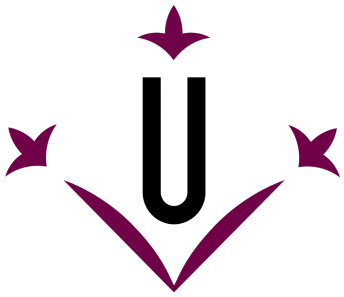

Universitat de Lleida

UNIVERSITAT DE LLEIDA

ESCOLA POLITÈCNICA SUPERIOR, EPS

GRAU ENGINYERIA INFORMÀTICA

Pràctica 1:

Programació d'aplicacions de xarxa

Març-Abril Curs 2022-2023

Professorat:
César Fernàndez, Enric Guitart,
Carles Mateu, Paula Gallucci
Autor:
David Carreras Castañ

Índex

Índex de figures

Introducció

La finalitat d'aquest informe és explicar el desenvolupament de la part d'un sistema de gestió de configuració usant el model de comunicació client-servidor. On el programari que es desenvolupa a la part del client seria la que s'installaria a l'equip de xarxa i que proporcionaria noves ordres per a la gestió de la configuració. D'altra banda, el programari del servidor s'hauria d'instal-lar a un NMS (Network Management Station) afegint noves funcionalitats per a la gestió de configuració.

