Memòria del TFC

Creació del nucli d'un servidor Web

XicHttpd

Miquel Fontanals Rofes
ETIS

David Carrera Pérez
Consultor TFC

9 de Desembre del 2003

Memòria del TFC: Creació del nucli d'un servidor Web; XicHttpd

per Miquel Fontanals Rofes,

David Carrera Pérez
Consultor TFC

9 de Desembre del 2003

Historial de revisions

Revisió 1.0 12-10-2003 Revised by: mfr
Creació del document inicial
Revisió 1.1 03-11-2003 Revised by: mfr
S'afegeix el capítol 2
Revisió 1.2 09-12-2003 Revised by: mfr
Es revisen alguns comentaris del consultor. El començament de l'etapa d'implementació m'ha fet replantejar algunes parts del disseny,
Revisió 1.3 09-01-2004
Arranjaments finals

Dedicatòria

Dedico aquest projecte a la meva família per el seu suport incondicional.

Sumari

Índex de taules

Índex de figures

Índex d'exemples

Estructura i contingut de la memòria

Aquest document està estructurat en varis blocs i alguns annexos:

- Conceptes teòrics fonamentals
- Anàlisi i disseny del servidor
- Servidor XicHttpd. Descripció, funcionament i proves efectuades

La primera part recull informació teòrica sobre el funcionament dels servidors Web, i sobre el protocol a implementar. També s'intenta donar una visió sobre l'estat dels servidors webs existents al mercat.

La segona part té com a objectiu obtenir l'anàlisi i el disseny del servidor que s'implementarà, de manera que es comença per establir-ne els requisits, per després fer-ne un anàlisi en funció de la funcionalitat que s'espera obtenir. Finalment es fa el disseny de l'aplicació seguint les pautes de l'anàlisi previ.

En la darrera part es descriuen els detalls de programació del servidor, així com els jocs de proves a que s'ha sotmès i els resultats dels *benchmarks* que s'hagin provat.

Finalment els annexos recullen tota la informació complementaria al projecte. La part més important és la documentació del servidor.

> **Nota:** El document original d'aquesta memòria està escrit en SGML utilitzant el DTD de DocBook per tal de poder generar simultàniament aquesta versió en pdf per facilitar-ne la lectura i la impressió, i una altra en html que de fet és la versió bona de la memòria.

Capítol 1. Conceptes teòrics.

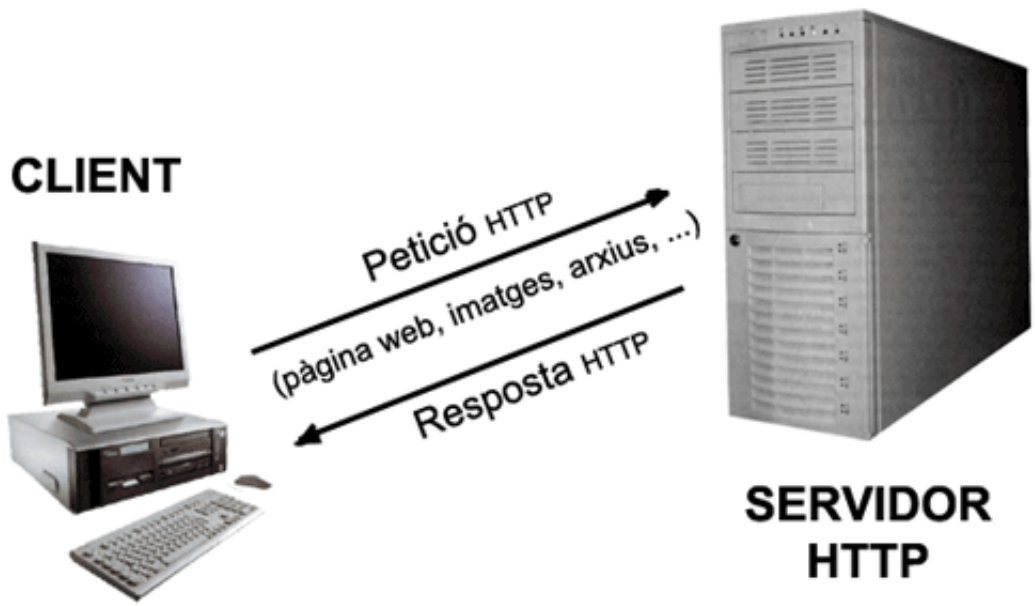
1.1. Introducció

Podem definir un servidor, en termes generals, com un ordinador que ofereix serveis, és a dir informació, a d'altres ordinadors en base als seus requeriments. D'aquesta manera qualsevol ordinador és un servidor en potència, i només li cal el programari adient.

Aquest tipus de programari acostuma a estar dissenyat seguint el model *client/servidor*, el qual defineix tant l'estructura de les aplicacions, com els protocols necessaris de comunicació entre els diferents elements que en formen part. Usualment la part servidor d'aquest model s'executa com un dimoni (*daemon*), que resta a l'espera de rebre peticions del client amb un determinat protocol de comunicació.

Així doncs, un servidor web és essencialment una entitat de programari que s'executa com un *daemon*, i resta a l'espera de rebre connexions en un port determinat (usualment el 80 per peticions estàndard i el 443 per connexions http segures, tot i que els ports acostumen a ser configurables). Quan el servidor detecta una connexió d'algun client tot l'intercanvi d'informació que hi ha entre les parts (peticions i respostes) es realitza seguint les pautes del protocol HTTP.

Figura 1-1. Exemple client/servidor



Un recurs demanat per un client pot fer referència a:

- Un arxiu: pàgina web, imatges, o en general qualsevol tipus d'arxiu.
- Una pàgina web dinàmica: el servidor crida a una tercera aplicació (amb paràmetres o sense) i en retorna la sortida al client. Cal no confondre aquestes pàgines generades en temps d'execució abans de ser enviades al client, amb les pàgines de contingut dinàmic de la banda del client, com per exemple les que utilitzen *Javascript*.

Alguns dels llenguatges utilitzats per generar pàgines web dinàmiques com el PHP[1] poden carregar-se com un mòdul al servidor Web, així s'executaran de forma més ràpida i eficient ja que ho faran al mateix espai de memòria que el mateix servidor, estalviant-se així les crides al sistema per fer el canvi de context.

1.2. Una mica d'història

1.2.1. HTTP

A finals dels anys vuitanta i començament dels noranta un grup d'ivestigadors del CERN (*Centre Europeu d'Investigació Nuclear*) van definir un protocol per accedir de manera senzilla i còmoda a la informació distribuïda entre les diferents seus del centre. Posteriorment aquest sistema es va estendre a altres organitzacions i països, i va néixer el que coneixem com a *World Wide Web* (WWW).

Aquest protocol s'anomenà HTTP (*Hypertext Transfer Protocol*), i a mesura que els desenvolupadors l'implementàven li van anar afegint noves funcionalitats fins que al 1996 es va publicar el document [RFC 1945], impulsat per Tim Berners-Lee, amb la definició de la versió HTTP/1.0 del protocol. Només un any més tard apareix l'especificació HTTP/1.1 recollida al [RFC 2068], i la última actualització del mateix apareix a l'[RFC 2616] al juny de 1999.

1.2.2. Servidors web (Apache)

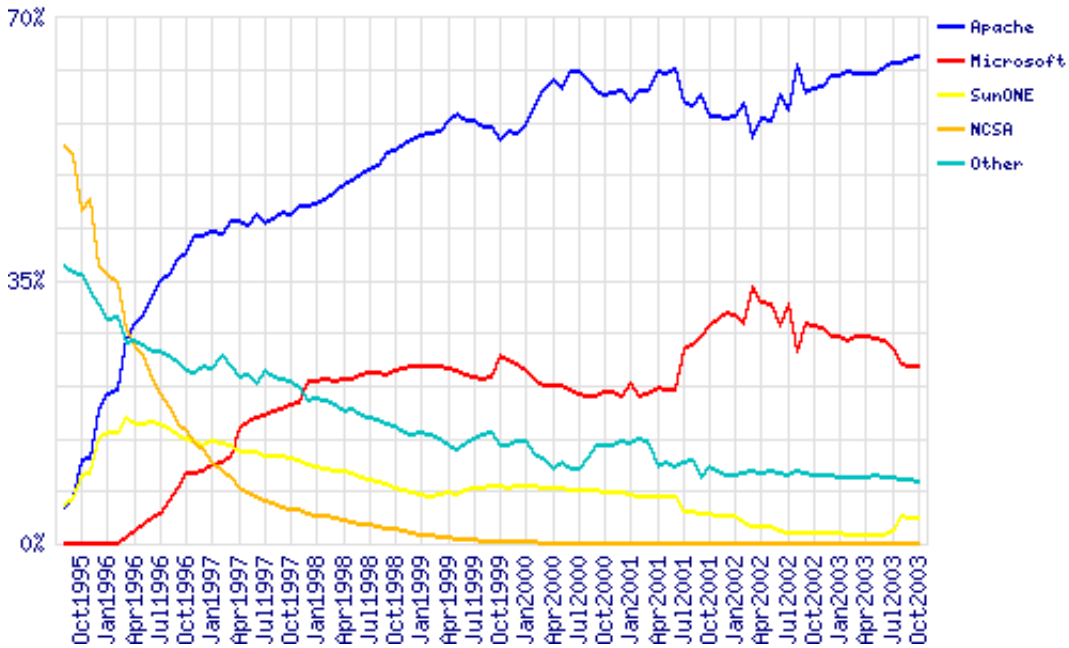
Als principis del noranta, el servidor web més popular i estès era el HTTPD de NCSA. Però a mitjans del 1994 els principals desenvolupadors del projecte el van abandonar, deixant a milers de webmasters davant del codi de NCSA. Posteriorment sen van implementar millores, i es corregiren errors de l'HTTPD que eren distribuïts en forma de pedaços del codi font. No va passar massa temps en que la quantitat de pedaços va fer el projecte intractable, i la necessitat de treure un nou estàndard de la comunitat de codi obert es va convertir amb alguna cosa més que important, de vida o mort.

Així a l'abril del 1995 apareix la primera versió de l'Apache (v 0.6.2) basada en la 1.3 de NCSA. El nom d'Apache surt precisament de que inicialment era una versió apedaçada de l'HTTPD de NCSA (*"A PAtCHy" release*). Posteriorment es va continuar desenvolupant el projecte, fins que l'1 de desembre del mateix any n'apareix la primera versió. Només sis mesos més tard, a mitjans del 1996, el servidor web Apache ja havia desbancat l'HTTPD de NCSA.

1.2.3. Panorama actual

Actualment podem trobar una gran quantitat de servidors web, tan comercials com lliures. Aquí només farem esment dels més importants des del punt de vista quantitatiu. De tota manera sen pot veure una llista bastant complerta a l'apèndix B.

Figura 1-2. Ús dels servidors web



Dades extretes de *http://news.netcraft.com* (Octubre del 2003).

Segons la mateixa font el percentatge de servidors web a internet era:

- Apache 64,61 %
- Microsoft 23,46 %
- SunONE 3,50 %
- Zeus 1,68 %

Veiem doncs com l'Apache, des de la seva aparició en la comunitat del codi obert, ha estat capdavanter en el mon dels servidors Web degut a que respecte els estàndards, és escalable, s'integra amb una gran quantitat de llenguatges de programació, és de construcció modular, multi-plataforma, ofereix un gran rendiment i estabilitat, és gratuït i a més gaudeix d'una merescuda popularitat.

Per la seva banda els servidors de Microsoft (IIS i PWS) prenen la seva importància degut al fet que hereten la simplicitat d'ús del seu tan extés sistema operatiu Windows, amb el que s'integra com una part més del sistema junt amb totes les altres tecnologies de la mateixa casa.

Bastant lluny ja hi trobem l'i-Planet formant part de la plataforma SunONE de SUN Microsystems, el qual es tracta d'un servidor Web d'altes prestacions, i la característica més rellevant és potser el seu suport sempre actualitzat de les tecnologies Java, més encarat als servidors d'aplicacions.

Finalment hi trobem el servidor Zeus de la companyia Zeus Technology dissenyat especialment per oferir una gran velocitat de resposta fins i tot amb quantitats crítiques de peticions, tot i això, el seu preu i el fet que només està disponible per plataformes UNIX/Linux fa que les empreses optin per altres solucions.

1.3. El protocol HTTP

En aquest apartat s'intenta donar una visió sintetitzada dels trets bàsics que caracteritzen el protocol HTTP. L'objectiu és utilitzar-lo de base de cara a recollir els requisits del protocol que haurà d'implementar el nostre servidor.

1.3.1. Introducció

EL protocol de transferència d'HiperText (*Hypertext Transfer Protocol*) és un protocol client/servidor senzill i eficient que defineix les regles que han de seguir les comunicacions entre el servidors web i els clients (navegadors, software de descàrrega, etc.).

HTTP està basat en operacions senzilles de solicituts-respostes (*Request-Response*), de manera que el client envia un missatge amb les dades de la solicitut al servidor, i aquest envia un missatge de resposta al client amb les dades sobre l'estat de l'operació i el seu possible resultat. Totes les operacions poden fer referència a l'objecte de recurs sobre el que actuen, i la manera de referir-se a aquests objectes és per mitjà de la seva URL (*Uniform Recource Locator*).

Les característiques més rellevants són:

- Tota la informació codificada en els missatges es transmet per mitjà de caràcters de 8 bits, el que permet de transmetre qualsevol tipus de recurs respectant-ne el format.
- Per a transferències multimèdia se n'identifica el format per mitjà de la seva classificació MIME[2], de manera que el client sempre sap el què li arriba i com tractar-ho.

- Existeixen tres comandes bàsiques: **GET** per demanar un recurs, **POST** per enviar informació al servidor i **HEAD** per demanar informació sobre un recurs. N'hi ha algunes més però el seu ús no està massa estès.
- En l'HTTP/1.0 la persistència era opcional i es controlava per mitjà d'una capçalera especial, però en l'HTTP/1.1 s'implementa la persistència per defecte, eliminant així la sobrecàrrega en totes les connexions.
- No es mantenen els estats, és a dir, cada operació s'efectua de manera independent, sense tenir en compte les transaccions anteriors, ja sigui del mateix client o de diversos.
- Cada objecte sobre el que actuen les operacions és identificat per la informació de situació del final de la URL.

1.3.2. Etapes d'una transacció HTTP

De qualsevol transacció HTTP sen poden distingir les següents etapes:

1. El client accedeix a una URL, ja sigui escrivint-la directament en el navegador, o per mitjà d'un hiperenllaç.
2. El client Web decodifica la URL, separant-ne les diferents parts, així n'identifica el protocol (HTTP en el nostre cas), la direcció DNS o IP del servidor, el possible port (per defecte el 80), i l'objecte demanat al servidor.
3. S'obre una connexió TCP/IP amb el servidor sobre el port 80 (o el que s'hagi especificat).
4. Es fa la petició: S'envia la comanda adient (**GET**, **HEAD**, **POST**, ...), seguida de la URL de l'objecte demanat, la versió del protocol, i un conjunt de capçaleres amb diferents tipus d'informació.
5. El servidor respon a la petició enviant un missatge de resposta que conté el codi d'estat, una serie de capçaleres entre les que destaquem el tipus MIME de la informació, seguit de la pròpia informació.
6. Tancament de la connexió TCP/IP.

Cal notar que si s'utilitzen connexions persistents (HTTP/1.1) els passos 4 i 5 s'anirien repetint fins que una de les dues parts tanqués la connexió explicitament, o fins que passés un cert temps d'inactivitat.

Els coneixedors del protocol TCP/IP s'adonaran que l'HTTP explicat fins ara, fa un ús ineficient de les característiques de rendiment del TCP/IP com és l'ús de finestres de connexió: Posem per cas que el client accedeix a una plana amb quinze imatges, aleshores el client faria una petició, i s'esperaria a rebre la imatge per fer la següent petició, i així successivament fins a les quinze vegades, si a més resulta que les imatges són molt petites (de la mida d'un píxel per decorar taules per exemple), tenim que mai s'arriba a omplir la finestra TCP provocant una ineficiència considerable en l'ús de la xarxa.

El **Pipelining** és una tècnica prevista per a la implementació de clients i servidors que permet de multiplexar múltiples peticions, i llurs respostes en una mateixa transferència i aprofitar així les característiques d'eficiència del TCP/IP, així se soluciona el problema descrit en l'anterior pàrraf, de manera que el client faria totes les peticions encadenades l'una al darrere de l'altra sense esperar per cada

una, i enviant-les totes de cop, el servidor per la seva banda prepararia les respostes en un buffer intermig, i quan tingui prou dades les enviaria al client. Cal que el servidor vigili de no fer esperar massa les respostes, òbviament no es tracta d'aprofitar l'eficiència per una banda penalitzant el rendiment per l'altra.

