# Informe Obligatorio 1

## Redes de Computadoras 2012

### INCO – Facultad de Ingeniería – UdelaR

### Setiembre 2012

Grupo 59
Álvaro Acuña – CI: 3826062-8
Gabriel Centurión – CI: 2793486-8
Germán Mamberto – CI: 3187102-8
Fernando Mangino – CI: 3621009-1

# Índice

# Objetivo

Aplicar los conceptos teóricos de capa de aplicación, mediante el desarrollo de una aplicación que funciona en red.
Familiarizarse con el uso de las API de sockets y threads en C/C++.
Consolidar los conocimientos acerca del manejo de protocolos, mediante el estudio particular del protocolo HTTP.

# Descripción del Proxy

Se desea implementar una aplicación que funcione como proxy HTTP, actuando al mismo tiempo como cliente y servidor, que permita controlar el tráfico web mediante ciertas políticas definidas por los administradores de la red.

La aplicación deberá interceptar las peticiones HTTP realizadas por los clientes web y controlar lo que realizará con ellas, pudiendo tanto denegarlas como aceptarlas. En caso de rechazarla, deberá retornar el mensaje de error correspondiente con un código adecuado. En caso de admitirla, podrá realizar la petición al servidor correspondiente para luego devolverle el contenido adquirido al navegador.

El proxy también debe proveer un servicio proxy-cache, o sea, permitir el almacenamiento de los objetos obtenidos. De esta forma, al aceptar una petición de un objeto que ya contenía en su cache, puede optar por enviarlo directamente desde allí, logrando mayor velocidad de respuesta evitando la consulta y transferencia del objeto desde el servidor.
Se podrá conservar una cantidad máxima de objetos almacenados en memoria, que será configurable por cualquier administrador. En caso de alcanzarse ese máximo, se utilizará una política de reemplazo LRU (Least Recently Used) para los nuevos ingresos, donde se eliminará de la cache el objeto que hace más tiempo no se ha solicitado.

La solución debe manejar el protocolo de transporte confiable, orientado a conexión, denominado TCP.
Además, debe soportar solamente los métodos GET y POST, considerando inválidos cualquier otro método.
La versión del protocolo HTTP aceptada será la 1.0, sin embargo el proxy también debe contemplar peticiones realizadas con la versión 1.1 del protocolo. En estos casos, las conexiones se trataran como se definen en HTTP/1.0, o sea, un pedido/respuesta por conexión, y luego se cierra, no permitiéndose realizar más de un pedido en una misma conexión.

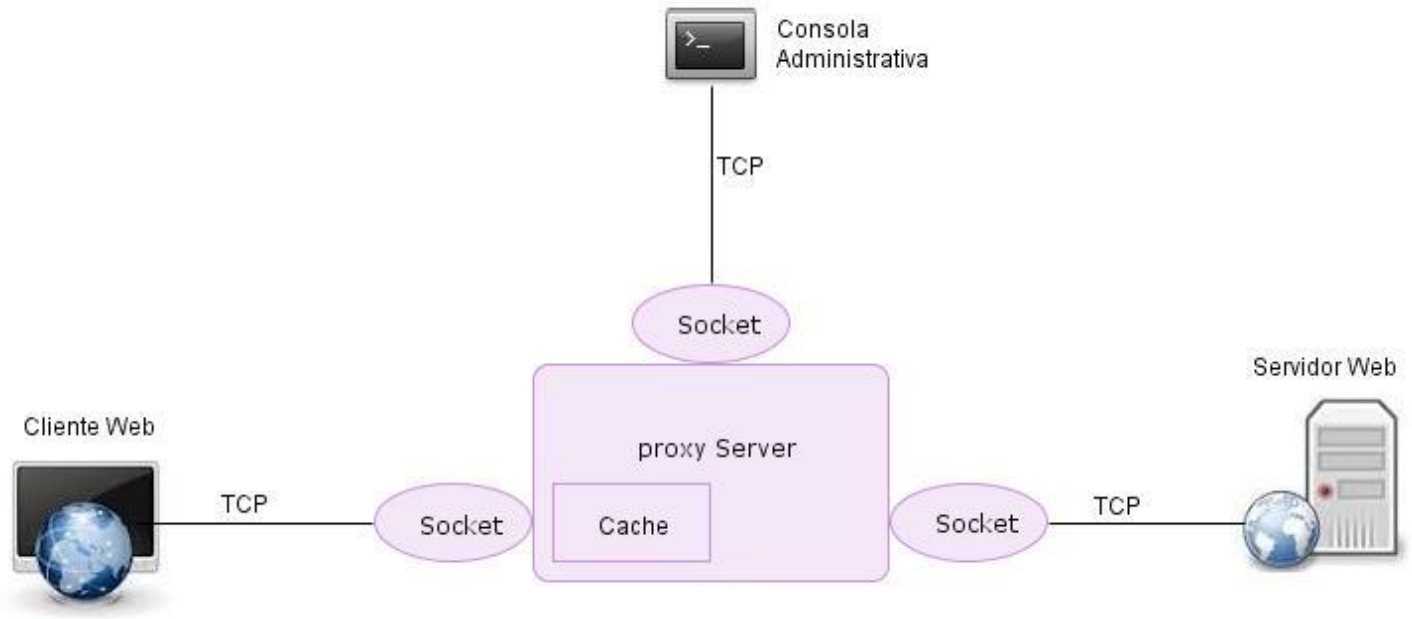
La aplicación debe actuar como servidor de dos tipos de usuarios: Clientes y Administradores.
Los clientes serán navegadores web y se conectaran en el puerto de datos 5555 por defecto. Los administradores serán terminales accedidas mediantes el comando telnet y se conectaran en el puerto de administración 6666 por defecto.
La dirección IP del proxy por defecto será la 127.0.0.1 (localhost).

