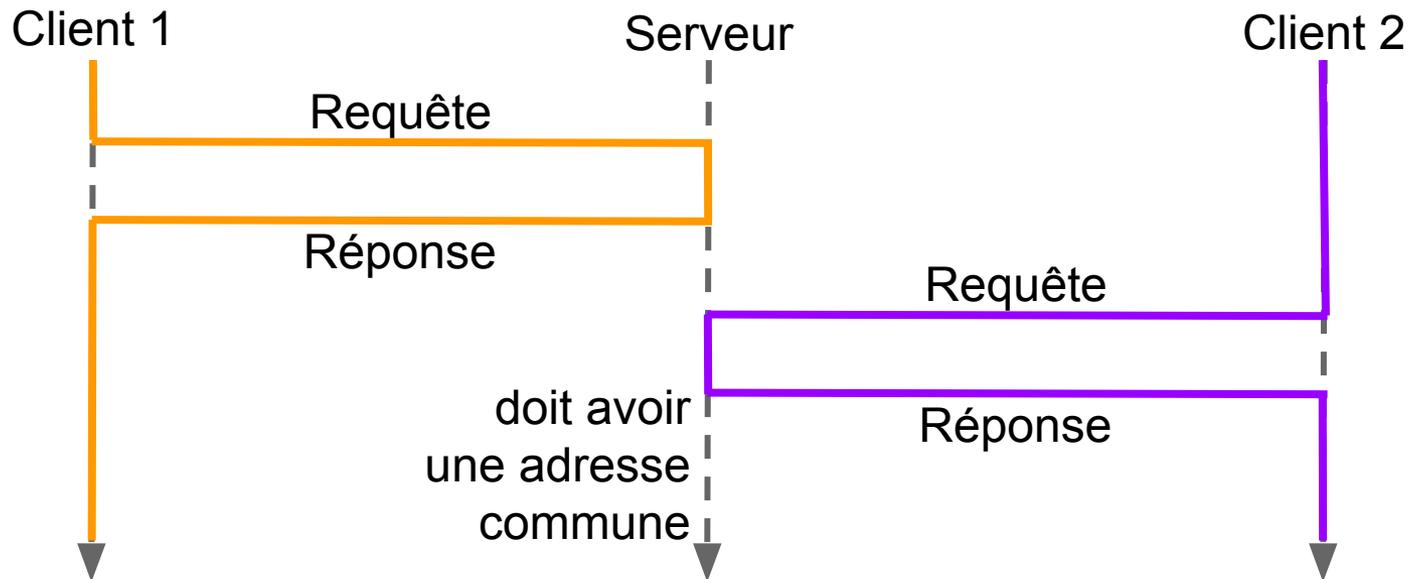
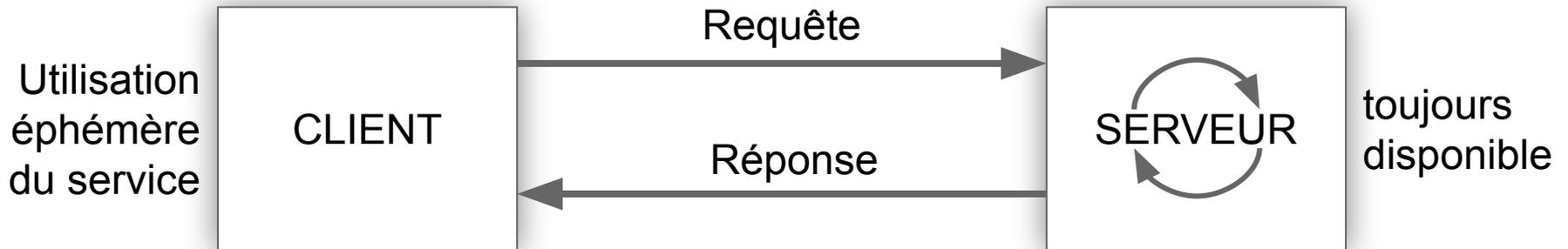


