PRÀCTICA 1

Programació d'aplicacions de xarxa

Xarxes

Grau en Enginyeria Informàtica

Universitat de Lleida

Data d'entrega: 17 d'abril de 2022

Autora: Laura Haro Escoi

Índex

Taula de figures

Introducció

En aquest informe es descriuran els diferents passos seguits per a realització de la primera pràctica de l'assignatura de Xarxes del curs 2021/22.

L'objectiu principal és exposar les idees prèvies al seu desenvolupament i els esquemes i estructures que s'han seguit per a la seva correcta implementació final.

També es descriuran els problemes que s'han presentat durat el procés i quines solucions s'han trobat i els seus motius.

Finalment, les conclusions a les quals s'ha arribat un cop finalitzada la pràctica i comentaris addicionals.

Model servidor-client

Per a la correcta implementació d'aquesta pràctica, tenir el concepte del model client-servidor és clau.

Aquest model estableix el procediment que han de seguir un client i un servidor per a comunicar-se. En el nostre cas, aquesta comunicació es farà de forma concurrent, cosa que provoca que un servidor reculli cada una de les peticions de servei i creï altres processos perquè s'encarreguin d'atendre-les. La seva estructura és la següent:

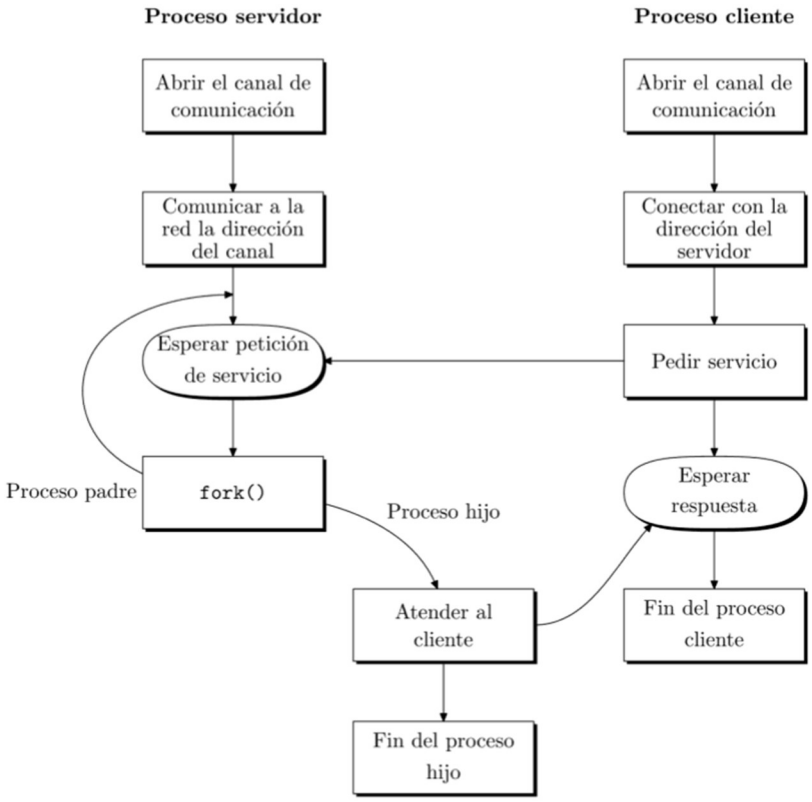

Figura 1: Estructura del model servidor-client concurrent

Sabent l'estructura que ha de seguir el model servidor-client concurrent, podem trobar diferents maneres de programar l'aplicació de xarxa segons els tipus de protocol que vulguem seguir. En aquesta pràctica s'ha d'aplicar el protocol UDP (User Datagram Protocol) i el protocol TCP (Transmission Control Protocol).