Estructura del Client i Servidor

Diagrama de blocs del Client

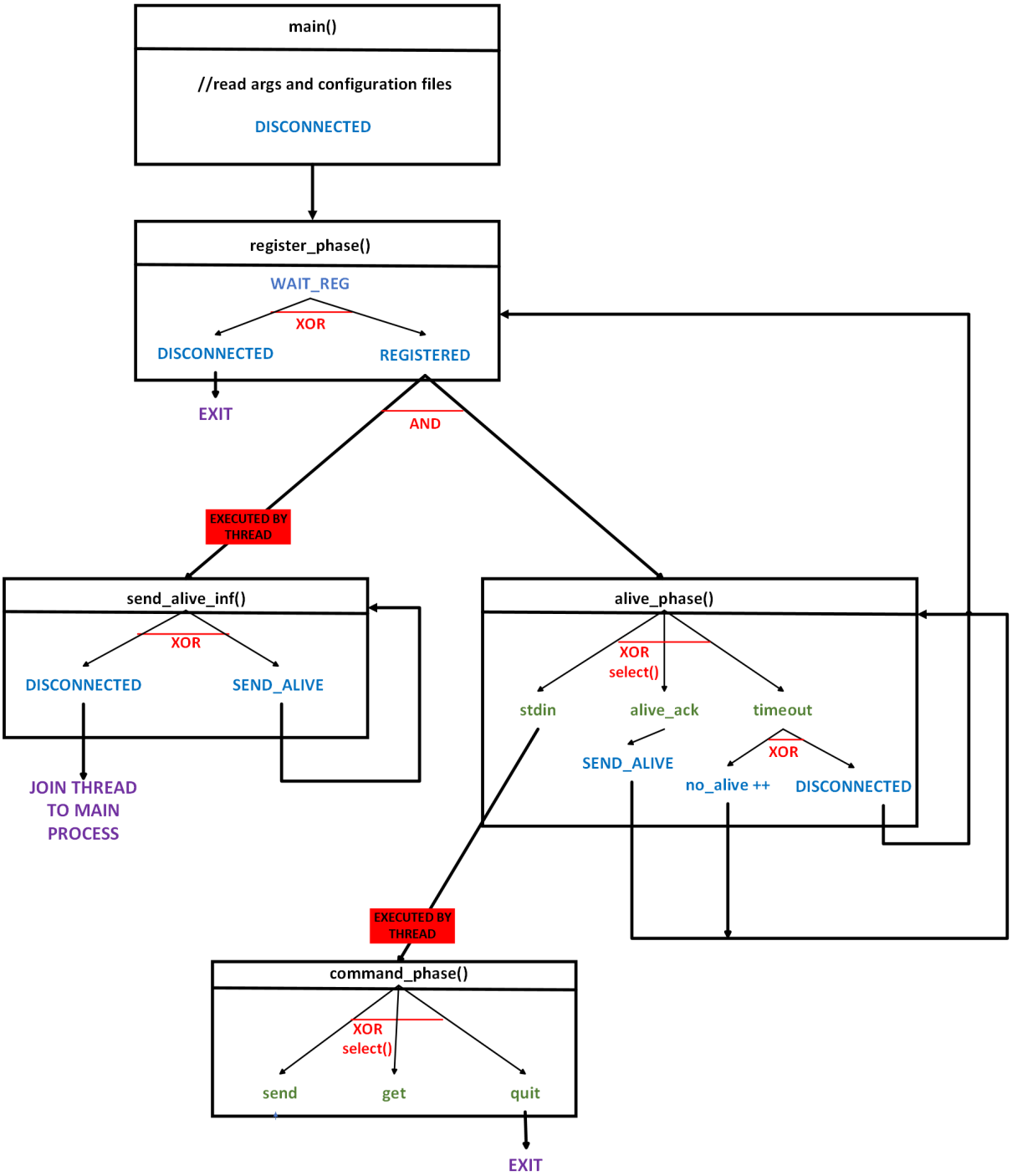

Figura 1: Diagrama de blocs del client

Diagrama de blocs del Servidor

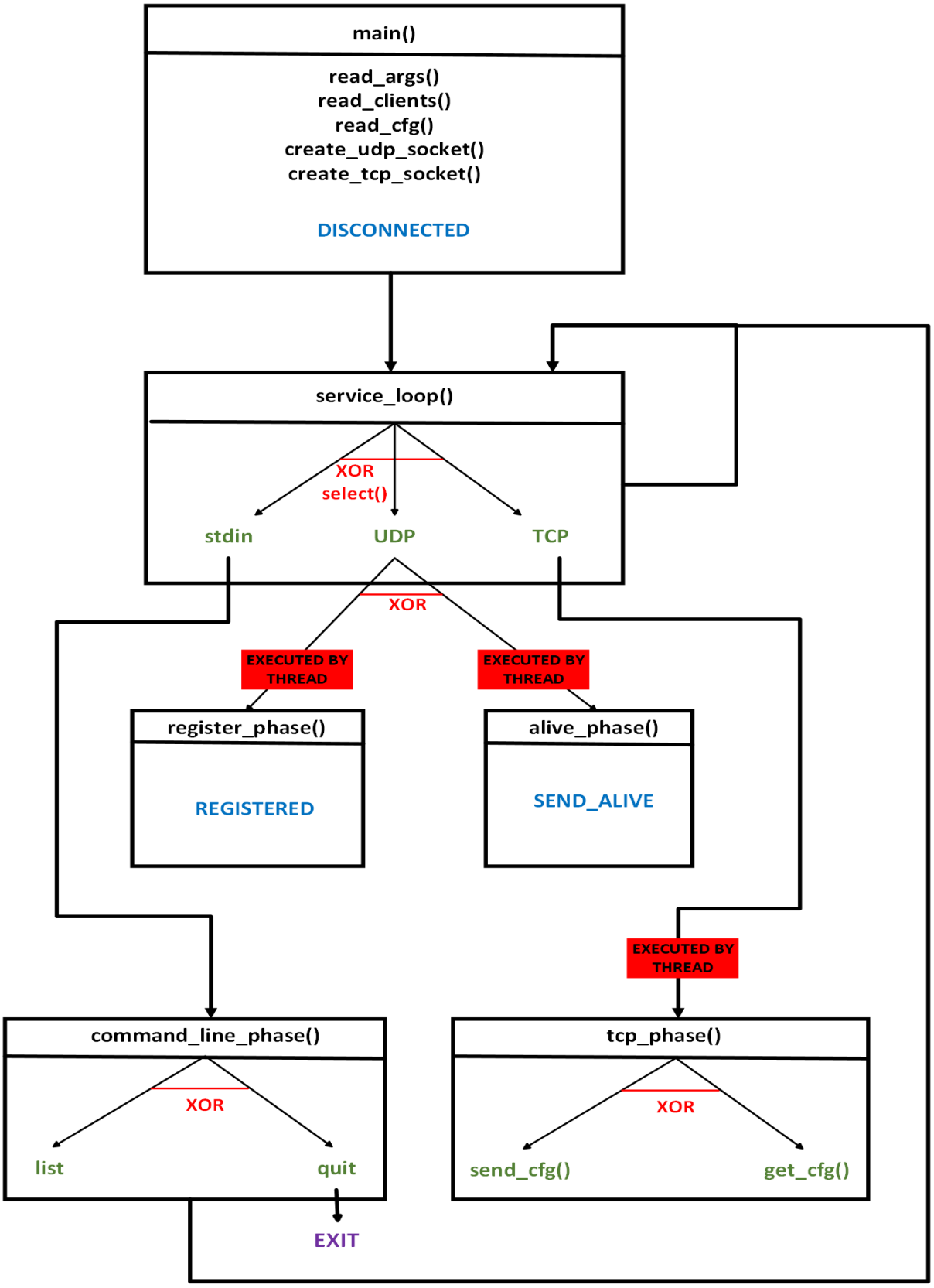

Figura 2: Diagrama de blocs del servidor

Manteniment de la comunicació

Estratègia emprada al Client

Per dur a terme el manteniment de la comunicació al client, primer s'ha d'haver realitzat amb èxit la fase de registre. Un cop l'equip ha canviat de l'estat DISCONNECTED a SEND_ALIVE, s'ha realitzat el manteniment de la comunicació amb dues funcions:

Una funció `send_alive_inf()` executada per un thread que s'encarrega d'enviar ALIVE_INF cada R segons i una segona funció `alive_phase()` executada pel fil principal d'execució que s'encarrega de rebre paquets. Aquesta segona funció, dins d'un bucle, monitoritza els descriptors de fitxer (sock_udp, stdin) sota un `select()` amb un timeout de R segons en cas de no rebre resposta.

