FACULTAD DE INGENIERIA – INSTITUTO DE COMPUTACIÓN

# REDES DE COMPUTADORAS

LABORATORIO Nº1

MATIAS PÉRES – NICOLAS VAZQUEZ - FEDERICO MUJICA – GERMAN RUIZ
AGOSTO Y SEPTIEMBRE DEL 2012

# 1 ÍNDICE

# 2 INTRODUCCIÓN

En el presente trabajo se describe el proceso de resolución del laboratorio número uno de la asignatura Redes de Computadoras.

La realización del trabajo se ha enfocado de manera tal que sirva como guía para el lector, no solo como solución inmediata de los clausulas, sino necesario, relevante, mediante el cual se pueda lograr una comprensión total y detallada de las cláusulas establecidas y de las soluciones.

Los objetivos son: cumplir con las cláusulas, ahondar en la utilización de los conceptos aprendidos, lograr un mejor manejo del entorno de desarrollo y ser lo más descriptivos posible de forma de lograr un desarrollo coherente, explicando las decisiones tomadas en cada sección del laboratorio.

En el capítulo 3, se realiza un acercamiento al marco teórico de las distintas herramientas utilizadas para la resolución del obligatorio. Entre estas destaca, el uso de funciones para creación de hilos, creación, bloqueo y desbloqueo de semáforos y funciones relacionadas a los Sockets en C. Se procede con una introducción a los protocolos y en especial al protocolo HTTP, de especial importancia en la resolución del obligatorio. Intentamos focalizar la mirada en los conceptos tratados más adelante.

En el capítulo 4 se describe la resolución del obligatorio. Se realizan las observaciones necesarias en cuánto a detalles técnicos, siguiendo las bases establecidas en la teoría e introduciendo cambios y decisiones. Es importante señalar que en ningún momento se introducen cambios ajenos a las cláusulas establecidas por el obligatorio.

El trabajo se encuentra respaldado por material bibliográfico confiable que ha sido seleccionado con atención, a efectos de que la información en la cual nos basemos para la realización del mismo sea certera, lo que brinda constantemente una seguridad en la lectura.

El proceso de elaboración del trabajo consistió en la división temática del mismo, asignando a cada uno de los integrantes del equipo una o varias secciones determinadas, teniendo en cuenta que el conocimiento sea general y llegando a una puesta en común antes de la división.

# 3 CONCEPTOS PREVIOS

## 3.1 CREACIÓN DE THREADS

### 3.1.1 Procesos, hilos

Antes de hablar de la llamada al sistema *pthread_create*, conviene hablar sobre procesos e hilos. Dos conceptos muy parecidos y relacionados, pero con un conjunto de sutiles diferencias.

Uno de los principales motivos de la existencia de la informática es imitar el comportamiento de la mente humana. En un comienzo surgieron los algoritmos, que no son más que una secuencia de pasos para conseguir un objetivo, a partir de los cuales surgió el pensamiento de "por qué no hacer varias cosas a la vez" y es precisamente de esta inquietud de donde surgen los hilos o threads.

Si queremos que nuestro programa empiece a ejecutar varias cosas "a la vez", tenemos dos opciones. Por una parte podemos crear un nuevo proceso y por otra, podemos crear un nuevo hilo de ejecución (un thread). En realidad nuestro ordenador, salvo que tenga varias CPU's, no ejecutará varias tareas a la vez. Esto se refiere a que el sistema operativo, en este caso Linux, irá ejecutando los threads según la política del mismo, siendo lo más usual mediante rodajas de tiempo muy rápidas que dan la sensación de simultaneidad.

En un sistema Linux, que como ya sabemos es multitarea (sistema operativo multihilo), se pueden estar ejecutando distintas acciones a la par, y cada acción es un proceso que consta de uno o más hilos, memoria de trabajo compartida por todos los hilos e información de planificación. Cada hilo consta de instrucciones y estado de ejecución.

Cuando ejecutamos un comando en el Shell, sus instrucciones se copian en algún sitio de la memoria RAM del sistema para ser ejecutadas. Cuando las instrucciones ya cumplieron su cometido, el programa es borrado de la memoria del sistema, dejándola libre para que más programas se puedan ejecutar a la vez. Cada uno de estos programas en ejecución son los procesos.

Los procesos son creados y destruidos por el sistema operativo, pero lo hace a petición de otros procesos. El mecanismo por el cual un proceso crea otro proceso se denomina bifurcación (fork). Los nuevos procesos son independientes y no comparten memoria (es decir, información) con el proceso que los ha creado.

En definitiva, es posible crear tanto hilos como procesos. La diferencia estriba en que un proceso solamente puede crear hilos para sí mismo y en que dichos hilos comparten toda la memoria reservada para el proceso.

Los hilos son similares a los procesos ya que ambos representan una secuencia simple de instrucciones ejecutada en paralelo con otras secuencias. Los hilos son una forma de dividir un programa en dos o más tareas que corren simultáneamente, compitiendo, en algunos casos, por la CPU.

La diferencia más significativa entre los procesos y los hilos, es que los primeros son típicamente independientes, llevan bastante información de estados, e interactúan sólo a través de mecanismos de comunicación dados por el sistema. Por otra parte, los hilos generalmente comparten la memoria, es decir, acceden a las mismas variables globales o dinámicas, por lo que no necesitan costosos mecanismos de comunicación para sincronizarse.

### 3.1.2 Creación de Threads

La librería *pthreads* es una librería que cumple los estándares POSIX y que nos permite trabajar con distintos hilos de ejecución al mismo tiempo.

Para crear un hilo nos valdremos de la función *pthread_create* de la librería y de la estructura de datos *pthread_t* que identifica cada hilo diferenciándolo de los demás y que contiene todos sus datos.

El prototipo de la función es el siguiente:

*int pthread_create(pthread_t* thread, pthread_attr_t *attr, void* (*start routine)(void*), void *arg)*

- *Thread* es una variable del tipo *pthread_t* que contendrá los datos del hilo y que sirve para identificar el hilo en concreto si nos interesa hacer llamadas a la librería para llevar a cabo alguna acción sobre él.
- *Attr* es un parámetro del tipo *pthread_attr_t* y que se debe inicializar previamente con los atributos que queramos que tenga el hilo. Si pasamos como parámetro NULL la librería otorga al hilo atributos por defecto.
- *Start_routine*: aquí pondremos la dirección de la función que queremos que ejecute el hilo. La función debe devolver un puntero genérico (void*) como resultado y debe tener como único parámetro otro puntero genérico.

- *Arg* es un puntero al parámetro que se le pasará a la función. Puede ser NULL si no queremos pasarle nada.

En caso de que haya ido todo bien, la función devuelve un 0 o un valor distinto de 0 de caso de que hubo algún error. Una vez llamado a esta función ya tenemos a nuestro hilo funcionando.