Un administrador mediante el ingreso de comandos en consola podrá realizar diferentes acciones para controlar el funcionamiento del proxy, como borrar todos los objetos cacheados en memoria, configurar el mayor tamaño de objeto transferible, determinar el mayor tamaño de objeto cacheable o la cantidad máxima de objetos a almacenar. Cuando un objeto a transportar desde un servidor hacia el proxy tiene un tamaño superior al mayor tamaño de objeto transferible, se rechazará esa petición de objeto y no se realizará la transferencia, enviando al navegador el mensaje de error correspondiente.

La aplicación debe asignar un hilo de ejecución independiente por cada navegador o administrador que se conecta al mismo, funcionando así como un programa multithreaded. Por tal motivo se deben de mutoexcluir las estructuras a las que acceden dichos hilos para evitar conflictos en el acceso o funcionamientos indeseables.

# Solución Propuesta

El siguiente diagrama describe la solución propuesta.




En el centro del diagrama podemos apreciar la aplicación que desarrollamos llamada proxyServer la cual contiene a la memoria cache. Los clientes web se conectan a la aplicación proxyServer y esta verifica si tiene el response correspondiente al request. Si lo tiene lo retorna, sino lo busca en el servidor web, lo cachea y luego lo retorna al cliente web que lo solicito.

Para lo anterior verifica que se cumplan las condiciones seteadas por el administrador. Los administradores mediante telnet setean valores y consultan utilizando la consola administrativa.

# Implementación

Al inicio de la tarea no contábamos con la maquina virtual recomendada, por tal motivo decidimos para acercarnos a la arquitectura a utilizar instalarnos una maquina virtual con Ubuntu Linux. En dicha maquina virtual instalamos Eclipse con el plugin CDT y las Linux Tools que nos facilitaran las tareas de debug y control de código como Valgrind. Para el control de versiones instalamos Rabbit SVN que sería utilizado por fuera del Eclipse. El repositorio utilizado se encuentra alojado en Google Code.

En una primera instancia se procedió a investigar las distintas tecnologías a utilizar. Se procedió a repasar el lenguaje C/C++, estudiar el manejo de sockets, multi hilos y sockets con la utilización de select, manejo de hilos con la librería pthread, relación de pthread con select, manejo de handlers en pthread, pasaje de parámetros, etc. Por otro lado se designo un integrante para dedicarse exclusivamente a la investigación de los RFC's inherentes a la tarea y todo lo relativo al proxy-cache. Otro al armado del entorno de desarrollo y definición de generalidades tanto de la tarea como del desarrollo de la misma. Un tercer compañero dedicado al manejo de sockets, análisis de la comunicación solicitada, desarrollo de librerías al respecto y la solución de manejo de los mismos. Y el cuarto integrante dedicado a implementar las áreas que en un momento u otro fuese necesario, dando soporte a los otros tres.

Se definieron las librerías a realizar y se fue integrándolas a la solución global. Se realizaron casos de test que abarcaron dichas librerías para encapsular los temas e ir tratando los inconvenientes de la forma más aislada posible.

Como en un principio no parecía ser un código muy extenso intentamos realizar la implementación lo mas modular posible para poder dividirnos más fácilmente las tareas y poder realizar un testeo unitario de los módulos.

# Programa Principal

Como programa principal se implementó proxyServer que contiene la rutina de programa principal y las estructuras a utilizar.

El siguiente es un pseudocódigo de la rutina principal sin manejo de errores.

***Main***

```
Inicializo variables y estructuras

While(true){
        Inicializo FDSET
        Seteo socket administrador al FDSET
        Seteo socket cliente al FDSET
        Quedo a la espera en select(FD_SETSIZE, &fdSet, NULL, NULL, NULL)
        //Si se detecta petición de consola administrativa
        IF( FD_ISSET nos indica si ha habido algo en el descriptor int dentro de fd_set y corresponde al administrador)
                socketAdministradorAceptado <- AceptarConexionComoServidor(socket administrador)
                //Creo hilo con la rutina de atención a las consolas administrativas
                pthread_create(Hilo administrador, Handler administrador, parámetro de handler administrador
        socketAdministradorAceptado);
        //Si se detecta petición de navegador
        IF (FD_ISSET nos indica si ha habido algo en el descriptor int dentro de fd_set y corresponde al administrador )
                socketClienteAceptado <- AceptarConexionComoServidor(socket cliente);
                //Creo hilo con la rutina de atención del browser
                pthread_create(Hilo administrador Cliente, Handler cliente, parámetro de handler cliente
        socketClienteAceptado);}
```

***Handler Administrador*** sin manejo de errores ni mutuo exclusión

El handler del Administrador es la rutina encargada de atender la consola administrativa.
Cuando el administrador se conecta utilizando telnet al puerto indicado para tal fin se reconoce dicha entrada, se acepta la conexión y se crea un hilo para dicho administrador el cual es atendido por este handler.
El administrador ingresa el comando, si es reconocido se ejecuta y se realizan las indicaciones del caso.
Para cada comando hay una función que pide los parámetros necesarios y se encarga ya sea de setear las variables indicadas por parámetro, como modificar estructuras o armar la salida de datos solicitados.
Existe previamente un chequeo del parámetro el cual es case sensitive, o sea, no interpreta comandos que no sean exactos a los indicados en la letra de la tarea, además en cada caso verifica que tengan el valor correspondiente para los casos que los solicitan. El largo máximo de el valor aceptado es 99999999 (antes era 99999999 pero en backtrack no compila con ese valor y fue modificado).

```
adminHandler(socket Administrador){
        while(not salir){
                comando <- LeerDelSocket (socketAdmin)
                mensaje a Administrador <- ejecutarComando(comando, socket Administrador ,salir);
        }
        retorno mensaje a Administrador
}

ejecutarComando(comando, socketAdministrador, salir){
        switch (comando) {
          case show_run:
                showRun(socket, memoriaUtilizada, cantidadEntregados, cantidadRequest, max_object_size,laCache)
          case purge:
                ejecutarPurge(socket, laCache)
          case set_max_object_size:
                setMaxObjectSize(socket, max_object_size,getValor(comando))
          case set_max_cached_object_size:
                 setMaxCachedObjectSize(socket, max_cached_object_size,getValor(comando),laCache)
          case set_max_object_count:
                setMaxObjectCount(socket, max_object_count,getValor(comando),laCache)
          case quit:
          ejecutarQuit(socket, salir)
        }
}
```