Veiem ara un exemple concret:

Exemple 1-1. Exemple de transacció HTTP

1. Un client solicita la URL *http://www.foo.com*
2. S'obre una connexió TCP/IP al port 80 del servidor *www.foo.com*
3. El client fa la solicitut enviant, per exemple:
```
GET / HTTP/1.1
Host: www.foo.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.0.0) Gecko/20020623
Accept: text/html, text/plain, image/*, application/*, video/*, */*;q=0.1
Accept-Language: ca, es-es;q=0.66, en;q=0.33
Accept-Encoding: gzip, deflate, compress;q=0.9
Accept-Charset: ISO-8859-15, utf-8;q=0.66, *;q=0.66
Keep-Alive: 300
Connection: keep-alive
Cookie: user_id=2003.9.25.14.0.28.188; idioma=a
<CRLF>
```
4. El servidor respon amb la següent informació:
```
HTTP/1.1 200 OK
Date: Sun, 19 Oct 2003 11:25:53 GMT
Server: Apache/1.3.28 (Unix)
Last-Modified: Tue, 15 Oct 2002 09:21:26 GMT
Etag: "24499-1075-3dabde16"
Accept-Ranges: bytes
Content-Length: 4213
Content-Type: text/html
Age: 34
<CRLF>
<HTML><HEAD><TITLE>Foo Web Page</TITLE>
<meta name="keywords" ...
...
```
5. Es tanca la connexió.

1.3.3. Estructura dels missatges HTTP

El protocol HTTP només defineix dos tipus de missatges: Les peticions o *Request* i les respostes o *Response*. Cada missatge està format per una línia d'inici, cap o més camps de capçalera i una línia en

blanc indicant el final de les capçaleres, i finalment el cos del missatge si s'escau. Les capçaleres segueixen el format definit a l'[RFC 822 [9]], de manera que cada camp ocupa una línia acabada amb `CRLF`, i s'usa ASCII per la codificació dels caràcters.

Els missatges doncs seguiran el següent esquema:

- ***Request.***

```
Comanda SP URL SP Versió-HTTP CRLF
Capçalera-1 CRLF
Capçalera-2 CRLF
...
Capçalera-n CRLF
CRLF (línia buida)
Cos-missatge (opcional)
```

- ***Response*.**

```
Versió-HTTP SP Codi-estat SP descripció-codi CRLF
Capçalera-1 CRLF
Capçalera-2 CRLF
...
Capçalera-n CRLF
CRLF (línia buida)
Cos-missatge (opcional)
```

On tenim que `SP` significa espai en blanc, i `CRLF` són els caràcters `CR` (*Carry return*) y `LF` (*Line Feed*)

1.3.4. Descripció de les comandes

Les comandes són aquelles operacions que el client demana sobre un recurs al servidor HTTP. La seva sintaxi és molt senzilla com s'ha vist en l'exemple de petició de l'apartat anterior:

```
Comanda SP URL SP Versió-HTTP CRLF
```

En cas que la comanda necessiti especificar alguna característica especial es fa ús de les capçaleres del missatge.

Les diferents comandes reconegudes a l'especificació del protocol [RFC 2612] són les següents:

- **OPTIONS.** Mètode utilitzat per demanar informació sobre les opcions de comunicació disponibles en l'intercanvi de peticions/respostes sobre un recurs. D'aquesta manera el client pot determinar les opcions i/o requeriments associats a un recurs, o les capacitats del servidor sense que això impliqui actuar sobre el recurs. Per exemple, es poden determinar les comandes permeses sobre un determinat recurs, o la codificació que s'emprarà per transmetre'l.

- **GET.** El mètode **GET** és el que s'utilitza per demanar qualsevol tipus de recurs, ja siguin pàgines web, imatges, o en general qualsevol tipus d'arxiu. Òbviament si l'URI fa referència a un programa o script (CGI, ASP, PHP, ...) aquest serà executat al servidor i sen retornarà la seva sortida.
- **HEAD.** Aquest mètode és idèntic al mètode **GET**, excepte que no s'envia el cos del missatge, sinó només les capçaleres.
- **POST.** Mètode utilitzat per enviar informació (o dades) al servidor, de manera que el servidor les passa al recurs identificat per l'URI (normalment un script o programa) per tal que les processi. El servidor depenent del resultat de la operació pot retornar la sortida del programa, o bé no retornar res (codi 204 No Content).
- **PUT.** Mètode que indica que el cos del missatge s'ha de deixar a l'URI indicada en la petició, o bé reemplaçar-lo si aquest ja existeix.
- **DELETE.** El mètode **DELETE** demana al servidor que elimini el recurs identificat per l'URI de la petició.
- **TRACE.** Aquest mètode permet al client de veure quina informació rep el servidor. Posem per exemple que hi ha un *proxy* entre mig de la comunicació, aleshores aquest servidor intermediari, quan rep una petició del client, la reenvia al servidor ficant-li les seves pròpies capçaleres. Així si el client fa un **TRACE** sobre un determinat recurs, el servidor li retornarà les capçaleres que ha rebut al cos del missatge de resposta.
- **CONNECT.** Aquesta és una comanda reservada per ser utilitzada amb els servidors *proxy* quan aquests poden convertir-se en un túnel dinàmicament (com ara un túnel SSL).

1.3.5. Les Capçaleres

Les capçaleres HTTP són un conjunt de variables que s'adjunten als missatges per tal de descriure alguna característica, o de complementar-ne el significat. El format de les capçaleres és el següent:

```
Nom-Varialbe: SP Valor-Variable
```

On segons el tipus de variable, `Valor-Variable` pot prendre un o més valors separats per algun caràcter separador.

Les capçaleres definides al [RFC 2616] són:

- **Capçaleres generals.**

```
Cache-Control
Connection
Date
Pragma
Trailer
Transfer-Encoding
Upgrade
Via
Warning
```

- **Capçaleres de les peticions (*Request*).**

  ```
  Accept
  Accept-Charset
  Accept-Encoding
  Accept-Language
  Authorization
  Expect
  From
  Host
  If-Match
  If-Modified-Since
  If-None-Match
  If-Range
  Max-Forwards
  Proxy-Authorization
  Range
  Referer
  TE
  User-Agent
  ```

- **Capçaleres de les respostes (*Response*).**

  ```
  Accept-Ranges
  Age
  ETag
  Location
  Proxy-Authenticate
  Retry-After
  Server
  Vary
  WWW-Authenticate
  ```

- **Capçaleres del contingut (*Entity*).**

  ```
  Allow
  Content-Encoding
  Content-Language
  Content-Length
  Content-Location
  Content-MD5
  Content-Range
  Content-Type
  Expires
  Last-Modified
  extension-header
  ```

Tot i això, la majoria d'elles no són imprescindibles per al funcionament del protocol, sinó que més aviat s'utilitzen en determinats tipus de transferències, o per donar informació addicional. Cal mencionar, però, que la capçalera `Host` si que és obligatòria per al totes les peticions HTTP/1.1 i el seu valor ha de fer referència al nom del servidor hoste i al port (si és diferent del 80) del recurs que es demana. També és necessari que tots els missatges que portin cos (`entity-body`) n'indiquin la seva mida, ja sigui per mitjà de la capçalera `Content-Length`, o per la combinació d'altres[3].

1.3.6. Codis d'estat del servidor

Com s'ha vist quan parlàvem dels missatges, la primera línia del missatge de resposta del servidor conté el codi d'estat, fent referència a la petició solicitada. Podem veure els codis d'estat definits per l'especificació del protocol a la següent taula:

Taula 1-1. Codis d'estat del servidor HTTP

Codi	Comentari
1xx	**Informació**
100	Continue
101	Switching Protocols
2xx	**Èxit**
200	OK
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
205	Reset Content
206	Partial Content
3xx	**Redirecció**
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
306	(No utilitzat)
307	Temporary Redirect
4xx	**Error del client**
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout
409	Conflict
410	Gone

Codi	Comentari
411	Length Required
412	Precondition Failed
413	Request Entity Too Large
414	Request-URI Too Long
415	Unsuported Media Type
416	Requested Range Not Satisfiable
417	Expectation Failed
5xx	**Error del Servidor**
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Suported

1.3.7. Notes sobre la cau de pàgines i els servidors Proxy

La majoria de clients Web mantenen una còpia local de les pàgines visitades anteriorment, amb la data de modificació. L'objectiu no és altre que reduir el nombre de connexions a internet, així quan el client accedeix a una web, primer es comprova si la pàgina es troba a la cau, si hi és s'envia un **HEAD** al servidor per comprovar la data de modificació de la plana, si és la mateixa que la que té a la cau, s'usa la còpia local, i sino es fa un **GET** per obtenir-ne la última versió, i a més s'actualitza el contingut de la cau.

Un servidor *Proxy* es situa entre el client i el servidor Web, de manera que el client es connecta al *Proxy* per demanar un recurs enlloc de fer-ho al servidor original, i el *Proxy* actua de manera molt semblant al que em explicat.

El principal avantatge d'utilitzar servidors *Proxy* en les organitzacions és que proporcionen un únic punt d'entrada/sortida a internet, el que permet tenir un control més eficient sobre l'ús d'internet i la seguretat, i a més sen redueix el tràfic considerablement, ja que els clients acostumen a accedir a conjunt semblant de pàgines.

Tot i això en alguns casos les cau poden tenir copies obsoletes, sobretot de pàgines generades dinàmicament. De tota manera el protocol HTTP permet de controlar certs aspectes de com gestionar les còpies de la cau per mitjà d'algunes capçaleres.

Notes

1. PHP és l'acrònim de *Hypertext Pre-Processor*
2. MIME és l'acrònim de *Multipurpose Internet Mail Extension*.
3. Per exemple quan el tipus de transferència és "*chunked*" s'indica la posició de les parts que s'envien amb la capçalera `Content-Range`.

Capítol 2. Anàlisi i Disseny del servidor Web

2.1. Anàlisi de requisits

En el capítol anterior ja hem vist quines eren les característiques més rellevants del protocol HTTP, i també que els servidors Web s'implementen seguint l'esquema client/servidor donant servei a múltiples clients, de manera que ara ja podem establir els requisits que haurà de tenir el servidor XicHttpd. Començarem doncs per analitzar-ne les característiques essencials, i a continuació establirem les funcionalitats addicionals que se li volen implementar.

2.1.1. Requisits del servidor

1. Cal implementar la part servidor de l'esquema *client/servidor* sobre el protocol TCP/IP.
2. El servidor ha de restar a l'espera de rebre connexions al port HTTP per defecte (port 80). Tanmateix s'hauria de permetre de canviar aquest port estàticament, és a dir, abans de l'execució del servidor ja sigui com a paràmetre de la línia de comandes, o per mitjà d'algun fitxer de configuració.
3. S'han de poder atendre simultàniament les peticions de múltiples clients. Això requerirà d'algun mecanisme que permeti l'atenció paral·lela de cada connexió.
4. Un cop acceptada la connexió d'un nou client, tot l'intercanvi d'informació de la capa d'aplicació entre aquest i el servidor es farà seguint la versió HTTP/1.1 del protocol HTTP. Donat que en aquesta versió es contempla la persistència per defecte, caldrà que el servidor tanqui les connexions després d'un cert temps d'inactivitat. En el cas que les peticions facin referència a una altra versió, el servidor haurà de respondre amb el codi d'estat `505 HTTP Version Not Supported`.
5. Els mètodes HTTP inicialment acceptats per aquest servidor seran **GET** i **HEAD**. En cas de rebren algun altre, el servidor haurà de respondre amb el codi d'estat `501 Not Implemented`.
6. Totes les peticions faran referència a un recurs situat en el sistema de fitxers de la màquina on s'executa el servidor, de manera que el servidor haurà de demanar el recurs al sistema de fitxers i servir-lo al client (mètode **GET**), o informar-ne de les característiques (mètode **HEAD**). Es poden donar dos situacions d'error: o bé que no es tingui permís per accedir al fitxer (o el directori on es troba), o bé que el fitxer no existeixi; en ambdós casos el servidor respondrà amb el codi d'estat `404 Not Found` ja que en aquesta versió no es fa distinció de clients, ni es suporta cap tipus d'autentificació.
7. Per cada petició caldrà que el servidor en verifiqui la sintaxis. Algunes situacions d'error que es podrien donar són:
   - L'identificador del recurs (`Request-URI`) és massa gran. El servidor respon amb el codi d'estat `414 Request-URI Too Long`.
   - Error general de sintàxis, o falta la capçalera `Host`. El servidor respon amb el codi d'estat `400 Bad Request`.
8. Les capçaleres que incorporaran els missatges de resposta són les següents:

- `Date` : Amb el valor de la data i hora local del servidor en format GMT.
- `Server` : El nom i la versió del servidor Web, i la plataforma on s'executa.
- `Last-Modified` : La data i hora de la última modificació del recurs demanat.

- `Content-Length` : La mida en bytes del recurs demanat, si es tracta d'un **GET** serà la mida del cos del missatge (`entity-body`).
- `Content-Type` : El tipus MIME del recurs demanat.
- `Content-Encoding` : Aquesta capçalera només hi serà present en el cas que s'hagi comprimit el cos del missatge, per tant primer caldrà que el client accepti aquest tipus de transferència. El seu valor serà `gzip` (veure següent apartat).

9. Cal tenir present, que existeix la possibilitat de passar paràmetres per mitjà de la URL a les pàgines Web (pas per **GET**), caldrà doncs que a l'hora de descodificar l'URI es tingui en compte que el caràcter separador és '`?`'.

2.1.2. Funcionalitats addicionals

1. **Pipelining.** Es permetrà que el client faci múltiples peticions enlloc d'haver-se d'esperar per cada resposta, així s'aconsegueix mantenir la finestra TCP/IP el màxim de plena possible, reduint així els temps d'inactivitat de la connexió.
2. **Compressió.** Amb l'objectiu de reduir el volum de tràfic que circula per la xarxa, sempre que sigui possible s'enviarà el cos del les respostes (`entity-body`) comprimit. Per tant, caldrà que el client accepti aquest tipus de transferència, així el servidor haurà de comprovar que la petició contingui la capçalera `Accept-Encoding` amb el valor `gzip` (de moment no es contemplen ni `compress` ni `deflate`), i el servidor enviarà el cos del missatge comprimit amb gzip i la capçalera `Content-Encoding: gzip`. En cas contrari el servidor enviarà el recurs tal qual es troba en el sistema de fitxers prescindint de la capçalera `Content-Encoding`.
3. **Logs.** El servidor registrarà una serie d'informació útil per als administradors. A continuació es descriu el contingut de cada fitxer:
   - `acces.log`: Contindrà a cada línia l'adreça IP de cada client, la data i l'hora local del servidor, la primera línia de la petició i el codi d'estat de la resposta del servidor en el següent format:

     ```
     Adreça-IP - [Hora-Local] "Mètode Recurs Versió" "Codi-Resposta Descripció"
     ```

     Per exemple:

     ```
     80.36.24.66 - [Sun. 02 Nov 2003 11:55:38 GMT] "GET /index.html HTTP/1.1" "HTTP/1.1 200
     ```

   - `errors.log`: Contindrà els errors que es puguin produir en l'execució del servidor. De moment el seu format és lliure, i quedarà definit en la fase de programació però com a mínim hauria d'inclouré un codi d'error amb una breu descripció del mateix.

Nota: Durant les primeres setmanes de recollida d'informació, vaig poder veure algunes implementacions del mètode **POST** que no tenien res a veure amb els estàndards CGI[1], ni amb el pas de paràmetres a llenguatges d'script. Donat que el servidor s'executarà en un entorn controlat, si es disposa de temps s'intentarà fer una implementació "particular" del mètode **POST** definint les regles bàsiques que hauria de tenir l'aplicació que en processi les dades.

2.1.3. Anàlisi

Podem distingir dues parts diferenciades a partir dels requisits anteriors, així tenim que els tres primers fan referència a la implementació del servidor, i la resta a la implementació del protocol.

Per satisfer el tercer requisit es podrien utilitzar subprocessos, utilitzar algun mecanisme de selecció de connexions (com *select* de la llibreria *sockets* del C) o bé utilitzar fils d'execució diferenciats. S'obta pels fils d'execució, ja que seleccionant connexions en realitat no tindríem paral·lelisme, i utilitzant subprocessos correm el risc de sobrecarregar la màquina en excés en càrregues altes, a part de la penalització que provoca la creació del mateix (crida al sistema, assignació de memòria, etc), a banda que totes les comunicacions entre processos es farien per mitjà de crides al sistema (*PIPE*, *mmap*, *IPC*, *signals*, etc.). D'altra banda els fils comparteixen per defecte l'espai de memòria, ens ofereixen mecanismes d'accés segur a les variables compartides, i un control adient de cada fil, tot per mitjà de les funcions de la llibreria escollida. Un altre tema és el tractament que faci el sistema operatiu dels fils, però aquesta discussió queda lluny de l'objectiu d'aquest document.