## 3.2 MULTIEXCLUSIÓN

#### 3.2.1.1 Sincronización de hilos en POSIX: Mutex

Estas funciones incluyen mecanismos de exclusión mutua (mutex), mecanismos de señalización del cumplimiento de condiciones por parte de variables, y mecanismos de acceso de variables que se modifican en forma exclusiva, pero pueden ser leídas en forma compartida. Las funciones para el manejo de zonas de acceso exclusivo tienen el prefijo pthread_mutex.

Un mutex es una variable especial que puede tener estado tomado (locked) o libre (unlocked). Es como una compuerta que permite el acceso controlado. Si un hilo tiene el mutex entonces se dice que es el dueño del mutex. Si ningún lilo lo tiene se dice que está libre (o unclucked). Cada mutex tiene una cola de hilos que están esperando para tomar el mutex. El uso de mutex es eficiente, pero debería ser usado sólo cuando su acceso es solicitado por corto tiempo.

#### 3.2.1.2 Creación e iniciación de mutex

```
#include <pthread.h>
int  pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mutexattr_t * attr);
```

Alternativamente podemos invocar:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

lo cual inicializa el mutex con los atributos por omisión. Es el uso que daremos en este curso. Es equivalente a invocar:

```
pthread_mutex_t mylock;
pthread_mutex_init(& mylock, NULL);
```

#### 3.2.1.3 Destrucción de un mutex

```
#include <pthread.h>
int  pthread_mutex_destroy(pthread_mutex_t * mutex);
```

Si el mutex lo tenía otro hilo y éste es destruido, POSIX no define el comportamiento de mutex en esta situación.

#### 3.2.1.4 Solicitud y liberación de un mutex

```
#include <pthread.h>
int  pthread_mutex_lock(pthread_mutex_t * mutex);
int  pthread_mutex_trylock(pthread_mutex_t * mutex);
int  pthread_mutex_unlock(pthread_mutex_t * mutex);
```

Con pthread_mutex_trylock el hilo siempre retorna, si la función es exitosa, se retorna 0 -como en los otros casos; si no se retornará EBUSY indicando que otro hilo tiene el mutex.

Ejemplo: Para proteger una zona crítica, usar:

```
pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&mylock);
/* Sección crítica */
pthread_mutex_unlock(&mylock);
```

## 3.3 SOCKETS EN C

Los sockets son interfaces que permiten que los procesos envíen y reciban mensajes a través de él. Si pensamos en los teléfonos, que permiten la comunicación entre las personas, podemos establecer una analogía entre los teléfonos y los sockets, quienes permiten la comunicación entre procesos.

Los procesos pueden comunicarse estando en un mismo sistema, o en distintos sistemas pero unidos mediante una red. En el segundo caso, lo más común es que la comunicación entre procesos se lleve a cabo mediante el modelo cliente-servidor. Con este modelo, tenemos que los procesos cliente se conectaran con el proceso servidor para hacerle pedidos (requests) de datos.

En este obligatorio trabajaremos con el segundo caso, por lo tanto el dominio de comunicación que utilizaremos será AF_INET. Utilizando la librería <*sys/socket.h*>:

Para poder crear una conexión, debe crearse un socket, y esto se hace mediante la función *socket()* que retorna un descriptor del socket, de tipo entero, retornando -1 en caso de error:

Ejemplo de invocación:

*int descriptorSocket = socket ( AF_INET, SOCK_STREAM, 0 );*

Luego de crear el socket, si queremos recibir conexiones (por ejemplo para un servidor) debemos asignarle una IP y un puerto por el cual escuchará, y para esto utilizamos la función *bind()*, que devuelve un entero, y también retorna -1 en caso de error.

Ejemplo de invocación:

*int bind(descriptorSocket, (struct sockaddr* ) my_addr, sizeof(my_addr);*

Luego, si queremos habilitar al socket (para que pueda recibir conexiones) debemos utilizar la función *listen()*.

*int listen (descriptorSocket, MAXQUEUE);*

donde MAXQUEUE representa el número máximo de conexiones en la cola de entrada, estas quedan en estado de espera hasta que se aceptan, mediante la función *accept()*.

Por último, el cliente utiliza la función *connect()* para iniciar la conexión con el servidor y mediante las funciones *send()* y *recv()* comienza con la transferencia de datos.

*int connect ( int sockfd, struct sockaddr *serv_addr, int addrlen )*

- *sockfd* es el descriptor de socket devuelto por la función *socket()*.
- *serv_addr* es una estructura *sockaddr* que contiene la dirección IP y número de puerto destino.
- *addrlen* debe ser inicializado al tamaño de *struct sockaddr*.

Después de establecer la conexión, se puede comenzar con la transferencia de datos. Estas dos funciones son para realizar transferencia de datos sobre sockets stream. *send()* y *recv()* son idénticas a *write()* y *read()*, excepto que se agrega un parámetro flags.

*send ( int sockfd, const void *msg, int len, int flags )*

- *sockfd* descriptor socket por donde se enviarán los datos.
- *msg* es un puntero a los datos a ser enviados.
- *len* es longitud de los datos en bytes.

- *flags* leer: man 2 send

*send()* retorna la cantidad de datos enviados, la cual podrá ser menor que la cantidad de datos que se escribieron en el buffer para enviar. *send()* enviará la máxima cantidad de datos que pueda manejar y retorna la cantidad de datos enviados, es responsabilidad del programador comparar la cantidad de datos enviados con len y si no se enviaron todos los datos, enviarlos en la próxima llamada a *send()*.

*recv ( int sockfd, void *buf, int len, unsigned int flags )*

- *sockfd* descriptor socket por donde se recibirán los datos.
- *buf* es un puntero a un buffer donde se almacenarán los datos recibidos.
- *len* es longitud del buffer buf.
- *flags* leer: man 2 recv

Si no hay datos a recibir en el socket, la llamada a *recv()* no retorna (bloquea) hasta que llegan datos, se puede establecer al socket como no bloqueante (ver: man 2 fcntl ) de manera que cuando no hay datos para recibir la función retorna -1 y establece la variable errno=EWOULDBLOCK. *recv()* retorna el número de bytes recibidos.

## 3.4 CONEXIONES PERSISTENTES Y NO PERSISTENTES

En muchas aplicaciones de internet, el cliente y el servidor están comunicados durante un periodo de tiempo amplio, haciendo el cliente una serie de solicitudes y el servidor respondiendo a dichas solicitudes. Dependiendo de la aplicación y de cómo se esté empleando la aplicación, las solicitudes pueden hacerse una tras otra, periódicamente a intervalos regulares o de forma intermitente. Cuando esta interacción se realiza mediante el protocolo TCP, el desarrollador de la aplicación debe tomar una decisión importante, deberá elegir entre enviar todas las Solicitudes/Respuestas en conexiones TCP separadas o enviar todas las Solicitudes/Respuestas a través de la misma conexión TCP. Si se utiliza el primer método se dice que la aplicación utiliza conexiones no persistentes, mientras que el segundo la aplicación está utilizando conexiones persistentes.