***Handler Cliente*** sin manejo de errores ni mutuo exclusión

El handler del Cliente es la rutina encargada de atender las peticiones del web browser.
Cada web browser genera una petición por cada link, en principio por la URL indicada en la barra del navegador, luego por cada objeto que sea solicitado por la primera pagina como así las siguientes. A cada petición realizada por un navegador lo atiende un hilo de cliente distinto.

```
clienteHandler(socketCliente){
        request <-  LeerDelSocketServWeb(socketCliente)
        cantidadRequest ++ // aumento la variable de cantidad de request en 1
        response = consultarCache(request);
        if(response!=NULL ){
                cantidadEntregados ++// aumento la variable de cantidad de entregados en 1
                retorno response
        }else{ // no se encontró en la cache
                response <- buscarEnWeb(request, socketCliente) // busco en web
                cachear(request,response);
        }
        Retorno response
}
```

# Interfaces

## Administración del proxy (admin.h)

```
const uint64_t MAX_VALOR_SOPORTADO = 99999999;

int getComando(string comando);
// retorna el entero correspondiente al comando (lo codifica)

uint64_t getValor(string valor);
// retorna el valor del parámetro ingresado en el comando

// operaciones

string showRun(int socket,uint64_t memoriaUtilizada, uint64_t cantidadEntregados,uint64_t cantidadRequest,uint64_t
max_object_size, cache* &ptrCache);

void ejecutarPurge(int socket, cache* &ptrCache);

void setMaxObjectSize(int socket, uint64_t & max_object_size , uint64_t newValor);

void setMaxCachedObjectSize(int socket, uint64_t & max_object_size , uint64_t newValor, cache* &ptrCache);

void setMaxObjectCount(int socket, uint64_t & max_object_size , uint64_t newValor, cache* &ptrCache);

void ejecutarQuit(int socket, int & salir);
```

## Atención a los navegadores (cliente.h)

Esta interfaz se encarga de la atención a los navegadores mediante dos operaciones que se muestran a continuación:

```
#define BUFFER_LECTURA_CLIENTE 1500

const char* buscarEnWeb(char* request, int largoRequest ,int socketCliente, int max_object_size, int& bytesLeidos);
// realiza la búsqueda en la web para el request indicado , verifica el largo máximo de objeto

int mandarReponseDesdeCacheANavegador(char * resLecura, int largoResponse, int socketCliente);
// se encarga de enviar el response desde la cache al navegador
```

La operación, buscarEnWeb, se encarga de, en base a la request recibida desde el navegador, obtener la response correspondiente obtenida desde el servidor web indicado en la request.
La operación se puede dividir en varias partes:

La primera se encarga de extraer el nombre de host desde la request mediante la operación GetHost provista por la ParseLib, y luego obtener la IP a partir de dicho nombre de host utilizando la función getaddrinfo ().

Luego comienza la segunda etapa, donde poseemos los insumos necesarios para abrir un socket como clientes mediante la operación AperturaSocketComoCliente provista por la librería SocketLib, obteniendo el id del socket para la comunicación con el servidor web. Paso siguiente, se escribe la request en dicho socket mediante la operación EscribirEnSocketServWeb, de esta forma enviamos la petición al servidor web, para luego leer el response que el mismo nos envía.

La tercer etapa realiza la lectura del response que nos envía el servidor web mediante la operación LeerdelSocketServWeb provista por la librería SocketLib. Dicha lectura se realiza dentro de una iteración donde se indica que se continúe leyendo hasta que se cumpla una de las siguientes condiciones:

- Que no queden datos por leer.
- Que ocurra algún error inmanejable en la lectura.
- Que el tamaño del response que se está leyendo supere el tamaño máximo de objeto transferible (se pasa como parámetro) establecido por una directiva administrativa.

La cuarta etapa consiste en la escritura del response obtenido, la misma se realiza en el socket provisto para la comunicación con el navegador web (dicho id de socket se pasa como parámetro), siempre y cuando la lectura haya culminado por la condición que indica que se leyó la totalidad del response. La escritura se realiza en una iteración, al igual que la lectura, y continua hasta que se cumpla una de las siguientes condiciones:

- Se escribió la totalidad del response.
- Ocurrió algún error inmanejable en la escritura.

Si la escritura se realizó con éxito, es decir, que se escribió la totalidad del response, se retorna una copia del response leído, de forma que el mismo sea procesado por la librería manejadora de la Cache.