# Accès Fichier NFS



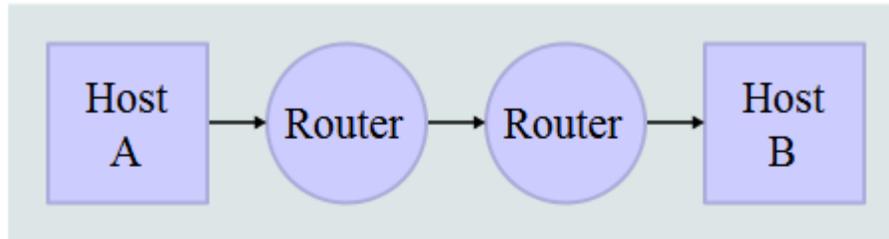
# SERVICES

Un serveur **rend** un service  
Un client **demande** un service



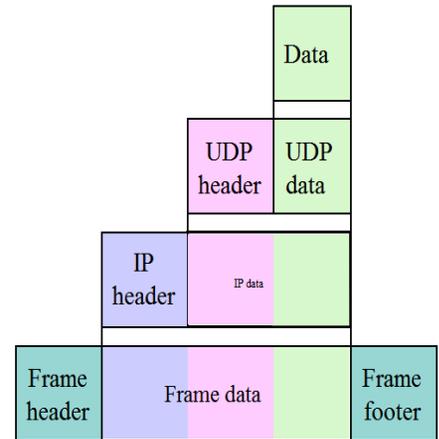
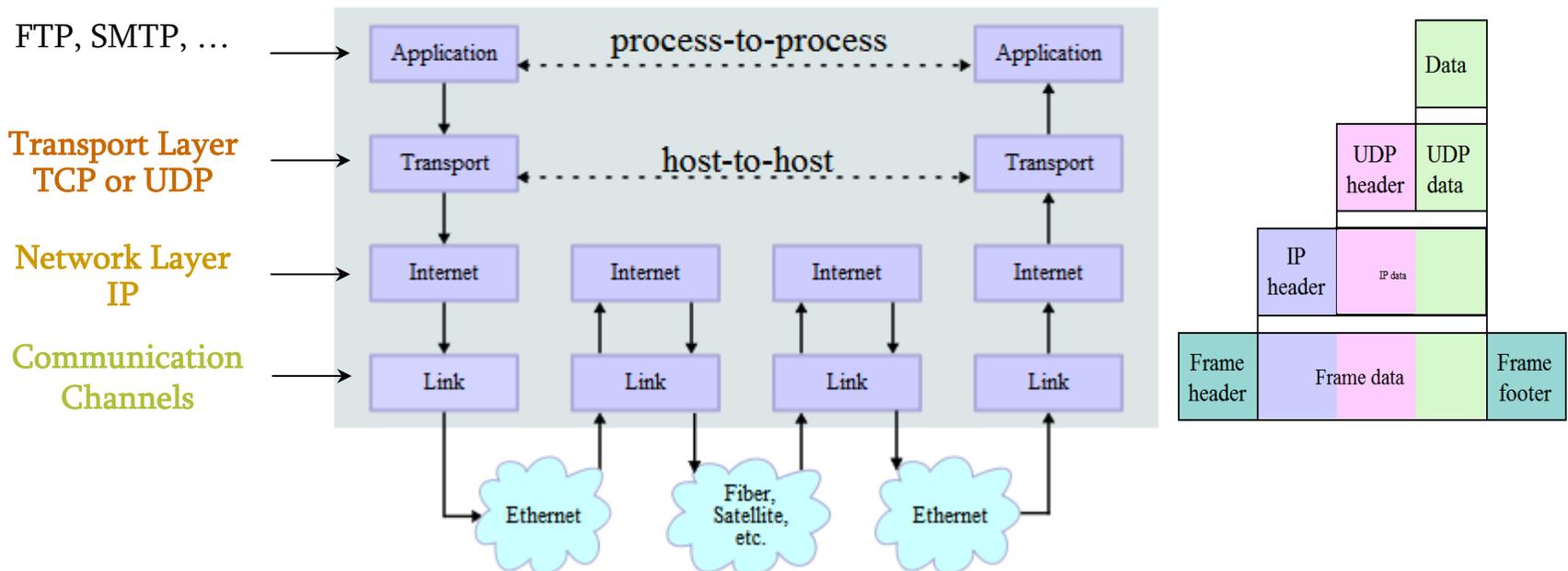
# TCP/IP

## Network Topology



\*

## Data Flow



\* image is taken from "[http://en.wikipedia.org/wiki/TCP/IP\\_model](http://en.wikipedia.org/wiki/TCP/IP_model)"

# Internet Protocol (IP)

- provides a **datagram** service
  - packets are handled and delivered independently
- **best-effort** protocol
  - may lose, reorder or duplicate packets
  