### 3.4.1 Conexiones no persistentes con HTTP

Los pasos de transferencia de una página web desde un servidor web para una conexión no persistente son los siguientes. Supongamos que la pagina

consta de un archivo base HTML y de 10 imágenes JPEG, residiendo los 11 objetos en el mismo servidor.

Hace el pedido del objeto base, en este caso el HTML, la conexión TCP se cierra una vez que el objeto es recibido por el cliente. En el objeto HTML se encuentran las 10 referencias a las imágenes JPEG, por lo cual el cliente debe abrir 10 conexiones TCP y cerrarlas una vez que recibe los objetos.

Para comparar estos 2 métodos vamos a definir el tiempo de ida y vuelta (RTT, Round Trip Time), es el tiempo que tarda un paquete pequeño en viajar del cliente al servidor y del servidor al cliente. El tiempo RTT incluye los retardos de propagación de los paquetes, los retardos de cola en los router y switches intermedios y los retardos de procesamiento de paquetes.

Para las conexiones no persistentes esto puede presentar un problema, en primer lugar se tiene que establecer y mantener una conexión completamente nueva para cada objeto solicitado. Para cada una de estas conexiones, deben asignarse los buffer TCP y las variables TCP tienen que ser mantenidas tanto en el servidor como en el cliente. Esto puede cargar de forma significativa el servidor web ya que puede estar atendiendo a miles de cliente a la vez. También sufre un retraso de entrega de 2 RTT, uno para establecer la conexión http y otro para enviar los objetos pedidos.

### 3.4.2 Conexiones persistentes con HTTP

En las conexiones persistentes, el servidor deja la conexión TCP abierta después de enviar una respuesta. Las subsiguientes solicitudes qué tienen lugar entre el mismo cliente y servidor pueden enviarse a través de la misma conexión. Estas solicitudes de objetos pueden realizarse una tras otra sin esperar a obtener respuesta de las solicitudes pendientes utilizando procesamiento en cadena (pipelining). El modelo por defecto de http utiliza conexiones persistentes con procesamiento en cadena.

## 3.5 PROTOCOLO HTML

Es un protocolo (conjunto de reglas a seguir) en el que su propósito es permitir la transferencia de archivos entre un cliente y un servidor, siguiendo el esquema de solicitud-respuesta (request-response). A grandes rasgos la comunicación entre el cliente y el servidor se divide en dos etapas: la primera es cuando el cliente realiza una solicitud http y la segunda es la respuesta del servidor.

Una solicitud por parte de un cliente consiste en especificar el tipo de documento solicitado, la dirección URL de donde obtenerlo, el método que se aplicara y la versión del protocolo utilizada por el cliente (generalmente esta es HTTP/1.0).
Una respuesta por parte de un servidor consiste en la versión del protocolo utilizada, el código de respuesta y el significado de este código.

### 3.5.1 Mensaje de Solicitud HTTP

La primera línea del mensaje de solicitud HTTP se denomina línea de solicitud y las siguientes son las líneas de cabecera.

La línea de solicitud consta de 3 campos:

1. El campo que especifica el método que puede tomar diferentes valores, entre los que se incluye GET, POST, HEAD, PUT, DELETE. La gran mayoría de los mensajes de solicitud HTTP utilizan el método GET
2. El campo de URL
3. La versión del HTTP

### 3.5.2 Cabecera de mensaje de solicitud

Los campos de cabecera del mensaje de solicitud le permiten al cliente pasar mayor información al servidor sobre el pedido y sobre el mismo cliente. Algunos por ejemplo son:

1. Authorization
    - Este campo lo puede utilizar el cliente para identificarse enfrente a un servido, básicamente se le envían los valores de las credenciales.
2. From
    - En este campo el cliente puede incluir una casilla de mail, para diferentes usos dentro del servidor.
3. If-Modified-Since
    - Campo utilizado con el método Get para condicionar el pedido

### 3.5.3 Métodos del mensaje de solicitud

1. GET:
    - El método Get significa que el cliente está solicitando que se le envié el objeto que se encuentra en la URL, si la URL apunta a un procedimiento que genera un archivo, lo que se devuelve en el cuerpo de la entidad en el response es el archivo y no el texto del procedimiento.
    - El método Get se puede condicionar si el mensaje de solicitud incluye el campo If-Modified-Since en su cabecera. Esto le indica al servidor

que devuelva el objeto únicamente si ha sido modificado desde la fecha enviada en el campo If-Modified-Since. Esto condicionamiento es un buen uso para reducir el tráfico en la red y permite a las entidades cacheadas actualizarse sin realizar múltiples pedidos ni transferir datos innecesarios.

2. HEAD

   - El método HEAD es idéntico al GET pero no devuelve el objeto en el cuerpo de la entidad en la respuesta, sin embargo toda la información devuelta en el cabezal de la respuesta a este método es idéntica como si se hubiera ejecutado un GET, esto sirve para obtener información del objeto sin tener que transferir todo el objeto en sí.

3. POST

   - El método POST envía un objeto en el pedido al servidor.
   - Este método es utilizado para que el servidor acepte el objeto enviado por el método y lo asocie con el recurso ingresad en la URL del pedido. Un típico uso del método POST es por ejemplo cuando se llena un formulario en una página web y este es enviado mediante un FORM.

### 3.5.4 Mensaje de Respuesta HTTP

La estructura general de este mensaje es la primera líneas se llama línea de estado, luego vienen las líneas de cabecera y al final el cuerpo de entidad. El cuerpo de entidad es la parte más importante del mensaje, ya que contiene el objeto solicitado en sí.

La línea de estado contiene 3 campos, el que especifica la versión del protocolo, el correspondiente al código de estado y el tercer campo que contiene el mensaje explicativo del estado correspondiente.

#### 3.5.4.1 El cuerpo de entidad

El cuerpo de entidad contiene un cabezal con información sobre el objeto si lo hay o sobre el recuso identificado por el pedido, algunos campos son:

1. Allow

- El campo Allow indica los métodos soportados por el recurso identificado por el pedido. Un proxy no debe modificar el campo Allow aunque no identifique alguno de los métodos especificados

2. Content-Encoding
    - Este campo indica que tipo de codificación adicional se le aplico al recurso, obteniendo el desarrollador directamente que decodificación debe aplicarse para obtener el recurso en sí. Este campo es básicamente usado para comprimir el recurso sin perder su identidad.

3. Content-Length
    - Este campo indica el tamaño del recurso o si no existe recurso en el cuerpo de entidad, el tamaño del recurso identificado por el request si se hubiera echo un método GET.

4. Content-Type
    - Este campo indica el tipo del recurso identificado por el request, en el caso que el método sea HEAD se envía el valor de este campo si se hubiera echo un GET.

5. Expires
    - Este campo indica con una fecha/hora cuando el recurso enviado va a dejar de ser válido. No da una idea de la volatilidad del objeto. La presencia de este campo no nos indica exactamente que el objeto va a cambia o dejar de existir cuando la fecha expire.