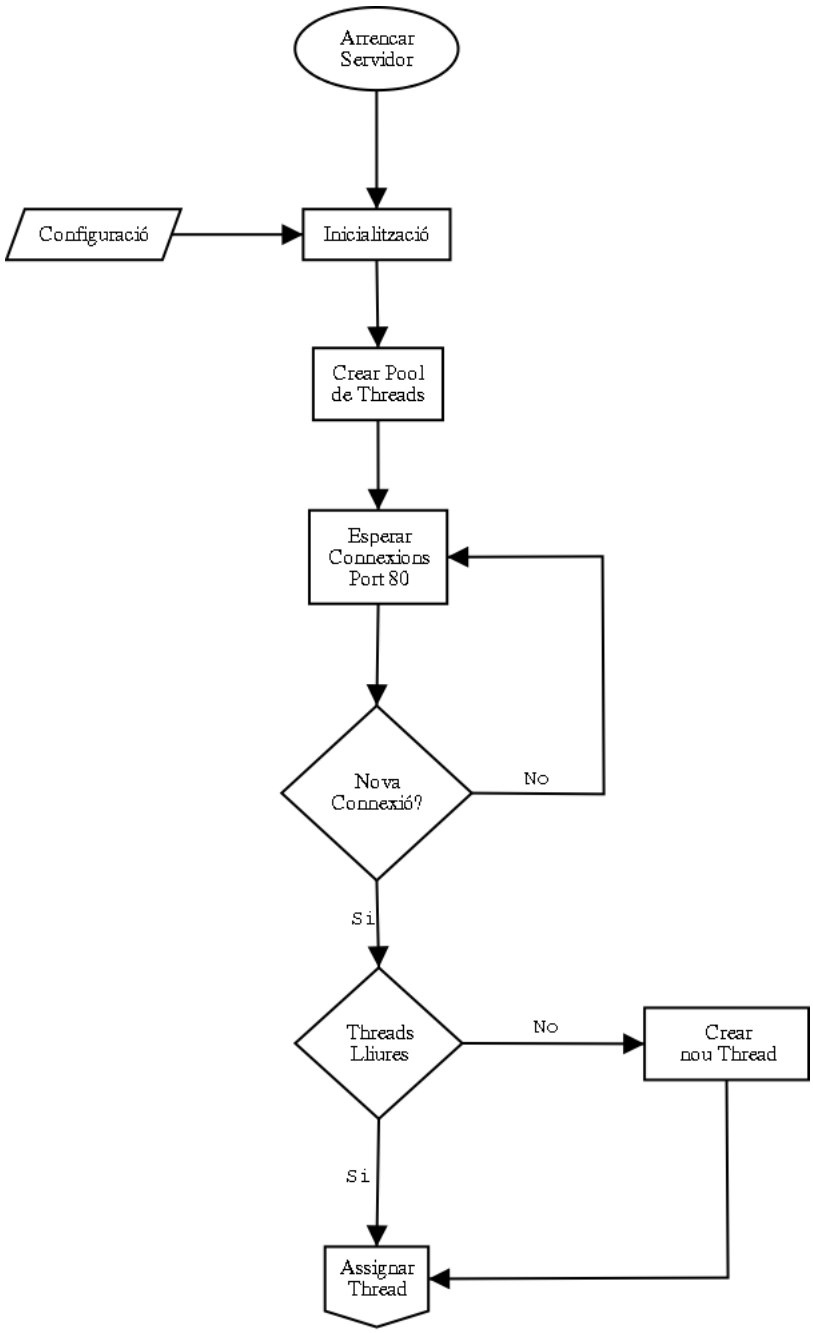
Tenint en compte el que s'ha escollit, inicialment es crearan una serie de fils estàticament que s'assignaran a cada connexió entrant, un cop estiguin tots assignats sen crearan de nous dinàmicament per atendre les noves connexions, això si, prioritzant sempre els fils estàtics que ja hagin acabat la seva tasca. Cal notar que si es permet de configurar la quantitat de fils estàtics i dinàmics, així com el *timeout* de les connexions podrem fer un *tunning* del servidor i comparar els resultats dels *benchmarks* escollits.

Vegem doncs els diagrames de flux sobre el funcionament del servidor Web. En primer lloc tenim el diagrama del fil principal fins a la creació/assignació dels fils per cada connexió, el següent fa referència a cada fil que és on estarà implementat el protocol HTTP.

2.1.3.1. Tasques del servidor

Vegem a continuació la descripció de tasques a realitzar pel procés principal del servidor Web, a partir del diagrama de flux següent:

Figura 2-1. Diagrama de flux del servidor Web



- **Inicialització.** Es llegiran les dades del fitxer de configuració que es creguin oportunes. Aquestes quedaran completament definides en la fase de disseny però com a mínim hi haurà el nombre de

threads estàtics que es crearan, el valor del *timeout*, el nombre màxim de clients que es permeten i el port on haurà d'escoltar el servidor si és diferent del port HTTP per defecte. També es crearan les estructures de dades necessàries i s'inicialitzaran les variables globals que calguin per al funcionament del servidor.

Es crearà el *socket* del servidor per tal que escolti en el port especificat.

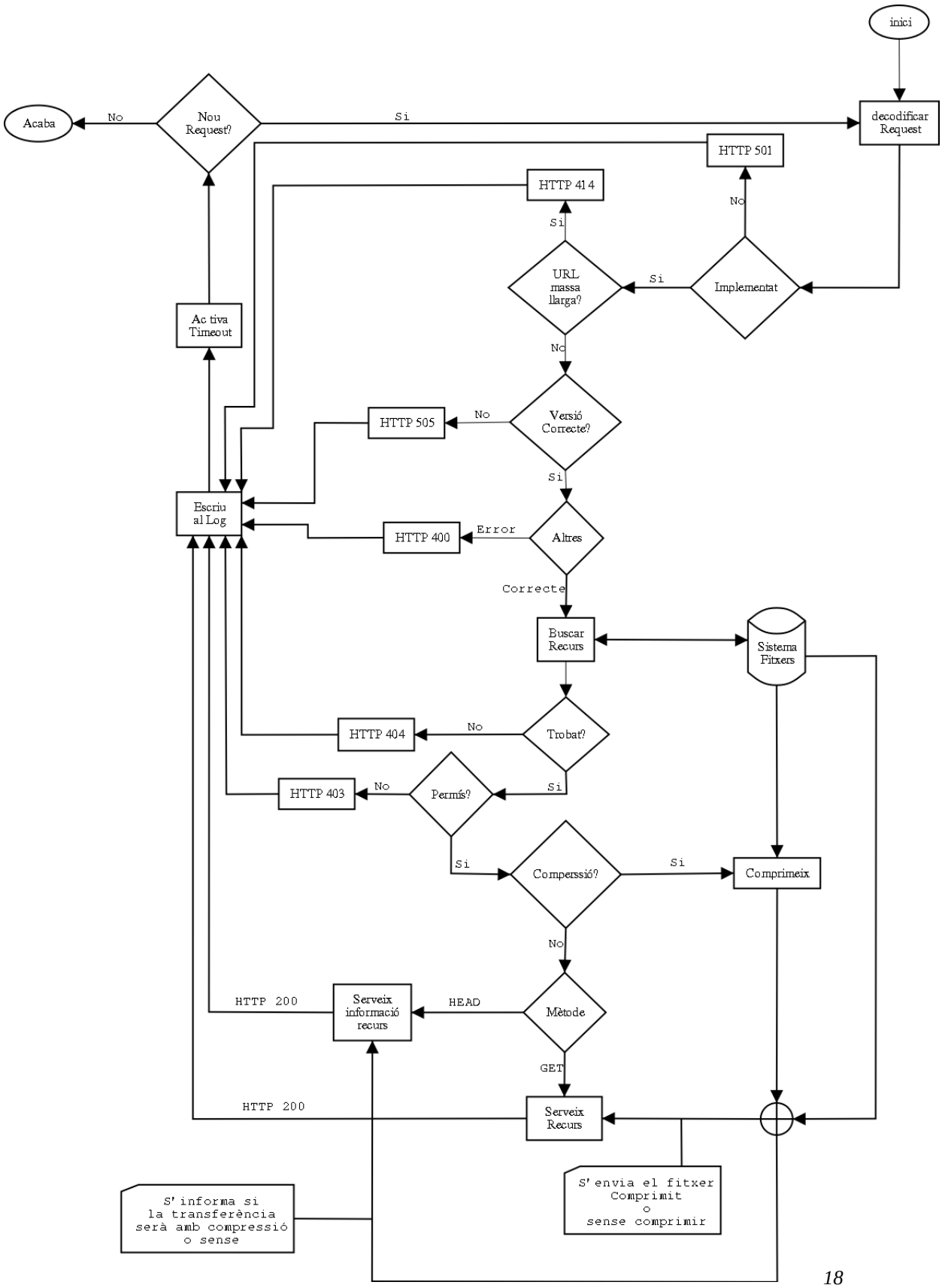
Caldrà també definir els senyals (*signals*) acceptats pel servidor per tal que el sistema operatiu el pugui tancar de manera segura.

- **Crear *Pool* de *Threads*.** Es crearan i s'inicialitzaran tants *threads* com s'indiqui al fitxer de configuració deixant-los preparats per quan arribin connexions
- **Esperar Connexions.** El servidor entrarà en un bucle infinit deixant el socket preparat per rebre les connexions. Quan arribi una connexió se li assignarà un *thread* del *pool* estàtic si n'hi ha de lliures, i si no sen crearà un de nou sempre i quan no s'hagi arribat al màxim de clients. El *thread* assignat rebrà per paràmetre l'identificador de la connexió del client, a més dels que defineixi la llibreria utilitzada (per exemple les funcions a executar).

2.1.3.2. Funcionament de cada fil

A continuació tenim el diagrama de flux de les tasques que haurà d'executar cada *thread* des de que se li assigna una connexió, fins que aquesta es tanca:

Figura 2-2. Diagrama de flux de cada fil d'execució



- **Descodificar *Request*.** Es llegirà la petició que arriba pel *socket* del client, per ordre de lectura farem les següents comprovacions:

  - Si el mètode no està implementat retornarem un *Response* amb el codi d'estat `501 Not Implemented`.
  - Si la mida de l'URI és massa gran per la mida definida en el programa el codi d'estat de la resposta serà `414 Request-URI Too Long`.
  - Si la versió del protocol indicada és diferent a HTTP/1.1 retornarem la resposta amb el codi d'estat `505 HTTP Version Not Supported`.
  - Si falta la capçalera `Host`, o bé es detecta qualsevol altre malformació sintàctica durant tota la lectura de la petició, es retornarà la resposta amb el codi d'estat `400 Bad Request`.

  A mesura que es llegeixen les capçaleres de la petició, s'aniran descartant totes excepte el `Host` ja comentat, i la capçalera `Accept-Encoding`, la qual utilitzarem per determinar si es pot enviar el recurs comprimit amb gzip o no.

- **Cercar Recurs.** S'intentarà accedir al recurs demanat en la petició, si el recurs no existeix es retornarà la resposta amb el codi d'estat `404 Not Found`, si d'altra banda intentem accedir a un directori on no es té permís, o bé no es té permís sobre el recurs demanat el codi d'estat de la resposta serà `403 Forbidden`. Si es troba el recurs es procedirà de la següent manera:
  - Si s'accepta compressió, es comprimirà el recurs per determinar-ne la mida i omplir la capçalera `Content-Length`, en aquest cas caldrà afegir també la capçalera i el valor `Content-Encoding: gzip`, sinó s'omplirà amb la primera amb la mida del fitxer sense comprimir.
  - S'afegira al missatge de resposta, les capçaleres `Date`, `Server`, `Last-Modified`, `Content-Type`, a més de les determinades de l'apartat anterior.
  - El codi d'estat de la resposta serà `200 OK`, si es tracta de la resposta a un **HEAD** s'enviarà el missatge, si la petició era un **GET** s'adjuntarà el recurs, comprimit o no segons calgui, després d'una línia en blanc en el mateix missatge.
- **Servir Recurs i Servir Informació.** Queda determinat pel punt anterior.
- **Escriu al Log.** Cada vegada que s'envia una resposta s'enregistra al fitxer `acces.log` seguint el format que hem detallat en l'apartat de funcionalitats addicionals.
- **Activa Timeout.** Cada vegada que s'envia una resposta s'activarà el *timeout* per tancar la connexió si no arriben més peticions en el temps establert.

  > **Nota:** Quan s'implementi el *Pipelining* hi haurà una cua de peticions, de manera que fins que no es respongui a la última no s'activarà el *timeout*.

2.2. Disseny del servidor Web

2.2.1. Tecnologies

El llenguatge de programació escollit per implementar el servidor Web és el C, i la plataforma de treball i execució serà el sistema operatiu Linux (compatible i386), junt al compilador que porta per defecte GCC.

Els directori arrel de treball seguirà l'estructura típica per aquest entorn:

- `src` : Contindrà els fitxers en codi font (`.c`)
- `src/include` : Contindrà els fitxers de capçalera (`.h`)
- `bin` : Contindrà el programa executable resultat de la compilació.
- `doc` : Contindrà la documentació del servidor web.
- `etc` : Directori per defecte on residirà el fitxer de configuració del servidor `xichttpd.conf`.
- `log` : Directori per defecte on s'escriuran els fitxers de log del servidor.
- `html` : Directori per defecte on hi ficarem els fitxers html, imatges, i qualsevol recurs que hagi de servir el nostre servidor Web.
- `html/cgi-bin` : Directori per defecte que contindrà els CGIs.

Les llibreries més rellevants del llenguatge C que s'utilitzaràn són:

- `socket` : Conté totes les funcions necessàries per la creació i el control dels canals de comunicació en xarxa.
- `pthread` : Conté totes les funcions necessàries per la creació i el control de fils d'execució, i defineix maneres segures d'accedir a les variables compartides.
- `zlib` : Llibreries de compressió.

2.2.2. Mòduls

Per tal de tenir les funcionalitats estructurades, les fonts dels programes es distribuiran en diversos fitxers:

1. **Fitxers de capçaleres**

- `server.h` - Contindrà les definicions de les estructures necessàries per al funcionament del servidor, i el nom d'algunes variables globals. Entre elles destaquem la que fa referència a una connexió del client i la que conté les dades de configuració del servidor:

```
typedef struct {
pthread_t fil; // El fil assignat a la connexió
unsigned long int id; // L'identificador
int socket; // L'identificador del socket
struct sockaddr_in adreca_client;
time_t ctemps; // Temps en que s'ha creat
time_t atemps; // Temps de l'últim accés
c_request *dades; // Capçaleres del client (http.h)
} connexio;

typedef struct {
char fitxer_config[255];
char dir_base[255];
char dir_cgi[255];
char dir_html[255];
char f_log[255];
char f_err[255];
char mime_types[255];
char nom_host[64];
short int port;
struct sockaddr_in addr;
int min_threads;
int max_threads;
time_t timeout;
} sconf;
```

- `http.h` - Conté les definicions de les estructures per emmagatzemar les capçaleres per les peticions i per les respostes, així com les definicions dels codis d'estat del servidor. En destaquem les que fan referència a les del capçaleres de les peticions i a la de les respostes:

```
typedef struct {
char metode[16]; // El mètode solicitat
char versio[16]; // La versió HTTP
char URI[1024]; // El recurs
char Host[64]; // Capçalera Host
char AcceptEncoding[16]; // Capçalera Accept-Encoding
char get_p[1924]; // Paràmetres del get
int ContentLenght; // Capçalera Content-Length (POST)
char Connection[64]; // Capçalera Connection (per als close)
} c_request;

typedef struct {
char estat[64]; // El codi d'estat
char Date[64]; // Capçaleres de la resposta
char Server[128];
char LastModified[64];
char ContentLength[64];
char ContentType[128];
char ContentEncoding[16];
} c_response;
```

```
/* Necessari per afegir al buffer de sortida dades binaries */
typedef struct {
char sbuf[MAX_SBUF]; // El buffer de sortir
int mida; // La mida del mateix
} buf_out;
```

- `mime.h` - Contindrà la definició de la llista amb els tipus mime, i un punter al primer element:

```
typedef struct mime {
char ext[16]; // L'extensió
char tipus_mime[64]; // El tipus mime
struct mime *next; // Punter al següent de la llista
} mime;

mime *mime_types; // Punter al primer de la llista
```

- `errors.h` - Contindrà les definicions dels missatges d'error del servidor.
- `main.h` - Contindrà tots els `include` necessaris per l'aplicació, tan els de les llibreries de necessàries com els nostres. Tots els fitxers font inclouran aquest fitxer de capçaleres per evitar duplicitats.

