UNIVERSITAT OBERTA DE CATALUNYA

Treball de Fi de Carrera (Xarxes)
ETIG - Enginyeria Tècnica en Informàtica de Gestió
Nucli d'un servidor web

Memòria

uaSW

(un altre servidor web)

per Joan Ardiaca Jové
Tutor: David Carrera Pérez
20 de juny de 2010

Índex de continguts

Descripció del projecte....3
Avaluació de la utilitat....3
Avaluació de la usabilitat....4
Contextualització....5
Molt breu història dels servidors web....5
Panorama actual....5
El protocol HTTP....6
Pla de treball....7
Instruccions de compilació i ús....8
Compilació....8
Configurar el servidor....8
Iniciar el servidor....10
Anàlisi....11
Funcionalitat bàsica: servir recursos per HTTP en paral·lel....11
Altres funcionalitats....12
Disseny....17
Estructures de dades....17
Mòdul fil.c - La rutina principal (mestre)...20
Rutina main()....20
Rutina daemonitza()....22
Rutina preparar_socket()....22
Rutina crear_fil()....23
Rutina modificar_nombre_fils()....23
Rutina preparar_pthread_attr()....23
Altres rutines....23
Mòdul configuracio.c - Lectura d'arxius de configuració....24
Rutina llegir_configuracio()....24
Rutina llegir_tipus_mime()....24
Mòdul fil.c - Rutines d'atenció a les connexions (treballadors)....25
Rutina atendre_sobrecarrega()....25
Rutina atendre_connexio()....26
Rutina esperar_peticio()....29
Rutina analitzar_peticio()....30
Rutina processar_peticio()....31
Rutina enviar_arxiu()....34
Rutina enviar_error()....34
Rutina acabar_fil()....35
Rutina deduir_mime()....35
Rutina uncork()....35
Mòdul cgi.c - Execució de programes CGI.36
Implementació....40
Versió 0.1 (19 d'abril de 2010, lliurament PAC2)....40
Versió 0.2 (23 de maig de 2010)....41
Versió 0.3 (13 de juny de 2010, lliurament final)....42
Proves realitzades....43
Anàlisi de rendiment....48
Errors i millores realitzables....53
Conclusions....56
Agraïments....56
Fonts utilitzades....57
Apèndix A: Altres servidors web....58

Nota: aquesta versió de la memòria ha estat lliurada posteriorment a la data de lliurament. A part d'una revisió general dels continguts, s'inclouen alguns continguts que estaven pendents de ser redactats: l'anàlisi de rendiment, les proves realitzades, una descripció del mòdul CGI, l'apèndix i unes breus conclusions. També s'han afegit algunes fonts utilitzades que no es mencionaven en el lliurament anterior.

Descripció del projecte

Aquest projecte tracta el desenvolupament d'un servidor HTTP, incloent planificació, anàlisi, disseny i implementació. Aquest programari, que he anomenat uaSW (sigles de “un altre servidor web”), s'ha desenvolupat en el llenguatge de programació C per a entorns GNU/Linux. Té un funcionament multifil per al processament en paral·lel (fent ús de l'estàndard *POSIX threads*) que el fa relativament senzill i ràpid.

Ja que un programa d'aquest tipus pot arribar a ser summament gran i complex i, per tant, inabastable per a ser realitzat per una sola persona, no s'ha buscat oferir una gran quantitat de funcionalitats, un alt rendiment o ni tan sols ser completament conforme amb els estàndards que implementa, sinó la senzillesa, rendiment òptim i funcionalitat: eficàcia i eficiència abastable. El nom del programari, “un altre servidor web”, ja denota que no s'ofereix res d'especial ni s'innova davant de programari similar. Després de tot, seria una tasca difícil, tenint en compte que en podem trobar fàcilment desenes.

Avaluació de la utilitat

En la seva forma actual, el servidor ofereix diverses funcionalitats estàndard de qualsevol servidor web. Pot ser utilitzat per a servir tant documents i fitxers estàtics a través del protocol HTTP, de manera que es poden accedir des de qualsevol navegador que suporti la versió 1.1 del protocol; com continguts dinàmics generats per programes CGI, tot i que aquesta funcionalitat s'implementa per ara només de manera experimental.

El programa és bastant configurable per a ser adaptat (fins a cert punt) a usos i circumstàncies diferents. Les proves realitzades demostren que és capaç de suportar una càrrega considerable amb un rendiment acceptable, fet que el fa adequat per a servir recursos estàtics que són accedits a l'hora per diversos clients.

Si bé no es pot considerar vertaderament una funcionalitat del programa pel seu caràcter preliminar, el suport de l'estàndard CGI faria de uaSW un servidor molt més útil, ampliant les seves capacitats en el camp de la generació dinàmica de continguts, que és avui dia la forma preeminent en que es generen els llocs Web.

Cap al final d'aquest document veurem algunes possibles millores que es poden fer al servidor, que

expandirien les possibilitats que ofereix.

Avaluació de la usabilitat

Sent realista, espero que cap organització o particular utilitzi uaSW en entorns de producció, al menys en la seva forma actual. Tal i com està ara mateix, no es pot considerar programari suficientment fiable (ni tans sols prou segur, s'haurien de fer proves més exhaustives) ni ofereix grans funcionalitats (comparant-lo amb altres solucions), tot i que els meus esforços han estat importants per a assolir-ho.

Això és especialment cert tenint en compte que hi ha disponibles moltes solucions gratuïtes, comercials i/o lliures en gran quantitat i de gran qualitat que han demostrat la seva fiabilitat durant anys de funcionament en entorns de producció i per a les quals es publiquen amb freqüència noves versions i modificacions per a oferir mes funcionalitats, rendiment i seguretat. El cas per excel·lència és el de Apache, un servidor web lliure que ha contribuït de forma excepcional al desenvolupament i progrés de la xarxa de xarxes i avui és encara el programari d'aquest tipus més utilitzat.

L'ús de uaSW que si consideraria raonable seria en l'àmbit de l'educació com a mostra d'un programa que resol un problema en paral·lel, fent ús de la xarxa i per a entorns GNU/Linux. El programa és relativament senzill i funcional i aplica tècniques diverses amb objectius diversos, que, en la meva opinió, són dignes de ser apresos i compresos per qualsevol estudiant (i no estudiant) que vulgui conèixer de més a prop aquest món.

Contextualització

Molt breu història dels servidors web

L'any 1989, Tim Berners-Lee, mentre treballava al CERN, va iniciar un projecte per a facilitar l'intercanvi d'informació entre científics usant un sistema d'hipertext. Com a resultat, va néixer el que avui coneixem com a Internet o WWW (tot i que no és l'únic ús que se li dona, és el predominant). Va desenvolupar amb el seu equip els primers navegador i servidor web, coneguts com WorldWideWeb i httpd, respectivament. La seva idea va quallar i aviat el sistema s'expandia exponencialment. Avui, vint anys després, el resultat de les seves creacions és gairebé omnipresent en les societats dels països desenvolupats (i cada cop més en els països en desenvolupament) i s'utilitza per a tot tipus de propòsits en quant a la comunicació es refereix, des de les compres electròniques fins al mer entreteniment, passant pel treball col·laboratiu i l'obtenció de tot tipus d'informació. Això li ha valgut diversos reconeixements de tot tipus d'institucions (incloent el nombrament de Doctor Honoris Causa per la Universitat Oberta de Catalunya, entre molts d'altres) i és considerat el “pare de de la web”.

Si bé la Web ha canviat des dels seus inicis fins ara, tant en els usos que se li donen com en quina forma s'usa, els fonaments segueixen sent els mateixos. S'utilitza el mateix protocol, HTTP, per a transmetre sobretot documents en format HTML usant el sistema de localització URL.

Panorama actual

Els servidors web són una peça fonamental d'Internet tal i com el coneixem avui i existeixen multitud de programes que realitzen aquesta funció. Els programes més populars en aquesta categoria són:

- Apache: Ha tingut un paper clau en el creixement inicial de la WWW i és actualment el més usat. Desenvolupat de forma oberta i multiplataforma per la Apache Software Foundation, serveix avui més de 100 milions de llocs web.

- IIS (Internet Information Services): Servidor web comercial desenvolupat per Microsoft per al seu sistema operatiu Windows. Serveix uns 50 milions de llocs web.

- nginx: Un servidor web orientat a l'alt rendiment i l'ús de pocs recursos. Publicat sota una

llicència oberta per a diverses plataformes.

Aquests són només alguns exemples. A la xarxa podem trobar fàcilment desenes, si no centenars, de programes similars per a tot tipus de plataformes, escrits en llenguatges diversos i amb funcionalitats diferents. Per tant, en aquest projecte no hi ha cap ambició d'innovar. Més aviat es tractarà de desenvolupar un nou programa d'aquest tipus que sigui funcional, eficient, simple i ràpid.

El protocol HTTP

El protocol HTTP (*Hypertext Transfer Protocol*, protocol de transferència d'hipertext) és el protocol sobre el qual funciona la major part d'Internet com el coneixem avui. Es tracta d'un protocol a nivell d'aplicació que es basa en un sistema de petició-resposta sobre un model client-servidor. El client envia peticions a un servidor i obté una resposta, normalment contenint un document HTML (*Hypertext Markup Language*, llenguatge de marcat d'hipertext) però en la pràctica pot ser qualsevol tipus de recurs, ja sigui estàtic o dinàmic.

No es limita a l'obtenció d'informació. El client també pot enviar tot tipus d'informació al servidor, que aquest processa. Així, els usuaris poden pujar fitxers i informació de formularis, per exemple al utilitzar aplicacions de correu electrònic sobre aplicacions Web. Això implica normalment la generació de continguts dinàmics, és a dir, documents HTML o altres recursos que són generats en el moment de preparar la resposta HTTP, enlloc de uns documents fixes i prèviament definits.

En la comunicació pel protocol HTTP poden participar altres actors a part del client i el servidor. Així, poden haver-hi servidors *proxy*, cachés web o portes d'enllaç, destinats a comunicar xarxes diferents o a accelerar la comunicació i evitar la saturació dels servidors.

Pla de treball

Aquest és el pla de treball que s'ha realitzat a l'inici del projecte i que he procurat seguir com a orientació durant la seva realització.

- 8 març – 14 març: anàlisi de requisits i especificació dels casos d'ús
- 15 març – 21 març: disseny de l'aplicació
- 22 març: revisió del pla de treball
- 23 març – 30 març: implementació de la lectura d'arxius de configuració
- 31 març – 4 abril: proves de la a lectura d'arxius de configuració
- 5 abril – 14 abril: implementació del procés principal (mestre)
- 11 abril: lliurament PAC2
- 15 abril – 18 abril: proves del procés principal (mestre)
- 19 abril – 28 abril: implementació del codi dels treballadors
- 29 abril – 2 maig: proves del codi dels treballadors
- 3 maig – 12 maig: implementació CGI
- 13 maig – 16 maig: proves CGI
- 16 maig: lliurament PAC3
- 17 maig – 23 maig: proves de l'aplicació sencera
- 24 maig – 3 juny: temps reservat per a possibles imprevistos
- 4 juny- 12 juny: preparació de l'entrega de la memòria i del producte
- 13 juny – 19 juny: preparació de la presentació
- 21 juny – 27 juny: període de preguntes del tribunal

Instruccions de compilació i ús

El codi lliurat (situat en el directori `codi`) inclou el projecte NetBeans sencer. També s'inclou el programa en binari en el directori `binari`. Els documents `memoria.odt` i `memoria.pdf` són aquesta memòria que esteu veient.

Compilació

La forma més senzilla de compilar el programa és accedint al directori `codi` i executant la comanda `make`. Això generarà fitxers objecte en el directori `codi/build/Debug/GNU-Linux-x86`" i el fitxer binari `codi/dist/debug/GNU-Linux-x86/uasw`.

També podeu optar per obrir el projecte amb l'IDE NetBeans[1] i compilar-lo des del mateix. Això, de fet, seguirà el mateix procediment que executant la comanda `make`, però us oferirà també un entorn configurat com jo el tenia per a desenvolupar el programa i realitzar diverses tasques fàcilment, com cercar errors en el programa i analitzar el seu funcionament.

Configurar el servidor

El servidor carrega els seus paràmetres configurables d'un arxiu de configuració, per defecte de l'arxiu `uasw.conf` que es trobi al mateix directori que l'executable. S'inclou un arxiu de configuració exemple en el directori `configuracio` (noteu que utilitza el port 8080 per a rebre les connexions entrants enlloc del 80, pel fet que aquest últim requereix tenir drets de superusuari per a ser utilitzat). A continuació veiem les configuracions possibles, juntament amb una breu explicació de cadascuna i els valors que prenen per omissió (entre parèntesis).

- `PortEscolta` (80): Defineix el número del port TCP/IP pel qual el servidor esperarà rebre les peticions. Recorda que el programa haurà de tenir permisos de superusuari si es volen usar ports per sota del 1024.

- `TempsEspera` (60): Temps en segons que esperarà el servidor des que una connexió entrant fins a rebre una petició HTTP sencera. També és el temps que es manté una connexió persistent oberta sense tenir activitat.

1 NetBeans és un entorn integrat de desenvolupament (IDE) implementat en la seva major part en Java i publicat sota una llicència oberta. Ofereix diverses funcionalitats que faciliten el desenvolupament de programari en diversos llenguatges de programació. Podeu trobar-lo a http://netbeans.org/.

- `MaxFils` (250): Nombre de treballadors (fils d'execució) que es poden crear alhora. Limita també quantes peticions es poden atendre alhora, així com el nombre de connexions simultànies. El nombre màxim està imposat pel nombre de fils i de connexions/fitxers oberts que admeti el SO per procés i haurà d'estar sempre per sota de 500 per limitacions del programari. Un cop s'esgotin es faran fils de sobrecàrrega.

- `MaxFilsSobrecarrega` (40): Nombre de màxim de treballadors que es crearan alhora per a avisar als clients d'un estat de sobrecàrrega del servidor.

- `MidaCuaEntrants` (50): Connexions TCP/IP entrants pendents màximes (veure `man listen`).

- `DirectoriDocuments` (`/var/www`): Directori que conté els documents a servir.

- `ArxiuPortada` (`index.html`): Document que es servirà al accedir a un directori.

- Host (`localhost`): URL base del servidor.

- `DirectoriDocumentsError` (`/var/www/errors`): Directori que conté les pàgines al mostrar quan ocorre un error (p.e. 404.html).

- `CharacterSet` (utf-8): Character set dels documents basats en text.

- `RutaCGI` (`/cgi-bin/`): Ruta per a a accedir als scripts CGI, p.e. si es vol que la ruta sigui `exemple.com/cgi-bin/`, establir el valor a `/cgi-bin/`. La ruta ha d'existir dintre del directori de documents. Els arxius en aquest directori i els seus subdirectoris s'executaràn si tenen el bit d'execució activat, si no, es serviràn com qualsevol altre document.