6. Last-Modified
    - Indica la ultima fecha/hora en la cual la terminal que envía el mensaje cree que el objeto fue modificado

# 4 RESOLUCIÓN

## 4.1 INTRODUCCIÓN

El laboratorio consiste en el desarrollo de una aplicación en C++ que permite controlar el tráfico web mediante políticas definidas por administradores de la aplicación. Se supone por tanto, una implementación que actúa de forma similar a un proxy HTTP, permitiendo el cacheo de datos y el análisis de los contenidos que son requeridos por navegadores y respondidos por servidores web. Es posible por tanto denegar un requerimiento o modificar una respuesta. A instancias de este laboratorio solo se solicita impedir el pasaje de respuestas que superan cierto tamaño configurable desde la administración.

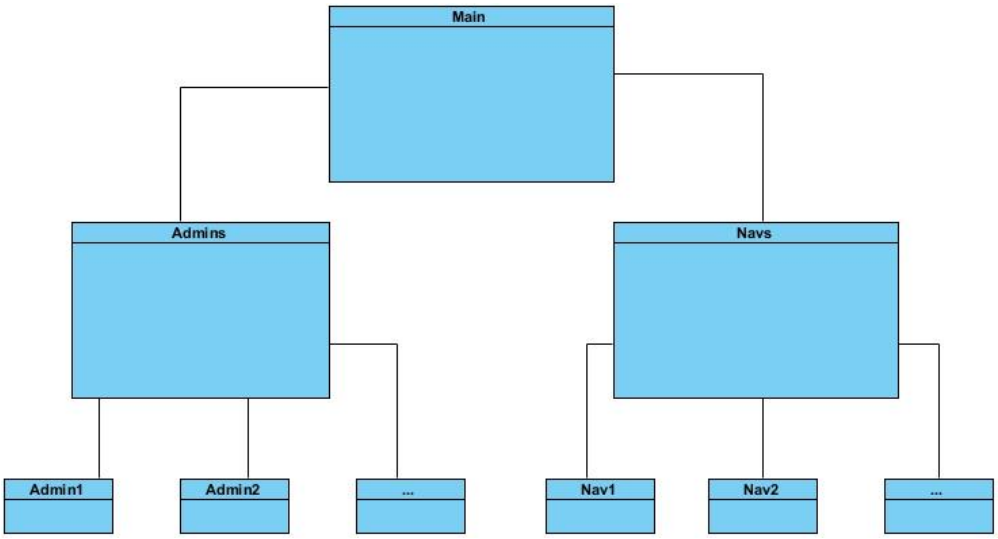
La aplicación constará por tanto de dos entidades que se encarguen de los navegadores y por otro lado de los administradores. En el siguiente capítulo se describe la arquitectura de la aplicación.

## 4.2 ESTRUCTURAS

Como es posible entrever de la descripción anterior, se tendrán dos grandes módulos o componentes que gestionen de forma independiente pero sincronizada los pedidos solicitados por navegadores y administradores.

Una vez comprendido los pasos necesarios para dar lugar a la creación de la aplicación, desarrollamos una arquitectura capaz de soportar dicho comportamiento pero que a su vez, permitiera la separación entre distintas funcionalidades del sistema, facilitando el trabajo en equipo y el desarrollo seguro hacia una aplicación sólida y eficiente. Otro de los aspectos tenidos en cuenta fue el desarrollo de un código comprensible y coherente, cuyo testeo no imponga dificultades ajenas a la intuición.

En primer lugar examinaremos un diagrama en primera aproximación a la estructura de la aplicación:




Aquí es posible observar la conexión entre el módulo principal del programa *main*, que da lugar a otros dos módulos, que serán los encargados de atender a los distintos clientes que se conectan a nuestra aplicación.

Ahora bien, el servidor proxy debe atender en dos puertos diferentes, uno para las conexiones de administradores y otro al que se conectarán los navegadores para hacer pedidos HTTP. En AMBOS casos, el servidor será una aplicación *multihilo*, es decir, cada vez que se conecta un cliente, el servidor le asignará un hilo de ejecución independiente, representados en el diagrama mediante puntos suspensivos.

Como se especifica en la letra del obligatorio la implementación del servidor proxy tiene como referencia las RFCs 1945 y 2612, que definen HTTP/1.0 y HTTP/1.1 respectivamente. El funcionamiento en cada nueva conexión con un pedido HTTP por parte del navegador es el siguiente:

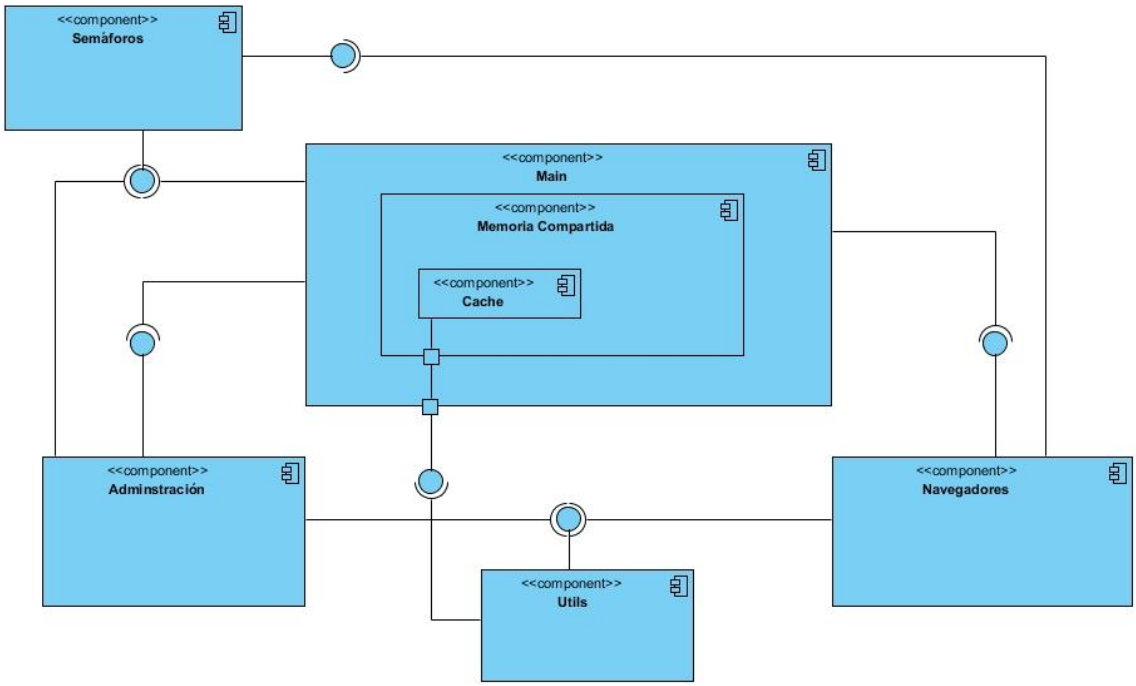
1. Es atendida en un hilo de ejecución independiente.
2. El pedido HTTP es recibido, analizado y procesado de acuerdo a:
a) Si es un método GET y está denegado administrativamente, se retorna únicamente el mensaje de error (con el código adecuado, para archivo demasiado grande). El único filtro a implementado es aquel relativo al tamaño de los objetos transferidos.
b) Si es un método GET y no se lo tiene en memoria, se realiza la conexión al servidor correspondiente, se obtiene el objeto, se lo entrega al cliente y si su tamaño es menor al tamaño máximo de objeto cacheable, se almacena en memoria.
c) Si es un método POST y está denegado administrativamente, se retorna únicamente el mensaje de error (con el código adecuado).

d) Si no es ni POST ni GET, se retorna únicamente el mensaje de error (con el código adecuado).
3. Se mantiene un máximo de objetos en memoria, y los más viejos son eliminados en caso de peticiones nuevas (política "Least Recently Used" - LRU).
4. Se eliminan objetos expirados de la cache y no se entregan a los usuarios.
5. Una vez que ya fueron obtenidos todos los datos, y fueron enviados (con modificaciones) al navegador, se cierra la conexión.