UDP

El protocol UDP no es orientat a connexió, ja que no estableix una connexió abans d'enviar les dades a traves de la xarxa per a comunicar-se. Aquest tipus de protocols segueixen aquest esquema a l'hora de programar-se:

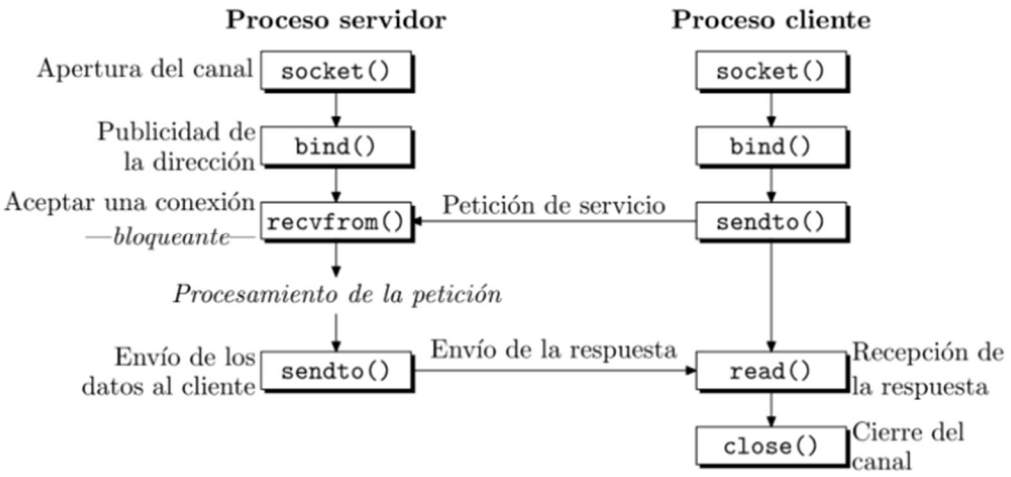

Figura 2: Seqüència de crides per a una comunicació no orientada a connexió

TCP

El protocol TCP es un conjunt de protocols orientat a connexió que assegura l'entrega de paquets de dades al seu destí i la transmissió segura de cada capa dels paquet. Aquest tipus de protocol segueix l'esquema següent:

