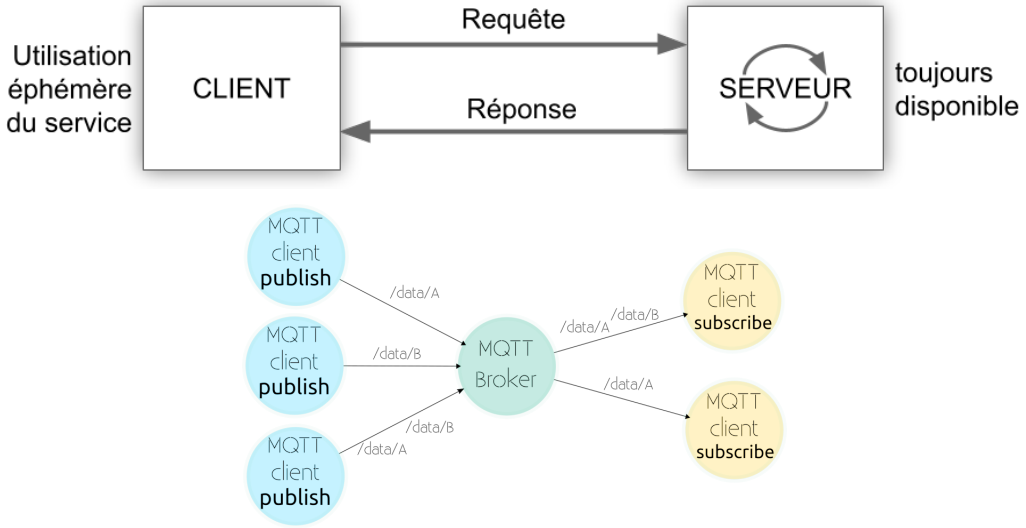
Modèle Client-Serveur - MQTT - cours 5 IOC

Ce document est une explication des slides présentés en cours



1. Page de titre

Le modèle client-serveur est essentiel pour la communication entre machines sur un réseau. L'objectif principal de la présentation est d'expliquer les bases de la communication en réseau dans un modèle client-serveur par l'utilisation des sockets et des protocoles TCP/IP et MQTT.

2. 3 couches

Une application ne communique pas directement avec le matériel, elle passe par le système d'exploitation. Le modèle est divisé en trois couches : applications, système d'exploitation et matériel. Les applications utilisent des appels système pour demander des services au système d'exploitation.

3. Appels système fondamentaux (user)

Les fichiers et périphériques sont accessibles via des appels système (`open`, `read`, `write`, `close`). Les communications réseau ont presque la même API (`read`, `write`, `close`) mais il y a des services spécifiques pour établir et gérer la connexion (`listen`, `accept`, `connect`,).

4. Accès fichier local

Exemple d'ouverture et de lecture d'un fichier local ou d'un périphérique (/dev/LCD). On a vu que les devices sont vus comme des fichiers.

5. Accès fichier NFS

Un fichier distant est accessible avec la même API que les fichiers locaux grâce à un système de fichiers réseau. Le système peut abstraire l'accès aux objets distants et conserver la même API. Sur la machine cliente, si on accède à un fichier dans une partition montée en nfs quand on ouvre un fichier et que l'on invoque la fonction read(), il y a des envois de requêtes vers le serveur nfs pour aller chercher les morceaux du fichier. Ces morceaux sont stockés dans le page-cache (le cache local des fichiers) comme pour un fichier local.

6. Modèle client-serveur

Un serveur fournit un service, les clients l'utilisent en envoyant des requêtes et recevant des réponses. Le serveur est **toujours disponible**, tandis que le client **utilise le service de manière temporaire**. Le serveur est identifié par 1 adresse IP et 1 numéro de port.

Les transparents suivants sont issus d'un tutoriel de 100 slides de Eleftherios Kosmas
https://www.csd.uoc.gr/~hy556/material/tutorials/cs556-3rd-tutorial.pdf

7. Architecture TCP/IP

Présentation des couches réseau : IP pour le routage, TCP/UDP pour le transport, et les protocoles applicatifs.