Todo lo que se acaba de mencionar induce a la introducción de herramientas que permitan el manejo de los datos globales de la aplicación. Dado que se trata de una aplicación multihilo, se tienen varias instancias de ejecución accediendo a los mismos datos que eventualmente podrían interceptarse y ocasionar fallas de memoria, por lo que se necesita también del uso de semáforos

Para la presentación de la arquitectura se ha optado por un diagrama de componentes y conectores. Cada elemento del diagrama, como es usual en un diagrama de componentes, se manifiesta durante la ejecución del programa (como ser objetos o librerías), consume recursos y contribuye con el comportamiento del sistema.

En la siguiente imagen se muestra el diagrama realizado. El diagrama muestra instancias, no tipos. Su realización contribuyo al desarrollo del obligatorio, ayudando a visualizar el camino de la implementación, permitiendo tomar decisiones tempranas. Se debe tener en cuenta que se trata de un diagrama primitivo, cuya semántica puede no estar perfectamente definida. En particular, esta vista ayuda a responder la pregunta de cuáles son las entidades más importantes durante la ejecución de la aplicación y cómo interactúan.




En el diagrama anterior es posible ver los principales componentes de la aplicación. Se observa el componente main, quien a través de las interfaces provistas por los componentes *Administración* y *Navegadores*, da lugar a la atención de peticiones.

También se observa que los componentes *Administración* y *Navegadores* hacen uso de los semáforos y de las utilidades provistas por *Utils*. De esta forma acceden a los recursos globales en forma sincronizada. Entre los recursos se encuentra la cache de datos donde se cachean los objetos transferidos entre el navegador y el servidor web, siempre y cuando no se supere el máximo tamaño de objeto cacheable ni la cantidad máxima de objetos en cache.

## 4.3 MAIN

Veamos un pseudocódigo del código principal *Main*,

```
//Definimos la memoria compartida por threads
unsigned int memoria_utilizada;
unsigned int requests_atendidos;
unsigned int objetos_cacheados_entregados;
unsigned int max_object_size;
unsigned int max_cached_object_size;
unsigned int max_object_count; // se multi excluye con cache_mutex
unsigned int object_count; // se multi excluye con cache_mutex
```

```
cache* Cache; // se multi excluye con cache_mutex

//Definimos los Semaforos utilizados por threads
pthread_mutex_t* memoria_utilizada_mutex;
pthread_mutex_t* objetos_cacheados_entregados_mutex;
pthread_mutex_t* requests_atendidos_mutex;
pthread_mutex_t* max_object_size_mutex;
pthread_mutex_t* max_cached_object_size_mutex;
pthread_mutex_t* cache_mutex;

…

int main(int argc, char** argv)
{

//Se crea el socket para escuchar a los admins
   admin_socket = socket(AF_INET, SOCK_STREAM, 0);
…
   printf("Creado el socket para escuchar a los administradores: %d\n", admin_socket);


//Se crea el socket para escuchar a los clientes de datos
   data_socket = socket(AF_INET, SOCK_STREAM, 0);
…
   printf("Creado el socket para escuchar a los clientes: %d\n", data_socket);


   //Bind del socket de administradores
   struct sockaddr_in admin_addr;
   socklen_t admin_addr_size = sizeof admin_addr;
   admin_addr.sin_family = AF_INET;
   admin_addr.sin_port = htons(ADMIN_PORT);
   admin_addr.sin_addr.s_addr = inet_addr(ip);
   if (bind(admin_socket, (struct sockaddr*) &admin_addr, admin_addr_size) < 0) {
…
   }
```

```
   //Bind del socket de datos
   struct sockaddr_in data_addr;
   socklen_t data_addr_size = sizeof data_addr;
   data_addr.sin_family = AF_INET;
   data_addr.sin_port = htons(puerto_datos);
   data_addr.sin_addr.s_addr = inet_addr(ip);
   if (bind(data_socket, (struct sockaddr*) &data_addr, data_addr_size) < 0) {
…
   }

   // Listen del admin_socket
   if (listen(admin_socket, MAX_QUEUE) < 0) {
…
   }

   // Listen del data_socket
   if (listen(data_socket, MAX_QUEUE) < 0) {
…
   }

//Creacion de hilo para atender a los administracion
   pthread_t thread_admin;
   if (pthread_create(&thread_admin, NULL, administracion, (void*) &admin_socket)
!= 0) {
…
   }

   //Creacion de hilo para atender navegadores
   pthread_t thread_nav;

   if (pthread_create(&thread_nav, NULL, clienteservidor, (void*) &data_socket) != 0) {
…
   }
```

Como es posible observar, sobre el final del pseudocódigo se crean dos hilos. El primero ejecuta la función *administracion* que es provista por el componente *Administración*. El segundo la función *clienteservidor* provista por el componente *Navegadores*. Estos hilos a su vez, se encargan de crear los hilos

correspondientes para cada nueva petición. Es necesario leer los siguientes capítulos para comprender completamente el esquema adoptado. A continuación se describe la funcionalidad principal del componente de *Administración* donde se atiende a los administradores del proxy.

## 4.4 ADMINISTRACIÓN

Los administradores se conectan a la aplicación utilizando el comando *telnet*, por ejemplo ejecutando en consola:

*telnet localhost 6666*.

6666 es el puerto por defecto en el que se atiende a los administradores aunque este parámetro es configurable al iniciar la aplicación. Una vez conectados, tienen la posibilidad de ejecutar los siguientes comandos:

- *show run*: muestra cantidad de memoria utilizada, cantidad de objetos en cache, cantidad de pedidos atendidos y cantidad de objetos cacheados entregados.
- *purge*: borra los objetos cacheados en memoria.
- *Set max_object_size XXX*: se setea el tamaño del mayor objeto transferible por el proxy. XXX es un número natural que representa el tamaño en KB.
- *set max_cached_object_size YYY*: se setea el tamaño del mayor objeto cacheable por el proxy. YYY es un número natural que representa el tamaño en KB.
- *set max_object_count ZZZ*: se setea la cantidad máxima de objetos a almacenar en la cache. (Por defecto el tamaño es de 200 objetos).
- *quit*: cerrar sesión.