- each packet must contain an **IP address** of its destination

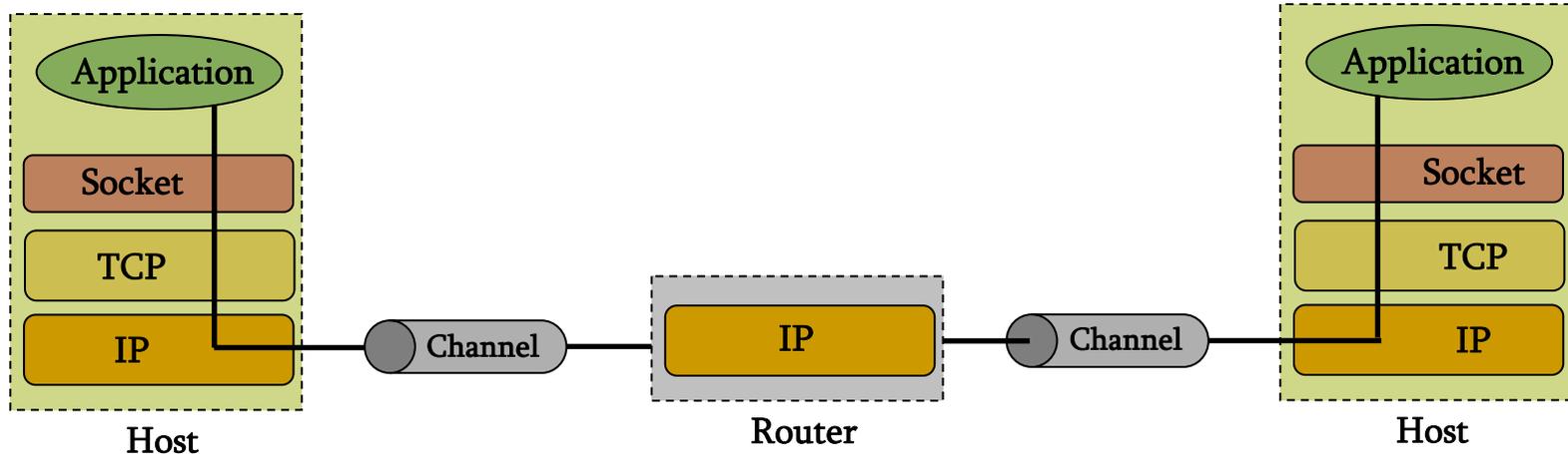


# TCP vs UDP

- Both use **port numbers**
  - application-specific construct serving as a communication endpoint
  - 16-bit unsigned integer, thus ranging from 0 to 65535
  - ☞ to provide **end-to-end** transport
- UDP: User Datagram Protocol
  - no acknowledgements
  - no retransmissions
  - out of order, duplicates possible
  - connectionless, i.e., app indicates destination for each packet
- TCP: Transmission Control Protocol
  - reliable **byte-stream channel** (in order, all arrive, no duplicates)
    - similar to file I/O
  - flow control
  - connection-oriented
  - bidirectional

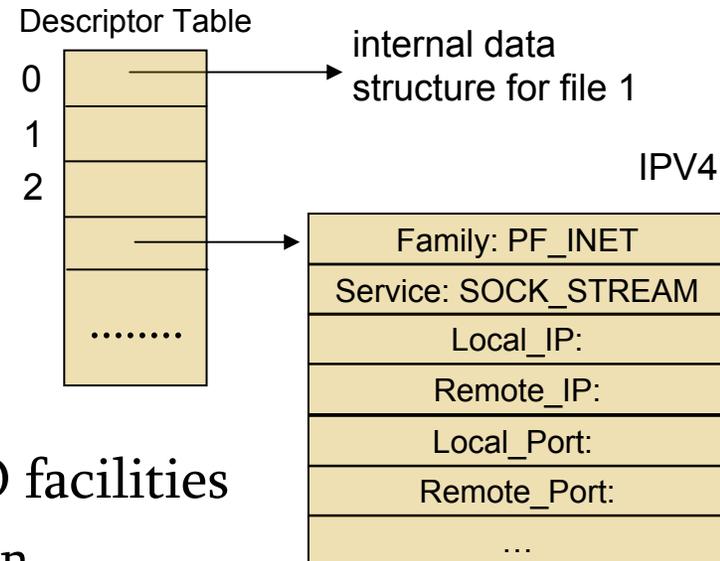
# Berkley Sockets

- Universally known as **Sockets**
- It is an abstraction through which an application may send and receive data
- Provide **generic access** to interprocess communication services
  - e.g. IPX/SPX, Appletalk, TCP/IP
- Standard API for networking