- `LogErrors` (`errors.log`): Ruta del log d'errors. Tigueu en compte que els logs només s'usaràn si s'inicia en mode dimoni (amb el paràmetre `daemon`).

- `Log` (`uasw.log`): Ruta del log general.

- `MimeTypes` (`mime_types.conf`): Ruta de l'arxiu amb els tipus MIME dels arxius per extensió

En el directori `configuracio` també s'inclou un arxiu de tipus MIME. Aquest no hauria de necessitar modificació per al correcte funcionament del servidor.

Iniciar el servidor

Per a iniciar el servidor, tan sols heu d'executar el fitxer executable. Opcionalment, podeu indicar a través del primer paràmetre si executar el programa en primer o en segon pla (passeu el paràmetre `n` o `normal` per a executar-lo normalment, `d` o `daemon` per a executar-lo en segon pla). Amb el segon paràmetre, podeu indicar la ruta a l'arxiu de configuració que vulgueu usar. Per defecte s'utilitzarà l'arxiu `uasw.conf` que es trobi al directori de treball.

Codi 1 - Executar el servidor en segon pla

```
uasw daemon
```

Codi 2 - Executar el servidor en primer pla determinant un arxiu de configuració

```
uasw n /etc/uasw.conf
```

Anàlisi

Funcionalitat bàsica: servir recursos per HTTP en paral·lel

La funcionalitat i cas d'ús primaris d'aquest servidor web és el mateix que el de qualsevol programari d'aquestes característiques: servir recursos per la xarxa als clients a través del **protocol HTTP**[2], normalment documents HTML però també qualsevol altre tipus de documents i arxius.

Per a oferir aquesta funcionalitat, el programari ha de seguir diversos passos:

- Inicialment ha de **preparar un port TCP/IP**[3] (que és el protocol sobre el qual s'usa normalment l'HTTP), el 80 per omissió, per a poder rebre connexions entrants.
- El servidor ha de romandre a l'**espera de connexions entrants** dels clients.
- Quan rebi una connexió, llegirà les dades fins a obtenir una **petició HTTP** sencera.
- Processarà aquesta petició, comprovant que és correcta i que pot ser atesa correctament. Si no és el cas, respondrà amb el missatge d'error corresponent, informant al client de la situació. Si tot és correcte, **servirà al client el recurs** que ha demanat.

Seria un limitant molt important en quant a la usabilitat del servidor si només es pugés atendre a un client en cada moment. A més, la tasca dels servidors HTTP és un problema fàcilment resoluble aplicant **computació paral·lela**, ja que en principi cada petició és independent de les altres (no han de compartir cap tipus de dades) i té clarament fases intenses en temps de processador i altres d'entrada i sortida, implicant que si es fa en paral·lel el sistema operatiu pot aprofitar millor els recursos del computador.

En sistemes GNU/Linux existeixen tres solucions bàsiques a aquest problema: implementar una solució multiprocés (usant `fork()`), multifil (usant *POSIX threads*) o dirigida per esdeveniments. Utilitzar **múltiples processos** és una solució clàssica, viable, fàcilment implementable i és capaç de treure profit de diversos processadors si el sistema en disposa, però te com a desavantatge l'ús

2 Vegeu el document RFC 2616 (“Hypertext Transfer Protocol – HTTP/1.1”), on es recull l'estàndard HTTP 1.1, a http://tools.ietf.org/html/rfc2616

3 TCP/IP són les sigles de *Transmission Control Protocol/Internet Protocol*. És el protocol més usat en les transmissions per Internet.

intensiu de recursos, ja que s'han de copiar una gran quantitat de dades (tot el PCB[4] del procés). El programari **dirigit per esdeveniments** també és una opció (tot i que no és un cas de computació en paral·lel), que treu profit dels moments en que el SO fa operacions d'E/S realitzant altres operacions, però no pot treure profit de sistemes multiprocessador i és, al menys al meu parer, més difícil d'implementar. Finalment tenim la opció **multifil**, que ofereix els avantatges d'una solució multiprocés però amb un rendiment molt superior[5], major facilitat per a comunicar els diferents fils (ja que tots comparteixen el mateix espai de memòria) i una facilitat d'implementació notable. Aquestes tres solucions es poden combinar de formes diverses en un mecanisme híbrid.

Pels motius que acabem de veure, he optat per utilitzar la solució multifil oferta per l'estàndard **POSIX threads**. Per a divisió del treball entre els fils es segueix un **disseny mestre/treballadors**: sempre hi ha un fil "mestre" o principal que és el que s'encarrega en primera instància de preparar l'entorn per a entrar a continuació en el bucle principal. Aquest bucle espera connexions entrants per un port (per defecte el 80) i, quan en rep una, crea un nou fil "treballador" que s'encarrega de rebre la petició o peticions del client i tornar-li les respostes de manera independent, tant del mestre com dels altres treballadors. Quan el treballador ha acabat la seva tasca, finalitza la seva execució alliberant recursos.

No s'implementa el protocol HTTP sencer. Si be es tracta de seguir l'estàndard **HTTP/1.1** (recollit al document RFC 2616) en gran mesura, només s'implementen parcialment alguns mètodes bàsics, concretament HEAD, GET i POST. A més, el comportament del servidor pot no ser estrictament el que es defineix en l'estàndard, tot i que es seguirà en termes generals per tal d'assegurar una compatibilitat i funcionalitats òptimes.

Altres funcionalitats

uaSW no només compleix el seu objectiu bàsic d'oferir recursos a través del protocol HTTP, sinó que ofereix altres que el fan més usable, útil i milloren el rendiment. Veurem aquí quines són.

- **Fitxer de configuració**. Com tot programari d'aquestes característiques, uaSW té diversos

4 PCB són les sigles de *Process Control Block* (bloc de control del procés). És un resigtre on el sistema operatiu agrupa tota la informació que necessita conèixer sobre un procés particular.

5 Una crida a `fork()` pot trigar fins a 50 cops més que a `pthread_create()` en certs sistemes, ja que un fil només precisa les dades necessàries per a mantenir un flux d'execució independent i no un procés sencer amb el seu PCB. Així s'estalvia, per exemple, tenir un espai de memòria propi i la llista d'arxius oberts. Vegeu la secció "Why Pthreads?" del document "POSIX Threads Programming" (https://computing.llnl.gov/tutorials/pthreads/#WhyPthreads), on es realitza un anàlisi del temps de processador consumit per fork() i pthread_create() en diferents sistemes.

paràmetres que poden ser configurats per l'administrador del servidor, com el port TCP/IP pel qual espera rebre les connexions dels clients o en quin directori es situen els documents a servir. El programa llegeix un fitxer de configuració que estableix el valor d'aquests paràmetres, comprovant si són correctes i permetent l'existència de comentaris en el fitxer. Si no s'estableix cap valor per un paràmetre, s'utilitza un valor per omissió raonable. A través del segon paràmetre en el moment d'executar el programa es pot establir quin fitxer utilitzar; si no s'indica es llegeix el fitxer `uasw.conf` en el directori de treball actual per omissió.

- **Tipus MIME[6] configurables**. De forma similar al fitxer de configuració, el programa carrega una llista de tipus MIME d'un fitxer. Aquest fitxer relaciona extensions de fitxers amb tipus MIME, que el servidor utilitzarà per a deduir el tipus MIME dels recursos servits. Si no coincideix amb cap entrada d'aquesta llista, s'utilitza el tipus `text/plain` per omissió. La ruta d'aquest fitxer es pot establir en el fitxer de configuració.

- **"Aturada cortès"** (*graceful stop*). Al rebre el senyal `SIG_TERM`, el servidor s'aturarà immediatament, avortant qualsevol operació que estigui realitzant en aquell moment (principalment, atendre les peticions dels clients). Però si rep el senyal `SIG_INT` no s'aturarà immediatament, sinó que atendrà les peticions que s'estan atenent en aquell moment, sense acceptar-ne de noves, i finalitzarà l'execució quan no quedin peticions en procés. Així s'evita que els clients rebin respostes parcials a les seves peticions.

- **Execució com a servei** / "dimonització". Si bé existeixen altres mètodes per a assolir el mateix efecte, l'administrador por demanar al servidor que s'executi com un "dimoni" o servei a través del primer paràmetre. Si el primer paràmetre és `normal` o `n`, uaSW procedeix com qualsevol comanda, mostrant els esdeveniments que ocorren a través de la sortida estàndard `(stdout)` i la sortida d'errors (`stderr`). Si s'executa en mode dimoni (passant el paràmetre `daemon` o `d`), uaSW tracta d'executar-se en segon pla i desa els esdeveniments en fitxers de registre.

- **Connexions persistents**[7]. Una novetat de la versió 1.1 del protocol HTTP és el suport de

6 MIME són les sigles de *Multipurpose Internet Mail Extensions*. Consisteix en una serie de convencions dirigides a facilitar l'intercanvi de dades de tot tipus, especialment a través d'Internet. L'estàndard està especificat per sis documents RFC: RFC 2045, RFC 2046, RFC 4288, RFC 4289 i RFC 2077.

7 Vegeu la secció "8.1 Persistent Connections" al document RFC2616 (http://tools.ietf.org/html/rfc2616#section-8.1), on es detalla el funcionament d'aquest mecanisme.

connexions persistents. Es tracta de mantenir una connexió oberta, enlloc de tancar-la després de cada petició al servidor, que permeti l'enviament de diverses peticions amb una sola connexió amb l'objectiu de minvar l'efecte de la latència de la xarxa i evitar el cost que suposa l'establiment de diverses connexions innecessàries, ja que els clients solen fer diverses peticions seguides en un període curt de temps en cada sessió, per exemple en el cas de documents HTML amb estils en un arxiu CSS[8] i imatges.

- **Pipelining**. Estretament relacionat amb les connexions persistents, uaSW suporta el pipelining o "encanyeriament" de peticions, és a dir, permet que el client envii diverses peticions al servidor per una mateixa connexió sense esperar les seves respostes, que el servidor atendrà una rere l'altra. Això contribueix a reduir encara més l'efecte de la latència de la xarxa i pot suposar un augment dramàtic del rendiment de la comunicació.

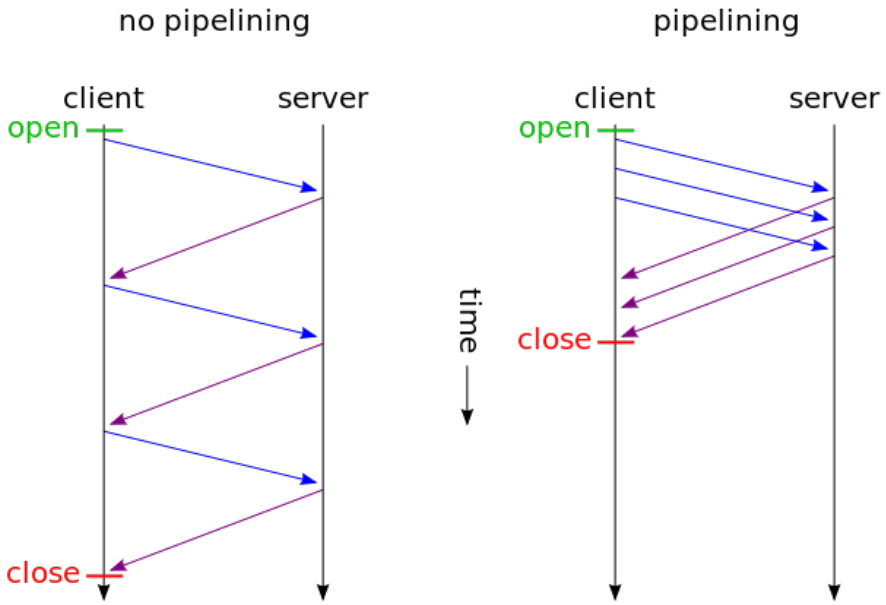

Diagrama mostrant el funcionament del pipelining. Il·lustra clarament l'augment del rendiment de la comunicació i l'efecte de la latència quan les peticions es fan sense utilitzar aquest mecanisme.
Font: http://en.wikipedia.org/wiki/HTTP_pipelining

- **Execució de programes CGI (experimental)**. uaSW permet de manera experimental l'execució de programes i scripts que generin continguts dinàmics de la manera recollida en l'estàndard CGI 1.1 al document RFC 3875[9]. Això s'assoleix iniciant un nou procés fill, al

8 CSS són les sigles de *Cascading Style Sheets* (Fulles d'estil en cascada). És un llenguatge usat per a definir la presentació i l'estil de documents i s'empra sovint en documents HTML.
9 Vegeu el document RFC 3875 ("The Common Gateway Interface (CGI) Version 1.1") a

qual se li posen a disposició les dades necessàries a través de variables d'entorn i l'entrada estàndard (`stdin`) envia les dades al client que emeti per la sortida estàndard (`stdout`), connectant-la al *socket* corresponent. Repeteixo que es tracta d'una funcionalitat experimental que no funciona correctament i té errors coneguts, per tant no espereu poder usar aquesta funcionalitat amb normalitat.

- **Avís de sobrecàrrega**. Quan és a punt d'esgotar els seus recursos (principalment la capacitat de crear nous fils d'execució) i de manera configurable (es reserva un nombre de fils d'execució a aquest propòsit tal i com es defineix en el fitxer de configuració), el servidor deixa de processar les noves peticions entrants i respon als clients amb l'estat HTTP 503[10] (servei no disponible). Ho fa de manera que consumeixi la menor quantitat de temps i recursos possible per tal d'evitar una major saturació.

- **Logs / fitxers de registre**. Quan el programa s'executa com a dimoni, desa els esdeveniments en fitxers de registre per tal de poder analitzar el funcionament i ús del mateix. La ruta dels fitxers és configurable. Es mantenen dos registres diferents: un per a esdeveniments generals, com aturades del servidor i accessos a recursos, i un per a indicar errors.

- **Documents d'error personalitzables**. El servidor permet que l'administrador utilitzi documents personalitzats que es mostraran al client quan una petició es resolgui amb un error (per exemple, quan no es troba un recurs o es fa una petició invàlida), permetent missatges informatius i amb un disseny personalitzable. La localització d'aquests documents és configurable.

- **Peticions condicionals**. Tot servei, especialment si s'ofereix per la xarxa, hauria de permetre que els clients i proxies mantinguin en memòria caché còpies dels recursos que ja han obtingut anteriorment. El protocol HTTP permet aquesta funcionalitat (entre d'altres mecanismes) a través de l'ús de diverses capçaleres que fan que una petició sigui condicional: només es serveix el recurs si es compleix certa condició. uaSW permet l'ús de la capçalera `If-Modified-Since`[11] (probablement la més utilitzada i útil en la majoria

http://tools.ietf.org/html/rfc3875, on s'especifica l'estàndard.