2. **Fitxers font**

- `main.c` - Contindrà el mètode principal que acceptarà un paràmetre amb el nom del fitxer de configuració que volem utilitzar si és diferent del que s'agafarà per defecte. També hi haurà les funcions de *logging* necessàries.
- `server.c` - Contindrà les funcions necessàries per l'execució del servidor. Les funcions principals són:
  - `void init ();` - Passos necessaris per arrencar el servidor, bàsicament es cridarà a les funcions que segueixen.
  - `void llegir_config ();` - Llegirà el fitxer de configuració i actualitzarà les variables necessàries.
  - `void crea_socket ();` - Crearà el socket del servidor actualitzant la variable global `server_sock`, posteriorment obtindrà l'adreça IP del servidor fent una cerca DNS seguint l'ordre establert a `/etc/hosts.conf` en funció del nom de host especificat al fitxer de configuració. Finalment s'assignarà l'adreça al socket. Qualsevol error detectat provocarà la finalització del servidor.
  - `void crea_fils_estatics ();` - Crearà els *threads* estàtics i els deixarà a punt per ser activats en rebre peticions.
  - `void escolta (int socket);` - Entrarà en un bucle infinit a l'espera de rebre connexions pel `socket` passat per paràmetre. Quan arribi una connexió se li assignarà un fil de la cua si n'hi ha de lliures, i si no sen crearà un de nou sempre hi quan no s'hagi arribat al límit de clients.
  - `void prepara_senyals ();` - Canviarà la rutina d'atenció a alguns signals.
- `http.c` - Contindrà la implementació del protocol que haurà d'executar cada fil. El paràmetre d'entrada `cid` d'algunes funcions fa referència a l'índex de la taula de clients on es troben les

dades del client actual, de manera que en puguem llegir o escriure els diferents valors. Les funcions principals són:

- `int deco_request (int cid);` - Decodificarà la petició del client. Els possibles valors de retorn seran:
  - `CE400` : Si el *request* està mal construït.
  - `CE411` : Un **POST** sense `Content-Length`.
  - `CE414` : Si l'URI és massa llarga.
  - `CE501` : Si el mètode demanat no està implementat.
  - `CE505` : Si la versió HTTP no està suportada.
  - `0` : Si tot és correcte.
  - `-1` : Si per qualsevol motiu no hi ha més dades per llegir però encara no hem decodificat el *request* completament.
- `void set_mime (c_response *resposta, char *extensio);` - Escriurà la capçalera `Content-Type` de la resposta actual apuntada per `resposta` en funció de `extensio`. Els tipus MIME seran carregats a l'inicialitzar el servidor. Si el fitxer no té extensió, el tipus serà `text/plain`, i si l'extensió és desconeguda el tipus per defecte serà `application/octet-stream`.
- `int envia (int sock, char *dades, int mida_dades, buf_out *sbuf, int mida_sbuf);` - Enviarà dades pel *socket* `sock`. Si hi ha *pipelining* les dades s'afegiran al buffer de sortida `sbuf`, si no hi ha *pipelining* o el buffer de sortida és ple s'enviaran les dades directament. Si `dades` és **NULL** es força l'enviament immediat de buffer de sortida.
- `int prepara_error (int ce, int cid, buf_out *sbuf);` - Prepararà el missatge d'error indicat per `ce` amb el codi d'estat, la versió, les capçaleres necessàries, a més de construir el missatge html descriptiu visible per al client. Tot el missatge s'afegirà al final del buffer de sortida `sbuf`, i es cridarà a la funció `envia`.
- `int cerca_recurs (int cid, buf_out *sbuf);` - Cercarà el recurs demanat en la petició dins el sistema de fitxers local, si tot és correcte prepara el missatge de resposta dins `sbuf` i cridarà a la funció `envia` on el paràmetre `dades` apuntarà al recurs. Els possibles valors de retorn seran els següents, excepte en els casos que es cridi a les funció `xic_cgi` que retornarà el que aquesta retorni (veure més avall):
  - `CE403` : Si no es té permís de lectura sobre el recurs demanat, o sobre el directori on es troba el recurs (*Forbidden*).
  - `CE404` : Si no es troba el recurs demanat.
  - `0` : Si tot és correcte.
  - `-1` : Si es produeix algun error en accedir al recurs.
- `int xic_cgi (int cid, buf_out *sbuf, char *f_cgi);` - Executarà el cgi especificat a `f_cgi`, esperarà a que aquest finalitzi i forçarà el tancament de la connexió. Els possibles valors de retorn són:
  - `CE500` : Error intern del servidor en executar el cgi.
  - `0` : Si tot ha anat correctament.

    - `-1` : Si sorgeix algun altre error inesperat.

- `int xic_gzip (int cid, char *file_in, int mida_in, c_response *resposta, buf_out *sbuf);` : S'encarrega de la compressió en gzip del fitxer `file_in`, donat que aquesta funció sempre serà cridada per `cerca_recurs` si per qualsevol motiu falla la compressió es retornarà un valor diferent de `0`, així `cerca_recurs` encara podrà intentar d'enviar el recurs sense comprimir. Per qualsevol error es retornarà `-1`.
- `void http_control (int cid);` - Aquesta funció és la que controlarà totes les altres, i és la que serà cridada a l'inici de cada *thread*.

3. **Fitxer de configuració**

- `xichttpd.conf` - A no ser que s'especifiqui el contrari per la línia de comandes, per defecte es llegirà aquest fitxer del directori `/etc`, `../etc`, `./` per aquest ordre de cerca, la sintaxis i les variables que s'acceptaran es mostren a continuació, qualsevol línia que comenci amb el caràcter # serà ignorada:

```
# Directori base on resideixen l'estructura de directoris del servidor
DIR_BASE = "/usr/local/xichttpd"

# Directori on residiran els cgi
DIR_CGI = "path/contingut/html/cgi-bin"

# Directori on residirà el lloc Web (fitxers html, imatges, fitxers, etc.)
DIR_HTML = "path/contingut/html"

# Nom del fitxer per logar els accessos al servidor amb el path complert
FITXER_LOG = "/usr/local/xichttpd/log/acces.log"

# Nom del fitxer per logar els errors interns del servidor
FITXER_ERR = "/usr/local/xichttpd/log/error.log"

# Path i nom del fitxer d'on carregar els tipus MIME
MIME_TYPES = "/etc/mime.types"

# El nom o l'adreça IP del host on s'executa el servidor
# Cal que estigui present al fitxer /etc/hosts, és a dir,
# ha de tenir una entrada DNS vàlida
NOM_HOST = "Natasza"

# El nombre mínim de threads en espera
MIN_THREADS = "5"

# El nombre màxim de clients simultanis
MAX_CLIENTS = "150"

# El valor del timeout de les connexions en segons
SERVER_TIMEOUT = "60"

# El port on escoltarà el servidor
PORT = "80"
```

Notes

1. CGI és l'acrònim de *Common Gateway Interface*

Capítol 3. Implementació

3.1. Introducció

Per tal de dur a terme la implementació del XicHttpd i no quedar-me a mig camí, he decidit separar la programació en tres fases, de manera que al finalitzar-ne cada una el servidor tingui una funcionalitat independent de les fases posteriors. Els objectius de cada fase són els següents:

- **Fase 1:** Programar el servidor per tal que resti a l'espera de rebre una única connexió simultània, i la implementació de les funcionalitats mínimes del protocol HTTP. També en aquesta fase es llegeix el fitxer de configuració i s'actualitzen les variables necessàries tot i que no s'utilitzin totes encara.
- **Fase 2:** Programació dels *threads* i del control dels mateixos, de manera que convertim el producte de la fase anterior en un servidor multi-fil que permeti l'accés de varis clients simultanis. També caldrà programar els *signals* que el servidor tractarà per tal de finalitzar la seva execució de manera segura.
- **Fase 3:** Programar les funcionalitats addicionals com ara l'enviament de contingut comprimit amb *gzip* i del sistema de *loggin* complert. També si em dona temps, implementar el mètode **POST** per solicitar contingut dinàmic en un entorn controlat.

3.2. Fases de la implementació

3.2.1. Fase 1

En aquesta fase, l'esforç es centra bàsicament en la implementació del protocol HTTP. Tot i així, en un primer moment vaig tenir certs problemes amb els *includes*, ja que el fet de tenir-los repartits feia que sorgissin problemes al compilar, per això finalment s'obta per tenir-los tots centralitzats al fitxer `main.h`, de manera que només cal incloure aquesta referència en tots els fitxers font i tot aclarit.

En aquest moment no em va semblar adient ni necessari d'utilitzar el port 80 per les primeres proves, ja que això implica tenir permisos de superusuari (de fet per qualsevol per sota del 1024), i sempre es pot escapar alguna coseta (mai millor dit) que provoqui algun efecte no desitjat a la màquina, així que en un primer moment se n'utilitza un d'aleatori, i després un per sobre del 5000 ja que, recordem que el sistema gestiona automàticament el rang 1024-5000 i no és gaire aconsellable interferir-hi.

L'altra plat nou per mi era obtenir la informació que necessitava dels fitxers, i comprovar-ne els permisos d'accés, ja que calia preparar les capçaleres i la resposta, però no volia tenir el fitxer obert mentre feia això, sobretot si es tractava de la resposta a un **HEAD** on necessito tota la informació però no em cal obrir el fitxer per res. Coneixia una funció anomenada `acces()` que em resolia el primer problema de permisos, però continuava necessitant alguna altra crida per la mida i la data de l'última modificació, així que després de passejar-me per diferents planes del manual sobre el tractament de fitxers, en una d'elles vaig trobar la referència a la crida `stat()`, en obrir-la vaig trobar el que buscava: recollir la informació

del fitxer d'un sol cop, junt als permisos i si la crida fallava només em calia fer un cop d'ull a la variable errno per saber-ne el motiu. La operació és tan senzilla com això:

Exemple 3-1. Exemple d'ús de la crida `stat()`

```
#include <sys/types.h>
#include <sys/stat.h>
...
struct stat st;
char *fitxer = "exemple.txt";
...
if (stat (fitxer, &st) != 0) {
if (errno == EACCES) // No tenim permisos
return CE403;
else return CE404; // Es poden comprovar tots si es vol
}
...
/* Ara a l'estructura st hi tinc tot el que em cal
 * st.st_mode - Màscara de permisos
 * st.st_mtime - El moment de l'última modificació
 * st.st_size - La mida
 * ....
 * man stat per més informació.
 */
```

Després de les primeres compilacions, i proves amb el telnet sembla que el XicHttpd contesta correctament, i retorna el recurs o els errors HTTP quan toca, així que em decideixo a fer la primer prova de foc, i obro el meu estimat mozilla, per fer-ho utilitzo la versió HTML d'aquest document ja que inclou varies pàgines i algunes amb imatges. L'índex s'obre correctament, portat per l'entusias-me començo a seguir els enllaços fins que arribo a una plana que inclou dues imatges, una de les quals no es carrega. En un primer moment no me'n preocupo gaire, i segueixo endavant... sembla que vagi bé, reinicio el servidor i ho torno a provar i ja funciona però no em convenç, així que netejo la memòria cau del mozilla i recarrego la plana, altre cop la imatge no es carrega... Després de mirar i remirar el codi i de múltiples `printf` no sé trobar el problema, fins que se m'encen la llumeta i faig una sessió amb l'ethereal[1] activat, i descobreixo que el mozilla, igual que altres navegadors que he provat més endavant, obre més d'una connexió quan li convé, el que em molesta una mica perquè un mateix client m'ocuparà varies connexions, però que hi farem. Això sí, des d'aquest moment fins al final del projecte l'ethereal es converteix en el meu company inseparable a cada prova.

La primera prova amb l'Internet Explorer de Microsoft també va fallar, però amb la nova eina vaig detectar de seguida que ves a saber per quins sets ous enviava salts de línia addicionals en les peticions, els descartem i problema solucionat. Es dona aquesta fase per finalitzada, i anem per la següent.

3.2.2. Fase 2

El primer que calia era veure quines de les variables globals que necessitaven els *threads* podien provocar incoherències per tal de bloquejar-ne l'accés, després de donar-li moltes voltes vaig veure que de fet no tenia cap regió crítica a protegir, ja que tots els accessos a aquestes variables eren de lectura i prou. Les estructures de les connexions dels clients, es troben en una taula global, però de nou cada *thread* únicament accedeix a la posició que té assignada de manera que tampoc hauria d'haver-hi problemes, ja que si hi escriu algú altre, aquest és el pare just abans de crear (o assignar) el fil. Únicament en la manera que el pare obté els identificadors (posició lliure de la taula) pot donar algun problema, ja que es fa un recorregut per la taula de clients i retorna la primera posició en que el terme de l'estructura que fa referència al socket té el valor 0, però com a molt el que passarà és que en una situació de càrrega màxima, mentre es fa el recorregut algun dels *threads* anteriors acabi, amb el que el pare òbviament no ho sabrà i refusarà la connexió del client. Per tant, s'obta per no bloquejar res conscient de que si més endavant alguna cosa no funciona serà força difícil de trobar el problema, però com a bon Snowboarder que sóc sé que sense risc no hi ha diversió, així que m'hi llanço de cap (actitud que m'ha costat més d'una costella ho reconec...).

En un primer moment no tenia molt clar com controlar els *threads* estàtics, així que començo per lo "fàcil", i quan funcioni el servidor multi-fil únicament amb *threads* dinàmics, ja em dedicaré a triar una de les solucions possibles que hi ha.

3.2.2.1. Threads dinàmics

La idea és senzilla, quan s'accepta una nova connexió, obtenim un identificador que farà d'índex de la taula de clients, deixem el descriptor del socket en aquesta posició, i creem el nou *thread* per tal que executi la funció `http_control()` ja comentada, i li passem l'índex de la taula on hi ha l'estructura sobre la que treballarà amb el nou socket a punt. Quan s'acabi la sessió HTTP ja sigui per un tancament explícit, perquè salti el *timeout* o bé per algun error, caldrà que el propi *thread* deixi el terme *socket* amb el valor 0, per marcar la connexió com a lliure després de ressetejar-ne els altres valors, i finalment finalitzi la seva execució, sense esperar cap resposta per part del pare:

Exemple 3-2. Exemple de creació d'un *thread*

```
#include <pthread.h>
#define MAX_FILS 100

pthread_t fils[MAX_FILS];
...
/* Funció que executen els threads dinàmics */
void *fil (int *cid)
{
clients[*cid].fil = pthread_self (); // L'identificador de procés
http_control (*cid); // "Conversa" HTTP
allibera_connexio (*cid); // Resetegem tots els valors
pthread_exit (NULL); // Fi del thread
}
...
/* Funció que accepta connexions i crea els threads
```

```
 * només amb lo bàsic per facilitar la comprensió*/
void escolta (int socket)
{
int sc, id;
...
for (;;) {
sc = accept (socket, (struct sockaddr *) &adreca_client, &mida_client);
id = getId ();
clients[id].id = id;
clients[id].socket = sc;

/* Creem el thread */
pthread_create (&fils[id], NULL, &fil, &clients[id].id);
/* Informem que ja no en volem saber res d'ell */
pthread_detatch (fils[id]);
...
}
}
```

Ara la pregunta podria ser, perquè no li passem directament l'adreça de la variable `id` com a quart paràmetre de la crida `pthread_create`? Fixem-nos que fins i tot si la primera instrucció de `fil` fos `int id = *cid;` correm el risc que just quan el `thread` s'acaba de crear, i es disposa a executar la primera instrucció, es produeixi un canvi de context, el pare accepta una nova connexió i obté un nou identificador, amb el que quan el fil recupera el processador l'adreça on anava a llegir l'identificador contindrà un altre valor.

En aquest punt, tot i poder provar el *multithreading* amb la doble connexió del mozilla, es fa necessari l'ús d'alguna altra eina que em generi una càrrega prou alta per veure si el XicHttpd aguanta. Es poden veure les eines que s'utilitzaran per generar les càrregues d'aquí en endavant en el següent apartat referent als Jocs de Proves i anàlisi de rendiment.