Como observaciones se puede acotar:

- Se realiza manejo de errores para los casos de flujos no exitosos, como por ejemplo la perdida de conexión o error general al momento de leer o escribir en el navegador o servidor web. A su vez se agrega control para el caso en que la lectura sea interrumpida debido a que el response que se está obteniendo supera la directiva de tamaño máximo de objeto transferible, dado el caso, se genera y envía al servidor web un response que denota un error 403 Forbidden indicando que el objeto a transferir es demasiado grande. Para el caso de error de conexión con el servidor, se genera y envía un response indicando error 502 Bad GateWay estableciendo que no se pudo realizar la conexión con el servidor así como la causa de ello.
- Otra ventaja de la implementación es que, al momento de leer el response desde el servidor web, el manejo de memoria que se realiza es dinámico, ya que el tamaño total del mismo es desconocido en este momento. El procedimiento que se realiza consiste en que, en cada iteración de la lectura se elimine la variable donde se va almacenando el response, antes copiándola a una variable auxiliar, para luego reservar más memoria incluyendo la cantidad de bytes leídos en la interacción en cuestión.

- Como observación final se puede destacar que la lectura del response desde el servidor web se hace independiente de la escritura de la misma hacia el navegador (se realizan en iteraciones separadas). Esto permite realizar la verificación de que el tamaño acumulado del response leído hasta el momento no supera la directiva de tamaño máximo de archivo transferible, todo esto sin necesidad de haber leído en su totalidad el response desde en servidor, ya que el tamaño del mismo podría superar ampliamente el tamaño determinado por la directiva administrativa. De esa forma se evita malgastar tiempo y recursos en una transferencia que resultara obsoleta.

La operación mandarResponseDesdeCacheANavegador se encarga de realizar un loop de escritura al igual que la función buscarEnWeb, pero con la finalidad de enviar el response almacenado en la Cache hacia el navegador.

# Sockets

La forma de comunicación entre procesos, en base al protocolo TCP/IP, se realizó mediante la programación de sockets, los cuales son básicamente el canal de comunicación entre dichos procesos.
La librería encargada de todo lo referente a la apertura de conexiones y comunicación entre los distintos nodos implicados es la socketLib, esta posee varias operaciones que provee a los distintos módulos de forma de llevar a cabo la comunicación, dicha interfaz se muestra a continuación:

## socketLib.h

```
int AperturaSocketComoServidor (int Puerto);
// abre un socket quedando en escucha de conexiones en el Puerto si ok retorna el id del socket , en caso de error retorna -1.

int AperturaSocketComoCliente (char* IPServidor, int Puerto);
//abre un socket de un servidor como cliente del mismo si ok, retorna id del socket abierto , en caso de error retorna -1.

int AceptarConexionComoServidor (int IDSocket);
// acepta la conexión de un cliente al socket indicado

int LeerDelSocketServWeb (int IDSocket, char* DatosALeer, int largoBuffer);
// lee del socket indicado ,la cantidad que sea posible hasta largoBuffer, carga lo leído en DatosALeer

int EscribirEnSocketServWeb (int IDSocket,const char*DatosAEscribir, int largoBuffer);
// escribe el contenido de DatosAEscribir al socket indicado ,la cantidad que sea posible hasta largoBuffer.
```

La primera operación, AperturaSocketComoServidor**,** es invocada desde el main principal, de forma de atender las solicitudes de los administradores y navegadores. Se encarga de la apertura de un socket, mediante el llamado a la función socket(), la cual retorna un descriptor de fichero, luego se llama a la función bind(), que se encarga de avisar al sistema operativo que se quiere abrir dicho socket y que debe asociarlo a nuestro programa, para ello se le indican la IP y puerto donde queremos atender las peticiones de conexión. Finalmente se invoca a la función listen de forma que efectivamente quedemos a la escucha de conexiones desde los clientes.

La segunda operación, AperturaSocketComoCliente, difiere de la primera dado el enfoque con el cual se quiere abrir un canal de comunicación, ya que la misma es invocada con un perfil de cliente. Dado esto la invocación de operaciones dentro de la misma también difiere. En primer lugar se ejecuta socket(),ya que como resultado obtendremos el descriptor para un socket, luego se invoca a la función connect(), la cual recibe como parámetros la IP y puerto del host con el cual se quiere establecer una conexión (en dicho host, se debe estar escuchando peticiones de conexión en dicho puerto). Como resultado de la invocación del connect() se obtiene el id de socket del servidor el cual oficiara de canal de comunicación con el mismo.

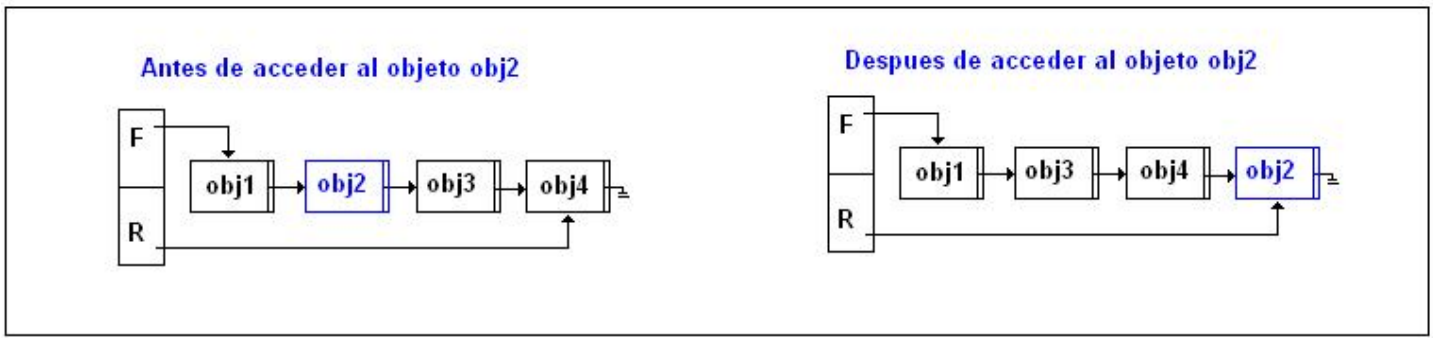
La tercera operación, AceptarConexionComoServidor, se encarga de aceptar las peticiones de conexión provenientes de los clientes, recibe como parámetro el id del socket con el cual los clientes desean establecer una conexión, luego invoca a la función accept() que realiza dicha acción, y tiene como retorno el id del socket creado con el fin de oficiar como canal de comunicación con el cliente.