10 Aquest codi s'utilitza per a indicar que el servei no és disponible, típicament per sobrecàrrega però pot ser-ho per qualsevol altre motiu. Vegeu l'ús del codi d'estat 503 a la secció "10.5.4 503 Service Unavailable" del document RFC 2616 (http://tools.ietf.org/html/rfc2616#section-10.5.4)

11 Vegeu l'ús de la capçalera If-Modified-Since a la secció "14.25 If-Modified-Since" del document RFC 2616

de casos) en les peticions dels clients, de manera que el recurs només es serveix si ha estat modificat a partir d'un moment concret. ja que el client ja disposa d'una còpia del recurs generada en aquesta data. Si el recurs no s'ha modificat des de llavors, s'informa d'això al client enviant-li el codi d'estat HTTP 304[12] (no modificat). D'aquesta manera s'estalvia l'ús innecessari de la xarxa, millorant el rendiment del servidor

- **Ús de l'estat HTTP 100 (continua)**[13]. El protocol HTTP permet l'ús d'aquest estat per a controlar el flux d'informació entre client i servidor. Si un client vol enviar dades al servidor usant el mètode POST, pot decidir tractar d'assegurar-se que el servidor les acceptarà. En aquest cas, a través de la capçalera `Expect` indica al servidor que esperarà la seva confirmació per a seguir amb la transmissió. El servidor pot respondre amb el missatge 100 (continua) si tot és correcte o amb un codi d'error si no.

- **Bon rendiment**. No és estrictament una funcionalitat del programa, sinó una propietat del funcionament del mateix. uaSW ha estat dissenyat i implementat tenint en compte el rendiment i la senzillesa alhora. Com ja hem vist, és difícil innovar en el camp dels servidors web, així que com a mínim es tracta de fer bé la feina, consumint pocs recursos i oferint un rendiment acceptable, com en la secció d'anàlisi del rendiment. Ja hem vist alguns dels mecanismes que utilitza el servidor per a oferir un bon rendiment.

(http://tools.ietf.org/html/rfc2616#section-14.25)

12 Aquest codi d'estat indica que el document no ha estat modificat des de la data indicada pel client en la capçalera `If-Modified-Since` o en una altra petició condicional. Vegeu l'ús del codi d'estat 304 en la secció "10.3.5 304 Not Modified" del document RFC 2616 (http://tools.ietf.org/html/rfc2616#section-10.3.5)

13 Vegeu l'ús del codi d'estat 100 en la secció "8.2.3 Use of the 100 (Continue) Status" del document RFC 2616 (http://tools.ietf.org/html/rfc2616#section-8.2.3)

Disseny

El programa es pot dividir en quatre grans parts o mòduls.

- **Rutina principal / mestre (`main.c`)**: És la rutina que s'executa al iniciar el programa i que inclou el codi del fil d'execució mestre. Prepara l'entorn del programari (inicialització, configuracions, *sockets*) i entra en un bucle per a atendre les connexions entrants. Quan rep una connexió, crea un nou fil d'execució (un treballador) executant la rutina d'atenció a les peticions. Al sortir, neteja l'entorn.

- **Lectura de l'arxiu de configuració (`configuracio.c`)**: Llegeix els arxius de configuració i de tipus MIME, comprovant i aplicant els valors desats.

- **Atenció a les peticions / treballador (`fil.c`)**: Aten les connexions entrants. Llegeix les dades enviades pel client, comprovant la capçalera enviada. Respon seguint fins a cert punt l'especificació del protocol HTTP i envia les dades demanades, sigui un arxiu o el resultat de l'execució d'un script CGI. Aquest és el codi que executen els treballadors.

- **Execució de scripts CGI (`cgi.c`)**: Executa programes seguint el mecanisme definit per l'estàndard CGI per tal de generar continguts dinàmics.

A més d'aquest mòduls, s'utilitza un fitxer font d'encapçalament (**`header.h`**) que conté totes les constants que utilitza el programa (mides de buffers, vectors i cadenes; configuracions per defecte; límits diversos; codis HTTP...), així com l'estructura de les tuples (`structs`) que s'utilitzen.

Estructures de dades

La principal estructura de dades que implementa el programa és un vector de tuples del tipus fil. El vector te una mida fixa definida en el codi del programa (per defecte 500 elements). Cada treballador utilitza una d'aquestes tuples per a desar les seves dades principals: l'objecte `pthread_t` del fil d'execució, el descriptor del *socket* de la connexió amb el client, l'adreça IP del client, les dades de la petició HTTP rebuda i les dades de la resposta HTTP enviada.

Codi 3 - Tupla `fil`

```
// Tupla amb les dades d'un fil d'execució
struct fil {
    pthread_t pthread_obj; // objecte pthread_t del fil
    int descriptor; // descriptor del socket connectat amb el client
    struct sockaddr_in adreca_remota; // adreça de xarxa del client
    struct resposta_http resposta; // dades de la resposta HTTP
    struct peticio_http peticio; // dades de la petició HTTP
};
```

Les dades de la petició i la resposta HTTP es desen en tuples separades que veiem a continuació.

Codi 4 - Tupla `peticio_http`

```
// Tupla per a peticions HTTP
struct peticio_http {
    char http_method[MIDA_HTTP_METHOD];
    char http_protocol[MIDA_HTTP_VERSION];
    char http_path[MIDA_HTTP_PATH];
    char http_entity[MIDA_HTTP_ENTITY];
    char http_query_string[MIDA_HTTP_PATH];
    char connection[20];
    long content_length;
    char content_type[MIDA_DEF_HTTP_HEADER_VALUE];
    char expect[20];
    char if_modified_since[MIDA_DATA];
    char user_agent[MIDA_DEF_HTTP_HEADER_VALUE];
};
```

Com es pot observar, no es desen totes les capçaleres HTTP definides en l'estàndard. Això implica, entre altres coses, que el servidor no es comportarà com s'espera en certes condicions (per exemple, al obviar la capçalera `Accept`[14] pot ser que el servidor serveixi un recurs que no serà utilitzable pel client), no es suporta l'autenticació HTTP, etcètera. Només es desen les informacions útils per a implementar les funcionalitats del servidor:

- Dades essencials: mètode HTTP, protocol, ruta del recurs demanat.
- Dades per a l'execució de programes CGI i l'implementació del mètode POST: query-string (part de l'URL posterior al primer caràcter '?'), entitat de la petició, capçaleres `Content-Length` i `Content-Type` (indiquen la mida i el tipus del contingut de la entitat, respectivament)

14 En una petició, els clients poden usar la capçalera `Accept` per a definir quin tipus MIME accepten o prefereixen. Això és especialment útil en termes d'accessibilitat. Vegeu més informació sobre aquesta capçalera a la secció “14.1 Accept” del document RFC 2616 (http://tools.ietf.org/html/rfc2616#section-14.1)

- Capçaleres per a connexions persistents i control de flux: `Connection` i `Expect`.
- Capçalera per a peticions condicionals: `If-Modified-Since`.
- Capçalera d'interès estadístic: `User-Agent` (indica quin programari ha generat la petició).

Codi 5 - Tupla `resposta_http`

```
// Tupla per a respostes HTTP
struct resposta_http {
    long content_lenght;
    char content_type[MIDA_MIME];
    char date[MIDA_DATA];
    char last_modified[MIDA_DATA];
    int status_code;
};
```

De nou, observem que només s'inclou una petita part de les capçaleres HTTP possibles, les mínimes necessàries per a garantir un bon funcionament del servidor:

- Dades essencials: codi d'estat de la resposta, capçaleres `Content-Length` i `Content-Type`.
- Capçaleres per a permetre l'ús de la caché per part del client i servidors *proxy*: `Date` i `Last-Modified` (indiquen la data actual a la màquina servidori de l'última modificació del recurs, respectivament)

Una última tupla que utilitza el programa és `tipus_mime`. Utilitza un vector de tuples d'aquest tipus per a relacionar extensions de fitxers amb els tipus MIME corresponents, que el servidor utilitza per a deduir el tipus dels recursos que serveix.

Codi 6 - Tupla `tipus_mime`

```
// Tupla per a desar tipus mime relacionant-los amb l'extensió de fitxer
// corresponent
struct tipus_mime {
    char extensio[MIDA_EXTENSIO];
    char mime[MIDA_MIME];
};
```

La resta de dades que utilitza el programa són variables locals i globals de tipus estàndard o cadenes

de caràcters.

En tot el codi no es fa cap assignació de memòria dinàmica (crides a `malloc()` o rutines similars), sinó que s'assignen totes les dades estàticament. Això es fa per tres motius: mantenir la màxima senzillesa del programa, evitar en mesura del possible fer crides al sistema (amb els seus relativament costosos canvis de context) i evitar fugues de memòria. Com a contrapartida tenim sorgeixen alguns desavantatges: les mides dels vectors i cadenes són fixes i no hi ha possibilitat de canviar-les i es consumeix memòria que no s'utilitza.

Mòdul `fil.c` - La rutina principal (mestre)

Aquest mòdul conté les rutines que són executades pel fil d'execució mestre, així com algunes variables d'abast general del servidor.

Rutina `main()`

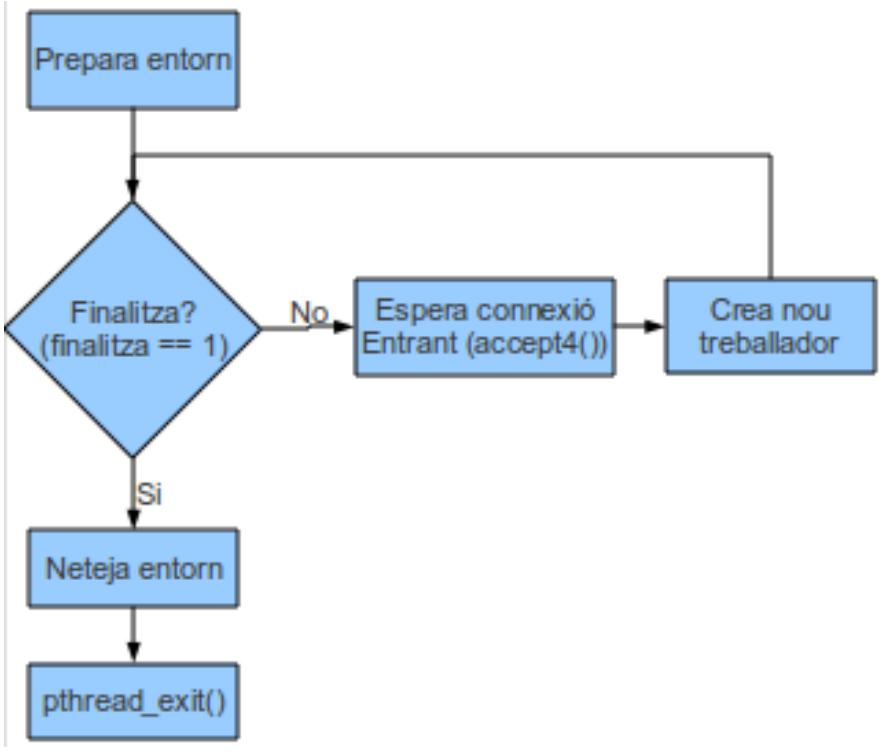

Diagrama il·lustrant de forma simplificada el funcionament de la rutina main()

La rutina `main()` és, com en tots els programes escrits en C, el punt d'entrada en l'execució del programa i conté les instruccions que executa el **fil d'execució mestre**. Veiem en el diagrama superior una versió simplificada del seu funcionament. Es basa en els processos següents:

- **Preparació de l'entorn**. Consta de les accions següents:

  - Estableix una **màscara als permisos dels fitxers** que el programa pugui crear[15] cridant a `umask()`, de manera que els nous arxius podran ser llegits i escrits per l'usuari propietari, només llegits pel grup d'usuaris propietari i es denegarà l'accés a altres usuaris.

  - Estableix les **rutines d'atenció dels senyals** mitjançant crides a `signal()`. El senyal `SIGINT` serà atès per la rutina `handle_sigint()` i el senyal `SIGTERM` per `handle_sigterm()`. El senyal `SIGPIPE` és ignorat[16].

  - Crida la rutina `llegir_configuracio()` del mòdul de configuració. Això **carrega l'arxiu de configuració i els tipus MIME**. Si s'ha proveït un segon paràmetre en el moment de l'execució, es tracta d'utilitzar el fitxer indicat per aquest.

  - Si així s'ha demanat a través del primer paràmetre, el programa passa a **executar-se en segon pla** cridant la rutina `daemonitza()`.

  - **Prepara el *socket* principal**, pel qual s'esperen rebre les connexions entrants, cridant la rutina `preparar_socket()`.

  - **Prepara els atributs dels futurs fils d'execució** cridant la rutina `preparar_pthread_attr()`.

  - **Posa el *socket* principal en l'estat `LISTEN`** a través de la crida a `listen()` per a començar a acceptar connexions.

- **El bucle principal**. Aquest bucle seguirà mentre el valor de la variable global `finalitza` sigui zero[17]. El funcionament del bucle és simple: espera connexions entrants amb la crida `accept4()` i, al rebre'n una, crida la rutina `crear_fil()` que tracta d'iniciar un nou

15 Els únics fitxers que crea el servidor són els fitxers de registre. Els programes CGI poden crear fitxers pel seu compte: en aquest cas també s'aplicarà la mateixa màscara.

16 El senyal SIGPIPE es rep quan es tracta d'escriure en una *pipe* és "trencada", és a dir, quan l'altre extrem ha tancat. Aquesta situació es dona sovint en el servidor (especialment quan suporta una càrrega notable), ja que el client pot tancar la connexió prematurament. Com el comportament per omissió és avortar el procés (i això no és pas desitjable en aquest cas, tot i que té sentit en molts altres casos), opto per ignorar-lo.

17 L'únic moment en el que el valor de `finalitza` canvia a 1 és quan es rep el senyal SIGINT, que causa una "aturada cortès" del servidor.

treballador. S'utilitza `accept4()` enlloc d'`accept()` per a activar la opció `SOCK_CLOEXEC`; més endavant, en l'explicació de la rutina `preparar_socket()` veurem perquè.

- Si surt del bucle, la rutina fa una crida a `neteja()`, que **allibera recursos**.

- Finalment, **finalitza el propi fil mestre** amb una crida a `pthread_exit()`. Es fa aquesta crida enlloc de `exit()` o `return` per a permetre que els treballadors segueixin executant-se. Hem de tenir en compte que el programa només arriba a aquest punt si realitza una "aturada cortès". Si es rep el senyal d'acabar immediatament (`SIGTERM`), la pròpia rutina d'atenció del senyal crida a `exit()`.