En començar doncs a generar càrrega, tot i que la majoria de peticions es resolen bé, es comencen a rebre aleatòriament senyals de tipus SIGPIPE (Pipe trencada) i SIGSEGV (Violació de segment). Per tal de trobar-ne l'origen començo per utilitzar el gdb[2] tot i que sense èxit, ja que no sé per quin motiu, però tots els *threads* que finalitzaven la seva tasca es quedaven en estat *zombie*, de manera que tot i no ocupar espai de memòria si que ocupen un descriptor, i quan el sistema té massa processos envia un SIGTERM a aquells que, per dir-ho així, li molesten més, i naturalment amb més de 300 instàncies del xic en estat *defunct* em tocava el rebre a mi. Així doncs que amb el gdb no vaig poder treure cap conclusió. Tot i això en una de les proves si que vaig rebre els SIGPIPE abans del SIGTERM comentat, així que vaig intentar treure'n informació:

Exemple 3-3. Exemple d'algunes comandes bàsiques del gdb

```
$ gdb xic
gdb> run
```

```
-- I un cop rebia el senyal, el gdb n'informa correctament i s'atura
-- però ara cal cercar informació, començo doncs per fer un backtrace
-- i veure les darreres funcions que s'han executat
gdb> bt

-- Però clar, cal veure quin thread l'ha provocat, així que em dedico
-- a anar canviant de thread i anar fent bt.
gdb> thread xxx (on x és el número de thread que informa el gdb)
gdb> bt

-- Finalment no ens deixa accedir als threads que ja han acabat
-- és a dir, els que estan zombie, i probablement en siguin
-- l'origen.
```

Així doncs cal alguna altra eina per intentar veure el que passa, i buscant una mica vaig descobrir l'`strace`

Exemple 3-4. Exemple d'ús simple de l'strace

```
$ strace -ff -o debug ./xic

-- L'obció -ff indica que cal tracejar tots els subprocesos (en aquest cas threads)
-- i escriure un fitxer de traça per cada un amb nom debug.PID
```

Cal notar que l'strace ressegueix les crides al sistema, però si el SIGSEGV pot aparèixer en qualsevol lloc, un SIGPIPE quasi segur que passa en fer alguna crida sobre un descriptor, així que l'strace ens serveix. Un cop rebut doncs, executo la comanda `$ rgrep "SIGPIPE" *` dins el directori on hi ha les traces i així sé quines traces han rebut aquest senyal i n'obro els fitxers, després d'una primera observació, repeteixo la prova varies vegades. Curiosament totes les vegades, tots els *threads* el rebien mentre estaven en la crida `select` excepte un que estava fent una serie de `recv`

Exemple 3-5. Exemple de sortida de l'strace

```
...
sigreturn()                             = ? (mask now [RTMIN])
select(9, [8], NULL, NULL, {15, 0})     = 1 (in [8], left {15, 0})
recv(8, "G", 1, MSG_PEEK)               = 1
recv(8, "G", 1, 0x4000)                 = 1
recv(8, "E", 1, 0x4000)                 = 1
recv(8, "T", 1, 0x4000)                 = 1
recv(8, " ", 1, 0x4000)                 = 1
recv(8, "/", 1, 0x4000)                 = 1
```

A *Google linux* (http://www.google.com/linux) hi vaig trobar un post[3] que en parlava, així que **man recv**, i per fi descobreixo com detectar que l'altre extrem tanca la connexió (la falta d'experiència ja les té aquestes coses, però bé, uns quants cops de cap contra la paret i seguim). Tot i tenir varies opcions per tractar-ho, em va semblar que el més adient és ignorar aquest senyal de manera que quan els `recv` (i els `send` també a partir d'ara) fallin sigui per un SIGPIPE o per qualsevol altre motiu sempre retornen `-1`, amb el que XicHttpd tanca la seva banda i allibera el *thread*.

Exemple 3-6. Ignorar el signal SIGPIPE en les crides `recv` i `send`

```
recv (sock, &c, 1, MSG_NOSIGNAL);
send (sock, buf, mida_buf, MSG_NOSIGNAL);
```

3.2.2.2. Threads estàtics

Per als *threads* estàtics el primer que ens cal és trobar un mecanisme per tal que estiguin pausats, esperant a que el pare els hi assigni una connexió. Tenia varies opcions, com ara el *polling* (Consultar una variable fins que pren cert valor), que els fils capturin els SIGUSR1 i facin un `pause()` i el pare els hi envii en rebre una connexio, utilitzar semàfors, però la pròpia llibreria **pthread** ja té les crides per aquestes situacions, com són l'ús de variables *mutex* i les variables condicionals, el que passava és que amb tota la documentació que havia mirat (veure Bibliografia) no m'acabava de quedar clar com adaptar-ho al meu cas, ja que tots parlen de l'accés a regions crítiques, però en el meu cas només em calia esperar i prou. Aquí agraeixo l'ajuda d'en David Carrera (el meu Consultor) que em va mostrar un exemple de l'ús de les variables condicionals, a més de remetrem al que ha sigut potser el *Tutorial de Pthreads* (http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html) més aclaridor de tots els que he vist.

Veiem doncs com es pot fer per tenir una serie de *threads* en espera, i com el pare els pot despertar, sense disposar realment de cap regió crítica ni variable compartida a consultar.

Exemple 3-7. Exemple de creació, bloqueig i activació de *threads*

```
pthread_t fils[MAX_THREADS];
pthread_cond_ti cond[MAX_THREADS];
pthread_mutex_t mutex[MAX_THREADS];
...
/* Funció que executen els threads estàtics */
void *fil_estatic (int *cid)
{
clients[*cid].fil = pthread_self();
for (;;) {
pthread_mutex_lock (&mutex[*cid]);
pthread_cond_wait (&cond[*cid], &mutex[*cid]); // Aquí s'atura fins rebre el senyal
pthread_mutex_unlock (&mutex[*cid]);
http_control (*cid);
allibera_connexio (*cid);
}
}
```

```
...
/* Creació de threads */
void crea_fils_estatics ()
{
int i;
for (i = 0; i < MAX_THREADS; i++) {
clients[i].id = i;
pthread_create (&fils[i], NULL, (void *) &fil_estatic, (void *) &clients[i].id);
pthread_detach (fils[i]);
pthread_mutex_init (&mutex[i], NULL);
pthread_cond_init (&cond[i], NULL);
}
}
...
/* Acceptem la connexió i despertem al thread */
void escolta (int socket)
{
int sc, id;
...
crea_fils_estatics ();
for (;;) {
sc = accept (socket, (struct sockaddr *) &adreca_client, &mida_client);
id = getId ();
clients[id].id = id;
clients[id].socket = sc;
/* Despertem al thread corresponent */
pthread_mutex_lock (&mutex[id]);
pthread_cond_signal (&cond[id]); // Enviem el senyal
pthread_mutex_unlock (&mutex[id]);
...
}
}
```

Les crides `pthread_cond_wait` i `pthread_cond_signal` internament el primer que fan és alliberar el bloqueig de la regió crítica en la que estan, i lo últim és tornar-lo a posar, de manera que ens cal l'ús de les funcions `pthread_mutex_lock` i `pthread_mutex_unlock` ja que d'altre manera sempre ens trobaríem l'accés bloquejat.

3.2.2.3. Tractament dels signals

La captura dels *signals* en aquesta versió del XicHttpd es fa únicament amb finalitats de *loggin* i donat que s'executa com un *daemon* també hem de poder parar el servidor de forma segura. Així el tractament per tots els signals que es capturen és el mateix, es loga i es para el servidor. Els signals capturats són:

- **SIGHUP**: Desconnexió.
- **SIGINT**: Interrupció (Ctrl+C).
- **SIGSEGV**: Violació de segment.

- **SIGPIPE**: Intent d'escriptura en una canyeria on no hi ha ningú llegint.
- **SIGTERM**: Finalització controlada.

3.2.3. Fase 3

Anem a veure doncs com s'han implementat les diferents funcionalitats addicionals:

3.2.3.1. Pipelining

En primer lloc es tracta el *request* com si fos una petició normal, llavors just abans d'enviar mirem si ens queden dades pendents de llegir del *socket* del client, si és així, entenem que el client utilitza pipelining, ja que encara no hem servit cap resposta. Aleshores si la resposta ens hi cap al buffer de sortida ho encuem i tractem el següent *request* fins que el buffer de sortida és ple, o ja no queden més peticions i llavors procedim a l'enviament de les respostes.

3.2.3.2. Enviament de dades comprimides amb *gzip*

La llibreria *zlib* redefineix els mètodes d'entrada/sortida sobre fitxers per als fitxers *gzip* amb el prefix *gz* així per exemple trobem `gzopen, gzwrite, gzread`, etc. De fet les operacions són molt senzilles, només hi ha un problema, i és que treballa sobre fitxers, quan lo ideal per al servidor Web fora que realitzés la compressió en memòria, nosaltres en llegim la mida, omplim la capçalera adient, i n'enviem el resultat. La única manera de fer això amb aquestes llibreries seria fer una implementació pròpia de les rutines de compressió utilitzant com a model les fonts de les *zlib* (Concretament les del fitxer `gzio.c`), però això s'allunya molt dels objectius marcats, per tant utilitzarem les funcions `gz*`.

Així doncs tenim dues necessitats, la primera és la creació d'un fitxer temporal sobre el qual poder fer la compressió, i la segona és obtenir la mida del fitxer comprimit abans d'enviar-lo per tal d'omplir la capçalera `Content-Length`.

Fitxer temporal. La tocada de pera és que fins que no es crida la funció `gzclose` no s'escriu el CRC al fitxer, però com que el fitxer queda tancat caldria tornar-lo a obrir i llegir per poder-lo enviar, i això no em feia gaire gràcia. Així que en un primer moment decideixo mapejar el fitxer a memòria, però com que encara no sabia com saber-ne la mida utilitzo la mida del fitxer original conscient del malbaratament de memòria que estava fent, però d'aquesta manera un cop tancat el fitxer la zona mapejada contindrà el fitxer comprimit complert amb CRC i tot, de manera que només em caldria enviar-lo i desmapejar-lo posteriorment, de moment deixem-ho aquí.

Mida del fitxer comprimit. No, un `strlen` sobre la zona mapejada no funciona ja que normalment a partir del quart byte de la capçalera del fitxer *gz* i ha alguns zeros, de manera que sempre retorna 4. Si, podem fer un `lseek` sobre la zona mapejada i restar la posició final de la inicial, però implica un parell de crides, o podem llegir fins a trobar l'*EOF* i anar comptant, però per enviar-ho repetim la lectura i tampoc no m'agrada. No desesperem, l'avantatge d'utilitzar productes de codi obert és que sempre ens

podem baixar les fonts i veure com treballen, ja que internament a un lloc o altre es deuen controlar el nombre de bytes de sortida per poder escriure'ls al fitxer. Certament repassant el codi del fitxer `gzio.c` em trobo que el **gzFile** (El nom ho diu tot) es pot "castejar" com un punter a una tal estructura **gz_stream** la qual té un membre que és una estructura **z_stream**, que ja podem trobar al fitxer de capçaleres `zlib.h`, i la qual té un membre sospitós anomenat **total_out**. Bé, anem a petar una mica lluny però si és el que buscàvem haurà valgut la pena, però en fer algunes proves i comprovacions sembla que sempre hi faltin 20 bytes, i aquí és on fem un cop d'ull a l'[RFC 1952] on es descobreix que la capçalera del *gzip* ocupa 10 bytes, el CRC n'ocupa 8 i els 2 que resten són una altra capçalera que ens indica el tipus de CRC utilitzat. En definitiva, obtenim el valor de *total_out* i li sumem 20 per obtenir la mida total del fitxer.

Resultat final. En el següent exemple es mostra el resultat prescindint del control d'errors i altre codi que pugui despistar.

Exemple 3-8. Exemple de compressió i enviament en *gzip* utilitzant les llibreries *zlib*

```
#include <zlib.h>
...
int xic_gzip (int cid, char *file_in, int mida_in, c_response *resposta, buf_out *sbuf)
{
/* Tros d'estructura extreta de la llibreria zlib
 * però està declarat com a "local" a gzio.c, i
 * nosaltres només necessitem accedir al primer membre
 */
 typedef struct xic_gz_stream {
  z_stream stream;
} xic_gz_stream;
int fdout, escrits, ret = 0, mida_ant;          // fd desti, ...
FILE *fdin, *tmp;                               // fitxer origen, temporal
char buf_in[mida_in];                           // buffer de lectura
gzFile gzfile;                                  // fitxer gz
caddr_t buf_mem;                                // per mapejar a memòria

/* Creem el fitxer temporal */
tmp = tmpfile ();
fdout = fileno (tmp);

/* Obrim l'original */
fdin = fopen (file_in, "r");

/* Preparem el buffer per fer la compressió d'un sol cop */
fread (buf_in, 1, mida_in, fdin);
fclose (fdin);

/* Comencem la compressió */
gzfile = gzdopen (fdout, "wb9");
gzwrite (gzfile, buf_in, mida_in);
gzflush (gzfile, Z_SYNC_FLUSH);

/* Escrivim la mida */
escrits = ((xic_gz_stream *) gzfile)->stream.total_out + 20; // 20 = Capçaleres + CRC
```

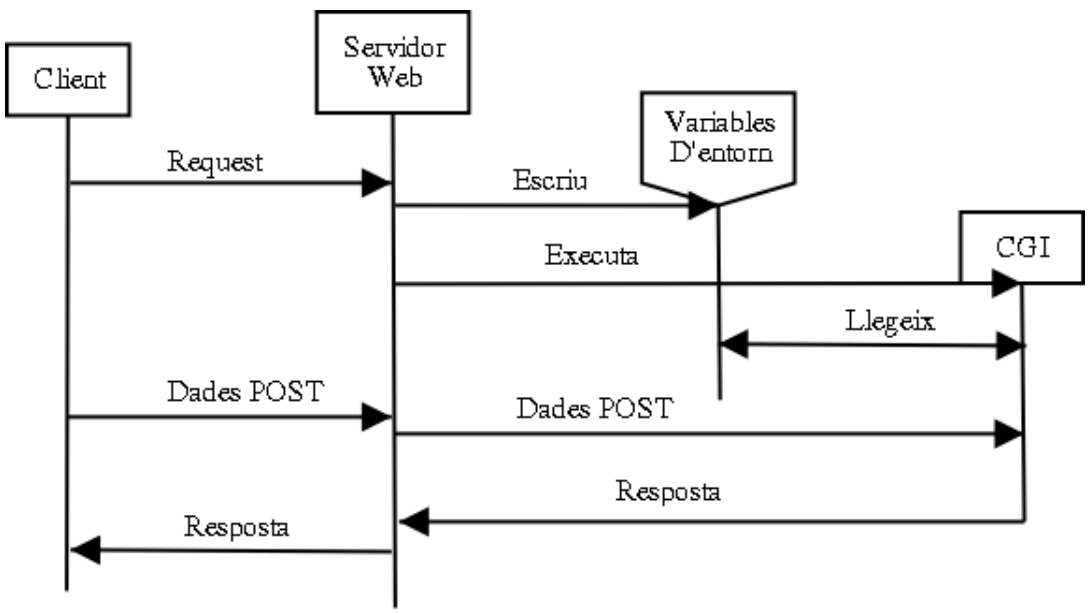
```
/* Projectem el fitxer comprimit en memòria ara que sabem quan ocuparà */
buf_mem =  mmap ((void *) 0, escrits, PROT_WRITE | PROT_READ, MAP_SHARED, fdout, 0);

/* Ja podem tancar el gzip */
gzclose (gzfile);

/* Omplim les capçaleres, etc. */
...
/* Enviem */
envia (clients[cid].socket, buf_mem, escrits, sbuf, MAX_SBUF);
...
}
```

3.2.3.3. CGI

El XicHttpd suporta el processament de dades per mitjà d'scripts o programes CGI, però de moment no es permet l'accés directament a aquests scripts per generar planes dinàmicament en el servidor. Ho he decidit així perquè penso que hi ha maneres molt més eficients de generar contingut dinàmic, aquell qui disposi de l'Apache, PHP i el mòdul PHP per l'Apache, pot provar a configurar l'execució dels PHP com a mòdul o com a CGI i comparar per ell mateix.

Per l'execució dels CGI cal que el servidor Web faci com de passarel·la entre el client i el CGI, per la seva banda l'script en qüestió espera trobar certs valors en certes variables d'entorn, un cop ha fet la feina retorna una sortida que el servidor envia directament al client:

Figura 3-1. Procés CGI



Per al pas de paràmetres per mitjà del mètode **GET** el procés és el mateix però sense la línia "*Dades POST*". Anem a veure doncs quines són aquestes variables d'entorn mínimes que li calen als CGI, i que de moment són les úniques que utilitza el XicHttpd:

- **REQUEST_METHOD**: Conté el mètode utilitzat pel client per enviar les dades i pot contenir els valors **GET** o **POST**.
- **QUERY_STRING**: Només s'utilitza en el mètode **GET** i contindrà tot el text posterior al caràcter '?' codificat en la URL de la petició.
- **CONTENT_LENGTH**: Només s'utilitza en el mètode **POST** i contindrà la mida de l'`entity-body` del *request*, és a dir, el valor de la capçalera `Content-Length` de la petició.