Finalmente la cuarta y quinta operación se encargan de escribir y leer respectivamente en un determinado socket, cuyo id es recibido como parámetro. A su vez se reciben otros dos, uno de ellos indica la cadena de caracteres donde se quiere dejar lo leído del socket, o bien lo que se quiere escribir en el socket, y el otro indica la cantidad de bytes que se requiere que sean leídos o escritos. La lectura o escritura se realiza invocando a las funciones read y write respectivamente las cuales reciben dichos parámetros y realizan la lectura o escritura respectivamente.

Todas estas operaciones poseen manejo de errores para cada una de las invocaciones de las funciones importadas de la librería socket.h además de algunos internos propios, sumando a su vez un manejo de error particular al momento de leer datos de un socket, indicando que si hubo error al leer pero debido a que la conexión se encuentra ocupada temporalmente, se vuelva a intentar luego de un intervalo de tiempo.

# Cache

Considerando las restricciones de la letra de contar con una cantidad dinámica de objetos en la memoria cache y una política de reemplazo LRU, se opto por diseñar la cache como una lista encadenada con punteros al comienzo y al final de la misma. Cada vez que se accede a un objeto de la cache, el mismo es trasladado al final de la lista. Los ingresos de nuevos objetos se hacen al final de la lista (R) y las eliminaciones al comienzo (F) como se muestra en la siguiente figura:




De esta forma, el objeto que hace más tiempo no es accedido será el primero en la lista y será el candidato a ser eliminado en caso de reemplazo, cumpliéndose LRU.

Cada objeto dentro del cache estará identificado por su URL absoluta, la cual será confeccionada en caso de que llegue una petición HTTP/1.1 con URI relativa buscando el header “Host” dentro de la misma y concatenándolo al comienzo de esa URI. El objeto también tendrá la información de su tamaño, su fecha de expiración y la fecha de su última modificación en el servidor.
Cada vez que se busque un objeto en la cache, se contrastará su fecha de expiración contra la fecha actual y en caso de que el mismo haya expirado, se eliminará de la cache y se notificará que el objeto no se encuentra en memoria.

Para resolver lo lógica del caching se implementó un manejador del cache que expone dos operaciones que utilizara el programa principal del proxy, una que consulta si el objeto está en la cache (consultarCache) y otra que almacena en el cache un nuevo objeto (cachear).
Para determinar esta lógica nos basamos en la RFC 1945 y la 2616, que definen los protocolos HTTP/1.0 y HTTP/1.1 respectivamente.

Al momento de llegar una petición al proxy se llevan a cabo una serie de chequeos:

El primer chequeo que realiza el handler al recibir un http request es verificar que se trate de un método valido (GET o POST) y en caso de falla, se retorna una respuesta http cuyo código de estado es 405 (Method Not Allowed) y cuyo cuerpo se compone de una página HTML que notifica el error.

En caso de que el método sea válido, se chequea si la petición contiene el header “Pragma” con el valor “no-cache” y en caso afirmativo, no se consulta por el objeto en el cache sino que se redirige la petición al servidor web correspondiente para obtener dicho objeto. Luego de conseguirlo se realiza una actualización del cache, en caso de contener ese objeto, y por último se lo retorna al navegador. Este comportamiento se describe en la sección 10.12 RFC 1945 (referencia 1)

Luego, el tercer chequeo verifica si la petición contiene el header “Authorization” y en ese caso no se busca en la cache ya que este encabezado indica que se requiere una autenticación en el servidor y todo contenido obtenido de esta forma no es cacheable. Este criterio se especifica en la sección 10.2 RFC 1945 (referencia 2)

El siguiente chequeo es verificar que la petición se trate de un GET y no de un POST, ya que de ser este ultimo tampoco se buscara en el cache ya que los objetos obtenidos mediante POST no son cacheables, como lo indica la Sección 8.3 RFC 1945 (referencia 3)

En caso que pasen estos chequeos, se procederá a buscar el objeto en el cache.
Para ello lo primero será controlar si la petición contiene el header “If-Modified-Since“ y guardar su correspondiente valor (modifiedSince). En caso de que aparezca este encabezado estamos en presencia de un GET condicional en el cual el navegador solicita que se le envíe el objeto siempre y cuando haya sido modificado luego de la fecha que indica en ese header.
Para resolver esto, buscamos el objeto en el cache y en caso de encontrarlo consultamos su valor de última modificación (lastModified) y lo comparamos con la fecha que nos envió el navegador.
Si lastModified es menor a modifiedSince, el objeto ha sido modificado por lo que se lo retorna al navegador.
En el otro caso se devuelve una respuesta http con código de estado es 304 (Not Modified) al navegador y no se incluye el objeto.

A continuación se muestra el pseudo código del consultarCache:

```
consultarCache (request, tamanioResponse) {
    IF ( NOT isGet(request) AND NOT isPost(request) )
        return (response405)  // metodo no permitido
    ELSE IF ( contienePragmaNoCache(request)  OR  contieneAuthorization(request)  OR  esPost(request) )
        return (NULL)  // ya sé que no está en el cache
    ELSE
        IF  (contiene “If-Modified-Since“)// es un GET Conditional
            IF ( encontre(Objeto) )
                IF ( fueModificado(Objeto) )    return (Objeto)
                ELSE                            return (response304)
            ELSE    return (NULL)  // no está en el cache
        ELSE
            IF( encontre(Objeto) )    return (Objeto)
            ELSE                      return (NULL)  // no está en el cache
```

Al momento de guardar un objeto en el cache se llevan a cabo una serie de chequeos:

Si la petición contiene el header “Authorization” sabremos que se solicitó una autenticación en el servidor por lo cual no se cachea. Este comportamiento se define en la sección 10.2 RFC 1945 (referencia 2).