Rutina `daemonitza()`

Aquesta rutina canvia la forma en que s'executa el servidor, passant-lo a segon pla. El seu funcionament consta de: la redirecció de `stdout` i `stderr` als fitxers de registre principal i d'errors, respectivament; la redirecció de `stdin` a `/dev/null` i en deixar orfe al procés[18].

Rutina `preparar_socket()`

Aquesta rutina tracta de demanar un *socket* TCP/IP al SO usable pel servidor en el port establert per la configuració. A aquest se li apliquen tres opcions: `SOCK_CLOEXEC` (com al *socket* principal) per a que es tanqui al fer una crida a `exec()` (això succeeix al executar un programa CGI: no interessa que aquest tingui accés als *sockets* del servidor), `SO_REUSEADDR` per a forçar l'ús del port TCP/IP encara que estigui en estat `TIME_WAIT`[19] i `TCP_CORK` per a evitar que el SO envii les dades escrites al *socket* immediatament[20].

18 Això s'assoleix amb una crida a `fork()`, on posteriorment el procés pare acaba, sent el fill el que continua amb l'execució normal. Al deixar orfe el procés, és "adoptat" per `init` (el procés principal del SO) i s'executa en segon pla.

19 Això succeeix quan el port ja no és en ús, però ho ha estat recentment. És especialment útil per a fer proves i desenvolupar el servidor, ja que s'inicia i atura el servidor molt sovint.

20 Això es fa perquè el servidor envia sempre primer les capçaleres HTTP i posteriorment el recurs demanat i no té sentit enviar-los en dos paquets separats si és possible fer-ho en un de sol. Vegeu l'explicació de la rutina `uncork()` de `fil.c`, la pàgina man de `sendfile()` i el document "TCP/IP options for high-performance data transmission" disponible a http://articles.techrepublic.com.com/5100-10878_11-1050878.html.

Rutina `crear_fil()`

Aquesta rutina tracta de crear un nou fil d'execució treballador que atengui una connexió entrant. El funcionament és simple: cerca en el vector de tuples `fil` una tupla no usada comprovant si el membre `pthread_obj` val zero. Quan la troba, copia el descriptor del *socket* i l'adreça IP del client en els seus membres corresponents. Si, d'acord amb els paràmetres establerts en la configuració, el servidor està saturat, el nou fil només informarà al client d'aquesta situació executant la rutina `atendre_sobrecarrega()`; sinó, el nou fil atendrà les peticions rebudes per la connexió executant la rutina `atendre_connexio()`. En executar una d'aquestes rutines (en un nou fil d'execució) es passa com a paràmetre un punter a la tupla `fil` que li correspon.

Rutina `modificar_nombre_fils()`

La rutina `modificar_nombre_fils()` s'usa per a modificar el valor de la variable `nombre_fils`. Com aquesta variable ha de ser modificada per tots els fils d'execució, s'ha d'utilitzar un *mutex* per a evitar condicions de competició (és un dels problemes típics de la computació en paral·lel, sorgeix quan dos processos volen modificar el valor d'una mateixa variable). Aquesta és la única variable en la que ocorre això, fet que demostra l'alta independència entre fils d'execució i l'aptitud dels servidors web per a aprofitar la computació en paral·lel.

Rutina `preparar_pthread_attr()`

La rutina `preparar_pthread_attr()` prepara un objecte `pthread_attr` que conté els paràmetres per als fils d'execució que crearà la rutina `crear_fil()`. Estableix la mida de la pila a un valor segur definit per la constant `MIDA_STACK_FIL` i fa que els fils d'execució s'executin de manera independent (*detached*), de manera que el fil d'execució mestre no hagi d'estar pendent de quan finalitzen els treballadors.

Altres rutines

La rutina `handle_sigint()` s'encarrega d'atendre els senyals `SIGINT`, provocant l'aturada "cortès" del servidor. En la pràctica, estableix el valor de `finalitza` a 1 i tanca el *socket* principal per a desbloquejar la crida a `accept()` del bucle principal (si no es fes això, el servidor no s'aturaria fins a rebre una connexió entrant, ja que aquesta rutina bloqueja per omissió l'execució

fins a rebre una connexió).

Els senyals `SIGTERM` són atesos per la rutina `handle_sigterm()`, que provoca l'aturada immediata del servidor. Ho fa amb una crida a `exit()`, cridant prèviament a la rutina `neteja()` per a alliberar recursos.

La rutina `neteja()` allibera recursos assignats pel programa: l'objecte `pthread_attr` creat per la rutina `preparar_pthread_attr()` i l'objecte `addrinfo` creat per `preparar_socket()`.

Mòdul `configuracio.c` - Lectura d'arxius de configuració

Aquest mòdul defineix les diferents variables que contenen els paràmetres configurables del programa i dues subrutines: l`legir_configuracio()` i `llegir_tipus_mime()`.

Rutina `llegir_configuracio()`

Aquesta rutina carrega els paràmetres desats en el fitxer de configuració. Ho fa obrint el fitxer per a lectura i entrant en un bucle que el llegeix línia per línia fent crides a `gets()`. El bucle obvia les línies buides o que comencin pel caràcter '#', permetent la inclusió de comentaris en el fitxer. Les demés línies les processa, llegint primer la primera paraula per a establir quin paràmetre es vol establir i posteriorment el valor del paràmetre que es troba a continuació en la línia.

Codi 7 - Fragment exemple d'un arxiu de configuració vàlid

```
# ArxiuPortada: Document que es servirà al accedir a un directori
ArxiuPortada index.html

# Host: URL base del servidor.
Host localhost:8080
```

Rutina `llegir_tipus_mime()`

La rutina `llegir_tipus_mime()` és molt similar a l'anterior. Obre el fitxer dels tipus MIME (la ruta del qual s'estableix en l'arxiu de configuració) i el llegeix línia per línia de forma idèntica a la rutina anterior permetent comentaris. En aquest cas, la primera paraula de cada línia indica l'extensió del fitxer i la resta de la línia el tipus MIME corresponent. El tipus mime pot contenir la

seqüència de caràcters "`%s`", que serà substituïda en temps d'execució pel conjunt de caràcters establert en l'arxiu de configuració.

Codi 8 - Fragment exemple d'un arxiu de tipus mime vàlid

```
# Això tan sols és un exemple amb un comentari
hdf  application/x-hdf
hqx  application/mac-binhex40
htm  text/html; charset=%s
html text/html; charset=%s
```

Mòdul `fil.c` - Rutines d'atenció a les connexions (treballadors)

Aquest mòdul conté les rutines que executen els treballadors amb l'objectiu d'atendre i processar les connexions i peticions HTTP entrants.

Els treballadors es creen quan la rutina `crear_fil()` de `main.c` s'executa amb èxit. Existeixen dues maneres de procedir diferents: si el servidor es troba sobrecarregat (això es decideix utilitzant els paràmetres `MaxFils` i `MaxFilsSobrecarrega` establerts a l'arxiu de configuració) el treballador comença executant la rutina `atendre_sobrecarrega()`. Si no és així, s'inicia amb la rutina `atendre_connexio()`.

Totes les rutines d'aquest mòdul identifiquen el fil d'execució en el que es criden a través d'un punter a una tupla de tipus `fil` del vector `fils`.

Rutina `atendre_sobrecarrega()`

Com acabem de veure, aquesta és la rutina que executa un nou treballador si el servidor es troba sobrecarregat. La forma de procedir és molt senzilla, ja que interessa que el servidor no es saturi encara més i que el treballador finalitzi quan abans: simplement ignora les dades enviades pel client, enviant-li el codi d'estat HTTP 503 (Servei no disponible). Tot seguit, finalitza el fil d'execució amb una crida a la rutina `acabar_fil()`.

Rutina `atendre_connexio()`

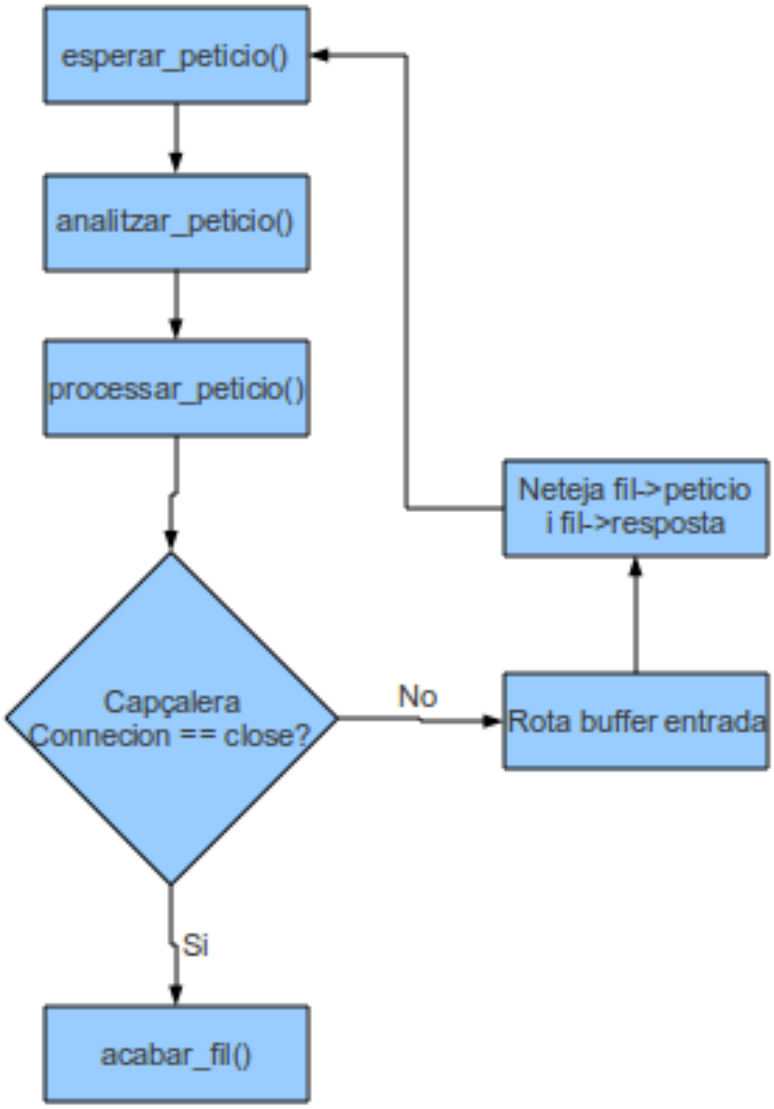

Diagrama il·lustrant, de forma simplificada, el funcionament de la rutina atendre_connexio()

Aquesta és la rutina d'entrada dels treballadors en circumstàncies normals (de no sobrecàrrega). No atén una sola petició, sinó que atén connexions, incloent les persistents; d'aquí la seva estructura cíclica que es pot apreciar en el diagrama.

La major part de la rutina forma part d'un bucle incondicional que repeteix tres grans fases (amb l'objectiu d'atendre les peticions HTTP que es vagin rebent) que corresponen a tres rutines diferents:

- **`esperar_peticio()`**: Espera a que es rebi al menys una petició HTTP sencera pel *socket*.

- **`analitzar_peticio()`**: Analitza la petició rebuda, desant les seves parts convenientment per al seu posterior processament en l'estructura de dades `peticio_http` de la tupla `fil`. Aquesta rutina és cridada per `esperar_peticio()` i no per `atendre_connexio()` directament.

- **`processar_peticio()`**: Tracta de respondre la petició amb una resposta HTTP.

Les dues primeres rutines no informen dels errors que puguin sorgir directament al client, sinó que ho fan amb el valor de retorn i és la rutina `atendre_connexio()` la que emprèn les accions necessàries, com informar al client i tancar la connexió si s'escau. En canvi, la rutina `processar_peticio()` realitza aquestes accions per si mateixa, ja que realitza el processament final de la petició rebuda. `atendre_peticio()` no s'interessa pel retorn de `processar_peticio()`, simplement segueix amb la següent petició suposant que l'anterior ha estat processada correctament, ja sigui servint el recurs demanat amb èxit o enviant un error al client.

La rutina `atendre_connexio()` actua de diferents maneres segons el retorn de `esperar_peticio()` (recordem que aquesta última és la que crida a `analitzar_peticio()`, i el seu retorn també pot contenir informació del resultat de l'execució d'aquesta última), ja que ens indica errors i circumstàncies davant les quals el servidor ha d'actuar. Veiem a continuació els possibles retorns d'`esperar_peticio()`:

- `EP_CLIENT_HA_TANCAT`: indica que el client ha tancat la connexió. En aquest cas, el treballador finalitza la seva execució immediatament.

- `EP_FINALITZA`: indica que el servidor està sent aturat. El treballador finalitza immediatament.

- `EP_TEMPS_ESPERA_EXCEDIT`: indica que s'ha excedit el temps d'espera fins a rebre una petició definit amb el paràmetre `TempsEspera` del fitxer de configuració. El treballador envia el codi d'estat HTTP 408 (temps d'espera excedit al rebre petició) al client i finalitza la seva execució.

- `EP_ERROR`: ha ocorregut algun error a la banda del servidor. S'envia al client el codi d'estat

HTTP 500 (error intern del servidor) i acaba l'execució del treballador.

- `EP_MASSA_LLARG`: El request-entity (les dades enviades amb el mètode POST) és massa llarg per a ser desat. Es respon amb el codi d'estat HTTP 413 (request-entity massa llarg) i el treballador acaba.

- `EP_PETICIO_NO_VALIDA` i `AP_INVALIDA`: La petició HTTP rebuda no és vàlida. Es respon amb l'error 400 (petició no vàlida) i el treballador finalitza.

- `AP_VERSIO_NO_SUPORTADA`: El client ha indicat que utilitza un protocol diferent a HTTP/1.1. Li informem que el protocol no és suportat amb el codi d'error HTTP 505 i el treballador finalitza.

- `EP_EXPECT_NO_VALID`: El client ha enviat amb la petició una capçalera Expect que no s'ha entès. Es respon amb el codi d'error 417 (expectació fallida) i finalitza el treballador.

Si la crida a `esperar_peticio()` no retorna un valor negatiu, és que ha pogut executar-se correctament i el treballador procedeix a respondre a la petició, cridant la rutina `processar_peticio()`.

El retorn d'`esperar_peticio()`, quan no és negatiu, indica la llargada de la primera petició rebuda en bytes. A través d'un paràmetre que és passat per referència anomenat `bytes_sobrants`, la rutina també retorna el nombre de bytes restants al buffer de recepció de dades que no corresponen a la primera petició trobada (que és la que es processarà). Aquest valor és desat en una variable. Un cop processada la petició (després de la crida a `processar_peticio()`), si el nombre de bytes restants en el buffer és zero, es fa una crida a `uncork()` per tal de forçar l'enviament immediat de les dades. Això només es fa quan no hi ha dades pendents de processar, ja que podem suposar que, si n'hi ha, molt probablement sigui d'una petició subseqüent. D'aquesta manera, permetem el *pipelining* i optimitzem l'ús de la xarxa.