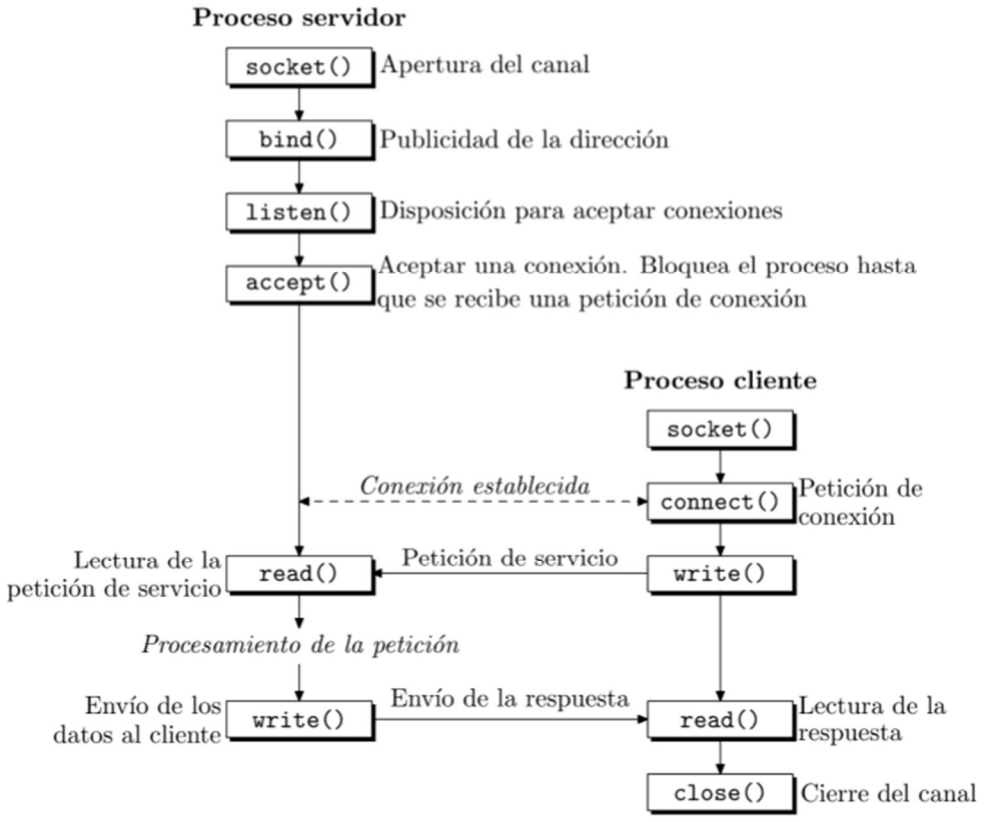

Figura 3: Seqüència de crides per a una comunicació orientada a connexió

Desenvolupament

Seguint les recomanacions del professorat, he començat desenvolupant el client. Cal dir que aquesta part es la que més temps ha ocupat a causa de tot el procés d'aprenentatge previ que comporta començar a desenvolupar aplicacions de xarxa per primera vegada.

Client

El procés de la implementació del client es pot dividir en 5 fases progressives:

- Registre
- Manteniment de la comunicació
- Introducció de comandes bàsiques
- Enviament de dades al servidor
- Recepció de dades del servidor

En el meu cas, he arribat fins la 3a fase, completant així tot el procés del protocol UDP.

El client utilitza diferents tipus de paquets i d'estats per a poder implementar el protocol sense problemes. En el següent diagrama d'estats es pot observar que segons quin tipus de paquet el client envia o rep, el seu estat canvia.

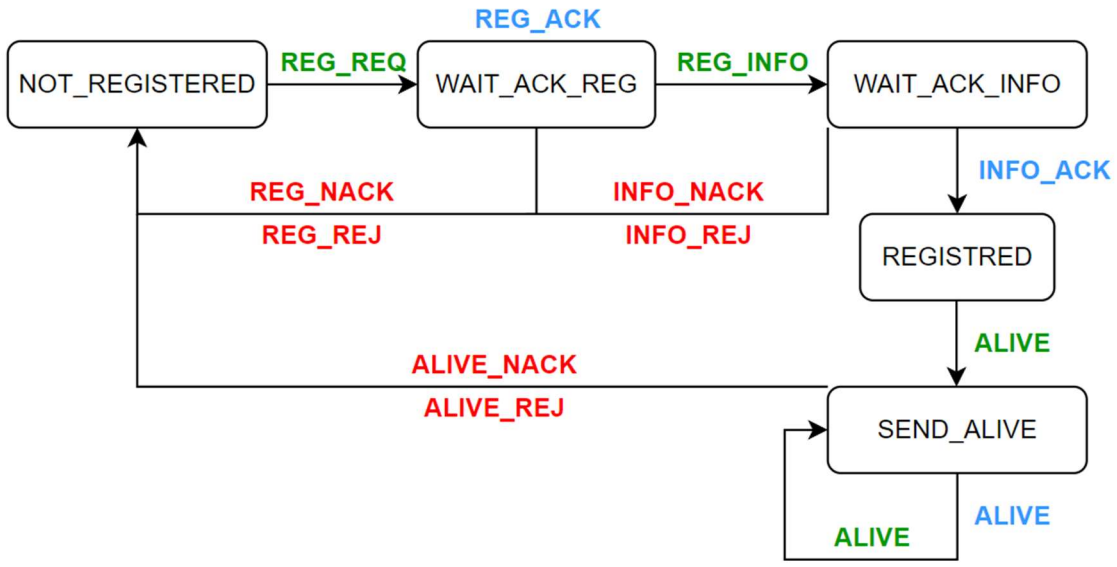

Figura 4: Diagrama d'estats del client

En verd tenim els paquets que envia correctament el client, el blau els paquets que rep correctament i en vermell els paquets que rep que aturen el procés de registre i el manteniment de la comunicació entre el client i el servidor.

