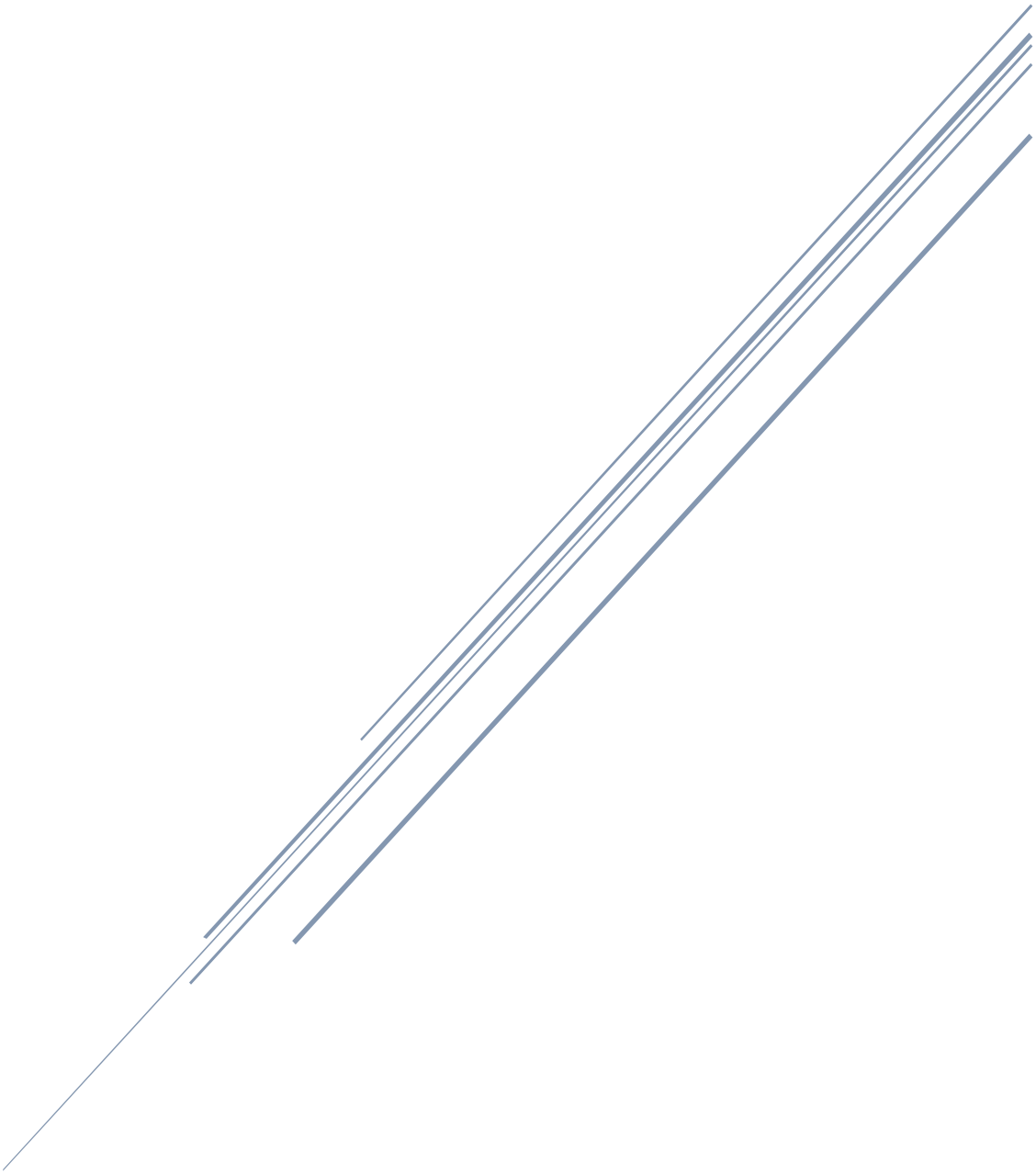
XARXES: PRÀCTICA 1

Programació d'aplicacions de Xarxa

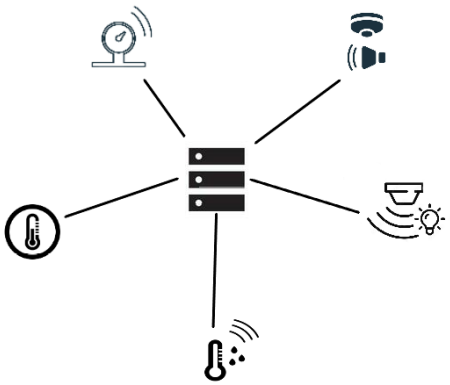


UdL EPS
Èric Bitriá Ribes 49755704B

Índex

1. Introducció

En la primera pràctica de programació d'aplicacions de xarxa, se sol·licita la implementació d'un model client-servidor per simular les comunicacions entre un conjunt de sensors i actuadors amb un servidor. Aquests dispositius estan classificats i identificats amb una sèrie de paràmetres, com sensors, ubicació, nom, adreça MAC etc. I poden estar distribuïts en diverses àrees d'un edifici. A continuació, es presenta un breu esquema del funcionament d'aquest sistema [1]:



[1] Representació de la comunicació del sistema de sensors amb el servidor

En aquesta implementació s'utilitzaran dos llenguatges de programació per la implementació del client i del servidor. Concretament, el client estarà implementat en *Python* versió 3.11.4 i el servidor en *C99*.

A tot llarg d'aquest document s'aniran mostrant les fases de desenvolupament, des del plantejament inicial: estructuració, diagrama d'estats, funcionalitats que ha de complir tant servidor com client, fins consideracions i decisions fetes a l'hora de fer la implementació en els dos llenguatges de programació.

Finalment, es presentaran unes conclusions i resum de tot el desenvolupament de la pràctica.

2. Plantejament

A l'hora de conèixer i entendre correctament quines funcionalitats ha de complir el sistema a desenvolupar es important crear un esquema visual que permeti identificar fàcilment cada punt i a més quines accions es poden fer en aquella situació. Per aquest motiu s'empra un diagrama d'estats mostrant a cada estat en quina situació es troba tant client com servidor.

Al tractar-se de dos dispositius amb funcionalitats, hem decidit de realitzar dos diagrames d'estats un per al client i l'altre pel servidor i en tot cas realitzar-ne un tercer per mostrar la interacció del model client-servidor.

2.1 Diagrama d'estats: Client

El client ha de realitzar una sèrie de tasques:

- Llegir Arxiu de Configuració.
- Iniciar Procés de subscripció amb el servidor.
- En cas d'haver d'estar subscrit amb el servidor, mantenir una comunicació periòdica.
- Interactuar amb el servidor enviant i rebent peticions de dades.

D'aquesta forma podem identificar de forma molt senzilla les fases que anirà realitzant el client. A continuació es mostraran diferents diagrames d'estats per cada fase.

La primera fase de lectura d'arxius de configuració no es rellevant en el context d'un diagrama d'estats específic per ell mateix, per aquest motiu l'obviarem i considerem que ja tenim les dades carregades en memòria.

2.1.1 Procés de Subscripció

El procés de subscripció s'inicia amb l'enviament de paquets al servidor mitjançant ***U**ser **D**atagram **P**rotocol* (UDP) i partint amb el estat del client en mode no subscrit *(NOT_SUBSCRIBED)*. Els primers paquets enviats durant el procés de subscripció sempre seran del tipus petició de subscripció *(SUBS_REQ)*. I tindran el següent format [2]:

SUBS_REQ	MAC	00000000	CTRL-XXX, BxxLxxRxxAxx

[2] Format del paquet SUBS_REQ, Tipus Paquet: SUBS_REQ, MAC del client, Identificador: 00000000, Dades: Nom del controlador i Situació.

A continuació el client canviarà el seu estat en espera de confirmació de subscripció *(WAIT_ACK_SUBS)*. Aquests paquets seran enviats de forma continuada fins haver superat un màxim d'enviaments o fins que el servidor respongui amb un paquet, en aquest últim cas el client avaluarà el tipus de paquet rebut i actuarà en conseqüència:

- ***SUBS_REJ:*** El servidor haurà rebutjat la subscripció, el client finalitzarà el procés de subscripció actual i n'iniciarà un de nou.
- ***SUBS_NACK:*** El servidor torna a demanar les dades, el client seguirà enviant peticions de subscripció.
- ***SUBS_ACK:*** El servidor ha acceptat la subscripció, el client pot avançar en el procés de subscripció.