Un cop processada la petició, es procedeix a rotar (si hi ha dades pendents de processar, usant `memmove()` i `memset()`) o netejar (si no hi ha dades pendents de processar, usant `memset()`) el buffer de recepció de dades. Per acabar, es netejen els elements `peticio_http` i `resposta_http` de la tupla `fil` del treballador. Un cop arribat aquí, el treballador torna a

començar el bucle principal de la rutina per a esperar i processar peticions subseqüents (recordem que el servidor suporta peticions persistents, per això és necessari aquest bucle).

Observem que el bucle és incondicional, però hi ha diverses maneres de provocar la finalització del treballador:

- Quan ocorre un error.
- Quan el client tanca la connexió.
- Quan s'excedeix el temps d'espera.
- Quan el client ho demana explícitament amb la capçalera `Connection: close`.
- El servidor s'està aturant.
- L'execució de programes CGI sempre força la finalització del treballador, per motius que veurem més endavant.

Rutina `esperar_peticio()`

Com ja hem vist breument, aquesta rutina s'encarrega de llegir les dades que es rebin pel *socket* connectat al client i comprovar quan s'ha rebut una petició HTTP sencera.

Aquesta rutina consta d'un bucle principal incondicional que abasta tota la rutina, del qual no es surt fins que es compleix una de les següents condicions: s'excedeix el temps d'espera, es rep una petició HTTP vàlida i completa, el client tanca la connexió, el servidor s'està aturant o ocorre un error (es rep una petició massa llarga, ocorre un error al rebre dades o a la crida a `select()`, etcètera).

Veiem a continuació les diferents accions que emprèn la rutina de manera simplificada:

- Comprova si el servidor s'està aturant (comprovant si la variable finalitza val 1, si és el cas, retorna `EP_FINALITZA`).
- Comprova si hi ha una petició sencera i vàlida al buffer. L'analitza fent una crida a `analitzar_peticio()`. Si s'ha rebut una petició sencera o invàlida, la rutina retorna el valor que correspon. Si el client demana l'enviament del missatge HTTP 100 (continua) a

través de la capçalera `Expect` abans de procedir, s'envia (només una vegada per petició).

- Comprova si s'ha excedit el temps d'espera.
- Amb una crida a `select()`, espera a rebre més dades durant tres segons.
- Espera a rebre dades addicionals amb una crida a `recv()`, desant-les al buffer.

Noteu com l'ordre en que es realitzen aquestes accions permeten que hi hagi una petició existent d'una lectura del descriptor anterior i com aquesta s'accepta sense llegir més dades del *socket*. Això es fa per a permetre el *pipelining* de peticions de manera eficient.

Noteu també que la crida a `select()` no espera indefinidament a rebre dades (o durant el temps definit en el temps d'espera). Això es fa per a que es pugui comprovar periòdicament si el servidor s'està aturant. Si no es fa així, el treballador quedaria bloquejat fins que es rebessin dades pel *socket*, posposant l'aturada del servidor.

Aquesta rutina te un paràmetre d'entrada i sortida anomenat `bytes_sobrants`. L'utilitza com a paràmetre de sortida per a indicar quants bytes del buffer d'entrada no corresponen a la primera petició HTTP rebuda, per a que es tingui en compte que aquestes dades segurament formin part de la següent petició HTTP enviada pel client. Com a paràmetre d'entrada, el fa servir per a saber quants bytes des de l'inici del buffer ja estan sent utilitzats, de manera que comenci a omplir-lo des d'aquell punt. D'aquesta manera es preserven els dades de peticions HTTP subseqüents, suportant el *pipelining* del protocol.

Rutina `analitzar_peticio()`

Aquesta rutina llegeix el buffer d'entrada del treballador per tal d'analitzar la petició HTTP que es trobi al seu inici. A part de comprovar amb diferents mètodes si la petició és vàlida i el protocol és el HTTP/1.1, copia i, si s'escau, converteix les dades significatives de la mateixa a la tupla `peticio_http` de la tupla `fil` que correspon al treballador.

Realitza aquestes accions analitzant el buffer línia per línia, primer analitzant la request-line de la petició (desant el mètode, la ruta del recurs demanat i la versió del protocol HTTP usada pel client) i les subseqüents línies de les capçaleres fins a trobar la seqüència de caràcters “`CRLFCRLF`“ (en una

cadena de C es representa amb la seqüència "`\r\n\r\n`"), que indica la fi de les capçaleres.

Retorna `AP_INVALIDA` si la petició no ha passat satisfactòriament les comprovacions de validesa, `AP_VERSIO_NO_SUPORTADA` si el protocol indicat pel client és diferent a HTTP/1.1 o zero si l'execució s'ha dut a terme sense problemes.

Rutina `processar_peticio()`

Aquesta és la darrera de les principals rutines que `atendre_connexio()` crida per a atendre les peticions HTTP. A diferència de les dues anteriors, aquesta no llegeix les dades de la petició del buffer d'entrada, sinó de la tupla `peticio_http` (que ha estat omplerta amb les dades corresponents per al rutina `analitzar_peticio()`) S'encarrega de construir la resposta HTTP en la tupla `resposta_http` i posteriorment l'envia al client d'acord amb l'especificat pel protocol.

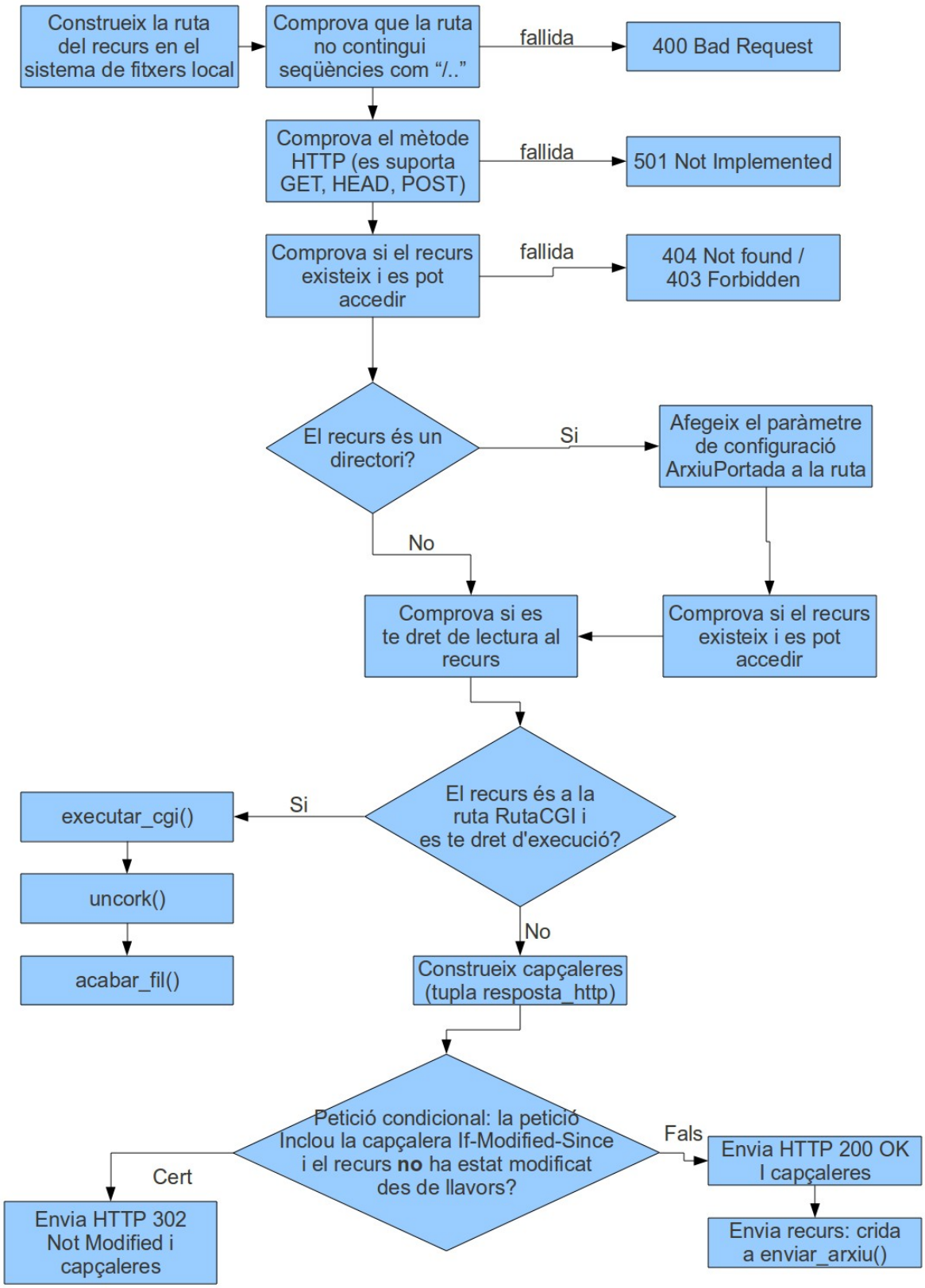

Diagrama il·lustrant de forma simplificada el funcionament de la rutina processar_peticio()

El diagrama anterior il·lustra prou be el funcionament d'aquesta rutina, tot i que la analitzarem breument aquí, seguint els diferents passos que segueix per a assolir el seu objectiu.

- Comença **construint la ruta del recurs** demanat en el sistema de fitxers local, concatenant les cadenes de la ruta demanada a la petició i el paràmetre configurable `DirectoriDocuments`.

- Comprova que la ruta no contingui seqüències que "tornin enrere" en la ruta, com "`..`" per a evitar problemes de seguretat, ja que els clients podrien demanar arxius que es trobessin fora del directori dels documents a servir.

- Posteriorment **comprova el mètode HTTP** de la petició. Es suporten GET, HEAD i POST. Als altres mètodes HTTP existents se'ls respon amb el codi 501 (no implementat) i a mètodes invàlids amb el codi 400 (petició no vàlida).

- Posteriorment, amb una crida a `stat()`, es **comprova si el recurs demanat existeix i és accessible**. Retorna el codi 404 (no trobat) si no existeix i 403 (prohibit) si no es pot accedir.

- **Comprova si el recurs és un directori**. Si és el cas, concatena la ruta del mateix amb el paràmetre configurable `ArxiuPortada` i repeteix el pas anterior (crida a `stat()`).

- **Comprova si es tenen drets de lectura al recurs**. Si no és el cas, retorna el codi 403 (prohibit).

- Si el recurs es troba a la ruta de **programes CGI** (configurable pel paràmetre `RutaCGI`) i es tenen drets d'execució, es fa una crida a `executar_cgi()`, que s'encarrega de dur a terme l'execució. Posteriorment es crida a `uncork()` per a forçar l'enviament immediat de les dades i acaba el fil immediatament. No seria segur permetre més peticions per la mateixa connexió, ja que en el cas dels programes CGI, de la forma en que està implementada, és el mateix programa qui ha d'enviar la capçalera `Content-Lenght` i no podem comptar que ho faci. Si no ho fa, el client pot no ser capaç de saber quan acaba la resposta HTTP actual i quan comença la següent.

- **Construeix les capçaleres HTTP** de la resposta en la tupla `resposta_http` de la tupla `fil` del treballador.

- Si el client ha fet una **petició condicional** amb la capçalera `If-Modified-Since`, comprovem si es compleix la condició. Si no es compleix, responem al client amb el codi 302 (no modificat).

- **Construeix el status-line i les capçaleres** de la resposta HTTP en el buffer de sortida i l'envia mitjançant una crida a `send()`.

- Si el mètode ha estat GET o POST (i no HEAD), **envia el contingut del recurs** mitjançant una crida a `enviar_arxiu()`.

La rutina retorna zero si s'ha pogut atendre la petició correctament, si no retorna -1. En els casos que ocorri un error, sempre tractarà d'enviar una resposta informant de la situació al client.

Rutina `enviar_arxiu()`

Aquesta rutina envia un arxiu al client (directament, sense capçaleres HTTP ni cap altre processament). Primer obre l'arxiu mitjançant una crida a `fopen()` i obté el descriptor del mateix amb `fileno()`. L'enviament es realitza amb una crida a `sendfile()`, de manera que el nucli del sistema operatiu s'encarrega de fer la transferència, evitant nombroses crides i canvis de context i oferint un millor rendiment.

La rutina retorna zero si s'ha executat amb èxit, 1 si hi ha hagut algun problema al obrir l'arxiu i 2 si hi ha hagut algun problema amb la crida a `sendfile()`.

Rutina `enviar_error()`

La rutina `enviar_error()` envia un missatge d'error al client mitjançant el protocol HTTP. Existeix la possibilitat de que envii un document d'error (configurable a través del paràmetre `DirectoriDocumentsError`) en els mètodes que ho permeten (GET i POST). El procediment que segueix consisteix en:

- **Comprovar si s'ha d'enviar el document d'error**, si aquest existeix i si es tenen drets de lectura del mateix.

- **Preparar les capçaleres** de la resposta HTTP en la tupla `resposta_http`.

- **Enviar el status-line i les capçaleres** mitjançant `send()`.
- Si s'escau, **enviar el document d'error** amb una crida a `enviar_arxiu()`.

La rutina sempre retorna zero.

Rutina `acabar_fil()`

Aquesta rutina finalitza l'execució d'un fil treballador. Ho fa de la següent manera:

- **Crida a `uncork()`** per a forçar l'enviament immediat de dades pel *socket* que puguin estar esperant.
- **Tanca el descriptor** del *socket*, i amb això la connexió TCP/IP, mitjançant `close()`.
- **Neteja** la tupla `fil` del treballador.
- Decrementa en u el valor de la variable global `nombre_fils` mitjançant `modificar_nombre_fils()`, ja que s'ha d'usar un *mutex* per qüestions de sincronització.
- **Finalitza** el fil d'execució amb una crida a `pthread_exit()`.

Rutina `deduir_mime()`

Aquesta rutina retorna, a partir de la ruta d'un fitxer, el seu tipus MIME, deduït a partir de la seva extensió i el vector `mimes`, que conté les relacions entre extensions i tipus MIME carregades del fitxer de tipus MIME.

Rutina `uncork()`

Els *sockets* creats pel servidor tenen la opció `TCP_CORK` activada. Això augmenta l'eficiència de l'ús de la xarxa implementant un simple mecanisme: enlloc d'enviar les dades escrites al *socket* directament, s'espera un temps (en la meva màquina, 200 milisegons) per tal d'esperar l'escriptura de més dades i així enviar paquets TCP/IP el més grans possibles enlloc de molts de petits. Això ajuda a donar un millor servei, però pot augmentar la latència en la resposta, ja que el SO pot estar esperant més dades que no s'escriuràn al *socket* (o, al menys, no en un temps raonable).