Amb la finalitat d'evitar problemes en les temporitzacions de l'enviament i la recepció d'ALIVE_INF i ALIVE_ACK, s'ha establert una variable global status perquè el thread enviï alives sempre que l'status no sigui DISCONNECTED. D'aquesta manera, la funció de recepció de paquets ja s'encarrega de controlar si s'estan rebent ALIVE_ACK correctament cada R segons.

Estratègia emprada al Servidor

Per realitzar el manteniment de la comunicació al servidor, un cop rebut un paquet REGISTER_REQ per part d'un client, aquest problema s'ha fraccionat en dues parts:

D'una banda, crearem un fil que simplement executarà una funció `alive_phase()` que s'encarregarà de respondre a tots els paquets del tipus ALIVE_INF i contestar adequadament.

D'altra banda, al fil d'execució principal del programa amb la funció `check_timeout()`, comprovarem els timeouts de cada client. Per resoldre aquest problema, s'ha creat una estructura de dades en forma de classe, on dos atributs de cada client ho monitoren. Un guarda la data de l'últim ALIVE_INF rebut (`last_received`) i per altra banda un comptador del nombre d'alives no rebuts(`non_received_alives`). Aquesta funció, a cada passada del bucle de servei, controlarà els timeout del client així com el nombre de paquets no rebuts.