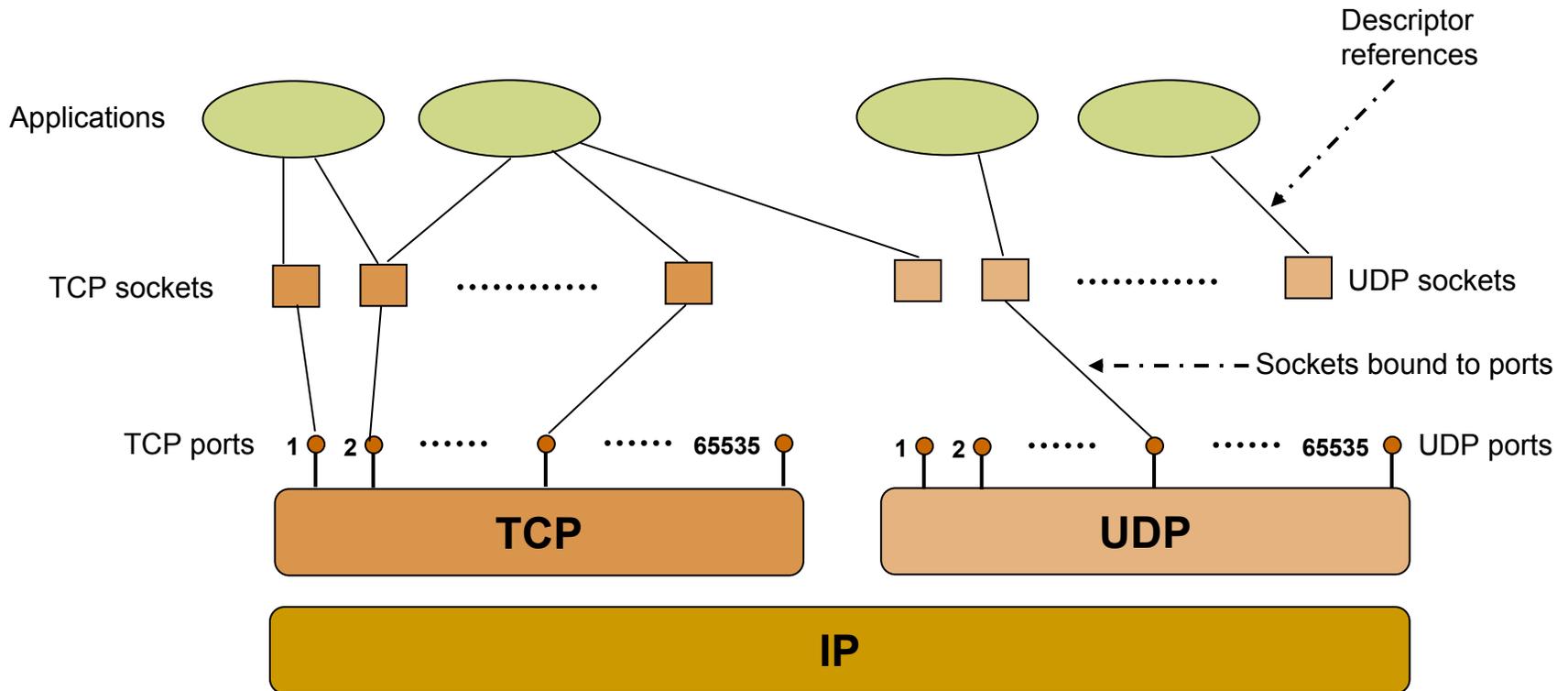


# Sockets

- Uniquely identified by
  - an internet address
  - an end-to-end protocol (e.g. TCP or UDP)
  - a port number
- Two types of sockets
  - **Stream** sockets (e.g. uses TCP)
    - provide reliable byte-stream service
  - **Datagram** sockets (e.g. uses UDP)
    - provide best-effort datagram service
    - messages up to 65.500 bytes
- Socket extend the conventional UNIX I/O facilities
  - file descriptors for network communication
  - extended the read and write system calls



# Sockets



L'usage des numéros de ports est standardisés :

- 80 pour les serveurs HTTP
- 22 pour les serveurs SSH

# Client-Server communication

## ■ Server

- ❑ passively waits for and responds to clients
- ❑ **passive** socket (écoute seule)

Nous allons voir qu'un Server va créer des sockets pour les transmissions qu'il accepte.