Per a gestionar el registre i la comunicació he fet us de diferents funcions junt amb un *switch* per a determinar que ha de fer el client segons el estat en el que es trobi.

Primerament, llegeix quin tipus d'argument se li passa a l'hora d'iniciar-se, per a que sàpiga quin fitxer llegir o si ha de fer debug o no.

A continuació, fa us d'una sèrie de funcions per a llegir els paràmetres d'entrada i configurar els structs adients i els sockets. A partir d'aquest moment, s'entra a dins d'un bucle que mentre no es desconnecti el client, aquest seguirà intentant registrar-se al servidor.

Dins d'aquest bucle tenim el *switch* esmentat anteriorment on primer entrem a cas NOT_REGISTERED. Aquí es configura la pdu per a enviar un REG_REQ al servidor amb l'ajuda d'una funció auxiliar anomenada *send_reg_req()*. Aquesta és la que gestiona els diferents cicles de registre del client i rep la resposta del servidor. Depenent de la resposta, l'estat del client canviarà o no. Això ho comprova amb la funció *check_package(),* la qual determina l'estat del client segon el tipus de paquet que li arribi. Dins d'aquest cas d'estat, també està contemplat el fet de que arribi un REG_REJ, per a reiniciar el procés de subscripció del client al servidor.

Seguidament tenim el cas WAIT_ACK_REG en el que fem un *recvfrom* i comprovem quin tipus de paquet ha arribat (per seguir amb el procés hauria d'arribar un REG_ACK) i en quin estat està el client a causa d'aquest. Si l'estat és correcte, enviem un REG_INFO i el client canvia el seu estat a WAIT_ACK_INFO.

En el cas WAIT_ACK_INFO s'utilitza la funció *config_select()* en la que es configura un select amb el timeout determinat. Aquesta utilitza el struct *timeval* de la llibreria <sys.time.h> per a poder contar els segons que passen abans de que arribi un paquet.

Si aquesta funció en determina que no ha arribat res en el temps estipulat, l'estat del client canvia a NOT_REGISTERED. En canvi, si tot va bé, rep el packet INFO_ACK i passa a l'estat REGISTERED.

Una vegada a l'estat REGISTERED, ja podem començar a enviar paquets ALIVE per a mantenir així la comunicació entre el client i el servidor. Per a poder fer-ho he creat una funció *send_alives()* en la que s'envia un paquet ALIVE i s'espera V segons a rebre una resposta. Si aquesta resposta no arriba, s'augmenta un contador per a portar un control sobre si es van enviant ALIVES de forma consecutiva. Quan aquest arriba a 3, vol dir que el servidor no ha contestat 3 vegades i per tant, s'inicia un nou procés de registre.

Aquesta funció d'aplica la primera vegada al cas REGISTERED, però quan el client passa a SEND_ALIVE, abans d'entrar a questa funció obra un thread per a poder llegir les comandes que li puguin arribar pel terminal. Aquestes poden ser *stat i quit.*

Quan el client passa a estat DISCONNECTED, es tanca el client i els seus processos.

Servidor

El procés de la implementació del servidor també es pot dividir en 5 fases progressives:

- Registre
- Manteniment de la comunicació
- Introducció de comandes bàsiques
- Enviament de dades al servidor
- Recepció de dades del servidor

En el meu cas, he arribat fins la 3a fase, completant així tot el procés del protocol UDP.

En el següent diagrama d'estats es pot observar que segons quin tipus de paquet el servidor envia o rep, el seu estat canvia.