Diagrama d'estats UDP

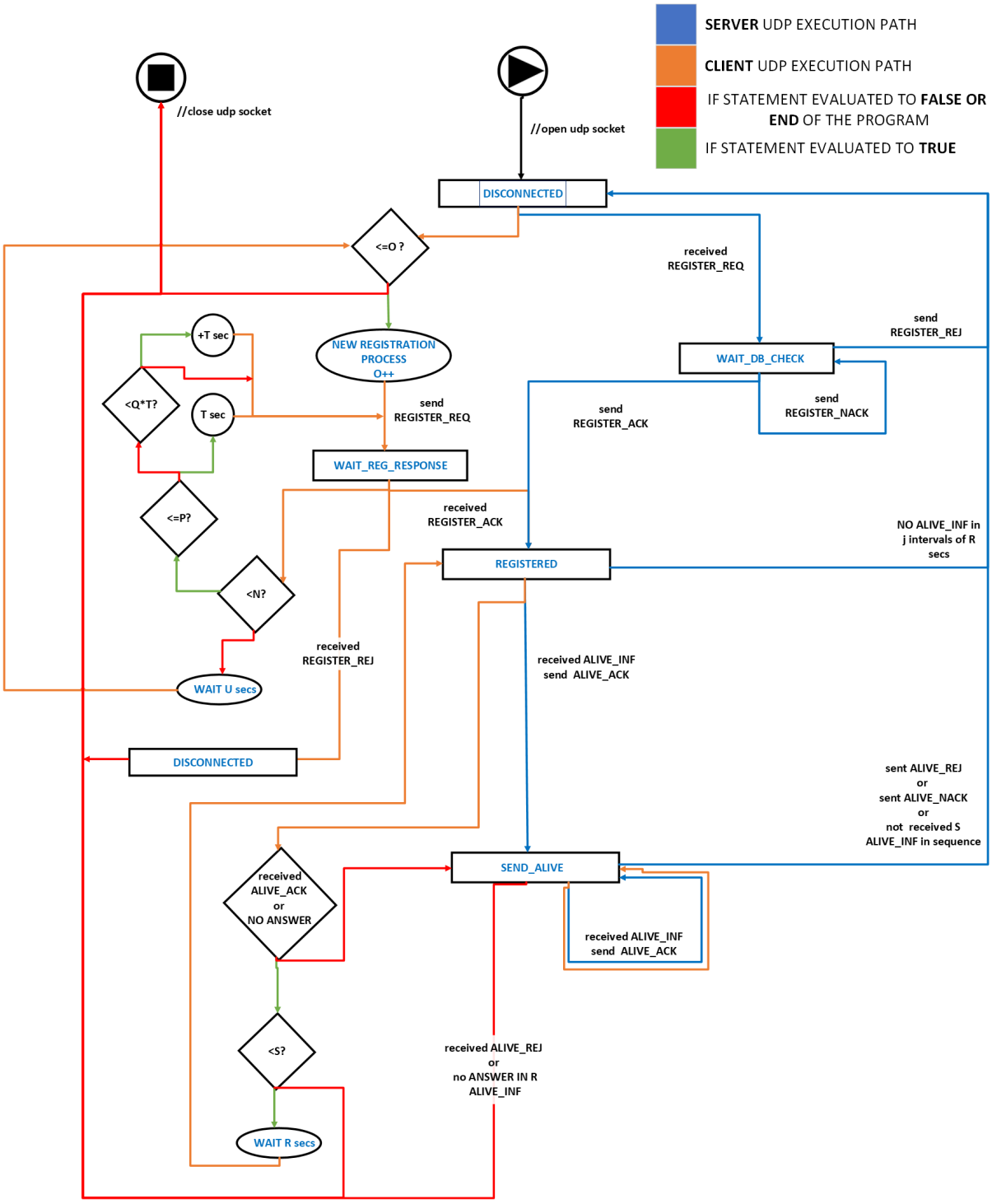

Figura 3: Diagrama d'estats del protocol implementat sobre UDP

Consideracions

Bucle en el test 3 del client

Al test 3 del client, quan s'acaba la seqüència de no enviar ALIVE_ACK als paquets 8,9,10,11, el client passa a l'estat DISCONNECTED i s'inicia un nou procés de registre. Com que en el programa facilitat pel professorat, en aquest cas no es considerava que calia incrementar el nombre d'intents de registre, s'ha implementat de la mateixa manera.
`-t 3: No s'envia [ALIVE_ACK] als paquets [ALIVE_INF] 2, 3, 5, 6, 8, 9,10....`
Això fa que estiguem en un bucle on el nombre d'intents de registre serà 1 a cada nou procés de registre. Per solucionar això, podria haver-se declarat la mateixa variable que porta el comptador del nombre d'intents de registre com a global, de manera que quan tornem a trucar a la funció per iniciar un nou procés de registre, el valor no es resetés a 1 de nou i per tant quedaria corregida aquesta consideració.