En l'únic cas de rebre la confirmació de subscripció *(SUBS_ACK)* el client deixarà d'enviar paquets de petició de subscripció i enviarà un paquet amb la informació del client *(SUBS_INFO)* [3]:

SUBS_INFO	MAC	XXXXXXXX	Port TCP, Controladors

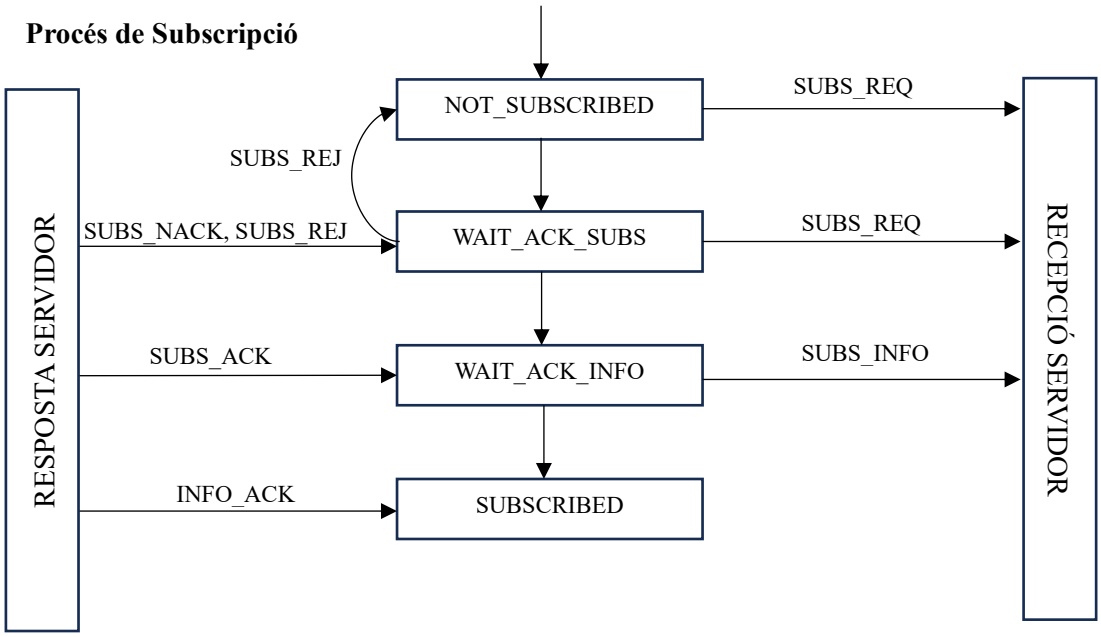
[3] Format del paquet informació de subscripció: MAC del client, identificador proporcionat pel servidor, Dades: Port TCP del client i llistat dels dispositius del client.

En aquest punt el client haurà d'emmagatzemar la identificació proporcionada pel servidor en el paquet rebut i utilitzar-la en els paquets posteriors. A més a més l'enviament del paquet amb informació de subscripció s'enviarà a través d'un nou port UDP especificat pel servidor en el paquet rebut.

Una vegada enviat el paquet amb la informació de subscripció *(SUBS_INFO),* el client canviarà el seu estat esperant la confirmació del paquet d'informació *(WAIT_ACK_INFO)*.

En cas de rebre el paquet de confirmació d'informació *(INFO_ACK)* el client avaluarà les dades del paquet i en cas de ser correctes finalitzarà el procés de subscripció canviant el seu estat a subscrit *(SUBSCRIBED)*.

Amb aquesta identificació dels paquets i estats que ha de rebre i enviar el client podem definir el diagrama d'estats del procés de subscripció[4]:



[4] Diagrama d'estats del client en funció dels paquets rebuts i enviats.

2.1.2 Comunicació Periòdica

El procés de comunicació periòdica entre client i servidor es realitza mitjançant l'enviament de dos tipus de paquets UDP:

- ***HELLO:*** Paquet amb les dades del client per confirmar la comunicació.
- ***HELLO_REJ:*** Rebuig del paquet *HELLO*, finalitzant la comunicació periòdica.

El procés de comunicació periòdica s'iniciarà després de finalitzar el procés de subscripció correctament i per tant el estat del client ha de ser subscrit *(SUBSCRIBED)*. Podem dividir el procés de comunicació periòdica en dos fase:

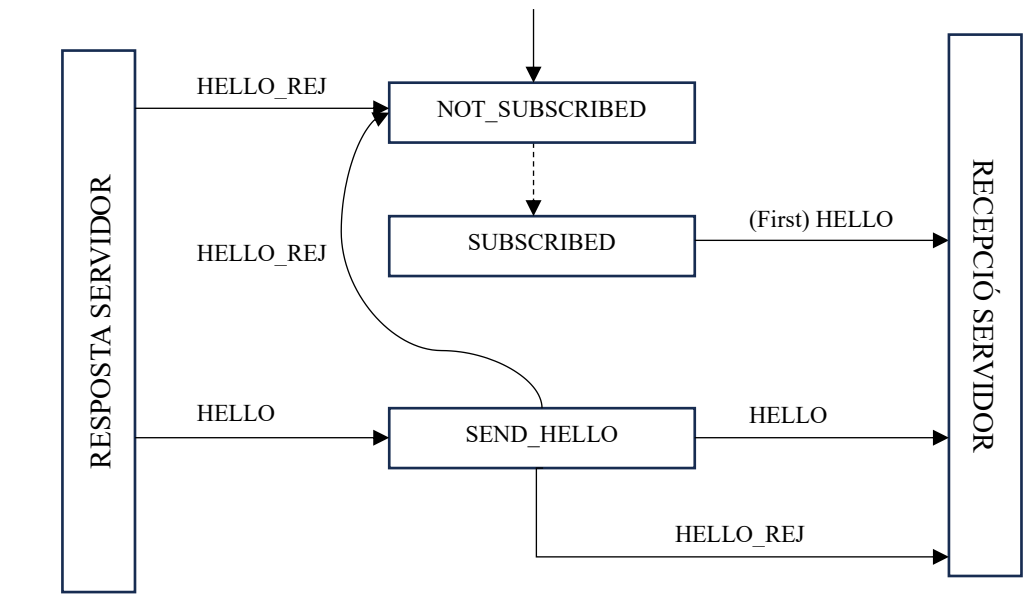
- **Inici de la comunicació:** El client enviarà el primer paquet *HELLO* i seguidament esperarà un màxim de 4 segons fins la resposta del servidor.
- **Continuació de la comunicació:** Un cop rebut correctament el primer paquet *HELLO* del servidor, el client canviarà el seu estat al enviament de paquets *HELLO (SEND_HELLO)*.

En el cas que el paquet *HELLO* rebut tingui discrepàncies en les dades d'identificació el client enviarà un paquet de rebuig *(HELLO_REJ),* i iniciarà un nou procés de subscripció. El format del paquet HELLO és el següent [5]:

HELLO	MAC	XXXXXXXX	CTRL-XXX, BxxLxxRxxAxx

[5] Format del paquet HELLO: MAC del client, identificador proporcionat pel servidor, Dades: nom del controlador i situació.

Amb els procés definit podem mostrar com quedaria el diagrama d'estats del procés de comunicació periòdica [6]:



[6] Diagrama del procés de comunicació periòdica mitjançant paquets UDP.

2.1.3 Enviament i Petició de Dades

La petició i recepció de dades solament estarà activa en el cas que la comunicació periòdica sigui correcta i per conseqüència el client ha d'estar subscrit.

En aquesta fase el client podrà rebre peticions d'informació dels seus dispositius mitjançant una connexió TCP oberta des del client. De forma similar el client podrà enviar dades dels seus dispositius al servidor establint una connexió TCP al port obtingut durant el procés de subscripció. Concretament el client haurà d'enviar dos tipus de paquets:

- ***SEND_DATA:*** Paquet amb el valor d'un dispositiu.
- ***DATA_ACK:*** Confirmació de les dades demanades pel servidor, sigui petició o modificació.

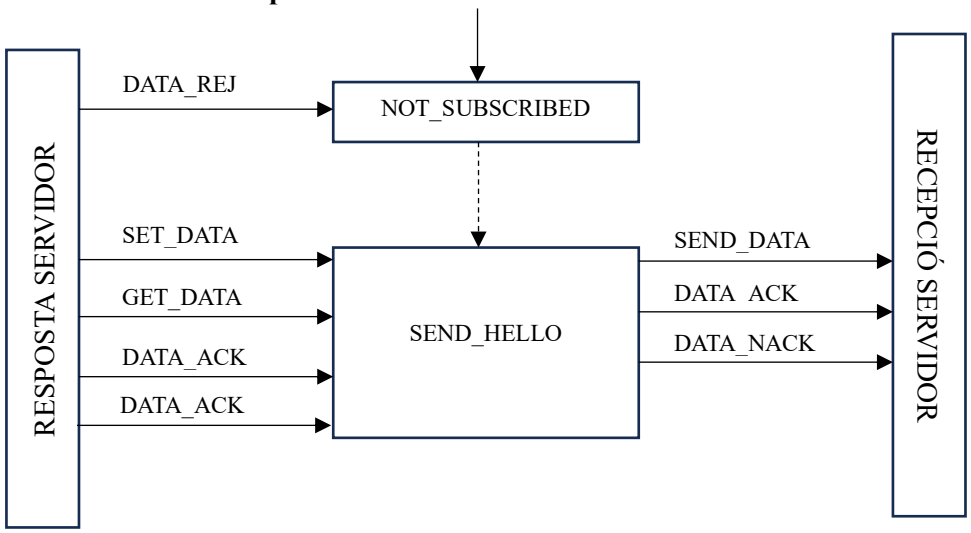
De forma similar esperarà rebre tres tipus de paquets:

- ***SET_DATA:*** Serà enviat pel servidor en qualsevol moment per el port TCP obert i serà una petició per modificar el valor del dispositiu d'entrada o actuador especificat en el paquet.
- ***GET_DATA:*** Serà enviat pel servidor en qualsevol moment per el port TCP obert i demanarà al client el valor del dispositiu especificat en el paquet.
- ***DATA_ACK:*** El client esperarà rebre aquest paquet com a confirmació de l'enviament de dades en el paquet *SEND_DATA*.