El código inicial de administración se expresa en el siguiente pseudocódigo,

```
void* administracion(void* admin_socket) {


  por siempre {
    //primitiva ACCEPT
    struct sockaddr_in client_addr;
    socklen_t client_addr_size = sizeof client_addr;
    int socket_to_client = accept(
          *(int*) admin_socket,
```

```
            (struct sockaddr *) &client_addr, &client_addr_size
            );


  //Creación de hilo para atender request
      pthread_t thread_admin_consola;
      if (pthread_create(&thread_admin_consola, NULL, adminconsola, (void*)
&socket_to_client) != 0) {
…
      }

   }
}
```

Como mencionamos anteriormente, este hilo crea a su vez otros hilos para atender los nuevos pedidos. Se ejecuta la función *adminconsola* que posee la siguiente estructura,

```
void* adminconsola(void* socket_to_client)
{
   mientras no quit
   {

      //primitiva RECEIVE
      int received_data_size = recv(socket_to_cliente, data, MAX_MSG_SIZE, 0);
      if(received_data_size <= 0)
      {
         …
      }
      else
      {
         procesarComando(data,argc,argv))


         if comando show run
         {
                  show_run(socket_to_cliente);
         }
         else if comando purge
         {
                  purge(socket_to_cliente);
```

```
        }
        else if commando set max_object_size XXX(KB)
        {
                set_max_object_size(socket_to_cliente, XXX);
        }
        else if comando set max_cached_object_size YYY(KB)
        {
                set_max_cached_object_size(socket_to_cliente, YYY);
        }
        else if comando set max_object_count ZZZ
        {
                set_max_object_count(socket_to_cliente,ZZZ);
        }
        else if commando quit
        {
                quit = true;
        }
        else
        {
                comando_invalido = true;
        }
    }
  }
  //Se cierra el socket
  close(socket_to_cliente);
}
```

Aquí recibimos el comando ejecutado por el administrador utilizando la función *recv*, luego analizamos el comando determinando si es correcto, y finalmente lo ejecutamos.

No detallaremos aquí la implementación de las funciones para cada comando. Solo resta decir que en ellas se utilizan las primitivas *P* y *V* definidas en el componente semáforos permitiéndonos multiexcluir las variables declaradas al comienzo de la función main.

## 4.5 CLIENTE-SERVIDOR

Este modulo como se menciono antes, se ocupa de crear hilos para atender a los clientes navegadores con el procedimiento *data_cliente(void* params).* El código de las funciones *clienteservidor*, *continuarCliente* y *nuevoCliente* sigue las pautas y comentarios establecidos en (1).

A cada uno le asignará un número de cliente y se lo enviará. Podremos lanzar hasta 10 clientes simultáneamente y todos serán atendidos. Utilizaremos la función **select().** Si tenemos varios sockets abiertos (incluido el socket que recibe a los clientes) y disponemos de sus descriptores, podemos pasárselos a la función **select()**. Si así lo deseamos, nuestro código se quedará dormido hasta que en alguno de los descriptores haya datos disponibles (un nuevo cliente que entra o un cliente ya existente que nos envía un mensaje).

Los parámetros de la función **select()** son los siguientes:

- **int** con el valor del descriptor más alto que queremos tratar más uno. Cada vez que abrimos un fichero, socket o similar, se nos da un descriptor de fichero que es entero. Estos descriptores suelen tener valores consecutivos. El 0 suele estar reservado para la stdin, el 1 para la stdout, el 2 para la stderr y a partir del 3 se nos irán asignando cada vez que abramos algún "fichero". Aquí debemos dar el valor más alto del descriptor que queramos pasar a la función más uno.
- **fd_set *** es un puntero a los descriptores de los que nos interesa saber si hay algún dato disponible para leer o que queremos que se nos avise cuando lo haya. También se nos avisará cuando haya un nuevo cliente o cuando un cliente cierre la conexión.
- **fd_set *** es un puntero a los descriptores de los que nos interesa saber si podemos escribir en ellos sin peligro. Si en el otro lado han cerrado la conexión e intentamos escribir, se nos enviará una señal SIGPIPE que hará que nuestro programa se caiga (salvo que tratemos la señal). Para nuestro ejemplo no nos interesa.
- **fd_set *** es un puntero a los descriptores de los que nos interesa saber si ha ocurrido alguna excepción. Para nuestro ejemplo no nos interesa.
- **struct timeval *** es el tiempo que queremos esperar como máximo. Si pasamos **NULL**, nos quedaremos bloqueados en la llamada a **select()** hasta que suceda algo en alguno de los descriptores. Se puede poner un tiempo cero si únicamente queremos saber si hay algo en algún descriptor, sin quedarnos bloqueados.

Cuando la función retorna, nos cambia los contenidos de los **fd_set** para indicarnos qué descriptores de fichero tiene algo. Por ello es importante inicializarlos completamente antes de volver a llamar a la función **select()**.

Estos **fd_set** son unos punteros un poco raros. Para rellenarlos y ver su contenido tenemos una serie de macros:

- **FD_ZERO (fd_set *)** nos vacía el puntero, de forma que estamos indicando que no nos interesa ningún descriptor de fichero.
- **FD_SET (int, fd_set *)** mete el descriptor que le pasamos en int al puntero **fd_set**. De esta forma estamos indicando que tenemos interes en ese descriptor.
  Llamando primero a **FD_ZERO()** para inicializar el contenido del puntero y luego a **FD_SET()** tantas veces como descriptores tengamos, ya tenemos la variable dipuesta para llamar a **select()**.
- **FD_ISSET (int, fd_set *)** nos indica si ha habido algo en el descriptor int dentro de fd_set. Cuando **select()** sale, debemos ir interrogando a todos los descriptores uno por uno con esta macro.
- **FD_CLEAR (int, fd_set *)** elimina el descriptor dentro del **fd_set**.

En nuestro programa de ejemplo del servidor tendremos un descriptor del socket servidor y un array con 10 descriptores para clientes. Inicializaremos **fd_set** con un **FD_ZERO()**, luego le añadiremos el socket servidor y finalmente, con un bucle, los sockets clientes. Después llamaremos a la función **select()**. El código sería más o menos

```
void* clienteservidor(void* data_socket)
{

   int socketServidor = *(int*) data_socket;
   int socketCliente[NUMERO_CLIENTES]; /* Descriptores de sockets con clientes */
   int numeroClientes = 0; /* Número clientes conectados */
   fd_set descriptoresLectura; /* Descriptores de interes para select() */
   …
   por siempre {

/* Cuando un cliente cierre la conexión, se pondrá un -1 en su descriptor de socket
dentro del array socketCliente. La función compactaClaves() eliminará dichos -1 de la
tabla, haciéndola más pequeña. Se eliminan todos los clientes que hayan cerrado la
conexión */
```

```
        compactaClaves(socketCliente, &numeroClientes);

 /* Se inicializa descriptoresLectura */
        FD_ZERO(&descriptoresLectura);

/* Se añade para select() el socket servidor */
        FD_SET(socketServidor, &descriptoresLectura);

/* Se añaden para select() los sockets con los clientes ya conectados */
        for (i = 0; i < numeroClientes; i++)
           FD_SET(socketCliente[i], &descriptoresLectura);

/* Espera indefinida hasta que alguno de los descriptores tenga algo que decir: un nuevo
cliente o un cliente ya conectado que envía un mensaje */
        select(maximo + 1, &descriptoresLectura, NULL, NULL, NULL);
```

Como no tenemos interés en condiciones de escritura ni excepciones, pasamos **NULL** en el segundo y tercer parámetro. El último lo ponemos también a **NULL** puesto que no tenemos otra tarea que hacer hasta que alguien se conecte o nos envíe algo.