Per a l'execució del CGI s'utilitza l'omnipotent crida `fork()` de Unix la qual després de reassignar els canals d'entrada/sortida cap a dues *pipes* prèviament creades, omple les variables d'entorn i fa un canvi d'imatge amb la crida `execl` per executar el CGI. Vejem-ne el funcionament en el següent exemple en pseudo-codi:

Exemple 3-9. Pseudo-codi de l'execució de CGI

```
int xic_cgi (char *cgi_file)
{
int cgi_in[2], cgi_out[2];
pid_t pid;
...
pipe (cgi_in);
pipe (cgi_out);

if ((pid = fork()) == -1) error;
if (pid == 0) { // Codi del fill
reassigna_entrada_sortida_pipe;
escriu_variables_entorn;
execl (cgi_file, cgi_file, NULL);
exit(0); // si falla execl

} else { // Codi del pare
tanca_canals_pipe_sobrants;
if (POST) {
mentre (rebre_dades)
escriu_dades(cgi_in[1]);
}
prepara_capçaleres;
mentre (llegir (cgi_out[0]))
envia_al_client;
tanca_pipes;
espera_fill (pid);
```

```
}
}
```

3.2.3.4. Sistema de *log*

En aquesta part apareix per primer cop l'accés concurrent d'escriptura per part dels *threads* sobre un mateix recurs, en aquest cas els fitxers de log, però de fet tractant-se de fitxers, el Sistema Operatiu s'encarrega dels bloquejos, i per tant no cal preocupar-se'n. Ara bé, el que si és imprescindible és fer una sola crida a la funció d'escriptura per cada entrada al log, ja que d'altra manera el resultat és impredictible, per tant cal que cada *thread* disposi d'un buffer intern on preparar el missatge per posteriorment fer la crida a la funció d'escriptura.

Log d'accessos. En arrencar el servidor s'obre el fitxer especificat per l'entrada `FITXER_LOG` del fitxer de configuració, en mode *append*, és a dir, afegint pel final, queda clar doncs que aquest fitxer creixerà desmesuradament i per això seria aconsellable configurà el sistema de rotació de logs del sistema per tal que també accepti aquest fitxer. Recordem a continuació el format de les entrades del fitxer, i vegem-ne un exemple del seu funcionament.

Exemple 3-10. Exemple de log d'accessos

```
/* Format del log
 *  Adreça-IP - [Hora-Local] "Mètode Recurs Versió" "Codi-Resposta Descripció"
 */
(fitxer server.h)
...
#define LOG_FORMAT "%s - [%s] \"%s\" \"%s\"\n"
FILE *f_log;
...
(fitxer server.c#init_log())
...
f_log = fopen (config.f_log, "a");
....
(fitxer http.c)
...
char log_temp[2048];
...
strncpy (log_temp, primera_linia_request, mida_linia);
...
/* Un cop sabem la resposta */
fprintf (f_log, LOG_FORMAT, inet_ntoa (clients[cid].adreca_client.sin_addr),
getData (), log_temp, resposta.estat);
```

Log d'errors. En aquest cas el fitxer d'errors especificat per l'entrada `FITXER_ERR` del fitxer de configuració s'obre en mode escriptura, de manera que a cada execució se'n buida el contingut, per tant és responsabilitat de l'administrador de conservar aquest fitxer si ho creu convenient. En la rutina d'inicialització del log d'errors, també es redireccióna la sortida estàndard d'errors **stderr** cap al fitxer, de manera que aquest tindrà dos tipus d'entrades diferents, per una banda les escriptures explicites fetes pel XicHttpd i per l'altre els errors que puguin produir certes crides. En el primer dels següents exemples

es mostra com redireccionar la sortida d'errors cap a un fitxer, i en el següent es pot veure el format dels errors i un exemple del seu funcionament:

Exemple 3-11. Exemple de redirecció de l'*stderr* a un fitxer

```
...
FILE *f_err;
...
*f_err = freopen (config.f_err, "w", stderr);
...
```

Exemple 3-12. Exemple de log d'errors

```
/* Format dels errors
 *
 * [Hora] Codi-Descripcio (FITXER->FUNCIO->CRIDA): Detalls
 *
 * On tenim que CRIDA i Detalls són opcionals */
(fitxer errors.h)
#define FORMAT_ERROR "[%s] %s (%s): %s\n"
...
(fitxers *.c)
...
char msg[UNA_MIDA]; // opcional
...
sprintf (msg, "%s", "Detalls Opcionals"); // opcional
...
fprintf (f_err, FORMAT_ERROR, getData (), XMISSATGE,
"FITXER->FUNCIO->CRIDA", msg);
...
```

Mostrem a continuació les errors que pot prendre la constant *XMISSATGE* de l'exemple anterior:

Taula 3-1. Errors possibles del XicHttpd (`errors.h`)

Constant	Codi Descripció	
XNOMEM	1 Falta Memòria!	
XLISTEN	2 Error en la crida listen	
XACCEPT	3 Error en la crida accept"	
XMASSACON	4 Massa connexions	
XSOCKET	5 Error creant socket servidor	
XDNS	6 No es pot obtenir l'adreça del host	
XBIND	7 No es pot publicar la direcció (port ocupat)	
XCONFIG	8 No es pot obrir el fitxer de configuració	
XCONFIG2	9 No es pot obrir CAP fitxer de configuració	
XCONFIG3	10 Fitxer de configuració erroni	

Constant	Codi Descripció	
XNODEFINIT	11 Error inesperat ! ??	
XSIGHUP	12 SIGHUP	
XSIGINT	13 SIGINT	
XSIGSEGV	14 SIGSEGV	
XSIGPIPE	15 SIGPIPE	
XSIGTERM	16 SIGTERM	
XSIG	17 Signal indefinit	
XREQUEST	18 No hi ha request per llegir	
XRECURS	19 No s'ha pogut obrir el fitxer	
XNOMEM2	20 Falta memòria per algun thread	
XWLOG	21 Error escrivint al log	
XOLOG	22 Error obrint log	
XMIME	23 Error carregant els tipus MIME	

3.2.3.5. Últims detalls importants

La compilació de programes que utilitzin la llibreria *pthread* és tan senzill com ficar l'include adient (`pthread.h`) i dir-li al compilador que l'enllaci (opció `-lpthread`), però hi ha algunes coses més a fer si utilitzem les llibreries glibc (...). Pensem per exemple amb la variable **errno** la qual és global a tot el procés i s'actualitza quan alguna crida falla, aleshores, si els diferents *threads* que hem creat comparteixen l'espai de memòria i dos d'ells fallen, és molt possible que quan en recollim el valor per fer alguna comprovació (com ara ENOMEM o EACCES) llegim el valor que no toqui. A més hi poden haver certes funcions que també tinguin un comportament indefinit quan treballem amb *threads* donat que internament defineixen variables estàtiques. Per evitar aquestes situacions cal que sempre que compilem programes multi-fil utilitzem:

```
#define _REENTRANT
(o bé gcc ... -D_REENTRANT)
```

La creuada no s'acaba aquí, donat que algunes funcions a les que podríem estar acostumats disposen de les equivalents però segures per als *threads* (*Thread-Safe*), com és el cas de la funció `strtok`, l'equivalent de la qual s'anomena `strtok_r` a la qual li hem de passar un nou paràmetre corresponent a l'adreça d'una variable local per tal que emmagatzemi els resultats intermitjos[4]

Una altra qüestió important són els permisos d'execució del servidor Web ja que els ports per sota del 1024 estan reservats, de manera que si els volem utilitzar calen permisos especials. Sempre el podem executar com a usuari root però mai és una bona idea fer això, l'altre opció que tenim és fer l'executable SUID root, amb el que el podrem executar com a un usuari normal però si ens hi fixem el procés tindrà

privilegis del superusuari per tant si es produeix algun desbordament no desitjat pot comprometre la seguretat de tot el sistema.

Per evitar aquesta situació de perill el XicHttpd fa el canvi d'usuari internament: Entre tots els identificadors coneguts pel procés, els que ens interessen són l'EUID (usuari efectiu) que correspon al propietari, i l'UID que és l'usuari que l'executa, així en arrencar el servidor amb SUID root des d'un usuari normal es canvia l'EUID per l'UID i només es restaura l'EUID en aquelles funcions en que els privilegis són estrictament necessaris. Anem a veure el següent exemple on es veurà més clar el procediment.

Exemple 3-13. Exemple de canvi del nivell de privilegis[5]

```
uid_t e_uid_inicial;
uid_t r_uid;
...
void init ()
{
e_uid_inicial = geteuid ();
r_uid = getuid ();

/* Canviem els privilegis d'execució */
seteuid (r_uid);
...
crea_socket ();
}
int crea_socket ()
{
...
/* Restaurem els privilegis originals */
seteuid (e_uid_inicial);

/* Crida(es) que necessita de privilegis */
bind (server_sock,  (struct sockaddr *) &config.addr, sizeof (struct sockaddr));

/* Tornem als privilegis de l'usuari */
seteuid (r_uid);
...
}
```

Per acabar es mostra com fer que un procés es converteixi en *daemon* per tal que s'executi com un servei més del sistema. Fixem-nos que ens calen dues coses, en primer lloc cal executar el procés en *background* i per fer-ho només tenim que "clonar" el procés amb la crida `fork()` i fer que el pare acabi la seva execució, ara bé el fill, igual que el pare quan s'ha creat, dependrà del terminal on s'ha cridat a la seva execució, de manera que si tanquéssim el terminal el procés acabaria. Per tant la segona cosa que cal fer és crear una nova sessió i així deslligar-nos de qualsevol terminal convertint-nos en el nou cap del grup de processos.

Exemple 3-14. Exemple de com convertir un procés en *daemon*

```
...
switch (fork ()) {
case -1: exit (-1); // Error
case 0: break; // El fill
default: exit (0); // El pare, sortim
}

setsid (); // Nova sessió
...
```

3.3. Jocs de Proves i anàlisi de rendiment

3.3.1. Entorn de treball

El XicHttpd s'ha provat en dues màquines diferents, i s'ha generat la càrrega des d'una tercera, a continuació se'n mostren les característiques:

1. **Desenvolupament i proves:**
   - Portàtil Pentium IV 2,4 GHz 512 KB cau L2 (clònic)
   - 512 MB de memòria RAM DDR
   - 1024 MB de memòria d'intercanvi (SWAP)
   - SO: Debian woody (3.0) 2.4.18-bf2.4
2. **Generador de càrrega:**
   - AMD-K6 3D 400 MHz (clònic)
   - 160 MB de memòria RAM (DIMM)
   - Memòria d'intercanvi per fitxer (uns 768 MB)
   - SO: Windows XP Professional V.2002
3. **Característiques de la xarxa:**
   - Ehternet amb parell trenat 10/100 UTP CAT 5
   - Entorn completament commutat
   - Connexions a 100 Mbps *Full-Duplex* PC-*Switch*

3.3.2. Proves bàsiques

Les primeres proves que es fan són per comprovar que realment el XicHttpd respongui el que toca, sobretot pel que fa als codis d'error de les respostes. Per aquestes comprovacions s'utilitza una connexió **telnet** des d'un altre terminal. Mostrem a continuació algunes d'aquestes sortides:

- Petició correcte

```
aposai@Natasza:~/tfc/memo$ telnet Natasza 80
Trying 192.168.1.13...
Connected to Natasza.
Escape character is '^]'.
GET /index_cgi.html HTTP/1.1
Host: Natasza

HTTP/1.1 200 OK
Date: Thu, 08 Jan 2004 09:30:54 GMT
Server: XicHttpd v0.1
Last-Modified: Tue, 30 Dec 2003 20:43:52 GMT
Content-Type: text/html
Content-Length: 2298
Content-Encoding: identity

<html>
<head>
<title>Exemple de CGI</title>
...
```

- Petició correcte HTTP/1.0 (XicHttpd no suporta persistència en aquests tipus de connexions)

```
aposai@Natasza:~/tfc/memo$ telnet Natasza 80
Trying 192.168.1.13...
Connected to Natasza.
Escape character is '^]'.
GET /index_cgi.html HTTP/1.0

HTTP/1.0 200 OK
Date: Thu, 08 Jan 2004 09:33:46 GMT
Server: XicHttpd v0.1
Last-Modified: Tue, 30 Dec 2003 20:43:52 GMT
Content-Type: text/html
Content-Length: 2298
Content-Encoding: identity
Connection: close <-- Força el tancament

<html>
<head>
...
Connection closed by foreign host.
aposai@Natasza:~/tfc/memo$
```

- Un **HEAD**

```
HEAD /index_cgi.html HTTP/1.1
```

```
Host: Natasza

HTTP/1.1 200 OK
Date: Thu, 08 Jan 2004 09:40:12 GMT
Server: XicHttpd v0.1
Last-Modified: Tue, 30 Dec 2003 20:43:52 GMT
Content-Type: text/html
Content-Length: 2298
Content-Encoding: identity

(Resposta idèntica al GET sense l'entity-body)

<-> HEAD acceptant gzip sobre el mateix recurs
HEAD /index_cgi.html HTTP/1.1
Host: Natasza
Accept-Encoding: gzip

HTTP/1.1 200 OK
Date: Thu, 08 Jan 2004 10:00:44 GMT
Server: XicHttpd v0.1
Last-Modified: Tue, 30 Dec 2003 20:43:52 GMT
Content-Type: text/html
Content-Length: 678 <-- La mida es redueix dràsticament
Content-Encoding: gzip
```

- A continuació alguns errors

```
<-> Plana inexistent
GET /foo.html HTTP/1.1
Host: Natasza

HTTP/1.1 404 Not Found
Date: Thu, 08 Jan 2004 09:42:47 GMT
Server: XicHttpd v0.1
Content-Length: 65
Content-Type: text/html

<html><body><center><h1>404 Not Found</h1></center></body></html>

<-> Versió estranya, en llegir la primera línia respon descartant tot lo demés
GET / HTTP/2.0
HTTP/1.1 505 HTTP Version Not Suported
Date: Thu, 08 Jan 2004 09:44:16 GMT
Server: XicHttpd v0.1
Content-Length: 81
Content-Type: text/html

<html><body><center><h1>505 HTTP Version Not Suported</h1></center></body></html>

<-> Mètode no implementat
TRACE / HTTP/1.1
HTTP/1.1 501 Not Implemented
Date: Thu, 08 Jan 2004 09:46:58 GMT
Server: XicHttpd v0.1
```

```
Content-Length: 71
Content-Type: text/html

<html><body><center><h1>501 Not Implemented</h1></center></body></html>

<-> HTTP/1.1 Sense la capçalera HOST
GET / HTTP/1.1

HTTP/1.1 400 Bad Request
Date: Thu, 08 Jan 2004 09:48:41 GMT
Server: XicHttpd v0.1
Content-Length: 67
Content-Type: text/html

<html><body><center><h1>400 Bad Request</h1></center></body></html>

<-> Intent d'accçes a un fitxer on no s'hi té permís
GET /cgi-bin/exemple.cgi HTTP/1.1
Host: Natasza

HTTP/1.1 403 Forbidden
Date: Thu, 08 Jan 2004 09:49:54 GMT
Server: XicHttpd v0.1
Content-Length: 65
Content-Type: text/html

<html><body><center><h1>403 Forbidden</h1></center></body></html>

<-> Igual que l'anterior però ara per processar dades
GET /cgi-bin/exemple.cgi?var1=1 HTTP/1.1
Host: Natasza

HTTP/1.1 200 OK
Connection: close <-- El CGI força el tancament
Content-Type: text/html; charset=ISO-8859-1

<?xml version="1.0" encoding="utf-8"?>
...

<-> POST sense Content-Length
POST / HTTP/1.1
Host: Natasza

HTTP/1.1 411 Length Required
Date: Thu, 08 Jan 2004 09:55:10 GMT
Server: XicHttpd v0.1
Content-Length: 71
Content-Type: text/html

<html><body><center><h1>411 Length Required</h1></center></body></html>

<-> Petició amb males intencions, no ens molestem ni a contestar
GET /../../../etc/passwd HTTP/1.1
```

```
Host: lala

Connection closed by foreign host
```