Durant aquesta fase, el client es mantindrà en l'estat de comunicació periòdica *(SEND_HELLO)* sempre i que no existeixin discrepàncies en la identificació dels paquets TCP rebuts i que no es rebi un rebuig de dades *(DATA_REJ)*. En aquest cas el client es desconnectarà i iniciarà un nou procés de subscripció. Si existeix algun tipus de problema amb el tractament del dispositiu i el seu valor, tan client com servidor enviaran un paquet amb el motiu de la discrepància *(DATA_NACK)*.

Amb els paquets i interaccions definides podem mostrar el diagrama de la fase d'enviament i recepció de dades a la figura [7]:

Procés d'enviament i recepció de dades



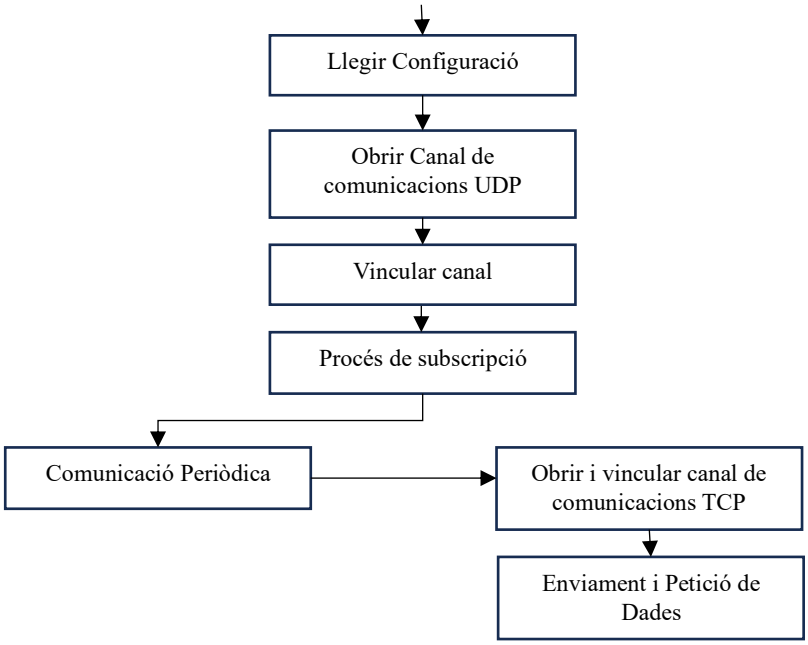
[7] Diagrama del procés de comunicació TCP amb el servidor.

El paquet de dades TCP que conté les dades a enviar tant per el client com servidor té l'estructura següent [8]:

TYPE	MAC	XXXXXXXX	DEVICE	VALUE	INFO

[8] Format paquet de dades: Type: El tipus de paquet mostrat anteriorment, MAC del que envia el paquet, identificador, dispositiu i valor enviat i informació addicional.

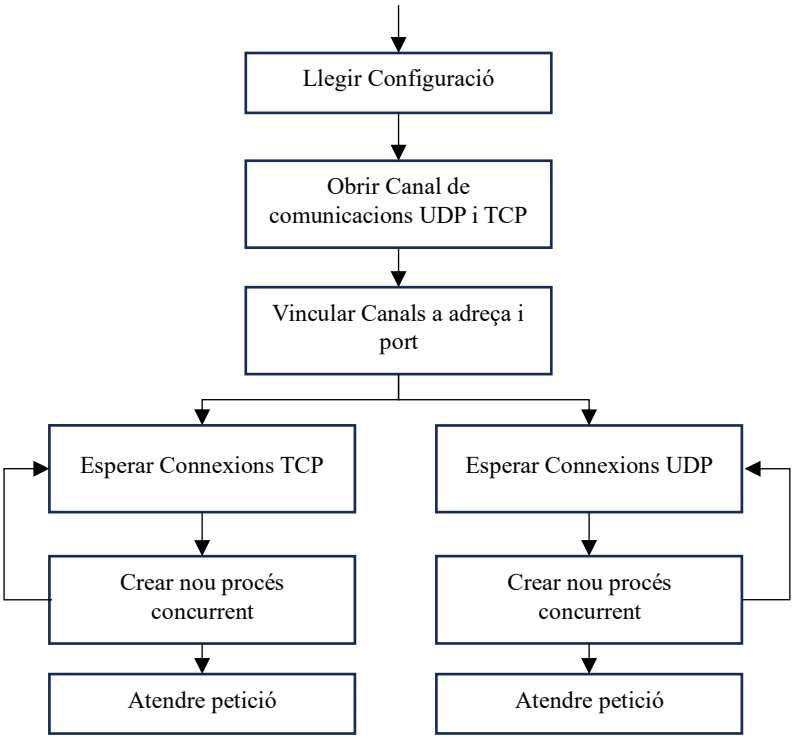
A continuació es mostra l'esquema de funcionament general que ha de seguir del client [9]:



[9] Diagrama final de les fases/funcionalitats del client. Nota: Les fases situades de forma paral·lela representen una execució concurrent.

2.2 Diagrama d'estats: Servidor

A diferència del client el servidor no s'estructura per un seguit de fases, en canvi a mesura que va rebent connexions dels clients serà ell l'encarregat de proporcionar i actualitzar les fases del clients. Per aquest motiu primer es mostra l'estructura del servidor concurrent amb les dues connexions que accepta [10]:



[10] Diagrama del funcionament del servidor atenent peticions UDP i TCP de forma concurrent.

Apart de la funcionalitat general del servidor aquest haurà d'efectuar un total de quatre tasques:

- Atendre Peticions de Subscripció.
- Mantenir Comunicació Periòdica.
- Rebre dades dels controladors.
- Demanar o enviar dades als controladors.

Les dues primeres es realitzaran mitjançant connexions amb UDP i les restants amb connexions TCP.

A continuació es mostrarà l'explicació i diagrama de cada tasca que ha d'efectuar el servidor per cada connexió entrant.

2.2.1 Peticions de Subscripció

El primer que haurà de comprovar el servidor en rebre una nova connexió de qualsevol tipus serà comprovar que el client sigui un controlador autoritzat en el sistema. Si resulta no estar autoritzat en el sistema el servidor enviarà un paquet de rebuig i finalitzarà la comunicació amb aquell client.

Si no ha estat rebutjat avaluarà el paquet rebut i en funció del estat en el que es troba el client dins del servidor realitzarà una acció o un altra.

Si partim d'un client desconnectat esperarem rebre un paquet de petició de subscripció *(SUBS_REQ)* [2]. Aleshores iniciaríem un nou procés de subscripció de forma concurrent per atendre a aquell client. De forma similar al procés de subscripció del client el servidor esperarà rebre diferents paquets durant aquest procés:

- ***SUBS_REQ:*** Paquet inicial de subscripció amb les primeres dades del controlador.
- ***SUBS_INFO:*** Següent paquet a rebre en la fase de subscripció amb dades del controlador.

Tanmateix el servidor haurà de respondre amb diferents paquets:

- ***SUBS_REJ:*** Rebuig de la subscripció, en cas de trobar discrepàncies durant el procés.
- ***SUBS_NACK:*** En cas de trobar algun error en el processament del paquet.
- ***SUBS_ACK:*** Primer paquet acceptat, el client pot seguir en el procés de subscripció.
- ***INFO_ACK:*** Rebudes i acceptades les dades finals pel procés de subscripció.

En els casos d'enviar confirmacions de subscripcions el servidor enviarà dades d'identificació i ports per continuar la comunicació:

- ***SUBS_ACK:*** En l'apartat de dades s'especificarà un nou port UDP pel que es continuarà el procés de subscripció. I en l'apartat d'identificador es crearà un codi únic durant la comunicació amb el client sempre i que estigui subscrit [11].

SUBS_ACK	MAC	XXXXXXXX	Nou Port UDP

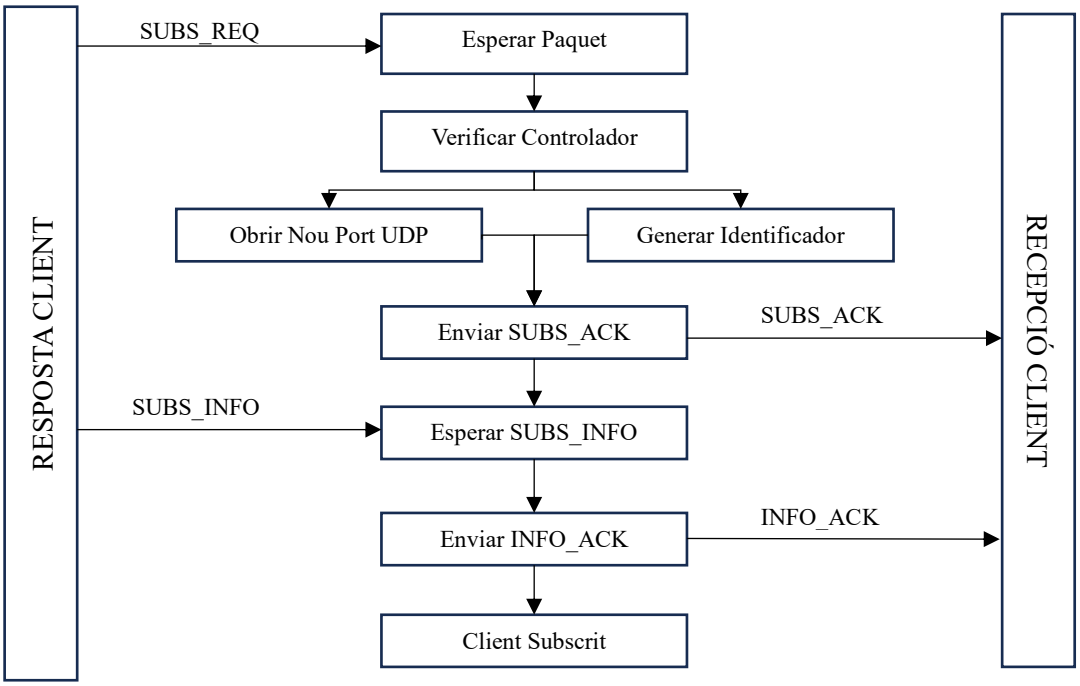
[11] Format del paquet SUBS_ACK

- ***INFO_ACK:*** En l'apartat de dades s'especificarà el port TCP del servidor per l'enviament de dades.