Cuando se salga del ***select()*** es porque: 1) se ha intentado conectar un nuevo cliente, 2) uno de los clientes ya conectados nos ha enviado un mensaje o bien 3) uno de los clientes ya conectados ha cerrado la conexión. En cualquiera de estas circunstancias, tenemos que hacer el tratamiento adecuado. La función ***select()*** sólo nos avisa de que algo ha pasado, pero no acepta automáticamente al nuevo cliente, no lee su mensaje ni cierra su socket.

Por ello, detrás del ***select()***, debemos verificar ***socketServidor*** para ver si hay un nuevo cliente y todos los ***socketCliente[]***, para ver si nos han enviado algo o cerrado el socket. El código, después del ***select()***, sería:

```
        /* Se comprueba si algún cliente ya conectado ha enviado algo */
        for (i = 0; i < numeroClientes; i++)
        {
           if (FD_ISSET(socketCliente[i], &descriptoresLectura))
           {
```

```
/* Se lee del socket. Si se lee un dato correctamente se crea un thread que ejecuta la
función data_cliente */
            if (continuarCliente(socketCliente[i], &numeroClientes) <= 0) {

 /* Hay un error en la lectura. Posiblemente el cliente ha cerrado la conexión. Scierra el
socket y se elimina del array de socketCliente[] */
                socketCliente[i] = -1;
            }
        }
    }
    /* Se comprueba si algún cliente nuevo desea conectarse y se le
     * admite */
    if (FD_ISSET(socketServidor, &descriptoresLectura))
        nuevoCliente(socketServidor, socketCliente, &numeroClientes);
  }
```

La función *continuarCliente* devuelve lo mismo que la función ***recv()***, es decir, el número de bytes leídos, 0 si se ha cerrado el socket o -1 si ha habido error. La función *nuevoCliente* se ejecuta cuando un nuevo cliente solicita conexión. Se acepta la conexión, se mete el descriptor en *socketCliente[]* y se sigue con la función *continuarCliente*.

La función *continuarCliente* luego de recibir en params el identificador del socket correspondiente al cliente y su request pasa a atender el pedido de la siguiente manera:

```
If (esGet(Request) || esPost(Request))

{

        If  (esGet(request)

        {

                Response=buscarEnCache(getURL(request));

//Variable definida por el admin

                If  (getTamaño(Response) > max_object_size)
                Denegado = true;
        }

        Else //Post
```

*If (getTamaño(Request) > max_object_size)*

*Denegado = true;*

*If (!denegado)*

*//Conectarse al servidor, mandar el pedido y esperar la respuesta*

*(1)*

*Else*

*Response = Error: El recurso es muy grande para su transferencia, denegado por el administrador*

*}*

*Else*

*Response = Error: Metodo no permitido*

*//Mandar respuesta al cliente*

*(2)*

En este pseudocódigo se puede observar la estructura principal del procedimiento. Se examina el pedido y si es un método permitido por nuestro proxy, en nuestro caso si es un método GET o POST, entonces se pasa procesar el pedido. Si no se retorna un response con el error de método no permitido.

Luego si es permitido y es un GET lo buscamos en la cache de la aplicación. Se busca el objeto con la url del request y si se encuentra se verifica que no se encuentre expirada esa entrada en cache. Cuando guardamos en cache una respuesta se examina su fecha de expiración, correspondiente al header de una respuesta HTTP *Expires*. Entonces al encontrarla se verifica que la fecha actual sea antes que la guardada. También se guarda el tamaño, y si este tamaño es mayor al permitido se deniega el pedido y no se lo pasa al cliente.

Al ser un método POST, lo único que hacemos es verificar que no exceda el tamaño permitido esta vez utilizando el header HTTP *Content-Length.* Al igual que con GET, si es más grande se deniega el pedido y el cliente recibe un mensaje de error correspondiente.

Como se puede ver (1) si el método no es denegado se pasa a conectarse con el servidor externo, mandarle el pedido y esperar al respuesta. A continuación se explica este caso:

```
//Se crea el socket del servidor externo

socket_to_server = socket(AF_INET, SOCK_STREAM, 0);

//Obtenemos la dirección del servidor, el addr y port se sacan de la URL, en res se guarda la dirección

getaddrinfo(addr, port, &hints, &res);

//Nos conectamos al servidor con la dirección antes obtenida

connect(socket_to_server, res->ai_addr, res->ai_addrlen);

//Antes de pasarle el pedido al servidor, lo modificamos, se explica más adelante

modificarRequest(Request);

//Se envía el request modificado

send(socket_to_server, data_request, msg_size, 0);

//Usando recv recibimos la respuesta del servidor, se explica más adelante

recv(socket_to_server, data_servidor, MAX_MSG_SIZE, 0);


if (enCache) //Si previamente estaba en cache

{

//Se examina la respuesta, si es una respuesta Not Modified entonces efectivamente se retorna la entrada de la cache. Si esta modificada, la entrada en cache antes encontrada no es válida, por lo tanto se borra

}

If (!enCache)

//No se encontraba en cache o fue modificada la pagina en el servidor

{
```

```
        //Se verifica el tamaño y si no se deniega
        If ( !denegado)
        {
                if (tamaño <= max_cached_object_size)
                {
                        If (object_count == max_object_count)
                                borrarUltimaEntradaCache(); //LRU
                        guardarEnCacheEntrada(Response);
                }
        }
}
(2) //Mandar Response al cliente
```

Luego de crear un socket para conectarse con el servidor externo y recibir los datos del cliente, se obtiene la dirección a partir de la URL y la función *getaddrinfo,* para poder conectarse al servidor. La conexión se realiza a través del socket creado especialmente para esto, *socket_to_server*, y pasamos como parámetros la dirección y el largo de la misma. Este procedimiento establecerá una conexión

TCP con el servidor externo y estaremos listos para comenzar a enviar el pedido.

Después de conectarse con el servidor modificamos el pedido realizado por el cliente web, la mayoría de estas modificaciones se deben a que utilizaremos HTTP 1.0. Cuando la respuesta está en cache debemos modificar la fecha del campo HTTP If-Modified-Since, para que tenga la fecha del pedido cuando se guardo en la cache. De esta forma el pedido GET se hace condicional. Más adelante se explica en detalles porque es que hacemos esta modificación.