Si el response contiene un código de estado diferente al estado 200 o si contiene el header “Pragma” con el valor “no-cache” tampoco se guardara en el cache.

Si el objeto llega sin fecha de expiración, indicada en el header “Expires”, consideramos que el objeto no expira nunca y en este caso lo guardaremos indefinidamente hasta que se realice un purge de la memoria. Y cuando llega con esta fecha, se chequea si es válida y se compara con la fecha que llega en el header Date. De ser menor o igual, se considera que el objeto vence inmediatamente y no se cachea. De ser mayor, se compara con la fecha actual y en caso de ser mayor tampoco se cachea. Para definir este comportamiento nos basamos en la sección 10.7 Expires RFC 1945 (referencia 4).

También antes de guardar un objeto se chequea que el tamaño del mismo no sea mayor al tamaño máximo permitido.

Si el objeto supera todos los chequeos, antes de efectivamente guardarlo en memoria, se debe obtener la fecha de su última modificación que llega con el header “Last-Modified”. En caso de que la respuesta no contenga esta información, se optó por tomar la fecha actual del sistema como fecha de última modificación, considerando que el servidor podría haber modificado al objeto el instante antes a enviárnoslo.

*cachear* (request, response, tamanioResponse) {

IF ( contieneAuthorization(request) OR
fecha de expiracion invalida OR
fecha de expiracion <= fecha Date OR
fecha de expiracion <= fecha Actual OR
tamaño del objeto > tamaño máximo permitido OR
(código de estado (response) != OK) OR
(contienePragmaNoCache(response)) )
**NO CACHEAR**

ELSE
IF ( es HTTP/1.1(response) )
cambioResponseHttp11A10(response);
URL = obtener URL (request);
IF (contiene fecha de última modificación)
lastModified = fecha de última modificación;
ELSE (contiene fecha Date)
lastModified = fecha de última modificación;
ELSE
lastModified = fecha actual;
IF ( contienePragmaNoCache(request) )
**SI EXISTE EL OBJETO SE ACTUALIZA, DE LO CONTRARIO SE INGRESA.**
ELSE
**INGRESAR OBJETO EN CACHE**

# Concurrencia e hilos

Para el manejo de hilos y su concurrencia se utiliza la librería threads. Dicha librería nos provee de las operaciones pthread_create, pthread_mutex_lock, pthread_mutex_unlock entre otras. Las operaciones mencionadas son las utilizadas por nuestra aplicación y pasaremos a explicarlas.

pthread_create:
En nuestra implementación toma la variable que hace referencia al hilo donde nos indica el id del hilo creado, el handler que ejecutará ese hilo y el parámetro del handler a ejecutar.

pthread_mutex_lock:
Indicándole el mutex definido para el propósito especifico, nos permite que los demás hilos accedan al código por debajo de este. Encolando a los hilos hasta que el mutex se libere. Actúa como un semáforo binario.

pthread_mutex_unlock:
El primer hilo que tomo el mutex, al momento de salir del código mutuo excluido debe liberarlo, para tal motivo se utiliza esta operación indicándole el mutex bloqueado.

# Bibliotecas

Si implementaron distintas bibliotecas de soporte para los módulos de atención a los navegadores y a los administradores.

## ParserLib

Se implemento una librería especializada en el parseo de strings para poder obtener consultar los headers que contienen los mensajes http y poder modificar los request y/o response en caso que sea necesario.
La implementación se baso fuertemente en el manejo de expresiones regulares para lo cual utilizamos la librería regex.h de C. También se implemento una función que modifica el encabezado de la respuesta de un servidor web en caso de que nos llegue en 1.1, lo cambiamos a 1.0.

```
bool  isPrefix(const char * pref, const char * cadena) ;
bool  isAuthorization(const char * cadena) ;
bool  isPragmaNoCache(const char * cadena) ;
bool  isExpires(const char * cadena) ;
bool  isGet(const char * cadena) ;
bool  isPost(const char * cadena) ;
bool  isHttp10 (const char * cadena) ;
bool  isProtocol(const char * cadena) ;
char* getContentLength (const char * cadena) ;
char* getHeader (const char * cadena);
char* getBody (const char * cadena);
char* getHost (const char * cadena);
char* getUrl (const char * cadena);
char* getStateCode(char * cadena);
int   getModifiedSince(const char * cadena) ;
int   getLastModified(const char * cadena);
int   getDate(const char * cadena) ;
int   getExpires(const char * cadena) ;
void  cambioResponseHttp11A10 (char* cadena);
char* regexp (const char *string, const char *patrn);
```

## stringLib

Como soporte de ayuda al manejo de strings también se realizó la librería stringLib que lo utilizamos para manejar los comando ingresados por los administradores y las conversiones de string a entero.

```
vector<string> split (const string &s, char delim);
string  intToStr (int number);
int  strToint (string strNumber);
uint64_t  strTouInt64_t (string strNumber);
string  int64_tToStr (uint64_t number);
```

## Cache

Como se menciono anteriormente para el manejo de la memoria cache se implemento un modulo aparte (cache) y su handler correspondiente (cacheHandler).