3.3.3. Proves de rendiment

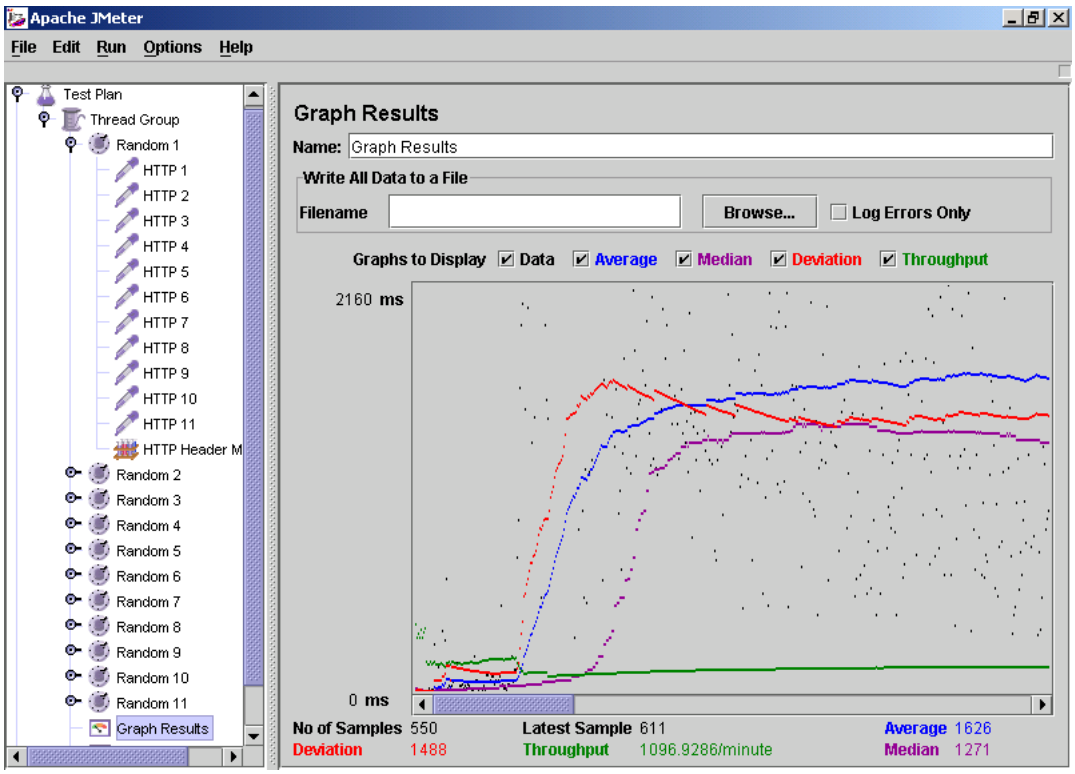
Les tres aplicacions de test que s'han aplicat són el Surge[6], el JMeter[7] d'Apache i el Web Stress[8] de Microsoft. A continuació es mostren algunes de les característiques de cada un i l'ús que n'he fet:

- **Surge:** Genera una serie de fitxers `*.txt` de diferents mides que cal copiar a un directori accessible pel servidor, i després permet definir la quantitat de *threads*, connexions per fil i temps de duració de la prova, un petit defecte és que el tipus de client s'ha de definir en el moment de la compilació, personalment el vaig compilar per a connexions HTTP/1.1 amb *pipelining*. Per al meu cas s'utilitzen els valors de configuració per defecte (2000 fitxers), tot i així la interpretació dels resultats no és massa senzilla i com que el temps apreta s'utilitza únicament per fer les primeres proves sobre *threads* i prou. D'altra banda cal notar que es resultats tampoc eren massa fiables donat que s'executa sobre la mateixa màquina que el servidor. Es mostra a continuació el Surge treballant.

  ```
  Natasza:/home/aposai/tfc/XicHttpd/benkmark# ./Surge 10 10 60
  SURGE: Scalable URL Reference Generator
  Running 10 clients with 10 threads/client for 60 seconds
  SURGEmaster:  90054 objects in name sequence
  SURGEclient 0: running 10 threads
  SURGEclient 0:  Object Count 0
  SURGEclient 1: running 10 threads
  SURGEclient 2: running 10 threads
  SURGEclient 3: running 10 threads
  SURGEclient 4: running 10 threads
  SURGEclient 5: running 10 threads
  SURGEclient 5:  Object Count 100
  SURGEclient 6: running 10 threads
  SURGEclient 7: running 10 threads
  SURGEclient 6:  Object Count 200
  SURGEclient 8: running 10 threads
  SURGEclient 9: running 10 threads
  SURGEclient 8:  Object Count 300
  SURGEclient 0:  Object Count 400
  SURGEclient 3:  Object Count 500
  SURGEclient 4:  Object Count 600
  SURGEclient 4:  Object Count 700
  SURGEclient 0:  Object Count 800
  SURGEclient 2:  Object Count 900
  SURGEclient 5:  Object Count 1000
  ....
  Natasza:/home/aposai/tfc/XicHttpd/benkmark# ./pbvalclnt Surge.log
  Mean transfer delay = 0.004702 seconds
  Varience of transfer delay = 0.000146 seconds
  ```

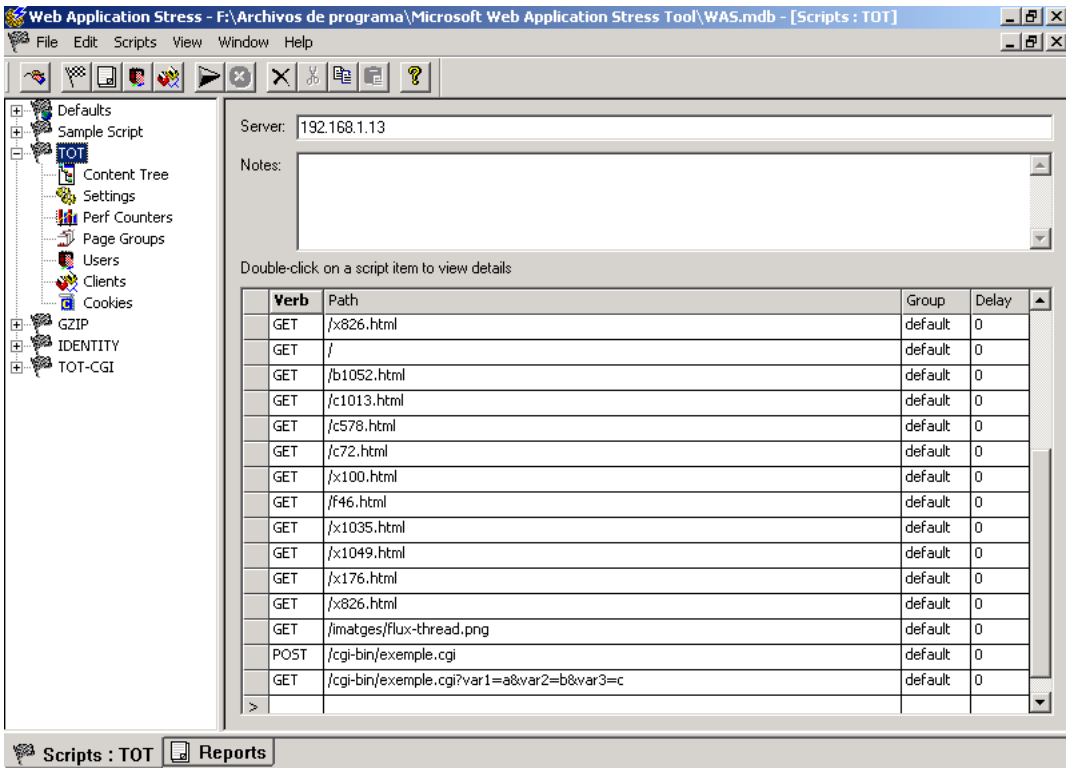
- **JMeter:** Està fet en Java i presenta un entorn gràfic prou entenedor. Per aquest cas cal definir completament l'entorn de proves, des de les URLs a les que es vol accedir, fins als mètodes, les capçaleres, etc. A l'hora de presentar els resultats tenim diferents opcions (*Listeners*) que ens permeten entre d'altres de veure gràfics, estadístiques d'accés per pàgina, la capçalera i el cos de les respostes... El problema és que consumeix molts recursos i la màquina de proves es quedava saturada molt abans que el servidor, de manera que tampoc ens podem refiar dels resultats. Tot i així em va permetre provar l'enviament de contingut comprimit amb *gzip*, a més de fer unes comparacions amb l'Apache (digueu-me agosarat), on la diferència més significativa era la dispersió en els temps de resposta del XicHttpd contra uns resultats més lineals de l'Apache. A continuació es pot veure una captura del JMeter en acabar un test fet amb 50 clients simultanis, compost d'11 grups d'accés aleatori a 11 planes cada un, en la meitat dels quals es demana compressió, l'objectiu és simplement comprovar el funcionament del XicHttpd en rebre accessos el màxim d'aleatoris possibles per simular un entorn real:

Figura 3-2. Captura del JMeter



- **Web Stress:** Tot i no mostrar uns resultats tan vistosos com el JMeter, tenim que també mostra una gran versatilitat a l'hora de preparar les proves, junt a que els requisits mínims són d'un P133 es converteix en un bon entorn per generar les proves de càrrega.

Figura 3-3. Captura del Web Stress



A continuació es defineixen les proves que s'han dut a terme amb aquest entorn:

- Totes les proves accediran a 11 pàgines html més una imatge contingudes en el servidor, de mides entre 2 i 100 KB.
- Test IDENTITY: Accés als arxius anteriors sense compressió.
- Test GZIP: Accés als arxius anteriors amb compressió (excepte la imatge que el servidor l'enviarà tal qual).
- Test TOT-CGI (tot menys CGI): Accés als arxius anteriors sense compressió, i se li afegeixen les 11 pàgines html demanant compressió.
- Test TOT: Igual que l'anterior, però substituïm dues pàgines per dues peticions CGI, una amb un **GET** i l'altre amb un **POST**.
- Cada test tindrà un *delay* aleatori entre cada *request* de 0 a 100 milisegons.
- Cada prova té una durada de 2 minuts.
- Les proves es comencen amb 100 clients i es repetiran amb increments de 100 per cada configuració del XicHttpd.

- La configuració del XicHttpd permetrà un nombre de sobres de clients dinàmics, i repetirem les proves variant el nombre de threads estàtics i el valor del timeout de les connexions.

Tot seguit es mostren els resultats que es creu que són més característics.

Resultats:

- En les següents taules **Hits** són la quantitat de peticions, **R/S** són els *request* per segon, i **Err** indica si es produeixen errors en les connexions dels *socket* (no confondre-ho amb respostes errònies que no s'han produït).
-

Taula 3-2. XicHttpd: 5 *threads* estàtics, 30 segons de *timeout*

Clients	TOT			TOT-CGI			IDENTITY			GZIP		
	Hits	R/S	Err	Hits	R/S	Err	Hits	R/S	Err	Hits	R/S	Err
100	18750	156.02	NO	19397	161.41	NO	19193	161.03	NO	13387	111.40	NO
200	15315	127.44	SI	12655	105.31	NO	26600	223.20	NO	8265	68.76	NO
500	18540	154.28	SI	6012	50.03	NO	27115	225.02	NO	6530	54.71	NO

-

Taula 3-3. XicHttpd: 50 *threads* estàtics, 15 segons de *timeout*

Clients	TOT			TOT-CGI			IDENTITY			GZIP		
	Hits	R/S	Err	Hits	R/S	Err	Hits	R/S	Err	Hits	R/S	Err
500	18756	156.07	SI	9144	76.09	NO	26771	224.28	NO	9012	75.62	NO

En els resultats anteriors, a partir de 500 clients en amunt els valors no varien gaire, el que em fa pensar que o bé el client no dona per més, o bé la limitació és l'ample de banda ja que la càrrega de la màquina on corre el XicHttpd es mantenia constant (excepte pel nombre de processos clar). S'observa doncs que quan el servidor realitza menys processament de les dades, és a dir, en la prova IDENTITY, s'aguanta perfectament la càrrega mantenint una taxa de R/X molt alta, i veiem que tot i canviar la configuració del XicHttpd la influència és mínima. No passa el mateix en les dues proves que requereixen de la compressió de les dades (GZIP i TOT-CGI) on en la primera configuració s'observa com la productivitat cau en picat en augmentar el nombre de clients, però d'altra banda si mirem els resultats de la segona prova veiem que aquí el fet de disposar de molts més *threads* estàtics i d'un valor menor del *timeout* provoca un augment considerable del rendiment del servidor. Finalment els resultats de la prova TOT encara que semblin incoherents no ho són tant, ja que el fet que apareguin errors de connexió (entre 10 i 120) provoca que el client faci una nova petició sense esperar al *timeout*, aquest errors són degut a la implementació de la funcionalitat CGI ja que no està molt depurada, i en càrregues altes alguns dels subprocessos CGI es queden en estat *zombie* fins que reiniciem el servidor

(no és tan crític quan tots els *threads* són dinàmics, ja que a la mort del *thread* també se'n moren els seus subprocessos).

- En la prova següent canviem la columna de clients per la del *timeout* configurat al XicHttpd, es configura el Web Stress per tal que simuli la sortida d'un mòdem de 56 Kbps, i el nombre de clients són 200.

Taula 3-4. XicHttpd: 50 *threads* estàtics, 200 clients a 56 Kbps

Time-out	**TOT**			**TOT-CGI**			**IDENTITY**			**GZIP**		
	Hits	**R/S**	**Err**	**Hits**	**R/S**	**Err**	**Hits**	**R/S**	**Err**	**Hits**	**R/S**	**Err**
60	13267	110.39	NO	5033	42.23	NO	8472	71.09	NO	9771	81.31	NO
15	12034	100.13	SI	7079	58.90	NO	8610	71.64	NO	12116	100.81	NO

L'alta productivitat de la prova TOT en comparació amb les altres s'explica pel fet que les respostes CGI ocupen poquets bytes i a més forcen el tancament de la connexió. Pel que fa a les demés podem trobar els resultats més representatius si comparem la prova IDENTITY amb la GZIP, on veiem que efectivament quan anem curts d'ample de banda el fet d'enviar el contingut comprimit augmenta considerablement la productivitat del servidor, i l'impacte es nota més encara si disminuïm el *timeout* del servidor.

Per acabar parlarem de les utilitats que s'han fet servir per tal de veure l'estat de la màquina sobre la que corre el XicHttpd ja que es considera un aspecte prou important. Tot i així el fet de determinar els requisits *hardware* mínims en funció de la càrrega ja no és un tema tan trivial, i s'haguéssin necessitat diferents entorns per tal de fer les proves.

- **top**. Per al meu gust és una eina més encarada a l'administració, que no pas al monitoratge, que és el que es pretén, però ens pot ser útil en determinats moments per tal de veure quins processos utilitzen el processador, i quins estan intercanviats per exemple, a més es clar de la memòria que utilitzen així com la que comparteixen amb d'altres processos.

Exemple 3-15. Exemple del top en plena prova "TOT"

```
17:00:36 up 57 min,  4 users,  load average: 3.67, 0.83, 0.26
220 processes: 165 sleeping, 46 running, 9 zombie, 0 stopped
CPU states:  54.8% user,  26.3% system,   0.0% nice,  18.9% idle
Mem:    512448K total,   269688K used,   242760K free,     4604K buffers
Swap:  1052216K total,        0K used,  1052216K free,    97712K cached

PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
958 aposai    19   0  7624 7624  2552 S     1.2  1.4   0:00 xic
632 aposai    17   0  7596 7596  2580 S     0.9  1.4   0:00 xic
```

```
1027 aposai    19   0  2160 2160  1160 R     0.9  0.4   0:00 exemple.cgi
1029 aposai    19   0  2156 2156  1160 R     0.9  0.4   0:00 exemple.cgi
 884 aposai    18   0  7604 7604  2556 S     0.7  1.4   0:00 xic
 887 aposai    14   0  7604 7604  2556 S     0.7  1.4   0:00 xic
 940 aposai    17   0  7604 7604  2552 S     0.7  1.4   0:00 xic
 982 aposai    17   0  7624 7624  7596 S     0.7  1.4   0:00 xic
1048 aposai    19   0  2152 2152  1156 R     0.7  0.4   0:00 exemple.cgi
```

- **ps**. La manera més ràpida de conèixer el PID del procés que volem matar, tot sovint però, l'he utilitzat per saber el nombre de *threads* del XicHttpd executant-se en un determinat moment, combinant la comanda amb un `grep` i un `wc`.