INFO_ACK	MAC	XXXXXXXX	Port TCP

[12] Format del paquet INFO_ACK

Amb els paquets definits podem mostrar el diagrama que hauria de seguir el servidor per el procés de subscripció [13].



[13] Diagrama del funcionament del procés de subscripció (obviant canvis d'estat del client intermitjos i possibles rebutjos de subscripció).

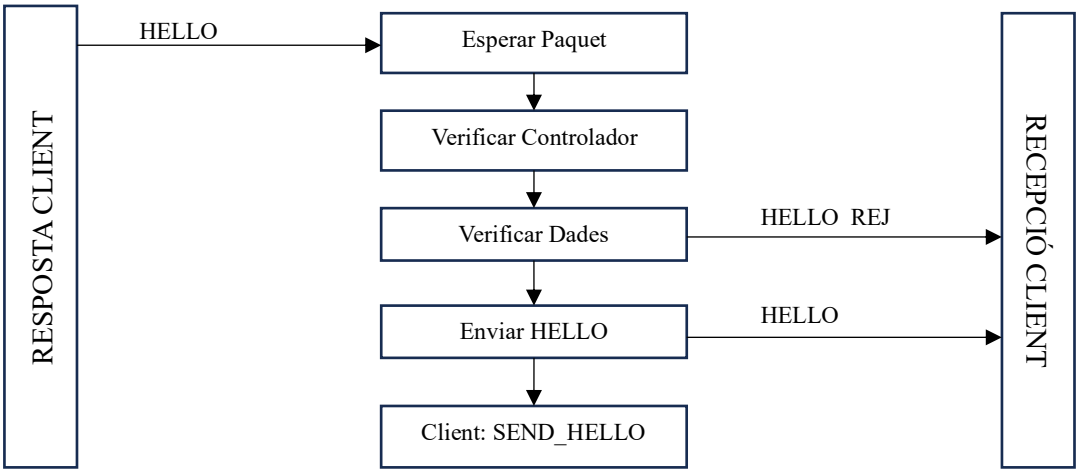
2.2.2 Manteniment de la Comunicació Periòdica

Per el manteniment de la comunicació periòdica el servidor simplement haurà de respondre als paquets HELLO dels controladors, i comprovar que cada controlador enviï aquests paquets de forma regular. Els paquets durant aquest procés seran els mateixos descrits en l'apartat 2.1.2.

Podem identificar dos processos que s'executaran de forma paral·lela:

- Enviament de paquets *HELLO*.
- Monitorització d'intervals dels paquets *HELLO*.

El procediment d'enviament de paquets *HELLO* simplement ha de comprovar que es tracti d'un controlador autoritzat en el sistema i que aquest paquet no tingui discrepàncies amb les dades del controlador carregades a memòria, el esquema de funcionament es pot veure en el següent diagrama:



[14] Diagrama del procediment de comunicació periòdica

Cada vegada que el servidor verifiqui i enviï un nou paquet *HELLO* haurà d'actualitzar el temporitzador de l'interval d'enviament de paquets. En cas que aquest interval superi el valor establert (3 paquets consecutius) el servidor desconnectarà el controlador.

2.2.3 Recepció de Dades

Aquest apartat és la resposta a la funcionalitat del client a l'hora d'enviar dades cap al servidor mitjançant el port TCP especificat pel servidor durant el procés de subscripció.

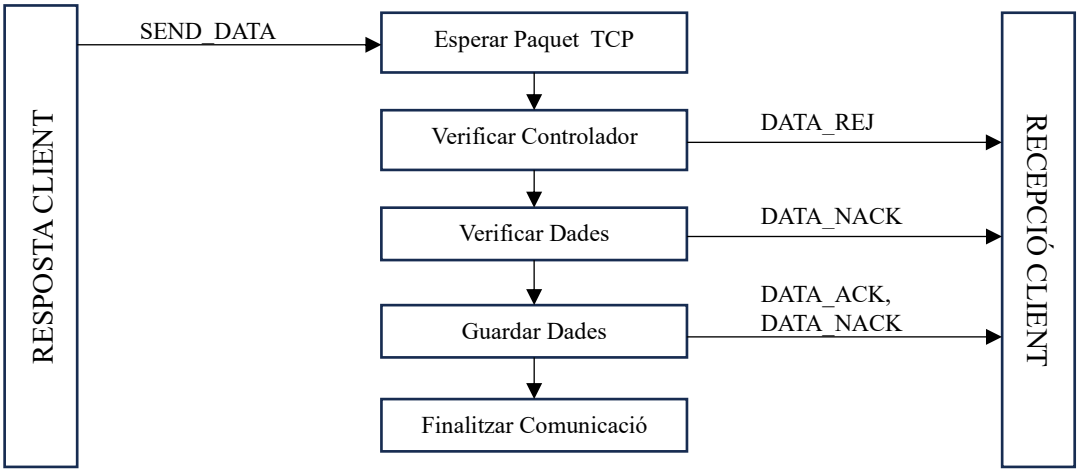
Durant aquest procés solament pot rebre un sol tipus de paquet:

- ***SEND_DATA:*** Paquet amb la informació del dispositiu del controlador a actualitzar i guardar.

En rebre el paquet, mitjançant la nova connexió TCP amb el controlador, el servidor haurà de verificar que es tracta d'un controlador autoritzat, i que el dispositiu especificat existeix. Si tot es correcte el servidor guardarà la informació en un fitxer especificant el controlador, data de recepció, dispositiu i valor. A continuació es mostren els paquets de resposta que el servidor haurà de respondre als controladors:

- ***DATA_ACK:*** El servidor haurà pogut emmagatzemar correctament les dades del dispositiu.
- ***DATA_NACK:*** El servidor no ha pogut emmagatzemar les dades del controlador.
- ***DATA_REJ:*** Solament s'enviarà en cas que existeixi algun problema amb les dades d'identificació del controlador.

A continuació es mostra el procés de recepció de dades [15]:



[15] Diagrama del procés de recepció de dades per part del servidor.

2.2.4 Petició de Dades

Per cada petició de dades que realitzi el servidor haurà d'iniciar una nova comunicació amb el controlador mitjançant el port TCP rebut durant el procés de subscripció. Existeixen dos tipus de peticions de dades:

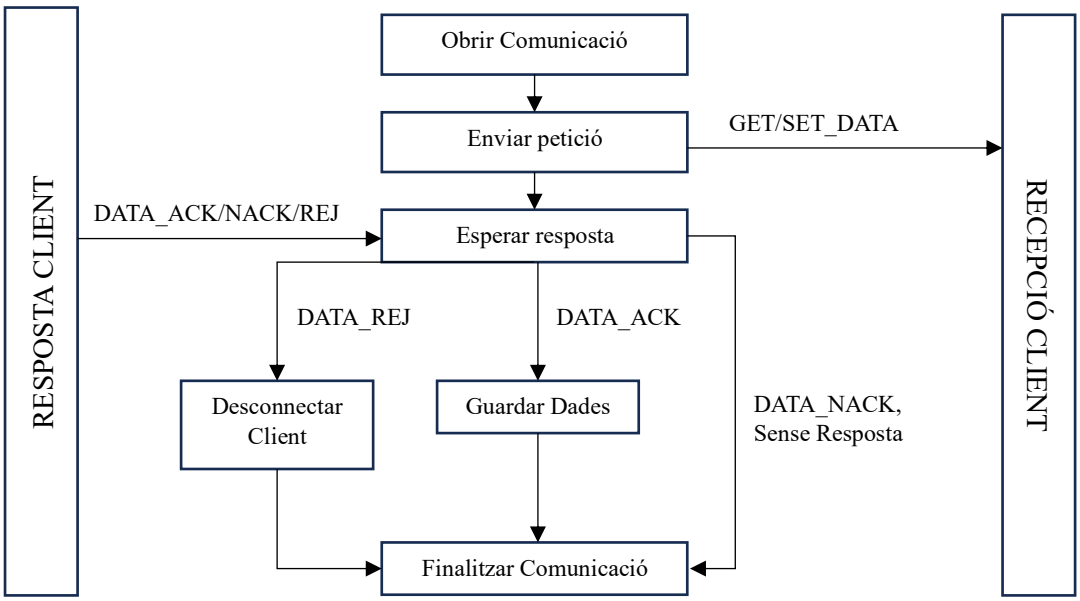
- ***SET_DATA:*** Demanarà al controlador modificar el valor d'un dels seus actuadors i en rebre confirmació l'emmagatzemarà.
- ***GET_DATA:*** Demanarà al controlador que enviï el valor d'un dels seus dispositius i seguidament en rebre confirmació l'emmagatzemarà.