- en haut : 2 applications communiquent par le réseau, les communications passent des routeurs qui sont chargés de l'acheminement des requêtes et des réponses de manière transparente (c'est-à-dire complètement abstraite)
- À gauche : les requêtes et les réponses traversent des couches applicatives, chacune offre un service spécifique. Les paquets sont routés à travers l'internet grâce au protocole réseau (IP)
    - La couche application qui offre des services de haut niveau pour les applications utilisateurs. Chaque service vient avec un protocole :
        - HTTP → WEB
        - FTP → échange de fichiers
        - SMTP → mails
        - DNS → nommage des URL
        - SSH → connexion distante
        - MQTT → Communication entre objets connectés (cf. plus loin)
    - La couche transport assure la communication de bout en bout entre deux applications.
        - Protocoles : **TCP** (Transmission Control Protocol) fiable,
          UDP (User Datagram Protocol) sans connexion
    - La couche **réseau** est responsable de l'acheminement des paquets de données d'un hôte à un autre à travers des routeurs.
        - Protocole : **IP** (Internet Protocol)
    - La couche Liaison (**Link**) gère la communication entre les machines connectées sur un même réseau physique. Elle gère l'accès au réseau physique.
        - Protocoles : Ethernet, WiFi, PPP (Point-to-Point Protocol)
- À droite : le dessin exprime que les données des couches applicatives sont prises en charge par les couches inférieures en ajoutant en entête de l'information nécessaire pour le protocole courant. Les données sont possiblement découpées en morceaux (pas sur le dessin), chaque morceau est envoyé indépendamment et le récepteur est responsable de la reconstitution de la donnée d'origine.

8. Protocole IP

IP est un protocole de la couche réseau qui permet :

- L'adressage unique des machines sur un réseau (via une adresse IP).
- Les données sont encapsulées dans des paquets IP pour être transmises.

L'acheminement des paquets d'un point A à un point B via des routeurs.
Il fonctionne selon un principe best-effort delivery, ce qui signifie :

- Il envoie les paquets, mais ne garantit pas qu'ils arriveront tous.
- Il ne garantit ni l'ordre ni l'intégrité des paquets envoyés.
- Il ne retransmet pas automatiquement les paquets perdus
  C'est le travail des couches dessus (TCP), mais pas (UDP)

9. Comparaison TCP vs UDP

Ce slide compare TCP et UDP qui sont les deux principaux protocoles de transport

- Les deux utilisent des ports permettant de savoir où envoyer les requêtes.
- UDP est sans garantie de livraison, mais rapide
- TCP assure un transport fiable et ordonné, il peut être plus lent si le réseau physique est perturbé, très utilisé ou que le destinataire est très occupé et qu'il jette les paquets qu'il reçoit.

TCP est le plus utilisé en termes de connexions mais UDP transporte plus de données en volume pour chaque connexion à cause du streaming vidéo (mais YT utilise TCP je crois).

10. Sockets de Berkeley

Une socket est une interface standard entre une application et la pile réseau, permettant d'échanger des données avec une autre application. Sous UNIX, une socket est représentée par un descripteur de fichier.

Le schéma du slide montre :

- 2 Applications utilisant des sockets pour envoyer et recevoir des données.
- Les données sont transportées via TCP/IP, qui gère la communication entre les applications.
- Un routeur est placé entre les hôtes pour acheminer les paquets.

11. Caractéristiques des sockets

Un socket est défini par trois éléments principaux :

- Une adresse IP (qui identifie une machine sur le réseau).
- Un numéro de port (qui identifie un service ou une application spécifique sur cette machine).
- Un protocole de transport (TCP ou UDP)

Un port

- localement, c'est un point d'écoute pour les paquets venant du réseau.
- à distance, cela désigne le point par lequel on veut que sorte les données, parce qu'on sait que c'est sur ce point qu'elles seront écoutées.
- On écoute un port local, on envoie sur un port distant.

Un serveur HTTP sur une machine ayant l'adresse 192.168.1.10
et écoutant sur le port 80 avec TCP serait identifié par : (192.168.1.10, TCP, 80)

Il existe deux grands types de sockets, en fonction du protocole de transport utilisé :

1. Stream Sockets (SOCK_STREAM - TCP)
    - Sur TCP : Garantit la livraison des données et leur ordre.
    - Fiable et orienté connexion : Nécessite un établissement préalable de la connexion. comme pour un appel téléphonique
    - Fonctionne comme un flux de données, similaire à un fichier.
    - Ex : HTTP, FTP (transfert de fichiers), SSH (connexion sécurisée).
2. Datagram Sockets (SOCK_DGRAM - UDP)
    - Sur UDP : Plus rapide mais sans garantie de livraison ou d'ordre.
    - Sans connexion : Chaque message (datagramme) est indépendant. comme un envoi postal
    - Capacité à envoyer des messages jusqu'à 65 500 bytes.
    - Ex : DNS (résolution de noms), VoIP (appels en ligne), jeux en ligne.