Per a evitar aquest augment de la latència, quan el servidor ja ha escrit al *socket* una resposta HTTP sencera (i no s'ha rebut cap altra petició del client), crida a la rutina `uncork()`, que simplement desactiva i torna a activar l'opció `TCP_CORK` del *socket* per a forçar l'enviament immediat de les dades.

Mòdul `cgi.c` - Execució de programes CGI

Repeteixo que la funcionalitat CGI del servidor és experimental i te errors coneguts. S'inclou com a demostració d'un treball preliminar, en cap cas complet.

Aquest mòdul consisteix d'una única rutina, que és cridada per `processar_peticio()` per a procedir a l'execució d'un programa CGI en els casos que ja hem vist anteriorment. El diagrama a continuació mostra el procediment que segueix.

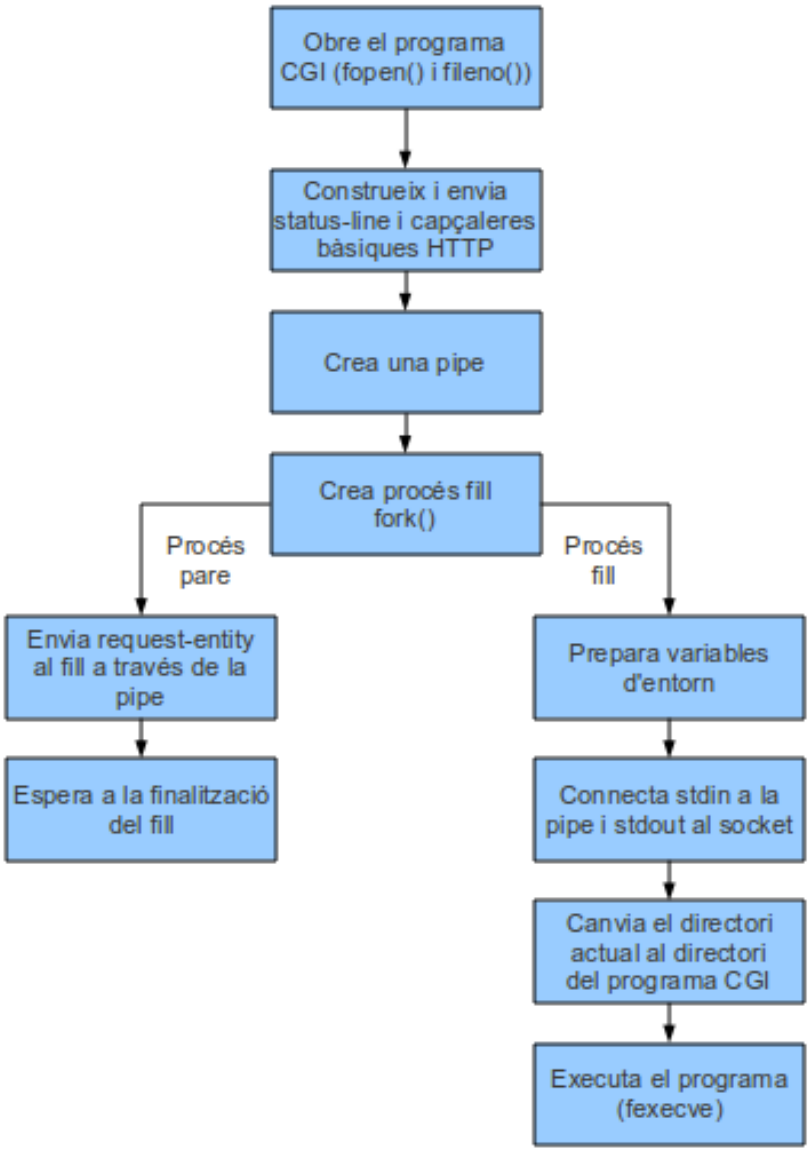

Diagrama mostrant de forma simplificada el funcionament de la rutina executar_cgi()

Com hem fet amb la resta de mòduls i rutines, veurem breument el procediment que segueix.

- Primer obre l'arxiu executable per a la seva lectura i obté el descriptor corresponent amb crides a `fopen()` i `fileno()`.

- Tot seguit prepara el *status-line* i les capçaleres de la resposta HTTP en el buffer de sortira i l'envia al client mitjançant `send()`. No s'indica el final de les capçaleres (la seqüència `CRLFCRLF` o `\r\n\r\n`), ja que el programa CGI podria enviar més capçaleres, i és aquest el que s'ha d'encarregar d'indicar el final de les capçaleres.

- Es crea una *pipe* per a comunicar el nou procés que es crearà amb el servidor per tal d'enviar-li l'*entity* de la petició HTTP més endavant.

En aquest punt s'inicia un nou procés mitjançant una crida a `fork()`. A partir d'aquest punt, tenim dos processos que s'executen per separat. Ambdós tenen accés a la *pipe* que tot just ha estat creada, amb la qual es comunicaran.

El procés pare, que seguirà sent la instància que actua com a servidor, segueix el procediment següent.

- Tanca l'extrem de lectura de la *pipe*, ja que no el necessita, i l'arxiu executable que s'ha obert al principi de la rutina.
- Li envia a través de la *pipe* l'*entity* de la petició HTTP, tal i com s'especifica en l'estàndard.
- Espera a la finalització del procés fill amb una crida a `wait()`.

Mentrestant, per la seva banda el procés fill realitza la execució del programa CGI seguint el procediment següent:

- Prepara un vector de cadenes que servirà per a establir diverses variables d'entorn en el programa CGI, per tal d'enviar-li diverses dades com s'especifica en l'estàndard CGI, com la mida de l'*entity*, el *query-string*, el nom del programari que actua com a servidor, etcètera.
- Realitza diverses operacions amb els descriptors amb crides a `dup()` i `close()` per tal que `stdin` quedi connectat a l'extrem de lectura de la *pipe* (per on es rebrà el *request-entity*, enviat pel procés pare) i `stdout` al *socket* connectat amb el client. La sortida d'error (`stderr`) no s'altera (no tindria sentit connectar-lo també al *socket* i, de fet, podria comportar problemes de seguretat) i es tanca l'extrem d'escriptura de la *pipe*. D'aquesta manera queda el procés comunicat com especifica l'estàndard CGI: per l'entrada es rep el *request-entity* i per la sortida s'envien els resultats de l'execució al client.
- Es canvia el directori de treball al directori on resideix el programa CGI (`chdir()`).
- Finalment, amb una crida a `fexecve()` s'executa el programa CGI, reemplaçant el procés fill i establint les diverses variables d'entorn.

La rutina en el seu estat actual executa amb èxit el programa i envia els resultats al client, però apareixen alguns problemes. Aparentment, hi ha algun problema en la comunicació entre els processos i el client no rep correctament les dades que necessita, per exemple, en l'enviament d'un formulari. No he analitzat la qüestió a fons i podria ser perfectament un error trivial.

Sigui com sigui, crec que és un bon punt de partida per a la implementació d'aquesta funcionalitat i els aspectes fonamentals del procediment funcionen satisfactòriament.

Implementació

La implementació d'aquest servidor s'ha dut a terme usant el IDE NetBeans. Aquesta eina ofereix una molt bona integració amb d'altres (com el compilador gcc), un sistema de depuració avançat i útil i eines per a monitoritzar l'execució dels programes visual i molt valuós, que inclou informació sobre els fils d'execució, ús de CPU i memòria, etcètera. Aquestes són només algunes de les característiques més notables, però n'ofereix moltes d'altres útils per al desenvolupament, tant en C com en altres llenguatges. És en definitiva un IDE molt potent, que m'ha facilitat enormement la feina.

El sistema operatiu utilitzat ha estat Ubuntu Linux (concretament les versions 9.10 i 10.04). Per a compilar i muntar el programa s'han usat les eines GNU.

Versió 0.1 (19 d'abril de 2010, lliurament PAC2)

Aquest ha estat el primer lliurament que s'ha fet del producte. Es tracta d'una versió preliminar, que comença a mostrar l'estructura i funcionament del programa d'una manera bàsica.

Aquest lliurament incloïa les parts nuclears del servidor:

- Preparava l'entorn de forma simple: crida a `umask()` per a establir la màscara de permisos als fitxers nous, preparava el *socket* i el posava en estat `LISTEN`, implementava la lectura de l'arxiu de configuració.

- El bucle principal esperava connexions i creava treballadors en rebre-les de forma similar a aquesta versió. S'implementava el *mutex* per a la modificació de `nombre_fils`. El pas de dades al treballador es feia de forma incorrecta però funcional en la majoria de casos (es passava com a paràmetre a `pthread_create()` un punter a una variable que podia variar abans de ser llegida pel treballador, un error en computació paral·lels).

- Els treballadors no processaven encara les dades rebudes del client, només les imprimien per pantalla. La resposta a les connexions entrants era sempre la mateixa, per a provar el funcionament bàsic del servidor. No s'implementava cap part del protocol HTTP.

- En acabar, duia a terme la neteja de l'entorn.

- Realització de proves a través de sessions *telnet*.

Si bé aquesta versió no era gens funcional, mostrava la senzillesa i viabilitat del disseny.

Versió 0.2 (23 de maig de 2010)

Aquesta versió no la vaig lliurar al tutor, bàsicament pel poc que vaig avançar la memòria. El codi, però, si va avançar bastant i es van implementar la major part de funcionalitats. A continuació llisto alguns dels progressos realitzats:

- El codi dels treballadors s'ha implementat en la seva major part, dividint-lo en les tres rutines principals (`esperar_peticio()`, `analitzar_peticio()`, `processar_peticio()`). Inclou també els fils de sobrecàrrega, així com les diverses subrutines (`enviar_arxiu()`, `enviar_error()`...). Inclou diverses comprovacions d'error i la resposta als mateixos.

- S'implementen i utilitzen les estructures de dades (*structs* `fil`, `peticio_http` i `resposta_http`).

- S'implementa el mecanismes per a personalitzar els documents d'errors HTTP.

- Correcció d'errors i millores diverses, com:

    - Creació dels *sockets* amb l'opció `CLOEXEC` (per a evitar que els programes CGI puguin accedir a ells) i `SO_REUSEADDR` (per a permetre la reutilització de ports TCP/IP en intervals breus de temps).

    - La creació dels fils d'execució es fa amb l'opció *detached* activada per a evitar haver d'esperar la seva finalització. Anteriorment cada creació d'un fil consumia recursos permanentment, inclús després de que acabés la seva execució. Al acabar la seva execució (`acabar_fil()`), es fa a una crida a `pthread_exit()`. També s'estableix la mida de les seves piles.

- Tractament dels senyals (`SIGTERM` i `SIGINT`, `SIGPIPE` s'ignora), implementació de l'"aturada cortès".

- Primeres proves d'estabilitat i rendiment amb navegadors (Chromium, Firefox i Epiphany) i l'eina httperf.
- Mòdul d'execució de scripts CGI

Versió 0.3 (13 de juny de 2010, lliurament final)

Aquest és el lliurament que es fa juntament amb aquesta memòria. En aquest punt la majoria de canvis són menors, ja que la major part del programa ja ha estat implementada.

- Lectura de tipus MIME del fitxer corresponent basada en la lectura del fitxer de configuració. Juntament amb això, ara s'informa al client del tipus MIME del recurs servit.
- Millorat l'ús de les estructures de dades.
- Implementació del sistema d'arxius de registre i de la rutina `daemonitza()` per a l'execució en segon pla.
- Comprensió de la capçalera HTTP `Expect` i enviament del missatge HTTP 100 (continua).
- Implementació de les connexions persistents i el *pipelining*.
- Ús de la opció `TCP_CORK` als *sockets* i implementació de la rutina `uncork()`.
- Processament dels paràmetres d'execució: el primer per a indicar execució en primer o segon pla, el segon per a definir una ruta de l'arxiu de configuració diferent al valor per defecte.
- Diverses correccions i optimitzacions al codi.
- Proves d'estabilitat i rendiment amb navegadors i eina httperf.

Posteriorment a aquest lliurament només s'ha fet una modificació: no s'envien les capçaleres `Content-Length` i `Content-Type` en una resposta HTTP amb codi 304 (no modificat), ja que aquest no és el comportament definit per l'estàndard. Noteu però, que el funcionament no estàndard de la versió 0.3 no ha donat cap problema ni comportament que provoqués situacions anòmales en les proves amb cap dels navegadors ni l'eina httperf. No ha estat fins que s'ha usat l'eina d'anàlisi de protocols Wireshark tenia problemes per a comprendre les respostes que no he detectat aquest error.

Proves realitzades

Durant i després de la implementació del programa he realitzat proves diverses per a comprovar el correcte funcionament del mateix. Repasso aquí alguns exemples de les que he realitzat.

Durant les primeres fases de la implementació, mentre el codi anava sent implementat progressivament, es feien proves dels mòduls implementats. Així, per exemple, la primera fase va ser el desenvolupament del mòdul de lectura dels arxius de configuració. Vaig fer un "prototip" de rutina principal, que només cridava la rutina principal d'aquest mòdul, `llegir_configuracio()`. Així podia fer comprovacions amb diversos arxius de configuració, tant vàlids com invàlids.

Codi 9 - Exemple arxiu de configuració erroni

```
# Aquesta configuració té un error tipogràfic i no és vàlida
PorEscolta 80

# Aquesta configuració no existeix
FooBar 1

# La suma del total de fils d'execució suma més de 500, que és el límit dur del
programa
MaxFils 475
MaxFilsSobrecarrega 50
```

Tractar d'arrencar el servidor amb aquest arxiu de configuració provoca la sortida següent, sense iniciar el servidor.

```
uaSW/0.3 - Joan Ardiaca Jové 2010
Configuració (uasw.conf) - configuració errònia: PorEscolta 80
Configuració (uasw.conf) - configuració errònia: FooBar 1
Configuració (uasw.conf) - configuració invàlida: MaxFils
```

Posteriorment vaig implementar la rutina principal i la majoria de les seves subrutines. En un principi tan sols preparava l'entorn esperava connexions entrants i mostrava un missatge per la sortida estàndard. És quan arriba la fase en la que implemento les rutines d'atenció a les connexions quan puc començar a fer proves més interessants, ja que es processen les peticions entrants. Les primeres les realitzava a través d'una connexió usant l'eina telnet.

Codi 10 - Exemple de petició i resposta HTTP vàlida usant telnet

```
$ telnet localhost 8080
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
GET / HTTP/1.1

HTTP/1.1 200 OK
Content-length: 209
Server: uaSW/0.3
Date: Sun, 20 Jun 2010 00:35:07 GMT
Last-Modified: Sun, 20 Jun 2010 00:34:56 GMT
Content-type: text/html; charset=utf-8

<html>
<head><title>Pàgina per defecte</title></head>
<body><h1>Benvingut!</h1>
Aquesta és la pàgina d'inici per defecte. Si veus això és que el servidor web
està funcionant correctament.
</body></html>
```

Codi 11 - Exemple petició invàlida usant telnet (mètode no implementat)

```
joan@joan-netbook:~$ telnet localhost 8080
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
TRACE / HTTP/1.1

HTTP/1.1 501 Error
Content-length: 0
Server: uaSW/0.3
Date: Sun, 20 Jun 2010 00:39:47 GMT
Content-type:
```

Més endavant també vaig realitzar diverses proves usant navegadors web. Recullo a continuació una connexió realitzada amb el navegador Chromium, capturada amb l'eina Wireshark.