### cache.h

```
cache* initCache();
// Inicializa el cache

char* getIdObj(nodoObjeto* nObj);
// Retorna una copia del id del objeto
// Si nodoObjeto == NULL, retorna NULL

char* getResponse(nodoObjeto* nObj);
// Retorna una copia del mensaje http: encabezado + objeto
// Si nodoObjeto == NULL, retorna NULL

int getTamanioObj(nodoObjeto* nObj);
// Retorna el tamaño del mensaje http (encabezado + objeto)
// Si nodoObjeto == NULL, retorna -1

int getVencimientoObj(nodoObjeto* nObj);
// Retorna la fecha de expiracion de objeto
// Si el objeto no expira, se retorna 0
// Si nodoObjeto == NULL, retorna -1

int getUltimaModificacionObj(nodoObjeto* nObj);
// Retorna la fecha de la última vez que el objeto fue modificado
// Si nodoObjeto == NULL, retorna -1

int getCapacidadCache (cache* c);
// Retorna la cantidad máxima de objetos posibles en el cache
// Precondición: El cache debe estar inicializado.

int getCantidadObjCache (cache * c);
// Retorna la cantidad de objetos almacenados en el cache
// Precondición: El cache debe estar inicializado.

int getTamanioPermitidoMaxCache (cache * c);
// Retorna el mayor tamaño de objeto cacheable en KB
// Precondición: El cache debe estar inicializado.

int getHitsCache (cache * c);
// Retorna la cantidad de objetos cacheados entregados
// Precondición: El cache debe estar inicializado

void setTamanioPermitidoMaxCache (int tam, cache * c);
// Setea el mayor tamaño de objeto cacheable
// Precondición: El cache debe estar inicializado.

int setCapacidadCache (int cap, cache * c);
// Setea la cantidad máxima de objetos posibles en el cache
// Si (getCantidadObjCache(c)>cap) se retorna -1 y no se setea dicho valor.
// Si (getCantidadObjCache(c)<=cap) se retorna 0 y se setea la capacidad del cache
// Precondición: El cache debe estar inicializado.
```

```
int esVacioCache (cache * c);
// Retorna true si c no contiene ningún objeto almacenado
// Precondición: El cache debe estar inicializado.

int esLlenoCache (cache * c);
// Retorna true si el cache contiene la máxima cantidad de objetos posibles
// Precondición: El cache debe estar inicializado.

int insertarObjetoCache (const char* ID, const char* obj, int tam, int expires, int lastModified, cache * &c);
// Si (getTamanioPermitidoMaxCache(c) < tam) se retorna -1 y no se permite insertar el nodoObjeto.
// Si (getTamanioPermitidoMaxCache(c) >= tam) se retorna 0 y se inserta el nodoObjeto al final de la cola.
// El nodoObjeto se identifica por 'URL', su mensaje http contiene el objeto "obj" y su tamaño es 'tam'.
// La fecha de expiración del objeto es "expires" y su fecha de última modificación es "lastModified".
// Precondición: El cache debe estar inicializado.

int actualizarObjetoCache (const char* URL, const char* obj, int tam, int expires, int lastModified, cache * &c);
// Se busca el nodoObjeto identificado por 'URL'
// Si no existe, se retorna -1 y no se actualizan sus datos.
// Si existe, se retorna 0 y se actualizan sus datos.
// Precondición: El cache debe estar inicializado.

nodoObjeto* buscarObjetoCache (const char* URL, int ahora, cache * &c);
// Busca el nodoObjeto identificado por URL en el cache
// Si lo encuentra y está vencido, lo elimina del cache y retorna NULL
// Si lo encuentra y no venció, lo mueve al final de la cola de nodoObjetos y retorna una copia del mismo
// Si no lo encuentra, se retorna NULL
// Precondición: El cache debe estar inicializado

void eliminarPrimeroCache (cache * &c);
// Elimina el primer nodoObjeto de la cola de nodoObjeto y libera la memoria.
// Si no hay nodoObjetos en la cache, la operación no tiene efecto.
// Precondición: El cache debe estar inicializado.

void destruirCache (cache * &c);
// Libera toda la memoria reservada para el cache deshaciendo la inicialización.
// Para volver a utilizar el cache se debe volver a ejecutar initCache()

void showCache (cache * c);
// Muestra información del cache.
```

## cacheHandler.h

```
void initCacheHandler(cache* &laCache);
// Inicializa el  cachea

int cachear(char* request, char* response, int tamanioResponseBytes);
// Si corresponde se cachea el objeto y se retorna 1
// Si no corresponde no se cachea y se retorna 0

char* consultarCache(char* request, int & tamanioResponseBytes);
// Consulta si se encuentra un objeto en el cache
// En caso que sea un GET Condicional, se retorna el objeto o se retorna el mensaje "304 Not Modified".
// En caso que sea un GET y se encuentre el objeto, se lo retorna.
// En otro caso se retorna NULL
```

## timeLib

Se implemento una librería especializada en el parseo de strings para poder obtener consultar los headers que contienen los mensajes http y poder modificar los request y/o response en caso que sea necesario.
La implementación se baso fuertemente en el manejo de expresiones regulares para lo cual utilizamos la librería regex.h de C. También se implemento una función que modifica el encabezado de la respuesta de un servidor web en caso de que nos llegue en 1.1, lo cambiamos a 1.0.

```
int stringToTime (const char *date);
// Convierte una fecha en formato string a un entero que representa la cantidad de segundos desde el 1 enero de 1970.
// Soporta los 3 formatos de dates especificados en el RFC 1945:
// 1: Sun, 06 Nov 1994 08:49:37 GMT     ; RFC 822, updated by RFC 1123
// 2: Sunday, 06-Nov-94 08:49:37 GMT    ; RFC 850, obsoleted by RFC 1036
// 3: Sun Nov  6 08:49:37 1994          ; ANSI C's asctime() format


int getToday();
// Retorna la fecha actual como un entero que representa la cantidad de segundos desde el 1 enero de 1970.
```

## logger

Esta biblioteca se utiliza para depuración del programa.

```
void log(string msg, int logLevelMsg);
// imprime el mensaje si logLevelMsg <= logLevel

void setLogLevel(int newLogLevel);
// setea el nivel de logueo de los mensajes
```

# Manual de ejecución

Se implemento una librería especializada en el parseo de strings para poder obtener consultar los headers que contienen los mensajes http y poder modificar los request y/o response en caso que sea necesario.