De forma similar a la descrita en l'apartat anterior el servidor esperarà resposta del client, existeixen tres tipus de resposta:

- ***DATA_ACK:*** El client haurà acceptat la petició i el servidor podrà emmagatzemar les dades demanades.
- ***DATA_NACK:*** El client haurà rebutjat la petició de dades.
- ***DATA_REJ:*** Discrepàncies amb la identificació del client, el servidor desconnectarà el controlador.
- ***Sense Resposta:*** Finalitzarà la comunicació amb el client.

Si el servidor no ha pogut establir connexió amb el client desconnectarà el client i finalitzarà la comunicació.

A mode de resum independentment del tipus de sol·licitud que realitzi el servidor, aquest haurà d'emmagatzemar les dades sempre que rebi un paquet de confirmació (*DATA_ACK*). A continuació és mostra el procés de petició de dades:



[16] Procés de petició de dades al client

3. Implementació

Tenint els esquemes del funcionament tant per part de servidor com client podem començar a realitzar la implementació d'aquest sistema. En aquest apartat es mostraran quines decisions de disseny s'han seguit per desenvolupar el codi font de la pràctica, comentant aquelles part del codi més importants o rellevants sense aprofundir explícitament en com s'ha fet pas per fas cada funció.

Si es vol aprofundir en com s'ha fet la implementació de cada secció del codi s'aconsella obrir el codi font i llegir els comentaris associats a cada funció, en aquest apartat es comentaran les funcions a nivell més general.

3.1 Estructuració del Projecte

Durant tot el desenvolupament del projecte s'ha tingut molt en compte una bona organització i estructuració del codi, tant a nivell de funcions auxiliars com comentaris i aclariments.

Per aquest motiu tant en la implementació del client com el servidor s'ha dividit el codi font en diferents fitxers, cadascun especialitzat en una funcionalitat concreta de les abans mencionades en els diagrames. Aquesta estructuració és molt més visible en la implementació del servidor, encara així existeixen elements compartits entre tots dos:

3.1.1 Arxius Base

Totes dues implementacions utilitzen fitxers amb funcions auxiliars per facilitar certes tasques:

- ***Logs:*** Els *logs* o registres son els comentaris mostrats per terminal durant l'execució del programa, aquests proporcionaran la informació necessària per determinar errors, informacions o avisos. Al requerir d'un format específic: *"[HORA:MINUT:SEGON] [TIPUS]"* era molt més senzill definir una sèrie de funcions que permetessin mostrar missatges amb aquest format per pantalla de forma senzilla. A més un dels requeriments era la opció *debug* la qual mostra missatges addicionals per realitzar un millor seguiment de l'aplicació, motiu pel que s'afegeix un paràmetre booleà que activarà aquests missatges.
- ***Llegir configuracions:*** És un arxiu que permet llegir i carregar en memòria les configuracions necessàries del arxiu de configuració *".cfg"*. També incorpora certes comprovacions i avisos per verificar que les configuracions introduïdes son les correctes per evitar possibles errors posteriors en l'execució del codi.
- ***Protocols PDU:*** Son les implementacions que es considerarien més importants. Aquests arxius incorporen tots el mètodes encarregats de tractar amb la recepció, enviament i conversió dels paquets enviats entre client i servidor, es comentarà en profunditat més endavant però el fet d'evitar treballar a nivell de bytes a nivells superiors i fer-ho solament amb paquets aporta un gran avantatge i simplificació del procés.

3.1.2 Arxius addicionals

Apart dels arxius que es troben en comú com hem vist en l'apartat anterior, cada codi font té les seves pròpies classificacions. En el cas de la implementació del client s'ha optat per no utilitzar tant aquest recurs pel fet de disposar d'un conjunt de variables globals i *flags* que es van comunicant entre processos.

En el servidor si que es va optar per seguir aquest procediment, ja que els codis solien ser molt llargs i suposaven un problema a la navegabilitat del codi font.

3.1.3 Comentaris

Totes les funcions, estructures, diccionaris, enumerats i cada fitxer estan degudament comentats amb una sèrie de descripcions i aclariments de les implementacions i funcionalitats que aquests aporten i tots segueixen una estructura en comú, en el cas de la implementació en *Python:*

```
"""
<Descripció de la funció / Classe>
Parameters:
- param1 (Tipus): Paràmetre que rep la funció.
- param2 (Tipus):
Returns:
- Element que retorna la funció
"""
```

En cas que el codi comentat no disposi de paràmetres o valors de retorn simplement no s'afegeix, però sempre es manté una breu descripció juntament amb una descripció més general de la implementació.

En el cas del servidor implementat en *C99:*

```
/**
 * @brief Descripció curta
 * @param Paràmetre que accepta la funció
 * @return El element a retornar
 */
```

I de forma similar si no es disposa d'algun apartat s'omet.

3.2 Abstracció del Codi

El fet de dividir el codi en diferents fitxers i en conseqüència diferents mètodes, redueix la complexitat a l'hora d'implementar certes funcionalitats. Es segueix molt el procediment d'encapsular un mètode a una sola funcionalitat i en el cas que aquest requereixi d'un altra secció crear una funció auxiliar per ajudar al mètode principal.

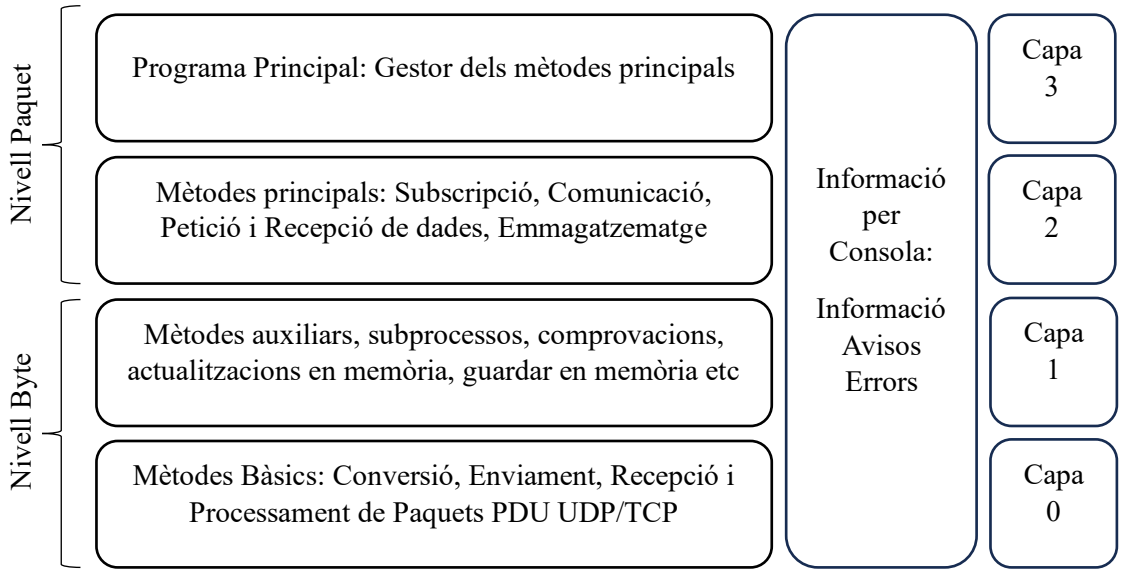
3.2.1 Capes

Encara que es un element bàsic de la bona programació s'ha volgut remarcar per que aquest procediment ha permès dividir el codi en diferents capes:

- **Capa 0:** Mètodes bàsics, recepció, enviament, i conversió de bytes.
- **Capa 1:** Mètodes auxiliars.
- **Capa 2:** Mètodes principals com processos de subscripció, comunicació etc.
- **Capa 3:** Programa Principal realitzant les crides i gestions necessàries dels mètodes principals de la capa inferior.

També tindrem en compte mètodes de carregar i processar configuracions, i mostrar informació per consola.

A continuació es mostra de forma esquemàtica com s'estructura el codi amb les capes i mètodes mencionats [17]:



[17] Disseny dels mètodes per capes

S'han classificat les dos primeres i dos últimes capes a nivell de byte i nivell de paquet ja que els mètodes principals al cridar als mètodes auxiliars rebran objectes o estructures en format paquet, menters que els mètodes auxiliars i bàsics hauran de fer les comprovacions i conversions dels paquets rebuts en format vector de bytes.

D'aquí la necessitat d'implementar mètodes que realitzen aquestes conversions:

- ***from_bytes():*** Converteix un *array* de bytes al format de paquet PDU UDP o TCP.
- ***to_bytes():*** Converteix un paquet PDU UDP/TCP a un *array* de *bytes*.
- ***recv():*** Rebrà un paquet donat un descriptor de fitxer i el retornarà en format PDU UDP o TCP realitzant la conversió internament.
- ***send():*** Enviarà un paquet al destinatari realitzant la conversió a *bytes* de forma interna.