A continuación le enviamos el pedido al servidor y esperamos la respuesta. Esta parte no es tan sencilla como se ve en el pseudocódigo. Al finalizar la ejecución del primer *recv* nos aseguramos que el servidor nos respondió y envió la

información, pero no podemos devolver esta información al cliente web hasta estar seguros de haber recibido todos los datos, puede haberse llenado el buffer ya que se le pasa un tope, en nuestro caso MAX_MSG_SIZE, por lo tanto tenemos que volver a ejecutar *recv* para recibir la siguiente parte. La manera en que se resolvió fue que se hagan todos los *recv* necesarios para traer toda la respuesta del servidor. Se utilizo un buffer auxiliar para ir guardando los pedazos de respuesta para luego copiarlos a la respuesta completa.

Cabe mencionar que si sabemos que el tamaño del recurso excede el máximo permitido entonces se deniega el pedido como se explico antes y no se sigue trayendo del servidor. Se implementaron dos controles para esto. En el primer *recv* si trae el header HTTP *Content-Length* y este es más grande que el permitido se cancela la operación. El otro control se hace en el bucle donde se va trayendo de a poco la respuesta. Con una variable se controla la cantidad de bytes traídos hasta el momento, si la misma excede el permitido se deniega y termina el pedido al servidor.

Un caso especial ocurre cuando la respuesta se encontraba en la cache. El pedido como se explico antes se modifica para transformarlo en un método condicional. La condición es que el recurso se trae solo si fue modificado. Entonces luego del primer *recv* se analiza la respuesta. Si esta indica que no fue modificada con el mensaje HTTP "304 Not Modified", entonces la entrada en cache es válida y no se sigue trayendo el resto de la respuesta si la hay como se menciono antes. En el caso que la respuesta indique que fue modificada, al entrada en cache expiro y se borra de la memoria, además que se sigue trayendo la respuesta nueva del servidor.

Luego si finalmente no se encuentra en cache, se realizan los controles de tamaños para el máximo permitido en la cache y se guarda en la misma la respuesta obtenida del servidor. Si se da el caso que se alcanzo la máxima cantidad de objetos en la cache antes se borra la última entrada en la misma, ya que se sigue la política LRU y en nuestro diseño la última entrada corresponde a la menos usada recientemente.

A continuación (2) se envía la respuesta encontrada al cliente web de la siguiente forma:

*send(socket_to_client, Response, Response_size, 0);*

*requests_atendidos++;*

*//Liberar memoria y cerrar sockets*

Como se puede observar se envía la respuesta al cliente web a través del socket que se obtuvo como parámetro de este proceso *socket_to_client*. Una vez realizado esto se aumenta la cantidad de pedidos atendidos y el procedimiento procede a cerrar todos los sockets creados y liberar la memoria utilizada para terminar correctamente.

# 5 CONCLUSIÓN

Hemos finalizado el informe sobre la realización del obligatorio número uno. Es importante mencionar los objetivos cumplidos.

El proceso hacia la formalización, ha tenido como propósito, según lo mencionado en ocasiones anteriores, ser coherentes en el desarrollo del informe, a diferencia de las ocasionales contradicciones que pueden encontrarse en piezas de información varias. Se pretendió fijar la atención en aspectos importantes y esenciales para la comprensión del desarrollo.

Podemos concluir que cumplimos nuestros objetivos cubriendo la totalidad de las cláusulas, implementando una aplicación con un fuerte uso de los conceptos aprendidos en el curso, logrando un mejor manejo del entorno de desarrollo y describiendo en la mejor forma posible la implementación logrando un desarrollo coherente y explicando las decisiones tomadas en cada sección del laboratorio.

Durante la preparación del trabajo los cambios y variantes, tanto de detalles como de perspectiva se hicieron numerosos y de múltiples estratos, sin embargo, en busca de una lectura confiable y coherente, se basaron las notas, observaciones y conclusiones en los textos bibliográficos.

Antes de la resolución del obligatorio, se intentó introducir al lector al aspecto teórico, y definir en forma precisa el vocabulario utilizado en etapas posteriores. Durante la explicación de la implementación se expusieron imágenes a modo de esclarecer la situación y agilizar la lectura. Se introdujo la arquitectura de la aplicación y se comentaron sus componentes principales. Se procedió a describir luego los componentes de *Administración* y *Navegadores*, donde se utilizaron los conceptos explicados anteriormente y donde surgieron decisiones de diseño especificas al grupo.

En todo momento se realizaron pruebas que verificaran el correcto comportamiento de la aplicación. Para dicho fin se utilizó la herramienta *Wireshark* desarrollada en el laboratorio anterior.

Debemos mencionar que las decisiones tomadas a lo largo de la implementación surgieron como respuesta a los resultados obtenidos hasta el momento.

# 6 BIBLIOGRAFÍA

1. [En línea] http://www.chuidiang.com/clinux/sockets/socketselect.php.

2. **Stalling, William.** *Comunicaciones y Redes de Computadoras.* Sexta edición.

3. **Kurose, James F. and Ross, Keith W.** *Computer Networking, A Top Down Approach.* Quinta Edición. 2010.

4. [En línea] Septiembre de 2012. http://opensylar.homelinux.net/exec/creahilos.pdf.

5. [En línea] Septiembre de 2012. http://www.rfc-editor.org/rfc/rfc2616.txt.

6. [En línea] Septiembre de 2012. http://www.rfc-editor.org/rfc/rfc1945.txt.

7. [En línea] Septiembre de 2012. http://www8.org/w8-papers/5c-protocols/key/key.html.

8. [En línea] Septiembre de 2012. http://es.wikipedia.org/wiki/Hypertext_Transfer_Protocol.

9. [En línea] Septiembre de 2012. http://www.linuxhowtos.org/manpages/2/recv.htm.

10. [En línea] Septiembre de 2012. http://alas.matf.bg.ac.rs/manuals/lspe/snode=55.html.

11. [En línea] Septiembre de 2012. http://www.eslinux.com/articulos/8591/programacion-sockets-lenguaje-c.

12. [En línea] Septiembre de 2012. http://es.kioskea.net/contents/utile/telnet.php3.

13. [En línea] Septiembre de 2012. http://www.wireshark.org/.

14. [En línea] Septiembre de 2012. http://www.xgc.com/manuals/xgclib/x5257.html.

15. [En línea] Septiembre de 2012. http://www.on-time.com/rtos-32-docs/rtip-32/reference-manual/socket-api/recv.htm.

16. [En línea] Septiembre de 2012. http://alas.matf.bg.ac.rs/manuals/lspe/snode=101.html.

17. [En línea] Septiembre de 2012. http://www.tuxmealux.net/nopaste/index.php/viewnopaste/9183102.Silent_http_proxy.

18. [En línea] Septiembre de 2012. http://www.jmarshall.com/easy/http/.