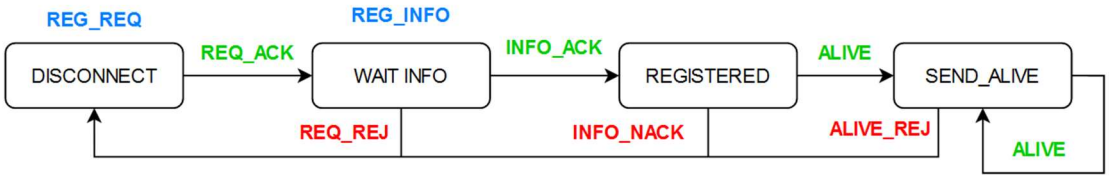

Figura 5: Diagrama d'estat del client

Per gestionar els registres i la comunicació concurrent entre els diferents clients que es puguin connectar al servidor, he utilitzat threads per a crear processos fills que atenguin a les diferents peticions dels clients.

Primerament he llegit els fitxers de configuració adients i he creat els sockets per a comunicar-me amb els clients.

A continuació obro un thread per a gestionar les peticions UDP de registre que puguin arribar. També obro un bucle per a poder llegir per comanda el que pugui arribar una vegada establert el estat SEND_ALIVE en un client.

Dins del thread obert, rebem el primer paquet i el derivem a un altre thread que gestionarà el registre del client.

Aquesta funció es la *UDP_process()*. En aquesta com que rebrem tant els paquets REG_REQ com els ALIVE, esta partida en aquest dos casos. Abans que saber quin tipus de paquet ha arribat, al tenir un servidor en python i un client en c, he creat unes funcions auxiliars per a transcriure les dades que arriben en un format que el servidor pugui llegir. Aquestes funcions son *UDP_decoder()* i *UDP_encoder()*. Així, cada vegada que arribi un paquet el desxifrarem i quan haguem d'enviar un paquet el xifrarem.

Quan ja hem tractat la pdu, mirarem si el client que ens està sol·licitant un registre està autoritzat. Si no fos així, enviaríem un paquet REG_REJ.

Si ha arribat un paquet REG_REQ i les dades son correctes, generarem un nombre aleatori amb la funció *randint()* de la llibreria random i la guardarem com a id de comunicació del client que estem atenent. A continuació crearem

un socket nou per el que atendrem la resta del procés de registre i anirem a la funció *wait_info()*.

En aquesta funció el primer que aplicarem serà un select de temps Z, i si arriba a temps un paquet, mirarem si es un REG_INFO. Si es donés el cas i totes les dades del paquet fossin correctes, enviaríem el INFO_ACK i registraríem al client. Si no es donés cap d'aquest cas, el client passaria a estat DISCONNECTED.

Ara que tenim el client registrat, el primer paquet de ALIVE arribarà pel primer socket que hem creat, i allí atendrem el paquet. Una vegada arribat el primer ALIVE, l'estat del client canviaria a SEND_ALIVE.

Quan s'arriba en aquest estat, ja es poden utilitzar les comandes *list i quit* per la línia de comandes.

Comentar que, per a guardar les dades dels clients atesos, he creat un diccionari en el que la clau es la id del client en qüestió i allí podem guardar totes les dades del client que ens facin falta com, la seva id de transmissió, el seu estat, etc.

Comentaris i conclusions

Al llarg del desenvolupament de la pràctica m'he trobat amb bastants problemes relacionats amb la falta de coneixements de la programació d'aplicacions de xarxa.

Mai havia utilitzat threads ni sockets però, he pogut aprendre molt sobre el seu funcionament.

També he tingut problemes a l'hora interpretar la informació que arribava del client al servidor, ja que arribava amb un format que python no sabia interpretar i he hagut de buscar la manera de desxifrar aquesta informació.

Malauradament, no he sigut capaç d'aplicar el protocol TCP i tot el que aquest comporta com l'enviament d'elements entre el client i el servidor, però he aconseguit els requisits que es demanaven per a l'entrega d'aquesta pràctica.

Bibliografía

Les figures 1, 2 i 3 formen part del llibre:

Márquez García, F. M. (1993). *Unix: programacion avanzada.* Grupo Editorial RA-MA.