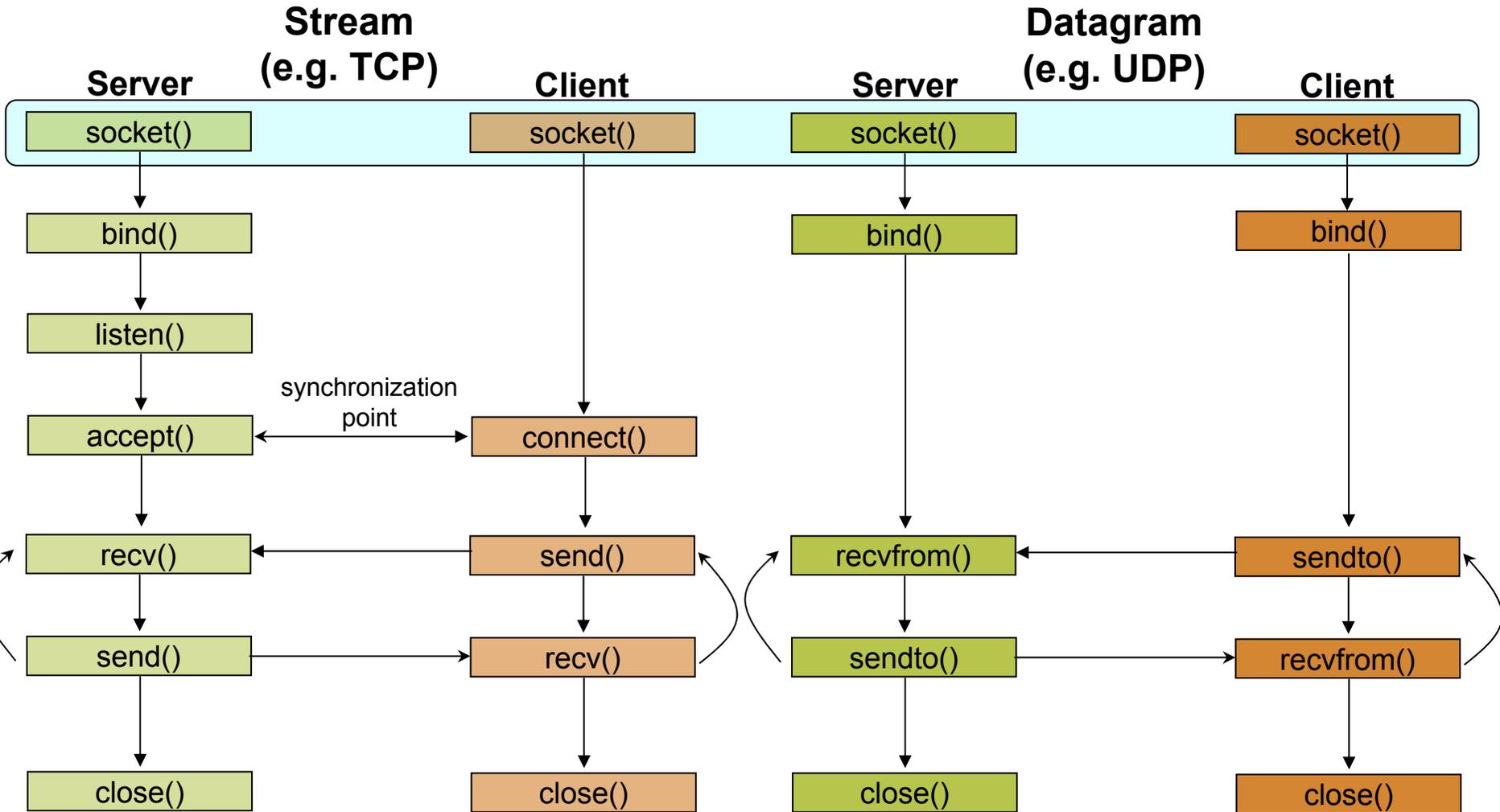
## ■ Client

- ❑ initiates the communication
- ❑ must know the address and the port of the server
- ❑ **active** socket (utilisé pour la transmission)

# Sockets - Procedures

| <b>Primitive</b> | <b>Meaning</b>                                  |
|------------------|---|
| Socket           | Create a new communication endpoint             |
| Bind             | Attach a local address to a socket              |
| Listen           | Announce willingness to accept connections      |
| Accept           | Block caller until a connection request arrives |
| Connect          | Actively attempt to establish a connection      |
| Send             | Send some data over the connection              |
| Receive          | Receive some data over the connection           |
| Close            | Release the connection                          |