Sur le dessin de droite, on voit que l'API des sockets étend l'API des fichiers et utilise les mêmes "file descriptors" pour l'identification côté user. On voit aussi que le socket doit connaître l'adresse IP local et qu'un numéro de port est attribué localement. Cette adresse IP locale et ce numéro de port local sont envoyés au serveur, pour qu'il puisse renvoyer une réponse. Ce numéro de port local est éphémère afin de permettre à une machine d'ouvrir plusieurs connexions simultanées vers un même serveur sans conflit.

12. Utilisation des sockets

Sur ce schéma, en haut les applications qui communiquent entre elles grâce aux sockets.

- Les sockets utilisent les ports pour recevoir les données. Certains ports sont standardisés (ex : 80 pour HTTP, 22 pour SSH).
- Une même application peut utiliser plusieurs sockets, évidemment, mais on voit aussi que deux applications peuvent partager le même socket (un père et son fils après fork(). Tous les threads d'une même application partagent les mêmes ports. La gestion du partage est à la charge de l'application (mutex, sémaphore, etc.)
- Il y a au plus 64k ports utilisables en TCP et 64k ports utilisables en UDP

13. Communication client-serveur

Le serveur écoute les connexions entrantes (passif), le client initie la connexion (actif).
Un serveur est persistant, il écoute un port pour savoir ce qu'il a à faire.
Un client va créer une socket (interface) localement pour échanger avec le serveur.
Quand le serveur accepte une connexion avec un client sur son port d'écoute, il peut créer un socket temporaire localement, qui durera le temps de la connexion et cessera d'être utilisé à la fin de la connexion.
Le nouveau socket est lié à la même adresse locale que le socket d'écoute, mais il est dédié à la communication avec ce client. Le socket d'écoute ne sert jamais à échanger des données, uniquement à accepter des connexions.

14. API Socket

Serveur :

- `socket()` → Crée un socket pour plus tard accepter des connexions.
- `bind()` → Associe le socket à une adresse IP et un port spécifique.
- `listen()` → Met le socket en écoute pour accepter des connexions entrantes.
- `accept()` → Accepte une connexion et crée un socket dédié pour l'échange..
- `send()` / `write()` → Envoie des données au client via le socket.
- `recv()` / `read()` → Reçoit des données envoyées par le client.
- `close()` → Ferme le socket et libère les ressources.

Client :

- `socket()` → Crée un socket pour plus tard établir une connexion avec un serveur.
- `connect()` → Se connecte au serveur sur son adresse IP et son port d'écoute.
- `send()` / `write()` → Envoie des données au serveur via le socket.
- `recv()` / `read()` → Reçoit des données envoyées par le serveur.
- `close()` → Ferme le socket une fois la communication terminée.

15. Communication client-serveur sous Unix

Sur ce schéma on voit la différence entre les communications TCP et UDP.
TCP propose un flux continu (STREAM) de données avec la garantie que les données sont reçues dans l'ordre d'émission, si vous envoyez send("bonjour"); puis send("le monde"); alors quand on reçoit avec un receive(), on peut recevoir "bonjour" puis plus tard "le monde" ou en une fois "bonjourle monde" ou n'importe quel autre découpage, selon la gestion des buffers du système et du réseau. TCP ne préserve pas la notion de message, seulement un flux d'octets.
Pour TCP : Le serveur

1. **socket**() pour créer le socket
2. **bind**() : associe le **socket** à une **adresse IP locale** et un **port**.
3. **listen**() : met le socket en écoute pour accepter des connexions entrantes
4. Ensuite il **accept**() les demandes venant de **connect**() d'un client.
   sort de cette étape s'il y a un **connect**() d'un client
   il y a création d'un socket temporaire pour l'échange
   le socket principal boucle en 4 avec une boucle while().

5. **recv**()/**read**() pour lire la requête envoyée par **send**()/**write**() du client. Les données peuvent arriver en plusieurs morceaux ou en une seule fois, selon la gestion des tampons
6. puis **send**()/**write**() une réponse qui sera **receive**()/**read**() par le client.
   (en général, mais pas obligatoire)
   Le socket temporaire peut rester ouvert pour plusieurs échanges avant d'être fermé par l'une des parties..
7. **close**() la connexion du socket temporaire.
   La fermeture est généralement demandée par le client, mais le serveur peut avoir un timeout d'inactivité.