Els noms del mètodes poden variar entre diferents implementacions del servidor i client, juntament amb el tipus de format de paquet que s'estigui tractant.

3.2.1 Llibreries

Anomenem llibreries a les diferents organitzacions dels mètodes del projecte, cada llibreria es tractarà d'un fitxer amb les declaracions i implementacions d'una funcionalitat en concret. Aquest apartat fa referència al punt 3.1.1 i 3.1.2.

Es treballa constantment amb el concepte d'encapsular llibreries que realitzin una sola tasca, això permet la estructuració per capes i una organització clara. En el cas de la implementació del client en *Python* al requerir d'un seguit de variables globals s'han mantingut moltes funcions en el arxiu principal, encara així s'ha comentat cada secció del codi amb comentaris d'inici i final:

```
############      ################
# <Secció> #      # <Secció END> #
############      ################
```

En el cas de la implementació del servidor si que s'han utilitzat arxius diferents.

A continuació es llistaran el conjunt d'arxius i una breu descripció de la seva utilitat, l'extensió d'aquests arxius és ".h" que representa la paraula *header* i simplement conté la definició dels mètodes i que aquestes han de realitzar:

- ***commons.h***: és un arxiu comú entre tots els fitxers del projecte i conté totes les llibreries utilitzades, d'aquesta forma evitem escriure explícitament per cada fitxer totes les llibreries que aquest requereix.
- ***logs.h:*** Llibreria per mostrar missatges informatius per pantalla, a més, permet activar la opció de *debug* per mostrar missatges addicionals.
- ***threadpool.h:*** S'explicarà en més profunditat en un apartat posterior.
- ***pdu/:*** Carpeta que contindrà les llibreries necessàries per la creació, recepció, enviament i conversió de paquets.
    - ***tcp.h i udp.h:*** Mètodes per convertir paquets a binari, de binari a paquet, enviament i recepció. També conté la definició de les estructures per definir els camps i tipus de paquets per cada tipus, respectivament.
- ***server/:*** Carpeta amb totes les llibreries necessàries per les diferents funcionalitats del servidor.
    - ***conf.h:*** Permet llegir i inicialitzar l'estructura que contindrà la configuració del servidor.
    - ***controllers.h:*** Una de les llibreries més important, ja que permet llegir, carregar en memòria els controladors autoritzats, gestionar-los, comprovar dades i en general tota operació que requereixi modificar o llegir controladors.
    - ***commands.h:*** Llibreria per gestionar les comandes del usuari que pot realitzar una vegada el servidor està inicialitzat.
    - ***subs.h:*** Conté les definicions de les diferents funcions per realitzar un procés de subscripció o mantenir la comunicació periòdica amb els controladors autoritzats.
    - ***data.h:*** Conté les definicions per l'enviament, recepció i emmagatzemant de dades dels controladors.

Apart de totes les llibreries implementades per el projecte s'importen d'altres de la llibreria estàndard.

4. Codi

Com s'ha comentat anteriorment no es mostrarà línia per línia la implementació del codi, per aquest motiu es mostrarà un "esquelet" o estructuració de com funciona per dins en una sèrie de passos.

4.1 Client

El client implementat en *Python 3.11.4* inicia amb una sèrie fases:

4.1.1 Inicialització

S'ha implementat un mòdul que s'encarregarà de llegir i verificar la configuració del client. A mesura que llegirà el arxiu de configuració ".cfg" anirà afegint els valors dins d'un diccionari que podrà ser accedit per la resta del programa:

```
client = {
    'status': status['DISCONNECTED'],
    'Name': None,
    'Situation': None,
    'MAC': None,
    'Local_TCP': None,
    'Srv_UDP': None,
    'Server': None,
    'Elements': {},
    'Server_Config':{}
}
```

De forma similar el client tindrà diccionaris per els seus dispositius del controlador i per la configuració del servidor. En cas que alguna configuració no compleixi el format correcte el client sortirà amb un missatge d'error.

Seguidament s'iniciaran altres comandes com el mode *debug* afegint el paràmetre "-d". I finalment iniciarà descriptors de fitxer per les comunicacions via PDU.

4.1.2 Subscripció

Per el procés de subscripció s'han utilitzat els estats del client juntament amb una variable global del nombre de subscripcions que permetran identificar quan el client està subscrit i mantenir un comptador dels intents de subscripcions realitzats.

```
subs_attempts = 0
```

Seguidament el programa principal "main" executarà un bucle infinit comprovant quant la bandera *disconnected* està activa, en aquest cas es farà una crida a la funció per iniciar un nou procés de subscripció.

```
if config.status['NOT_SUBSCRIBED'] and subs_req():
```

Aquesta funció retornarà un valor booleà indicant si el client s'ha subscrit correctament (*true*) o no s'ha pogut subscriure (*false*). Aquesta funció estarà organitzada en diferents fases:

1. Enviar petició de subscripció (SUBS_REQ):
2. Esperar resposta.
    1. Si es rep confirmació de subscripció (SUBS_ACK) seguir al següent pas.
    2. Si es no es rep confirmació (SUBS_ACK) tornar al pas 1.
    3. Si es rep rebuig de subscripció (SUBS_REJ) finalitzem procés retornant *false*.
    4. Si no rebem res, incrementem temps d'enviament de petició i retornem al pas 1.
3. Enviar informació de subscripció (SUBS_INFO) pel port rebut al SUBS_ACK.
    1. Si rebem confirmació client estarà subscrit i retornem *true.*
    2. Si no rebem confirmació finalitzem procés retornant *false.*

També tenim establert un màxim de cops que es pot enviar una petició de subscripció fixada en 7, en cas de superar-la retornarem *false*. Per implementar aquest funcionalitat s'ha optat per un bucle for que contindrà les dues primeres fases:

```
for packets_sent in range(n):
    send_subs_req(…) # Enviar petició
    subs_ack, address = pdu_udp.recvUDP(…)
    # Processar paquet
else:
    return false
```

En *Python* podem utilitzar *for* i *else* i en cas que no existeixi alguna condició que ens obligui a sortir del bucle s'executarà aquest bloc. En el nostre cas es molt útil ja que ens permet identificar el màxim de coms que hem enviat la petició de subscripció i per tant superat el límit establert.

En cas de rebre un paquet de confirmació podrem continuar a la següent fase situada sota del bucle. Aquí esperarem rebre un paquet de confirmació de subscripció i finalment retornarem *true* mostrant que el client s'ha subscrit correctament.

En cas que el client no s'hagi subscrit el comptador s'anirà incrementant fins a un màxim de 3 processos de subscripció, en aquest límit el client es tancarà mostrant un missatge d'error.

4.1.3 Comunicació Periòdica

Si la subscripció ha estat correcta el client iniciarà un procés de comunicació periòdica:

```
hello_process = threading.Thread(target=hello_process_thread)
hello_process.start()
```

Ho iniciem en un procés concurrent així podrem realitzar altres tasques d'enviament i recepció de dades. La comunicació periòdica la podem dividir en dos processos paral·lels:

4.1.3.1 Enviament de paquets HELLO

Cada 2 segons el client s'encarregarà d'enviar un paquet HELLO al servidor, això ho podem realitzar en un bucle infinit:

```
while config.status['SUBSCRIBED'] or .status[SEND_HELLO]:
    pdu_udp.send(…)
    time.sleep(2)
```

Solament deixarà d'enviar paquets HELLO en el cas que el client es desconnecti, l'encarregat d'actualitzar aquesta bandera serà el segon procés responsable de la comunicació periòdica:

4.1.3.2 Recepció de paquets HELLO

El procés de recepció de paquets HELLO es una mica més complex ja que ha d'efectuar més tasques:

1. Esperar primer paquet
   1.1. Si no es rep en els primers 4 segons finalitzar comunicació i iniciar nova subscripció.
   1.2. Si es rep paquet, confirmar que es tracta d'un paquet correcte i seguir al pas 2.
   1.3. Si el paquet no es correcte finalitzar comunicació i iniciar nova subscripció.
2. Esperar paquets
   2.1. Si no es rep paquet en 2 segons incrementar comptador de paquets no rebuts, si es superen 3 paquets, tancar comunicació i iniciar nova subscripció.
   2.2. Si es rep paquet correcte tornar al punt 2.
   2.3. Si el paquet no es correcte finalitzar comunicació i iniciar nova subscripció.

El procés principal esta format per un bucle *while* infinit sempre i que es rebin paquets de forma continuada:

```
while missed < 3:
    hello, addr = pdu_udp.recvUDP(sock_udp)
    if hello is None:
        missed += 1
    elif hello.packet_type == HELLO_REJ:
        # Finalitzar comunicació
    else:
        if check_hello(hello, addr):
            # Reinicar comptador
            missed = 0
            subs_attempts = 0
        else:
            # Finalitzar comunicació
else:
    # Finalitzar comunicació
```

Tornem a utilitzar la condició *else* per finalitzar la comunicació en cas que no es rebin tres paquets de forma consecutiva.