# Client - Server Communication - Unix



# Socket creation in C: `socket()`

- `int sockid = socket(family, type, protocol);`
  - `sockid`: socket descriptor, an integer (like a file descriptor)
  - `family`: integer, communication domain, e.g.,
    - `PF_INET`, IPv4 protocols, Internet addresses (typically used)
    - `PF_UNIX`, Local communication, File addresses
  - `type`: communication type
    - `SOCK_STREAM` - reliable, 2-way, connection-based service
    - `SOCK_DGRAM` - unreliable, connectionless, messages of maximum length
  - `protocol`: specifies protocol
    - `IPPROTO_TCP` `IPPROTO_UDP` (il y en a d'autres possibles)
    - usually set to 0 (i.e., use default protocol)
  - upon failure returns -1
- ☞ NOTE: socket call does not specify where data will be coming from, nor where it will be going to – it just creates the interface!

# Assign address to socket: `bind()`

- associates and reserves a port for use by the socket
- `int status = bind(sockid, &addrport, size);`
  - `sockid`: integer, socket descriptor
  - `addrport`: struct `sockaddr`, the (IP) address and port of the machine
    - for TCP/IP server, internet address is usually set to `INADDR_ANY`, i.e., chooses any incoming interface (dans le cas où ~~server~~ ~~usieurs~~ ~~interface~~, ~~un~~ ~~usieurs~~ ~~adresses~~)
  - `size`: the size (in bytes) of the `addrport` structure
  - `status`: upon failure -1 is returned

# bind() - Example with TCP

```
int sockid;
struct sockaddr_in addrport;
sockid = socket(PF_INET, SOCK_STREAM, 0);

addrport.sin_family = AF_INET;
addrport.sin_port = htons(5100);
addrport.sin_addr.s_addr = htonl(INADDR_ANY);
if(bind(sockid, (struct sockaddr *) &addrport, sizeof(addrport)) != -1) {
    ...}
}
```

htons() : Host TO Network Short

→ pour convertir les nombres dans l'endianess utilisé par le réseau

htonl() : Host TO Network Long

# Await incoming connections : `listen()`

- Instructs TCP protocol implementation to listen for connections
- `int status = listen(sockid, queueLimit);`
  - `sockid`: integer, socket descriptor
  - `queueLen`: integer, # of active participants that can “wait” for a connection
  - `status`: 0 if listening, -1 if error
- `listen()` is **non-blocking**: returns immediately
  - Si le nombre de clients excèdent `queueLimit`, ils sont informés du refut de connexion
- The listening socket (`sockid`)
  - **is never used for sending and receiving**
  - is used by the server only as a way to get new sockets

# Establish Connection: connect ()

- The client establishes a connection with the server by calling connect ()

- `int status = connect(sockid, &foreignAddr, addrlen);`
  - `sockid`: integer, socket to be used in connection
  - `foreignAddr`: struct sockaddr: address of the passive participant
  - `addrlen`: integer, sizeof(name)
  - `status`: 0 if successful connect, -1 otherwise
- connect () is **blocking**

# Incoming Connection: `accept ()`

- The server gets a socket for an incoming client connection by calling `accept ()`
- ```
int s = accept(sockid, &clientAddr, &addrLen);
```

  - **s**: integer, the new socket (used for data-transfer)
  - **sockid**: integer, the orig. socket (being listened on)
  - **clientAddr**: struct `sockaddr`, address of the active participant
    - filled in upon return
  - **addrLen**: `sizeof(clientAddr)`: value/result parameter
    - must be set appropriately before call
    - adjusted upon return
- `accept ()`
  - is **blocking**: waits for connection before returning
  - dequeues the next connection on the queue for socket (`sockid`)

# Exchanging data with stream socket

- `int count = send(sockid, msg, msgLen, flags);`
  - `msg`: `const void[]`, message to be transmitted
  - `msgLen`: integer, length of message (in bytes) to transmit
  - `flags`: integer, special options, usually just 0
  - `count`: # bytes transmitted (-1 if error)
- `int count = recv(sockid, recvBuf, bufLen, flags);`
  - `recvBuf`: `void[]`, stores received bytes
  - `bufLen`: # bytes received
  - `flags`: integer, special options, usually just 0
  - `count`: # bytes received (-1 if error)
- Calls are **blocking**
  - returns only after data is sent / received