Pour TCP : Le client

1. **socket**() pour créer le socket
2. Se **connect**() et il sort de cette étape s'il y a un **accept**() du serveur
3. **send**()/**write**() sa requête qui **receive**()/**read**() par le serveur.
4. **receive**()/**read**() la réponse attendue envoyé par **send**()/**write**() du serveur
   boucle éventuellement en 2.
5. **close**() la connexion et donc le socket

Il est recommandé de fermer les connexions inutilisées rapidement, car un serveur a un nombre limité de connexions simultanées. Laisser trop de connexions ouvertes peut entraîner un déni de service (DoS).

16. Création d'un socket en C

Fonction `socket()` pour créer une interface réseau et retourne un descripteur de fichier

17. Association d'une adresse avec `bind()`

Associe un socket à une adresse IP et un port spécifique.
Côté serveur : `bind()` permet d'associer le socket à un port d'écoute.
Côté client : ce n'est généralement pas nécessaire, car le système choisit un port local éphémère automatiquement.

18. Exemple d'utilisation de `bind()`

Code C qui illustre comment lier un socket à un port spécifique.

19. Mise en écoute avec `listen()`

Le serveur se met en attente de connexions entrantes avec une file d'attente limitée.
`listen()` transforme le socket en socket d'écoute.

20. Établissement d'une connexion avec `connect()`

`connect()` établit une connexion TCP avec le serveur distant., c'est une opération bloquante. L'appel se termine lorsque la connexion est établie ou échoue.

21. Acceptation des connexions entrantes avec `accept()`

Le serveur crée un nouveau socket pour chaque client connecté.
Le socket d'écoute continue d'accepter d'autres connexions.

22. Échange de données via socket en mode stream

`send()` et `recv()` permettent d'envoyer et recevoir des données via TCP.
Ces appels peuvent envoyer ou recevoir un nombre partiel d'octets ; il faut donc vérifier la valeur retournée, sachant que TCP transmet un flux d'octets, pas des messages.

23. Fermeture d'un socket en C

`close()` libère les ressources et termine la connexion et libère le port utilisé.

Ici, juste une introduction, suffisante pour le projet. Un cours complet est disponible ici : https://largo.lip6.fr/trac/sesi-peri/chrome/site/cours/Pietro_Manzoni_MQTT_v2.pdf
Le cours de Pietro Manzoni est beaucoup trop détaillé pour les besoins que nous aurons

25. MQTT

MQTT est un protocole léger de messagerie conçu pour les objets connectés et les communications M2M (Machine-to-Machine).

26. Introduction à MQTT

- MQTT s'appuie sur TCP pour le transport fiable des messages.
- MQTT permet une communication asynchrone via un broker : les clients ne se connaissent pas directement et n'ont pas besoin d'être connectés les uns aux autres.
  C'est le broker qui distribue les messages.
- Un client **publisher** (producteur) publie des données sous la forme (topic, valeur).
- Un client **subscriber** (consommateur) souscrit (subscribe) à un ou plusieurs topics pour recevoir les mises à jour.
- Le broker reçoit les messages et les distribue aux clients abonnés.
- Si aucun client n'est abonné au moment de la publication, le message est perdu, sauf s'il est marqué comme retain.

27. Glossaire MQTT

Explication des termes clés : broker, client, topic, publish, subscribe, QoS.
QoS 0 : envoi simple, sans garantie
QoS 1 : garanti au moins une fois (possibles doublons)
QoS 2 : garanti exactement une fois

28. MQTT commandes shell

On peut utiliser le shell pour envoyer échanger des messages entre clients
Ces commandes permettent de tester le protocole MQTT sans avoir à écrire un programme.
On démarre le broker, puis les clients peuvent communiquer

29. MQTT API Python

L'API MQTT est disponible en C, Python et dans de nombreux autres langages.
Ici, on illustre son utilisation en Python pour un client
Le broker doit déjà être lancé sur une machine accessible par le réseau.
Le client MQTT ne communique jamais directement avec un autre client, mais uniquement avec le broker.

- On importe la bibliothèque MQTT
- On crée un client MQTT.
- On définit éventuellement des fonctions de rappel (callbacks).
- On se connecte au broker.
- On publie ou on s'abonne à des topics.
- On lance la boucle d'attente des messages.

30. Conclusion et TME

TCP : canal fiable entre un client et un serveur
MQTT : échange de message entre clients

En TME :
Mise en œuvre d'une application de vote en mode client-serveur lors du TME.