4.1.4 Enviament d'informació

L'enviament d'informació per part del controlador es realitzarà mitjançant un nou procés concurrent encarregat d'establir una connexió TCP amb el servidor per realitzar l'intercanvi d'informació. Aquest s'iniciarà en el moment que s'activi la comanda *send*. Els passos a seguir son:

1. Obrir comunicació TCP amb el servidor.
2. Enviar paquet de dades.
3. Esperar resposta.
   3.1. Si no es rep resposta en 3 segons es finalitzarà la comunicació.
   3.2. Si es rep resposta i la verificació és correcta es finalitzarà la comunicació.
   3.3. Si es rep resposta amb verificació incorrecta es finalitzarà la comunicació amb el client i s'iniciarà un nou procés de subscripció.

La implementació d'aquest apartat es bastant bàsica a nivell de codi, per més detalls veure el codi de la funció *"send_data()"*.

4.1.5 Recepció d'informació

El procés de recepció d'informació solament es pot realitzar un cop establerta la comunicació periòdica amb el servidor 4.1.3 (estat *SEND_HELLO* del client). En aquest estat s'obrirà un descriptor de fitxer associat al port TCP enviat pel servidor durant el procés de subscripció.

Per obrir el canal de comunicació s'utilitza la funció *"open_comm()"* i serà executada pels threads encarregats de la comunicació periòdica.

Així doncs, amb la comunicació via TCP oberta el client haurà de realitzar un seguit de fases per atendre una petició:

1. Esperar nova comunicació TCP.
2. Atendre nova connexió.
3. Esperar paquet.
   3.1. Si no es rep paquet, finalitzar comunicació.
   3.2. Si es rep paquet comprovar credencials.
      3.2.1. Si les credencials son correctes anar al pas 4.
      3.2.2. Si hi ha discrepàncies enviar rebuig de dades finalitzant comunicació amb el servidor i iniciant nou procés de subscripció.
4. Determinar si es tracta d'un paquet de petició o de modificació de dades.
   4.1. Si es tracta d'un paquet de modificació de dades comprovar que el dispositiu a modificar es un actuador.
      4.1.1. Si el dispositiu existeix i es un controlador actualitzar dades i enviar confirmació. Finalitzem comunicació.
      4.1.2. En cas contrari enviar rebuig de dades depenent de si no es troba el dispositiu o si no es tracta d'un actuador.
   4.2. Si es tracta d'una petició de dades comprovar que existeix el dispositiu.
      4.2.1. Si existeix enviar confirmació amb les dades del dispositiu i finalitzar comunicació.
      4.2.2. En cas contrari enviar rebuig de dades i finalitzar comunicació.

Per aquesta implementació s'utilitzarà el mètode *select* que permet monitoritzar descriptors de fitxer obert, en el nostre cas l'utilitzarem per comprovar comandes del usuari i quan detectem peticions de dades per part del servidor:

```
readable, _, _ = select.select(…)
for sock_or_input in readable:
    if tcp_on.is_set() and sock_or_input is sock_tcp:
        recv_data()
```

A continuació comprovem les credencials del paquet (en cas de rebre'l) i processem cada petició en funció del tipus de paquet:

```
if packet.ptype == pdu_tcp.packet_type['SET_DATA']:
    set_data(packet,server_socket)
elif packet.ptype == pdu_tcp.packet_type['GET_DATA']:
    get_data(packet,server_socket)
```

Cada funció realitza el procés descrit anteriorment, simplement es requereixen d'una sèrie de comprovacions i enviar les dades pel port TCP.

Aquesta seria la última funcionalitat del client, en la explicació s'ha obviat el canvi d'estat del client però en cada fase com les descrites en l'apartat de plantejament s'anirà actualitzant l'estat del controlador.

4.2 Servidor

El servidor està implementat en el llenguatge C99 i a nivell de codi és més complex i extens que el client, ja que ha de suportar la comunicació concurrent entre múltiples clients a l'hora. A continuació es mostraran certes funcionalitats clau que permet el funcionament del servidor.

4.2.1 Configuració

La lectura de configuració es basa en dos parts, lectura de la configuració del servidor i lectura del nombre de controladors autoritzats.

La lectura de la configuració del servidor es com qualsevol altra lectura d'un fitxer, en canvi la lectura dels controladors es realitza de forma dinàmica, així no tenim "límit" teòric del nombre de controladors i evitem crear espai innecessari en cas que el nombre de controladors sigui menor al reservat inicialment.

Per aquest motiu tenim definides diferents funcions que ens permeten carregar en memòria a mesura que es llegeixen controladors del arxiu ".dat":

- ***loadControllers():*** Es la funció principal i s'encarrega de realitzar crides a les altres funcions per finalment retornar el nombre de controladors carregats.
- ***getNextLine():*** Llegeix cada línia del arxiu ".dat" retornant una parella de nom-controlador més mac-controlador.
- ***addController():*** Rebrà la parella de la funció *getNextLine* i s'encarregarà d'inicialitzar una estructura amb les dades necessàries per representar un controlador.

Amb els controladors i dades de configuració carregades el servidor podrà inicialitzar els descriptors de fitxers amb els ports per els que esperarà rebre les comunicacions.

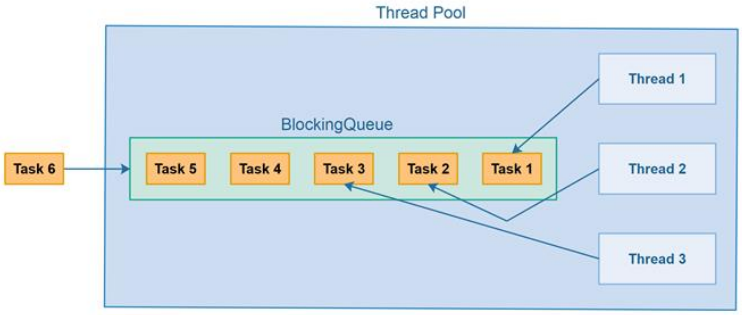
4.2.2 ThreadPool

Al final de la implementació es va decidir utilitzar una *ThreadPool* per facilitar la creació i gestió dels processos concurrents: fils o *threads* en angles. El concepte es relativament senzill:

S'inicialitza una estructura per la creació de la *ThreadPool* que contindrà variables per gestionar l'accés concurrent entre els fils. Seguidament es crearan un nombre fix de *threads* anomenats *worker threads*. Aquests s'encarregaran de processar "tasques" que els hi demanem.

Per representar aquestes tasques es crearà una cua entrellaçada que contindrà funcions i arguments que els *worker* aniran agafant per processar-les.

A mode de resum cada vegada que vulguem executar un procés concurrent s'enviarà a una cua de tasques de la que els *workers* aniran llegint i processant fins que la cua quedi buida. Podem veure una representació d'una *ThreadPool* en el diagrama següent [18]:



[18] Representació esquemàtica d'una ThreadPool, on veuen es 3 worker threads processant les tasques de la cua.

Per crear aquest sistema s'utilitzen 4 mètodes principals:

- ***thread_pool_create()***: S'encarregarà de crear la *thread pool*, inicialitzant la estructura de variables i creant els *worker thread* encarregats de fer les tasques.
- ***worker():*** Es la funció que defineix a cada *worker thread* i estarà en un bucle infinit comprovant la cua en busca de noves tasques a realitzar.
- ***thread_pool_submit():*** Rebrà com a paràmetre un punter a la funció que es vol executar juntament amb un punter als arguments d'aquesta funció. Tot seguit aquesta funció afegirà aquesta tasca a una cua (sempre i que aquesta no estigui en el seu màxim). Finalment avisarà als *workers* que existeix una tasca a la cua en cas que aquests estiguin en mode "hivernació" per estalviar recursos.
- ***thread_pool_shutdown():*** S'encarrega de finalitzar la *thread pool* alliberant de memòria les estructures i finalitzant els *workers.* La forma de fer-ho en aquest implementació és utilitzant un concepte anomenat *poison pills.*
    - ***Poison pills:*** Son tasques que la seva única funcionalitat es informar al *worker* que ha de finalitzar la seva execució, així el *worker* al llegir una nova tasca de la cua detectarà que es tracta d'una *poison pill* i finalitzarà.

L'objectiu de la pràctica no està enfocat en un maneig de processos concurrents però em va semblar una implementació curiosa i a més que aportava certs beneficis amb una petita adició:

A més de gestionar els processos paral·lels la *thread pool* s'encarrega d'alliberar de memòria els paràmetres de la tasca:

```
(task.function)(task.argument);
free(task.argument);
```

Així no ens hem de preocupar pels diferents casos o excepcions que puguin aparèixer dins de l'execució de la funció per alliberar memòria abans d'hora.

En general aquesta implementació ha permès oblidar-nos dels problemes que podria suposar executar múltiples *threads* nous cada cop que es vol realitzar una nova tasca i limitar la seva creació solament en la inicialització del programa, a més de facilitar l'execució i neteja de les funcions amb els seus paràmetres.

4.2.3 Main

El codi principal del servidor estarà situat en un bucle infinit i aquest efectuarà 4 tasques:

- Atendre paquets pel port UDP.
- Atendre connexions TCP.
- Realitzar comandes del usuari.
- Actualitzar temporitzadors de les comunicacions periòdiques amb els controladors.