Fase de comandes i timeouts al servidor

En una primera implementació, la funció `command_line_phase()` es va implementar amb un fil d'execució com s'ha fet per `alive_phase()` o `tcp_phase()` i el seu funcionament era correcte, però després de nombroses proves, no va caldre la creació d'un fil. Això és perquè aquesta fase tan sols ha d'executar les ordres `list` o `quit`, el cost de les quals és negligible, per la qual cosa finalment es va implementar perquè fos executat pel fil principal dins del `service_loop()`.

Una cosa semblant passa amb la funció `check_timeout()`, que també és executada pel fil principal de programa a cada passada del `service_loop()`. Però aquesta, a diferència de la `command_line_phase()`, a petita escala com és el nostre cas no cal crear un fil, però si es decidís escalar el servidor per rebre milers de connexions el cost d'aquesta funció és textbf*O(n)*. Això es tradueix que hauríem de crear un fil si esperem rebre moltes connexions, ja que podria produir timeouts als clients i fins i tot errors al sistema si tan gran fos el nombre de clients.
Això és una manera d'evitar l'ús innecessari de recursos per a tasques que no requereixen un gran cost i que s'executen gairebé de manera instantània. Resulta molt important avaluar el cost de les funcions en base a l'ús que li donarem, ja que això pot significar una millora en el consum de recursos del programa, o per contra, produir errors al sistema.

Outputs

Tant al client com al servidor, s'han considerat 3 tipus d'outputs:

1. `ERROR:` Errors en l'execució o el funcionament d'alguna funció, es mostraran sempre
   (Exemple: `No es pot fer el bind amb el socket UDP`)
2. `INFO:` Missatge d'informació, es mostraran sempre.
   (Exemple: `Acceptat registre. Equip: id=Sw-001, ip=127.0.0.1, mac=23F474D2AC67 alea=000000`)
3. `DEBUG:` Missatges que es mostren només quan s'activa el mode debug tal que: `./client -d`.Utilitzats per informar amb més detall a l'usuari sobre tots els esdeveniments que estan passant.
   (Exemple: `Finalitzat procés per gestionar comanda sobre arxiu configuració`)

Conclusions

El desenvolupament de la pràctica ha estat tediós pel fet que en un primer moment no tingués gaire clar com ni per on començar a afrontar el problema. En primera instància es va decidir implementar la lectura d'arxius de configuració i els paràmetres per començar a agafar la dinàmica del desenvolupament. Després d'això es van adquirir i treballar els conceptes de sockets, protocols UDP i TCP, així com el funcionament i la implementació del processament multifil. A més va caldre familiaritzar-se amb el treball de les macros dels descriptors de fitxer per a funcions com a select() tant a nivell del client, com a servidor pel que fa al select().

A poc a poc, després d'haver adquirit aquests coneixements, la pràctica va ser pràcticament qüestió de lectura, comprensió i implementació del protocol descrit a l'enunciat. A més que a mesura que es van provar els tests, va resultar molt útil disposar de l'executable facitilat pel professorat per comprendre de manera més pràctica i visual el producte que es volia desenvolupar.

Segurament, si hagués de destacar la part més complexa de la pràctica m'arriscaria a dir que el més complicat és el disseny de l'estructura del client i del servidor, mostrats a les figures Figura 1 i Figura 2, realitzats prèviament a la implementació.

El treball realitzat en aquesta pràctica ha estat molt útil per entendre com funciona i què hi ha darrere d'una estructura client-servidor, cosa que està present constantment en el nostre dia a dia encara que no reflexionem sobre el seu funcionament. A nivell de programació, s'han adquirit nous conceptes com ara les estructures, threads, programació de sockets i diccionaris en el cas de Python.