# Socket close in C: `close()`

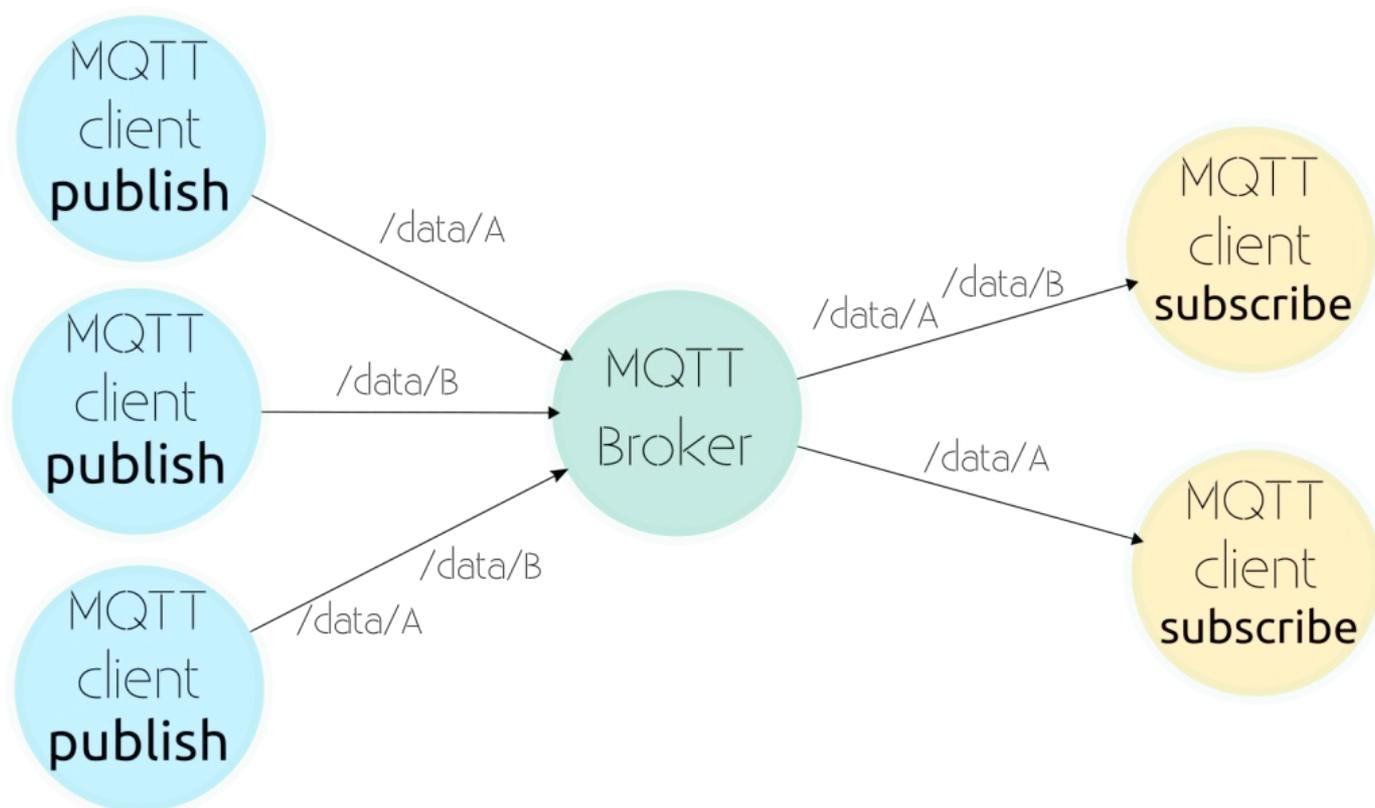
- When finished using a socket, the socket should be closed
- `status = close(sockid);`
  - `sockid`: the file descriptor (socket being closed)
  - `status`: 0 if successful, -1 if error
- Closing a socket
  - closes a connection (for stream socket)
  - frees up the port used by the socket

MQTT

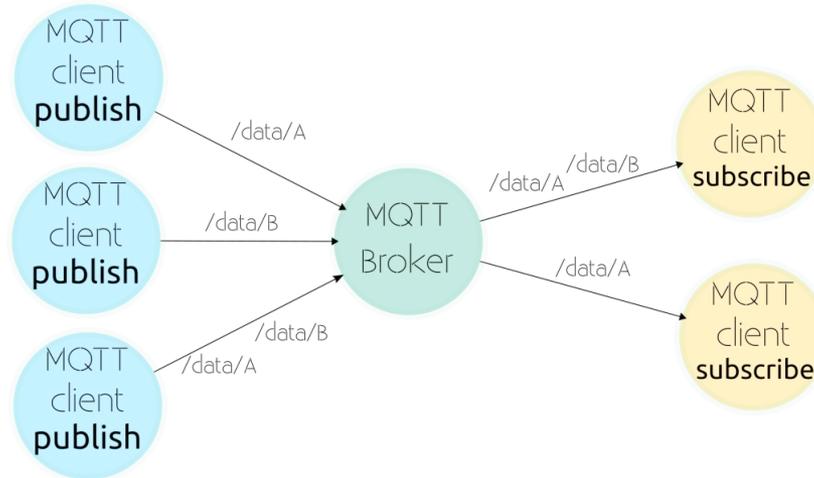
# MQTT : Caractéristiques générales

MQTT : Message Queuing Telemetry Transport

- modèle publish (topic-value) /subscribe (topic)
- communication décentralisée et asynchrone  
les client n'ont pas besoin de se connaître