Codi 12 - Captura de tres peticions en una connexió al servidor usant un navegador

```
GET / HTTP/1.1
Host: localhost:8080
Connection: keep-alive
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US) AppleWebKit/533.4 (KHTML,
like Gecko) Chrome/5.0.375.38 Safari/533.4
Cache-Control: max-age=0
Accept:
```

```
application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png
,*/*;q=0.5
Accept-Encoding: gzip,deflate,sdch
Accept-Language: ca,es;q=0.8,en-US;q=0.6,en;q=0.4,de-DE;q=0.2
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3

HTTP/1.1 200 OK
Content-length: 209
Server: uaSW/0.3
Date: Fri, 18 Jun 2010 01:11:02 GMT
Last-Modified: Fri, 18 Jun 2010 00:34:56 GMT
Content-type: text/html; charset=utf-8

<html>
<head><title>P..gina per defecte</title></head>
<body><h1>Benvingut!</h1>
Aquesta ..s la p..gina d'inici per defecte. Si veus aix.. ..s que el servidor
web est.. funcionant correctament.
</body></html>


GET /favicon.ico HTTP/1.1
Host: localhost:8080
Connection: keep-alive
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US) AppleWebKit/533.4 (KHTML,
like Gecko) Chrome/5.0.375.38 Safari/533.4
Accept: */*
Accept-Encoding: gzip,deflate,sdch
Accept-Language: ca,es;q=0.8,en-US;q=0.6,en;q=0.4,de-DE;q=0.2
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3

HTTP/1.1 404 Error
Content-length: 0
Server: uaSW/0.3
Date: Fri, 18 Jun 2010 01:11:03 GMT
Content-type:


GET / HTTP/1.1
Host: localhost:8080
Connection: keep-alive
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US) AppleWebKit/533.4 (KHTML,
like Gecko) Chrome/5.0.375.38 Safari/533.4
Cache-Control: max-age=0
Accept:
application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png
,*/*;q=0.5
Accept-Encoding: gzip,deflate,sdch
Accept-Language: ca,es;q=0.8,en-US;q=0.6,en;q=0.4,de-DE;q=0.2
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
If-Modified-Since: Sun, 20 Jun 2010 00:34:56 GMT
```

```
HTTP/1.1 304 Not modified
Server: uaSW/0.3
Date: Fri, 18 Jun 2010 01:11:04 GMT
Last-Modified: Fri, 18 Jun 2010 00:34:56 GMT
```

S'han afegit algunes línies en blanc a la captura original per a facilitar la comprensió

Noteu com aquesta captura, a més de mostrar el bon comportament del servidor, mostra la captura d'una sola connexió persistent. Recull una primera petició a un recurs, i una posterior petició condicional al mateix document, el la que el navegador usa la capçalera HTTP `If-Modified-Since` per a avisar al servidor que ja en té una còpia a la memòria caché. Com el document no s'ha modificat, li respon amb el missatge 304 (no modificat). També es realitza una petició a l'arxiu `favicon.ico` entre les ambdues esmentades. Aquesta és la ruta per omissió que utilitza el navegador per a obtenir una icona que representi el document (per exemple, en els marcadors).

Si reproduïu aquesta prova en la versió lliurada, possiblement noteu que no es produeix el mateix resultat, ja que en aquest exemple hi manquen les capçaleres `Content-Length` i `Content-Type` en la resposta amb el codi 304. De fet, aquest és el comportament especificat a l'estàndard, ja que aquestes capçaleres només s'han de incloure si hi ha realment un contingut a enviar, i en el cas d'aquest missatge, no s'envia el document. Aquest és un error que he observat al utilitzar l'eina d'anàlisi de protocols Wireshark. Noteu com en el missatge d'error 404 (que tampoc envia cap document) si que s'inclouen aquestes capçaleres, però en aquest cas `Content-Lenght` val zero i la capçalera `Content-Type` no te cap valor, de manera que sí es segueix el comportament estàndard; tot i que es podrien obviar aquestes capçaleres.

Aquests serien tres exemples dels tipus de proves realitzades, excloent les proves amb l'eina httperf que veurem en l'apartat següent per a analitzar el rendiment del servidor. Evidentment no són les úniques que s'han realitzat, i durant i després de la implementació s'han fet una multitud de proves no documentades, tot i que en general similars a les que hem vist, amb l'objectiu de comprovar la resposta del servidor davant tant de situacions donades en l'ús normal i legítim dels serveis, com de l'ús erroni o potencialment maliciós. Això incloent l'enviament de capçaleres massa grans per a tractar de provocar sobreeiximents i peticions a recursos no vàlids o fora del directori de documents.

També he realitzat algunes proves en quant a l'execució de programes CGI que no recullo en

aquesta memòria. Aquestes m'han permès saber que existeix algun problema a l'hora de transmetre dades al procés del programa CGI, però podria haver-hi més. Pel caràcter preliminar d'aquesta funcionalitat, no crec que aquestes proves siguin rellevants.

Anàlisi de rendiment

Per a comprovar el rendiment que ofereix uaSW, he utilitzat l'eina d'anàlisi de rendiment de servidors HTTP anomenada httperf[21]. Aquest programa realitza peticions al servidor HTTP recull informació sobre el rendiment obtingut. La informació que considero més rellevant i que recullo en aquest apartat són: el nombre de respostes per segon, el temps de resposta mitjà (suma de temps de connexió, temps de resposta i temps de transferència; és a dir, des de l'inici fins a la recepció completa de la resposta), la taxa de transferència i el nombre d'errors que han ocorregut.

Per a realitzar les proves s'ha comprovat el rendiment del servidor a l'hora de respondre quantitats considerables de peticions a dos fitxers: un document HTML molt breu de 209 bytes i un document PDF d'aproximadament 1,3 megabytes. La configuració usada és la mateixa que la que s'ha adjuntat amb el lliurament del producte (límit de 250 treballadors i 30 fils de sobrecàrrega).

Per a comparar el rendiment ofert per uaSW amb un altre servidor web, s'han realitzat les mateixes proves en el popular servidor Apache. S'ha utilitzat la configuració per omissió que s'inclou en la instal·lació d'aquest programari a través dels repositoris de la versió 10.04 d'Ubuntu.

La màquina en que s'han realitzat les provés és un computador portàtil amb un processador Intel Atom N270 (1,6 GHz amb tecnologia HyperThreading), 1 GB de memòria RAM DDR2 i un disc dur SATA amb el sistema operatiu Ubuntu 10.04. L'eina httperf s'executa en la mateixa màquina que el servidor, per tant no hi ha comunicació real per una xarxa, però ens permet comparar el rendiment d'ambdós programes, encara que sigui en una situació artificial. Així també evitem el coll d'ampolla que comporta la xarxa i podríem comparar aquesta situació amb l'ús d'una comunicació de xarxa d'alt rendiment, de la qual no disposo.

Vegem en aquesta taula el rendiment obtingut en diverses proves usant el document PDF de 1,3 megabytes. S'utilitza com a temps màxim d'espera 10 segons; les peticions que tarden més que aquest interval en obtenir respostes es consideren un cas d'error. Es fan tant connexions amb peticions individuals com connexions persistents amb diverses peticions per connexió. Si els servidors responguessin perfectament, cadascuna de les proves duraria 20 segons (en la pràctica prenen una quantitat variable més de temps).

21 httperf és una eina publicada sota una llicència lliure. Més informació disponible a http://www.hpl.hp.com/research/linux/httperf/.

	Apache				**uaSW**			
Proves amb el document PDF (1,3 MB)	Respostes/s	Temps total	Taxa transf.	Errors	Respostes/s	Temps total	Taxa transf.	Errors
Prova 1: 700 connexions / 35 per segon / 1 petició per connexió	**35**	**15 ms**	47.280 KB/s	0 (0%)	**34,7**	**444 ms**	46.239 KB/s	0 (0%)
Prova 2: 1000 connexions / 50 per segon / 1 petició per connexió	**50**	**17 ms**	67.466 KB/s	0 (0%)	**49,5**	**481 ms**	66.881 KB/s	0 (0%)
Prova 3: 1500 connexions / 75 per segon / 1 petició per connexió	**75**	**17 ms**	101.181 KB/s	0 (0%)	**74,2**	**436 ms**	100.155 KB/s	0 (0%)
Prova 4: 2000 connexions / 100 per segon / 1 petició per connexió	**99,9**	**23 ms**	134.886 KB/s	0 (0%)	82,4	4.651 ms	117.038 KB/s	**96 (4%)**
Prova 5: 3000 connexions / 150 per segon / 1 petició per connexió	92,7	7614 ms	127.235 KB/s	**280 (9%)**	132,7	5.079 ms	111.409 KB/s	**1171 (39%)**
Prova 6: 300 connexions / 15 per segon / 5 peticions per connexió	**75**	**42 ms**	101.243 KB/s	0 (0%)	**72,8**	1.296 ms	96.633 KB/s	0 (0%)
Prova 7: 400 connexions / 20 per segon / 5 peticions per connexió	**100**	**47 ms**	135.012 KB/s	0 (0%)	**96,5**	1.483 ms	127.839 KB/s	0 (0%)
Prova 8: 500 connexions / 25 per segon / 5 peticions per connexió	**124,9**	**83 ms**	168.486 KB/s	0 (0%)	95,6	6.091 ms	136.098 KB/s	0 (0%)
Prova 9: 700 connexions / 35 per segon / 5 peticions per connexió	114,3	9.612 ms	155.862 KB/s	0 (0%)	99,6	11.675 ms	133.549 KB/s	**148***
Prova 10: 1000 connexions / 50 per segon / 5 peticions per connexió	102,5	15.369 ms	141.167 KB/s	**231***	116,2	9.394 ms	137.116 KB/s	**445***

*Aquests errors poden ser d'una sola petició o d'una connexió completa. Per això és difícil calcular el percentatge de peticions que produeixen un error.

Veiem que en pràcticament totes les proves el servidor Apache supera el rendiment d'uaSW, com era d'esperar. Tant en el temps total que prenen les peticions, com en les taxes de transferència i d'errors uaSW es troba en gairebé totes les proves per darrera, tot i que tampoc excessivament.

Les xifres del nombre de respostes per segon poden resultar desconcertants (vegeu les proves 5 i 10, on uaSW sembla poder respondre més peticions que Apache, una xifra que sembla no correspondre's amb els altres resultats). Això es degut a que httperf conta qualsevol resposta HTTP, incloent aquelles en que el servidor retorna un codi d'error. La majoria d'errors d'Apache són o bé *timeouts*, o bé el tancament prematur de la connexió, mentre uaSW mostra poc aquest

comportament, però retorna molts més respostes HTTP 503 (servei no disponible) pel fet que entrin en joc els fils de sobrecàrrega. Aquest efecte es podria minimitzar optimitzant la configuració del servidor, tot i que s'ha de tenir en compte que la configuració actual ja és a prop dels límits que imposa el meu sistema, que no em permet tenir més de 382 fils d'execució. És possible que aquest límit es pugui elevar, però és una qüestió que no he investigat en profunditat.

En quant al rendiment per si, veiem que uaSW respon adequadament a taxes de fins a 75 peticions per segon, gairebé 100 si s'utilitzen connexions persistents; mentre Apache aconsegueix unes taxes de 100 i 125 respectivament. Un cop superats aquests límits, es disparen els temps que prenen els servidors en respondre. uaSW mostra uns temps de resposta en general molt superiors (exceptuant els casos en que entren en joc els fils de sobrecàrrega, on es redueix aquest temps, però moltes de les respostes no les podem considerar vàlides, tot i que prenguin menys temps). Si seguim augmentant el nombre de connexions, les taxes d'error són superiors en el cas d'uaSW. En quant a les taxes de transferència, Apache ofereix un millor rendiment, però per un marge escàs.

Tot i que el rendiment ofert per uaSW es notablement menor al d'Apache, crec que es pot considerar un èxit, tenint en compte que Apache ha estat desenvolupat durant anys durant un equip i una comunitat, mentre uaSW s'ha fet en uns mesos per una sola persona. A més, el fet que el servidor arribi a servir fins a 100 cops per segon un fitxer de 1,3 MB sense massa problemes es pot considerar molt satisfactori, i deixa la porta oberta a un millor rendiment si es duen a terme diverses optimitzacions possibles.

Vegem a continuació proves similars realitzades en aquest cas amb un document HTML de 209 bytes. Com la mida de l'arxiu és molt inferior (gairebé és tan curt com les capçaleres HTTP que s'envien amb la resposta), la transferència en si pren molt menys temps i permet un major nombre de peticions per segon.

	Apache				**uaSW**			
Proves amb el document HTML (209 bytes)	Respostes/s	Temps total	Taxa transf.	Errors	Respostes/s	Temps total	Taxa transf.	Errors
Prova 2: 12000 connexions / 600 per segon / 1 petició per connexió	**600**	**2 ms**	325 KB/s	0 (0%)	**599,9**	**10 ms**	264 KB/s	0 (0%)
Prova 3: 15000 connexions / 750 per segon / 1 petició per connexió	**750**	**2 ms**	406 KB/s	0 (0%)	**749,8**	**11 ms**	331 KB/s	0 (0%)
Prova 3: 17000 connexions / 850 per segon / 1 petició per connexió	**850**	**2 ms**	460 KB/s	0 (0%)	**849,7**	**17 ms**	375 KB/s	0 (0%)
Prova 4: 20000 connexions / 1000 per segon / 1 petició per connexió	**1000**	**3 ms**	542 KB/s	0 (0%)	799,3	646 ms	271 KB/s	**2987 (15%)**
Prova 5: 28000 connexions / 1300 per segon / 1 petició per connexió	**1265**	160 ms	704 KB/s	0 (0%)	631,5	933 ms	163 KB/s	**20167 (72%)**
Prova 6: 8000 connexions / 300 per segon / 5 peticions per connexió	**1499,5**	**5 ms**	812 KB/s	0	**1499,8**	**16 ms**	662 KB/s	0
Prova 7: 7000 connexions / 350 per segon / 5 peticions per connexió	1692,1	585 ms	921 KB/s	0	**1741,5**	**58 ms**	769 KB/s	0
Prova 8: 8000 connexions / 400 per segon / 5 peticions per connexió	1482,7	2088 ms	688 KB/s	**540***	1690,3	1049 ms	733 KB/s	**1287***
Prova 9: 10000 connexions / 500 per segon / 5 peticions per connexió	1461	2320 ms	704 KB/s	**2648***	1694	1052 ms	703 KB/s	**3743***
Prova 10: 15000 connexions / 750 per segon / 5 peticions per connexió	1419,9	2233 ms	663 KB/s	**7848***	449,9	2163 ms	126 KB/s	**20890 ***

*****Aquests errors poden ser d'una sola petició o d'una connexió completa. Per això és difícil calcular el percentatge de peticions que produeixen un error.