Exemple 3-16. Exemple d'ús del ps per comptar el nombre de *threads*

```
aposai@Natasza:~/tfc/XicHttpd/bin$ ps -e |grep xic |wc
    407    1636   11476
  (xics)
```

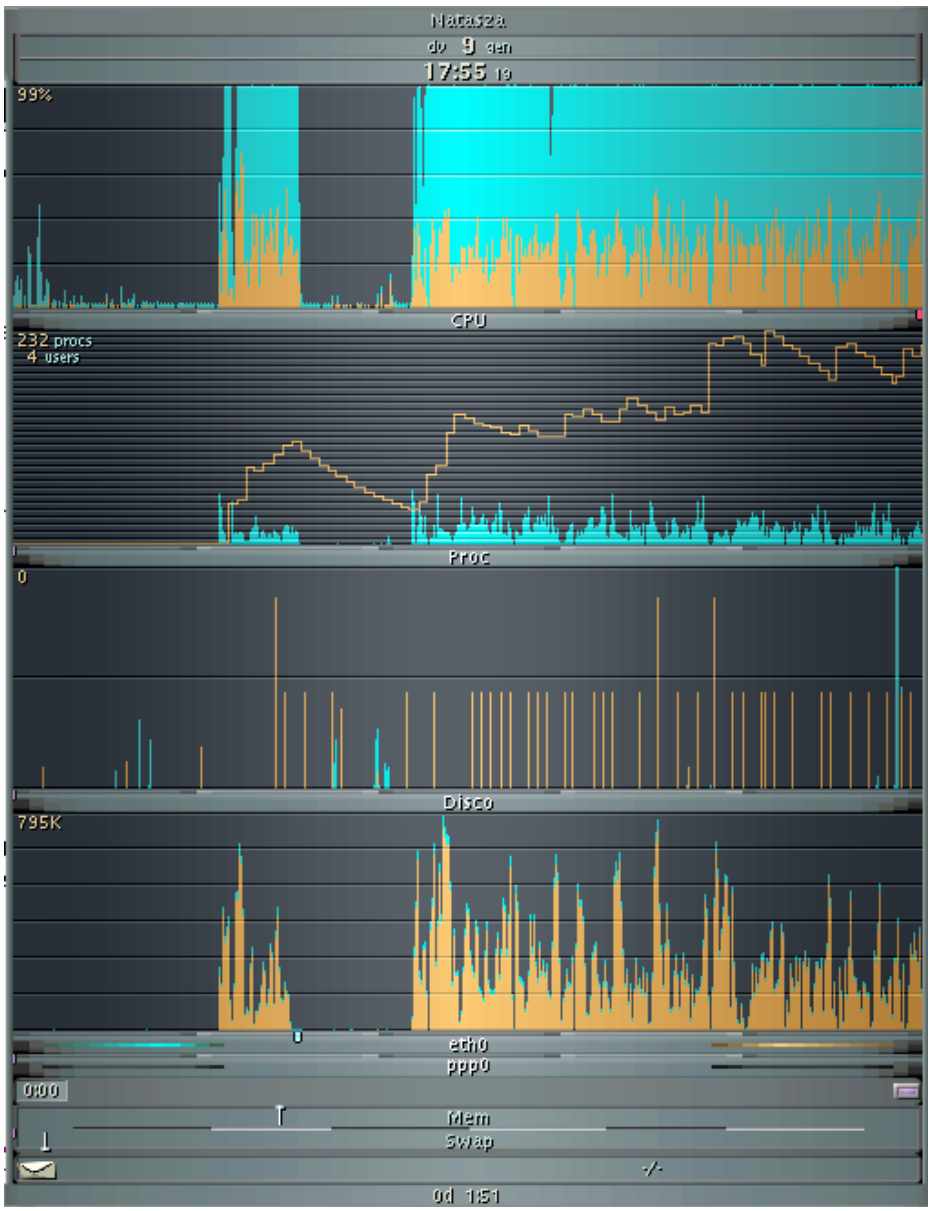
- **vmstat**. Aquest monitor treu estadístiques generals del sistema, de manera que no sabrem quins processos hi ha, però en canvi ens permet veure l'estat del sistema durant un interval de temps i a més si volem en podem emmagatzemar les dades per analitzar-les més tard.

Exemple 3-17. Exemple del vmstat en plena prova "TOT"

```
aposai@Natasza:~/tfc/XicHttpd/bin$ vmstat 1 4000 (es mostra la segona plana)
 procs                  memory    swap          io     system         cpu
  r  b  w   swpd   free   buff  cache  si  so    bi    bo   in    cs  us  sy  id
  8  0  1  38308 353884    568  21320   0   0     0     0  750   345  83  17   0
  7  0  1  38308 363088    568  21324   0   0     0     0  969   502  59  41   0
 10  0  1  38308 366388    568  21332   0   0     0   128 1117   544  64  36   0
 14  0  1  38308 343636    568  21344   0   0     0     0 1401   568  65  35   0
 17  0  1  38308 343684    568  21356   0   0     0     0 2308   772  66  34   0
 20  0  1  38308 348708    568  21384   0   0     0     0 2867   811  50  50   0
 25  0  1  38308 332876    568  21396   0   0     0     0 2446   700  65  35   0
 27  0  1  38308 324940    568  21420   0   0     0     0 2813   757  64  36   0
 25  0  1  38308 351300    568  21428   0   0     0     0 1820   592  62  38   0
 41  0  1  38308 333188    568  21440   0   0     0     0 1783   575  66  34   0
 23  0  0  38308 359168    568  21460   0   0     0     0 2237   681  62  38   0
```

- **GKrellM**[9]. És un monitor per les X, que ens permet de veure gràficament l'evolució de l'estat del sistema, així com l'ús de la xarxa. Sovint un cop d'ull al gkrellm ens ofereix la informació que necessitem ràpidament.

Figura 3-4. Exemple del GKrellM en plena prova "TOT"



Notes

1. L'ethereal és un sniffer per les X (*http://www.ethereal.com*). Té una funció que m'ha estat especialment útil al provar els *threads* que s'anomena *Follow TCP Stream* del menú que apareix amb

el botó dret del ratolí quan seleccionem una trama , així es pot veure tota la conversa d'una connexió TCP concreta.

2. gdb és l'acrònim de GNU Debugger
3. POST: *http://www.ussg.iu.edu/hypermail/linux/kernel/0006.3/0209.html*
4. Es pot trobar una bona explicació d'aquests detalls a *http://www.linuxjournal.com/article.php?sid=1363*.
5. Article interessant sobre programació segura: *http://www.tldp.org/linuxfocus/Castellano/January2001/article182.shtml*
6. Surge: *http://www.cs.bu.edu/faculty/crovella/links.html*
7. JMeter: *http://jakarta.apache.org/jmeter/index.html*
8. Web Stress: *http://www.microsoft.com/technet/itsolutions/intranet/downloads/webstres.asp*
9. GKrellM: *http://freshmeat.net/projects/gkrellm*

Capítol 4. Conclusions

4.1. Conclusions

La primera cosa que m'ha sorprés sobre el protocol HTTP és que és un protocol força relaxat en el sentit que el XicHttpd només n'implementa una petita part, i tot i així (com a mínim en les proves que he fet) es capaç de servir planes a dos dels clients més utilitzats, com són el Mozilla i l'Internet Explorer, a banda és clar d'aguantar l'"Stress" de les proves. Diguem que, igual que a Matrix, hi ha algunes regles que s'han de cumplir i d'altres que es poden saltar, el perment de definir uns requisits mínims en la implementació, i després anar-los escalant.

També he pogut comprovar que és realment difícil poder definir un entorn fiable per fer les proves, ja que hi ha molts factors que influeixen en el rendiment com ara l'ample de banda, el tipus de contingut (dinàmic/estàtic), el tipus de transferència (gzip), la màquina de proves, etc. I tot això provoca que no hi hagi una configuració general bona per als servidors Web, sinó que en cada cas s'han de controlar molt bé les diferents variables ja comentades, i anar provant fins a trobar la que ens convé.

No vull acabar sense dir que em sento molt satisfet d'aquest projecte ja que és el primer en el que he pogut fer de dalt a baix, i a més penso que el producte resultant satisfà prou bé els objectius que m'havia marcat, tan personals com funcionals (...). A més de la quantitat de coses que he aprés, he de dir que he disfrutat molt en aquestes últimes parts de proves, depuració i tunning, veient com es comportava el meu XicHttpd.

4.2. Agraiments

- Al Karpin, cap d'Informàtica de maimai (*http://www.maimai.com*), sense el qual mai no hagués reprès els meus estudis d'Informàtica, i el qual sempre m'ha animat i ajudat quan ha calgut.
- Als meus pares que sempre em criden per sopar.
- Al Roger, el meu germà petit, que sempre que tinc el cap calent ve a l'habitació i ens posem a tocar els tambors per acabar-lo d'escalfar.
- A la Glòria, la meva cunyada, que se l'emporta.
- I finalment al Raül i al Xic, els meus companys d'Snowboard i amics de sempre, per haver marxat a la neu sense mi més d'una vegada, perquè jo pogués acabar aquest projecte.

Bibliografia

Brian W. Kernighan Dennis M. Ritchie, *El lenguaje de programación C*, Segunda edición, Pretince-Hall, Inc., ISBN 968-880-205-0.

Francisco Manuel Márquez García, *UNIX Programación avanzada*, Segunda edición, ra-ma.

Christopher Negus, *La biblia de Red Hat Linux 7*, Anaya.

Jordi Íñigo Griera, *Xarxes de computadors II*, Universitat Oberta de Catalunya.

Varis, *Perl, CGI y JavaScript*, Anaya.

Norman Walsh Leonard Muellner, *DocBook: The Definitive Guide*, O'Reilly & Associates, Inc., *http://docbook.org/tdg/en/html/docbook.html*, Desembre, 2003.

The World Wide Web Journal, 2, 1, O'Reilly & Associates, Inc., The World Wide Web Consortium, Hivern, 1996.

Posix Threads Programing, *http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html*.

Schumi (inacabat), *Tutorial Pthreads*, *http://members.tripod.com/webprototype/tutorial_pthreads_menu.html*.

Multi-Threaded Programming With POSIX Threads, *http://users.actcom.co.il/~choo/lupg/tutorials/multi-thread/multi-thread.html*.

Internet Request for Comments (RFC), *http://www.cis.ohio-state.edu/cs/Services/rfc/index.html*.

Jim Frost, *BSD Sockets: A Quick And Dirty Primer*, *http://world.std.com/~jimf/papers/sockets/sockets.html*.

Unix Socket FAQ, *http://www.ibrado.com/sock-faq/*.

CGI: Common Gateway Interface, *http://www.w3.org/CGI/*.

Alex, *Server Multi Client*, *http://dsl.upc.es/docs/papers/serverMultiClient.txt*.

Ismael Ripoll, *Real-Time Linux*, *http://bernia.disca.upv.es/~iripoll/rt-linux/rtlinux-tutorial/06-sincronizacion/06-sincronizacion.html*.

Zlib FAQ, *http://www.gzip.org/zlib/FAQ.txt*.

Paul Rusty Russell, *Guía Informal al Bloqueo*, *http://es.tldp.org/Manuales-LuCAS/linux-bloqueo/single-html/*.

The GNU C Library: Low-Level Input/Output, *http://docs.linux.cz/glibc-manual/libc_13.html*.

David A. Wheeler, *Secure Programming for Linux and Unix HOWTO*, *http://en.tldp.org/HOWTO/Secure-Programs-HOWTO/*.

Frédéric Raynal, Christophe Blaess, Christophe Grenier, *Evitando los agujeros de seguridad durante el desarrollo de aplicaciones*, *http://www.tldp.org/linuxfocus/Castellano/January2001/article182.shtml*.

HTTP/1.1, *RFC 2616*, *http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc2616.html*.

GZIP, *RFC 1952*, *http://www.faqs.org/rfcs/rfc1952.html*.

Apèndix A. Instruccions de compilació i instal·lació del XicHttpd

```
XicHttpd Web Server v 0.5
per Miquel Fontanals

Aquest servidor és experimental, l'autor no es fa responsable
dels danys que pugui causar.

- XicHttpd és un servidor Web que fa una implementació simple
del protocol HTTP/1.1, suportant les comandes GET, HEAD i POST.

- També es permeten connexions HTTP/1.0 però sense persistència.

- El contingut s'enviarà comprimit en gzip sempre que es pugi

- El suport CGI és experimental i pot presentar problemes en
 situacions de càrregues altes. Vagi amb compte !

 Compilar:

 Al directori bin hi ha una versió precompilada que hauria de
 funcionar en la majoria dels casos. Cal però que tingui els
 permisos adients, executi com a root:

 $ chown root:root xic
 $ chmod u+s xic

 És necessàri tenir les llibreries pthread i zlib.
 S'aconsella disposar de les llibreries del C Glibc >= 2.0

 Entrar al directori src i executar make com a root.
 Això deixarà l'executable xic al directori bin.

 Probablement apareixeran 2 warnings, un referent al formateig
 de les dates, i l'altre a la funció getline (donat que no és portable),
 no en faci cas, el servidor funcionarà igualment.

 Executar:

 Accedir al directori bin i excutar ./xic com un usuari normal.
 El make haurà ficat el SUID actiu de l'executable, i el xic
 internament s'executarà com a l'usuari normal excepte en aquelles
 funcions on calen permisos especials, com ara la publicació del
 socket en un port reservat (veure server.c per més detalls).
 >
```

Apèndix B. Llistat de Servidors Web

- 4D WebSTAR - 4D, Inc *http://www.webstar.com/*
- Abriasoft Merlin Server - AbriaSoft *http://www.abriasoft.com/*
- AllegroServe - Franz Inc *http://www.franz.com/*
- AOLserver - America Online *http://www.aolserver.com/*
- Apache - Apache Software Foundation *http://www.apache.org/*
- BadBlue Personal Edition - Working Resources Inc *http://badblue.com/*
- Baikonur Web App Server - Epsylon Technologies *http://www.epsylontech.com/*
- BFOOT Embedded Ethernet Controllers - Hewlett-Packard Co *http://www.hp.com/*
- Caudium - The Caudium Group *http://caudium.net/*
- Commerce Server/400 - I/NET, Inc. *http://www.inetmi.com/products/webserv/webinfo.sht*
- Covalent Fast Start Server - Covalent Technologies, Inc *http://www.covalent.net/*
- Domino Go Webserver - IBM *http://www.software.ibm.com/webservers/dgw/*
- EMWAC HTTP Server - Edinburgh University *http://emwac.ed.ac.uk/*
- Enterprise Ready Server - Covalent Technologies, Inc *http://www.covalent.net/*
- Enterprise WebServer for NetWare - Novell *http://www.novell.com/products/*
- ESAWEB - Velocity Software, Inc. *http://velocitysoftware.com/esaweb.html*
- First Class Intranet Server - Centrinity Inc. *http://www.centrinity.com/*
- GOAhead WebServer - GoAhead Software *http://www.goahead.com*
- Hawkeye - Hawkeye Project *http://hawkeye.net/*
- iPlanet Web Server 6.0 - SUN Microsystems *http://www.sun.com*
- iTools - Tenon Intersystems *http://www.tenon.com/products/itools/*
- Java Server - Sun Microsystems *http://java.sun.com/*
- Jigsaw - W3C *http://www.w3.org/Jigsaw*
- Lancelot Server - Abriasoft *http://www.abriasoft.com/*
- Lotus Domino - Lotus *http://www.lotus.com/home.nsf/welcome/domino* (http://www.lotus.com/home.nsf/welcome/dominohttp://www.lotus.com/home.nsf/welcome/domino)
- Microsoft Internet Information Server - Microsoft Corp. *http://www.microsoft.com*
- Microsoft Personal Web Server - Microsoft Corp. *http://www.microsoft.com*
- Microsoft Site Server - Microsoft Corp. *http://www.microsoft.com*
- Mac OS X Server - Apple Computer, Inc *http://www.apple.com/*
- Netscape Enterprise Server - Netscape *http://www.netscape.com*
- Netscape FastTrack Server - Netscape *http://www.netscape.com*

- RapidSite - RapidSite, Inc. *http://www.rapidsite.com/about.html*
- RomPager Embedded Web Server - Allegro Software Development Corp. *http://www.allegrosoft.com/products.html*
- Roxen WebServer - Roxen Internet Software *http://www.roxen.com/*
- Savant - Michael Lamont *http://hera.wku.edu/~lamonml/*
- sedum - Guido Frassetto *http://www.frassetto.it/*
- Servertec Internet Server - Servertec *http://www.servertec.com/*
- SGI Internet Server Environment - Silicon Graphics (SGI), Inc *http://www.sgi.com/*
- Shadow Web Server - NEON Systems, Inc. *http://www.neonsys.com/*
- Simpleserver:WWW - AnalogX *http://www.analogx.com/contents/download/network/sswww.html*
- SITEFORUM WebServer - SFS Software *http://www.siteforum.com/*
- Stronghold - C2Net Software, Inc *http://www.c2.net/*
- StWeb - Teta Srl *http://www.teta.it/*
- Sun ONE Web Server - Sun Microsystems *http://wwws.sun.com/software/product_categories/web_servers.html*
- Tcl Web Server - Tcl Developer Exchange *http://dev.scriptics.com/software/tclhttpd/*
- URL Live! - Pacific Software Publishing, Inc. *http://www.pspinc.com/NTSG/htm/pro-vb.htm*
- Viking - RobTex *http://www.robtex.com/viking/*
- VqServer - vqSoft *http://www.vqsoft.com/vq/server/index.html*
- WebBase - Webbase, Inc. *http://www.webbase.com/http://www.webbase.com/* (http://www.webbase.com/)
- Web Forum Server - MiniHTTPServer Co. LTD *http://www.minihttpserver.net/*
- WebSite - Deerfield.com *http://www.deerfield.com/products/website/* (http://www.deerfield.com/products/website/http://www.deerfield.com/products/website/)
- WN Web Server - Northwestern University Math Dept *http://math.nwu.edu/home.html*
- Xitami - iMatix *http://www.xitami.com/*
- Zeus Appliance Edition - Zeus Technology *http://www.zeus.com/*
- Zeus Web Server - Zeus Technology *http://www.zeus.com/*