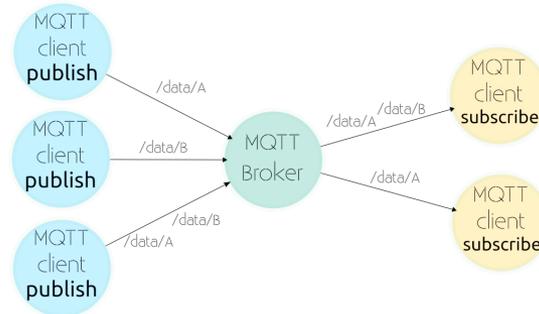


# Introduction MQTT



- Message Queuing Telemetry Transport
- Protocole de messagerie basé sur TCP-IP.
- Développé par IBM en 1999.
- Protocole léger de messagerie machines to machines
- Communication asynchrone → Pas besoin de connexion permanente.
- Transmission de données très faible bande passante.
- Différents niveaux de QoS pour garantir la livraison des messages.
- Adapté aux réseaux sans fil.
- Faible consommation énergétique.
- API Python, C, C++, bash, java, etc.

# Glossaire MQTT



- Broker Distribue les informations aux clients intéressés
- Client Connecté au Broker pour envoyer ou recevoir des informations.
- Topic Nom du message. Les clients publient, ou souscrivent à un Topic.
- Publish Envoi d'informations par un client au Broker qui les redistribue aux clients abonnés au Topic
- Subscribe Abonnement à un Topic pour recevoir les messages publiés  
Désabonnement possible
- QoS Qualité de service  
On peut spécifier une qualité de service au Broker avec une valeur entre 0 et 2.
  - 0 : au plus une seule fois sans qu'un accusé de réception
  - 1 : au moins une fois, message envoyé plusieurs fois jusqu'à la réception de l'accusé de réception
  - 2 : spécifie exactement une fois, Clients expéditeurs et destinataires ont la garantie d'une seule copie du message

# MQTT : commandes

Le shell propose des commandes pour faire communiquer les clients

- `mosquitto` démarre le broker
- `mosquitto_pub` client publisher
- `mosquitto_sub` client subscriber

Par exemple dans 3 terminaux de la même machine

- `mosquitto`
- `mosquitto_pub -h localhost -t "hello" -m "salut Franck"`
- `mosquitto_sub -h localhost -t "hello"`

Pour des machines clients distantes du broker, il faut ajouter l'adresse IP

- `mosquitto -c mosquitto.conf`
- `mosquitto_pub -h IP -p port -t "hello" -m "salut Franck"`
- `mosquitto_sub -h IP -p port localhost -t "hello"`

Par défaut le port d'écoute est 1883 mais on peut le changer dans `mosquitto.conf` avec la ligne `listener port_personnalisé`

# MQTT : API Python

Python propose la bibliothèque paho-mqtt (paho existe aussi en C)

Si ce n'est pas installé : `pip install paho-mqtt`

```
import paho.mqtt.client as mqtt
def on_connect(client, userdata, flags, rc): # Callback connexion broker
    print("Connecté " + str(rc))
    client.subscribe("hello")
def on_message(client, userdata, msg): # Callback subscriber
    print("Message reçu : {msg.topic} {str(msg.payload)}")

client = mqtt.Client() # Création d'une instance client
client.on_connect = on_connect # Assignment des callbacks
client.on_message = on_message
client.connect("adresse_ip_du_broker", 1883, 60) # Connexion au broker

client.publish("hello", "Salut Franck") # Publication d'un message
client.loop_forever() # pour attendre les messages
```

# Conclusion

- TCP fournit un canal de communication fiable entre 2 applications
- MQTT fournit un système de messagerie au-dessus de TCP qui découple les application (puisque l'on passe par un broker)

TME

Programmer une application de vote client-serveur  
directement avec TCP/IP  
puis avec MQTT