De nou, hem de comptar en que les proves que causen errors no podem comparar el rendiment pel motiu que ja he comentat abans. En aquests resultats veiem que en quan no s'utilitzen connexions persistents els resultats són similars a les proves de la taula anterior: uaSW ofereix un menor rendiment, tant en quant a la taxa d'errors com en els temps de resposta, taxes de respostes per segon i de transferència. En quant a les proves que utilitzen connexions persistents ens podem sorprendre que uaSW ofereixi un major rendiment en quan a latència i taxa de respostes per segon

(proves 5 i 6). Això molt probablement sigui degut a que la mida de les capçaleres (i el cost de la seva generació) en les respostes HTTP (que en aquesta prova tenen un pes molt major comparat amb el document) és notablement inferior en el cas d'uaSW (181 bytes) que en Apache (284 bytes). Així s'explica també que les taxes de transferència siguin notablement majors en el cas d'Apache.

En aquest cas, uaSW suporta sense degradar el rendiment fins a 850 peticions per segon i gairebé 1750 amb l'ús de connexions persistents, mentre en el cas d'Apache suporta gairebé 1300 i 1500 respectivament.

Com a apunt final, he de reconèixer que la comparació entre uaSW i Apache és un tant injusta, ja que aquest últim té moltes més funcionalitats i és més complet que uaSW. En aquest sentit, uaSW te un clar avantatge. Per l'altra banda, Apache conta amb molts anys de proves, desenvolupament i optimització.

Tot i així, els resultats reflecteixen un rendiment del servidor uaSW que considero molt satisfactori.

Errors i millores realitzables

UaSW és en el seu estat actual un programa bastant complet, estable i funcional, que compleix acceptablement els seus objectius. Tot i així, també he de reconèixer que te diverses mancances i errors que no el fan tan bon servidor com podria arribar a ser-ho.

Aquí tracto de recopilar algunes de les millores que es podrien fer al servidor tal i com està, la majoria sense que impliquin grans canvis en la seva estructura i funcionament.

- Es podria implementar la transmissió HTTP comprimida amb l'algorisme gzip. Això augmentaria l'eficiència del servidor i del seu ús de la xarxa, de manera que seria possible oferir un major rendiment. Seria una funcionalitat relativament senzilla d'implementar amb avantatges notables sobre l'estat actual del programa.

- El codi del servidor és actualment exclusivament per a sistemes GNU/Linux. Utilitza algunes crides (com `accept4()`) que són exclusives d'aquest sistema operatiu, tot i que la majoria estan definides en l'estàndard POSIX. Es podria aconseguir, sense excessives modificacions, que fos totalment conforme amb POSIX per a permetre la seva compilació i execució per a tots els sistemes operatius que implementen aquest estàndard i donar així al programa una major portabilitat. Es podria assolir modificant el codi o utilitzant directives de preprocessador per a fer una compilació condicional, mantenint així un ús més eficient del sistema operatius en sistemes GNU/Linux.

- Els fitxers de registre actualment només s'utilitzen quan el programa s'executa en segon pla de la manera definida pel programa, ja que la forma d'escriure en aquests fitxers actualment està implementada redirigint els descriptors de `stdout` i `stderr` cap als fitxers. Si be aquest disseny és funcional, no te realment sentit que no s'utilitzin quan el programa no s'executa en segon pla. Això és així perquè aquesta funcionalitat no va ser implementada fins a les últimes etapes del desenvolupament i no va ser tinguda en compte en el disseny. Seria positiu modificar aquest funcionament per tal que els fitxers de registre s'utilitzin en tots els casos. A més, s'hauria de millorar la forma en que s'enregistren els esdeveniments en aquests fitxers, per exemple informant de la data i hora de cadascun, incloure quin tipus de programari utilitza el client en cada petició, etcètera.

- Es podria millorar molt la conformitat del servidor amb l'estàndard HTTP 1.1. Actualment uaSW no es comporta en moltes situacions com ho especifica l'estàndard i obvia diversos requeriments

  - Actualment no te en compte moltes de les capçaleres de les peticions, com Accept, `Accept-Charset` i similars; les capçaleres per a realitzar peticions condicionals exceptuant `If-Modified-Since`; i altres.

  - Actualment permet l'ús de mètodes no idempotents en les connexions persistents, fet que no està permès per l'estàndard.

  - El servidor només entén les dates en el format RFC1123, quan l'estàndard requereix explícitament que s'acceptin dos altres formats (RFC 1036 i el format utilitzat per `asctime()` d'ANSI C)[22], tot i que també exigeix als clients usar només el primer format.

  - El servidor només accepta peticions de recursos sencers. El protocol estableix mecanismes per a permetre demanar i servir continguts parcials.

  - Es podrien suportar altres mètodes HTTP a part de GET, HEAD i POST definits en l'estàndard. Si be alguns són escassament utilitzats (com PUT i DELETE), altres (com OPTIONS) poden ser interessants per a alguns usos.

- uaSW actualment suporta només el protocol HTTP en la seva versió 1.1. La versió 1.0, tot i que està disminuint el seu ja escàs ús progressivament, encara és la que utilitzen alguns clients minoritaris. Per les grans similituds entre ambdues versions, no hauria de ser excessivament costós assolir la compatibilitat, tot i que augmentaria la complexitat del servidor.

- El programa actualment només suporta l'ús del protocol IP en la seva versió 4. Aquesta, tot i ser la més utilitzada amb diferència, està sent reemplaçada per la versió 6. El canvi de versió és quelcom que es produirà inevitablement (de fet, ja s'està produint) per diversos motius, un dels quals és l'exhauriment d'adreces IPv4 disponibles, que són limitades i avui gairebé totes estan en ús. Suportar el protocol IPv6 no implicaria grans canvis en el codi del servidor.

22 Vegeu més informació sobre les dates en el protocol HTTP 1.1 a http://tools.ietf.org/html/rfc2616#section-3.3

- El suport de programes CGI del servidor és actualment experimental i no funciona com hauria. Aquesta funcionalitat donaria més interès al servidor (i donaria sentit al suport que dona al mètode HTTP POST, ja que si no és per a executar programes CGI, les dades enviades pel client en la request-entity són obviades). Amb l'ús correcte de programes CGI es podria enllaçar el servidor amb altres tecnologies, com per exemple PHP. Es podria anar encara més lluny i implementar altres solucions relacionades com SimpleCGI i FastCGI.

- Tal i com s'implementa actualment la funcionalitat multifil del servidor i les connexions persistents, els fils d'execució es passen la major part del temps esperant a rebre dades del client. Això podria millorar, ja que representa un consum innecessari de recursos. Una possible solució seria tenir un fil d'execució dedicar a monitoritzar la recepció de dades dels clients (vigilant totes les connexions obertes) que creés nous fils d'execució treballadors al tenir dades disponibles.

- Quan uaSW s'executa amb drets de superusuari, tot el programa s'executa en mode privilegiat. Això no és un bon comportament en quan a la seguretat, i el millor seria utilitzar els privilegis només allà on és necessari. La implementació d'aquests canvis no seria complexa.

- Finalment, en un afany de millorar la capacitat i el rendiment del programa, enlloc de l'actual solució multifil per la qual he optat, es podria implementar una solució híbrida que utilitzés tant fils d'execució com múltiples processos. Això alleujaria alguns dels límits actuals del servidor (com el nombre d'arxius oberts o el límit en el nombre de fils d'execució) i es podria assolir un major rendiment. A més, es podria implementar mantenint igual gran part del codi actual.

Conclusions

El desenvolupament d'uaSW, tot i haver-me suposat un esforç important, ha estat també apassionant. Veure com he estat capaç de desenvolupar en solitari un programa de certa complexitat amb èxit i comprovar que funciona correctament en situacions diverses ha estat una motivació important. Implementar per primer cop un sistema de computació en paral·lel usant fils d'execució ha estat un aprenentatge important, així com l'aprofundiment que he fet en el protocol HTTP. Quan el servidor ja estava funcionant, també ha estat interessant veure com influïen els canvis en la configuració i algunes optimitzacions en el rendiment i comportament.

Si bé es poc probable que aquest programa s'usi mai en entorns de producció, estic orgullós de la meva producció. A més d'una tasca d'enginyeria, el considero una petita obra d'art.

Agraïments

Als meus pares, Josep i Anna Maria, que han fet possible (entre moltíssimes altres coses) que avui estigui tan a prop de finalitzar els meus estudis.

Al Roger i l'Andrea, per aguantar-me a mi i les meves dèries inclús en els pitjors moments.

A la Laura per ser com és, no li cal més.

A tants dels meus amics que s'interessen pel que faig, encara que la majoria no ho entenguin, i a més em donin el seu suport.

A tots aquells que han col·laborat o format part de la gran comunitat del programari lliure, que han fet possible que avui tinguem una gran varietat de programari lliure i d'alta qualitat i que han ajudat a fer del món un lloc una mica millor.

Fonts utilitzades

- S'han utilitzat intensivament les pàgines “man” del meu sistema GNU/Linux, especialment les dedicades al desenvolupament per a plataformes POSIX.
- Beej's Guide to Network Programming: Using Internet Sockets (http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html)
- How HTTP and CGI Work (http://www.evc-cit.info/cit042/how_cgi_works.html)
- HTTP Made Really Easy (http://www.jmarshall.com/easy/http/)
- Introduction to Parallel Computing (https://computing.llnl.gov/tutorials/parallel_comp/)
- POSIX thread (pthread) libraries (http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html)
- POSIX Threads Programming (https://computing.llnl.gov/tutorials/pthreads/)
- POSIX Threads: Semi-FAQ Revision 5.2 (http://www.cognitus.net/html/howto/pthreadSemiFAQ.html)
- RFC #2616: Hypertext Transfer Protocol – HTTP 1.1 (http://www.enderunix.org/docs/eng/daemon.php)
- RFC #3875: The Common Gateway Interface (CGI) Version 1.1 (http://tools.ietf.org/html/rfc2616)
- String Handling: <string.h> (http://www.cs.cf.ac.uk/Dave/C/node19.html)
- TCP/IP options for high-performance data transmission (http://articles.techrepublic.com.com/5100-10878_11-1050878.html)
- The Linux HTTP Benchmarking HOWTO (http://www.xenoclast.org/doc/benchmark/HTTP-benchmarking-HOWTO/)
- Unix Daemon Server Programming (http://www.enderunix.org/docs/eng/daemon.php)
- What is HTTP pipelining? (http://www.mozilla.org/projects/netlib/http/pipelining-faq.html)
- Altres fonts per a resolució de dubtes (cerques per Internet, informació general de Wikipedia, fòrums, blogs, etcètera)

Apèndix A: Altres servidors web

Com ja hem vist, hi ha un gran nombre de projectes similars a aquest, tant d'altament complexos com sorprenentment senzills. Veurem aquí alguns exemples dels que podem trobar a la xarxa i d'altres que formen part de la història de la web.

- Apache HTTP Server (http://httpd.apache.org/): el servidor web per excel·lència, és desenvolupat per una comunitat oberta amb el suport de Apache Software Foundation. Programat en C, suporta diversos sistemes operatius, incloent tots els majoritaris. Té diverses funcionalitats ampliables a través de mòduls. És avui en dia el servidor més utilitzat, serveix més de la meitat de llocs web existents.

- Internet Information Services (http://www.microsoft.com/iis) és la solució comercial oferta per Microsoft. Té moltes funcionalitats i suporta diversos protocols, cosa que el fa més que un servidor web i és capaç de funcionar com a servidor FTP i de correu, per exemple. Només està disponible per al sistema operatiu de la mateixa empresa, Windows. Actualment serveix aproximadament un quart dels llocs web existents, sent el segon servidor més usat. Alguns problemes greus de seguretat en el passat li han donat una fama indesitjada.

- nginx (http://www.nginx.org/) és un servidor orientat a oferir un alt rendiment amb un baix consum de recursos portable a diversos sistemes operatius. És una inclusió relativament nova en aquest àmbit i ha tingut un èxit efervescent pel seu rendiment i la necessitat de molts sistemes de poder suportar cada cop una major càrrega. En quant a utilització, es troba en tercer lloc.

- lighttpd (http://www.lighttpd.net/) és un servidor HTTP optimitzat per a oferir un alt rendiment, mantenint la conformitat amb els estàndards i sent segur i flexible. Va ser originalment ideat per a solucionar el que s'anomena el problema c10k: com atendre 10.000 connexions en paral·lel en una sola màquina. És utilitzat per a servir diversos llocs web que suporten altes càrregues.

- mini_httpd (http://www.acme.com/software/mini_httpd/) és un servidor web orientat a la senzillesa, sense pretensió d'oferir un alt rendiment. Desenvolupat per Jef Poskanzer en el llenguatge C, m'ha servit com a inspiració per a la realització d'aquest projecte, tot i que el

funcionament d'ambdós es radicalment diferent.

- micro_httpd (http://www.acme.com/software/micro_httpd/) és un servidor web desenvolupat pel mateix autor que mini_httpd. També desenvolupat en C, no està dissenyat per a ser utilitzat realment, sinó que és un exemple d'un servidor web extremadament petit, que ofereix les funcionalitats bàsiques en unes 200 línies de codi. Es pot considerar més una curiositat que un servidor i molt probablement és un dels servidors més petits que existeixen.

- CERN httpd (més tard conegut com a W3C httpd, http://www.w3.org/Daemon/) va ser el primer servidor web desenvolupat. Va ser creat per Tim Berners-Lee ("el pare de la web") l'any 1990 amb ajut d'altres. S'executava inicialment sota el sistema operatiu NeXTSTEP. Implementa totes les funcionalitats del protocol HTTP i és de domini públic.

- NCSA HTTPd va ser el segon servidor HTTP que es va crear. Desenvolupat per Robert McCool i altres a l'NCSA, va arribar a servir el 95% dels llocs web en les primeres etapes de la web fins a l'aparició d'Apache, el qual es basava en aquest servidor.

- Oracle iPlanet Web Server (http://www.oracle.com/technology/products/iplanetws/index.html) originalment desenvolupat per Netscape, va ser una de les primeres solucions comercials per a grans empreses. Avui segueix existint en propietat d'Oracle i sota un altre nom. El seu codi ha estat publicat i està disponible per a diversos sistemes operatius.