Al tractar-se de moltes possibles tasques a la vegada amb crides a funcions bloquejants que s'han d'executar de forma concurrent, d'aquí sorgeix la necessitat d'una *thread pool.* Per detectar quan s'ha d'atendre un tipus de procés o un altre s'utilitzarà la llibreria sys/*select.h* per monitoritzar els *sockets*. A continuació es mostra l'estructura implementada:

```
while (true) {
    /* Start file descriptor macros */
    FD_ZERO(&readfds);
    FD_SET(STDIN_FILENO, &readfds);
    FD_SET(tcp_socket, &readfds);
    FD_SET(udp_socket, &readfds);

    /*Start monitoring file descriptors*/
    select(...)

    /* Check if UDP file descriptor has received data */
    if (FD_ISSET(udp_socket, &readfds)){
        thread_pool_submit(...);
    }
    /* Check if the TCP file descriptor has received data */
    if (FD_ISSET(tcp_socket, &readfds)){
        thread_pool_submit(...);
    }
    /* Server commands */
    if (FD_ISSET(STDIN_FILENO, &readfds)){

    }
}
```

Per cada procés nou que es requereixi s'afegirà a la cua de la *thread pool* amb la funció *thread_pool_submit()*. A continuació es mostrarà com funciona cada procés que ha d'executar el servidor.

4.2.4 Subscripció

El procés de subscripció s'inicia un cop el paquet rebut pel port UDP ha estat verificat que es tracta d'un controlador autoritzat i que aquest es troba en l'estat desconnectat. Arribat a aquest punt inicia l'intercanvi d'informació amb el client.

La primera tasca que realitza el servidor es crear un identificador pel controlador, simplement es tracta d'un nombre aleatori, això ho realitza la funció *generateIdentifier()* i es mantindrà aquest identificador sempre que el controlador no es desconnecti.

A continuació el servidor ha d'obrir un nou port UDP aleatori per continuar amb la connexió del controlador. Podem deixar que el propi sistema operatiu s'encarregui d'obrir un port que estigui disponible, simplement no hem d'especificar quin port volem obrir per la connexió:

```
newAddress->sin_family = AF_INET;
newAddress->sin_addr.s_addr = htonl(INADDR_ANY);
newAddress->sin_port = 0;
```

Seguidament crearem un nou *socket* i el vincularem a l'adreça creada.

```
/* Create UDP socket */
if((newUDPSocket = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    lerror("Error creating UDP socket.", true);
}
/* Bind the socket to the address */
if(bind(newUDPSocket,newAddress, sizeof(*newAddress)) < 0) {
    lerror("Error binding UDP socket", true);
}
```

Pel que fa a la continuació del procés de subscripció es molt similar a la implementació del client doncs solament ha de realitzar comprovacions de la informació i enviar les dades requerides:

1. Atendre petició de subscripció SUBS_REQ.
2. Comprovar credencials.
    1. Si envia dades correctes anar al pas 3.
    2. Si les dades no son correctes enviar rebuig de subscripció finalitzant comunicació i desconnectant el client.
3. Generar identificador i nou *socket* per continuar amb el procés de subscripció.
4. Esperar paquet d'informació SUBS_INFO.
    1. Si no es rep res finalitzar comunicació i desconnectar controlador.
    2. Si es rep paquet i les credencials son correctes anar al pas 5, sinó enviar rebuig de subscripció.
5. Enviar confirmació de subscripció INFO_ACK emmagatzemar dades del controlador i canviar el seu estat a subscrit.

4.2.5 Comunicació Periòdica

El procés de subscripció s'inicia un cop el paquet rebut pel port UDP ha estat verificat que es tracta d'un controlador autoritzat i que aquest es troba en l'estat subscrit o en comunicació periòdica.

El seu procediment es basa en:

1. Rebre paquet HELLO.
2. Comprovar credencials.
    1. Si són correctes continuar al pas 3.
    2. Si són incorrectes enviar paquet de rebuig HELLO_REJ.
3. Crear nou paquet HELLO amb les dades del servidor.
4. Enviar nou paquet HELLO.

La implementació de codi són diverses comparacions i crear paquets, no val la pena explicar en detall com s'han implementat comparacions de variables.

Un altra tasca que ha de realitzar pel manteniment de la comunicació periòdica és actualitzar els temps entre paquets que es reben de cada controlador. Aquesta comprovació la realitza el procés principal *main* i funciona de la manera següent:

1. Iterem per tots aquells controladors que estiguin en comunicació periòdica.
2. Comprovem el temps de recepció de l'últim paquet rebut.
    1. Si el temps supera l'establert pel protocol, 6 segons que representen 3 paquets no enviats, desconnectem controlador.
    2. Sinó, no fem res.

El codi per implementar aquesta funcionalitat utilitza les funcions *time* per obtenir els temps dels paquets i guardar-ho en una nova variable dins de la estructura del controlador:

```
if (current_time - controllers[i].data.lastPacketTime > 6)
```

I cada vegada que rebem un paquet correcte actualitzem aquest valor.

```
controller->data.lastPacketTime = time(NULL);
```

4.2.6 Recepció de Dades

La recepció de dades es realitzarà mitjançant el port TCP de la configuració del servidor i es crearà un procés concurrent per atendre-la. El procediment de recepció és el descrit a continuació:

1. Esperar paquet.
    1. Si no es rep paquet finalitzar comunicació.
    2. Si es rep paquet i és del tipus SEND_DATA seguir al pas 2.
2. Comprovar informació del paquet.
    1. Si existeix alguna discrepància tancar connexió i desconnectar al controlador.
    2. Si tot es correcte continuar al pas 3.
3. Guardar informació del paquet.
4. Enviar paquet de confirmació o problema amb les dades en cas de no poder guardar la informació.

El codi per implementar aquesta funcionalitat tornen a ser moltes comprovacions dels tipus i crear els missatges corresponents. S'utilitza una funció apart per guardar les dades a disc amb el format de paquet especificat:

NOM_CONTROLADOR-SITUACIÓ_CONTROLADOR.data

```
sprintf(name,"%s-%s.data",contrl->name,contrl->situation);
```

I les dades dins d'aquest arxiu:

DATA,HORA_RECEPCIÓ,TIPUS_PAQUET,DISPOSITIU,VALOR

4.2.7 Petició de Dades

La petició de dades es realitzarà mitjançant comandes del usuari per la terminal del servidor. Existeixen dos tipus de comandes:

- ***set***: Modifica el valor del actuador especificat.
- ***get***: Demana al Controlador especificat la informació associada al dispositiu.

El tractament del format de les comandes es realitza en una funció específica pels arguments d'usuari i les funcions associades amb petició de dades assumeixen que el format, mides i nombre de paràmetres, ja sigui correcte. Una vegada executada la petició es realitzaran una sèrie de passos:

1. Establir Connexió amb el client. Crear *socket*, vincular-lo amb el port del controlador etc.
2. Si la connexió es correcta es procedirà a enviar el paquet d'informació SET_DATA per la comanda *set* o GET_DATA per la comanda *get.*
3. Esperar resposta del controlador.
    1. Si es reben credencials incorrectes es finalitzarà la comunicació desconnectant al controlador.
    2. Si les credencials són correctes però no es rep confirmació es finalitzarà la comunicació.
    3. Si les credencials són correctes i s'ha confirmat les dades anar al pas 4.
4. Emmagatzemar les dades en disc i finalitzar comunicació amb el client.

La funció per emmagatzemar les dades serà la mateixa que la utilitzada en la recepció de dades ja que ha de disposar del mateix format.

5. Observacions

Aquesta pràctica ha suposat un repte a nivell de programació en el sentit de crear un projecte nou des de zero. La majoria de pràctiques realitzades fins aquest curs solen ser molt pautades completant implementacions d'altres funcions o realitzar programes relativament senzills.

En aquest cas el projecte era molt més complex i tenia que disposar de múltiples funcionalitats interconnectades entre elles. Quan es treballa amb projectes més grans s'ha de mantenir una bona organització tant a nivell de codi com estructura i plantejament de com s'anirà implementant cada funcionalitat. Es de gran ajuda afegir comentaris descriptius i complets per cada apartat més complicat, juntament amb una sèrie de diagrames i explicacions del que ha de realitzar.

En la majoria de seccions sobre la implementació del codi no s'ha mostrat com es realitzen les comprovacions dels atributs de paquets, client, servidor etc pel simple fet que son comparacions de cadenes de caràcters o d'altres tipus i no mereixen molta menció en les justificacions de les implementacions, per entrar més en detall es recomana fer un seguiment del codi de les funcions relacionades amb aquests aspectes.

També destacar la complexitat que suposa treballar amb C a aquests nivells, on els problemes de memòria ja suposen un repte al treballar amb punters, adreces i sobretot *threads* amb variables que poden ser accedides de forma simultània. Per sort existeixen diferents eines que han ajudat molt a trobar males pràctiques o implementacions en el tractament de memòria, especial menció a l'eina "valgrind" tant per ajudar en els *memory leaks* [19] com en els problemes de sincronització dels *threads*.

```
Exiting via SIGINT...
==3961==
==3961== HEAP SUMMARY:
==3961==     in use at exit: 0 bytes in 0 blocks
==3961==   total heap usage: 75 allocs, 75 frees, 32,432 bytes allocated
==3961==
==3961== All heap blocks were freed -- no leaks are possible
==3961==
==3961== For lists of detected and suppressed errors, rerun with: -s
==3961== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

[19] Resum del anàlisi de memòria realitzat per l'eina valgrind mostrant que no existeix cap error en accessos ni reserves de memòria.