Todos los archivos necesarios para ejecutar deben encontrarse en el mismo directorio (lab1) y junto con ellos se construyó un archivo makefile que permite la compilación automática de la aplicación,
Ejecutando make clean, se elimina cualquier objeto que pueda haber compilado (archivos objetos y ejecutables).
Ejecutando make, se compila el proyecto entero y se genera el archivo ejecutable proxyServer.

Se contará con 3 formas posibles de ejecución:

**./proxyServer** - donde se toman por defecto la IP 127.0.0.1 y el puerto de datos 5555
**./proxyServer IP** - donde se toman por defecto el puerto de datos 5555
**./proxyServer IP puerto** - donde se puede setear la IP y el puerto de datos elegidos.

También es importante configurar el navegador web para que se conecte a internet a través de la IP y puerto elegidos.

En cuanto al administrador, el mismo se debe conectar por telnet a la dirección ip correspondiente y puerto 6666.
Los comandos brindados son los siguientes 6:

- **quit** : cierra la ejecución de la consola administrativa.
- **purge** : elimina los datos cacheados.
- **show run**: muestra los valores actuales y datos estadísticos como memoria total utilizada, cantidad de memoria en KB de objetos cacheados, cantidad de objetos cacheados, cantidad de request atendidos y cantidad de hits en cache.
- **set max_object_size XXX** : determina el tamaño del mayor objeto transferible en KB, tomando por defecto 10 MB.
- **set max_cached_object_size YYY :** determina el mayor tamaño de objeto a cachear, tomando por defecto 100 KB.
- **set max_object_count ZZZ** : determina la cantidad máxima de objetos a cachear, tomando por defecto 200.

# Dificultades

- Realizar la tarea en c++ llevo a tener que dedicar tiempo a sus particularidades como son el manejo de memoria y punteros. Varios bugs de la tarea se encontraban en esas aéreas que fueron resueltas usando el Valgrind para detectar Memory leaks. Entendemos que si la tarea la realizábamos en un lenguaje donde nos sintiéramos más cómodos y capacitados podríamos haber ahorrando algunos días de trabajo.

- El almacenamiento y la concatenación del response desde el servidor web fue lo que nos tomó más tiempo, tuvimos que realizar varios prototipos y testear varias soluciones. Dado que no solamente se trata de caracteres sino también de archivos binarios como imágenes o archivos comprimidos gzip no pudimos usar las funciones habituales para manejo de caracteres como concatenar y copiar (strcpy). En su lugar tuvimos que usar memcpy y memmove que trabajan con datos binarios. De esta forma las transferencias y guardado en la cache se hace en forma binaria.

- Nos insumió tiempo tomar decisiones de diseño por falta de detalle en las especificaciones, aunque leímos los rfc y documentos afines algunas informaciones poseían ambigüedades o poco detalle por lo que en estos casos tomamos decisiones arbitrarias previamente consultadas en monitoreo.

# Posibles Mejoras

- Una posible mejora podría ser el cambio de encabezados de peticiones http 1.1 a http1.0.Por falta de tiempo nos quedo pendiente la eliminación de los headers exclusivos de http 1.1 para machear con la compatibilidad de http 1.0. Por lo que en nuestra solución se envían los headers de http1.1 indicando que es http 1.0 sin quitar dichos headers.

- Otra posible mejora que evaluamos tiene que ver con que en varias áreas del sistema utilizamos int en vez de uint64_t lo cual podría llegar a limitarnos en algún caso con alto consumo. Si bien se realizó refactoring al respecto en varios puntos críticos, queda por refactorizar algunos casos no tan críticos.

- Test de stress. Sería conveniente realizar tests de stress que además de verificar los limites de nuestro sistema nos permita mediante cambios en las variables del sistema, como ser tamaños de buffers de lectura y escritura, etc., evaluar la performance del mismo y así encontrar la relación de dichas variables que sea más conveniente. Si bien utilizamos Valgrind como herramienta para verificar nuestro ejecutable, mediante una prueba de estress se podría determinar si hay memoria colgada para refinar algún detalle al respecto.

# Referencias

**1- Sección 10.12 RFC 1945**

When the "no-cache" directive is present in a request message, an application should forward the request toward the origin server even if it has a cached copy of what is being requested. This allows a client to insist upon receiving an authoritative response to its request. It also allows a client to refresh a cached copy which is known to be corrupted or stale.

**2- Sección 10.2 RFC 1945**

Responses to requests containing an Authorization field are not cachable.

**3- Sección 8.3 RFC 1945**

Applications must not cache responses to a POST request because the application has no way of knowing that the server would return an equivalent response on some future request.

**4- Sección 10.7 Expires RFC 1945**

If the date given is equal to or earlier than the value of the Date header, the recipient must not cache the enclosed entity.

**5- Referencias para TimeLib:**

http://pubs.opengroup.org/onlinepubs/007904975/functions/strptime.html

**6-Referencias para tema RegEx:**

http://www.zytrax.com/tech/web/regex.htm explica sobre la librería que uso

**7-Referencias usadas para programación de Sockets:**

Guía de sockets en C: http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html
Diferentes API de sockets: http://stackoverflow.com/questions/4199185/socket-api-or-library-for-c
Guía en español: http://www.chuidiang.com/clinux/sockets/sockets_simp.php
Ejemplo de pthread con Sockets :
http://cboard.cprogramming.com/c-programming/113221-pthreads-sockets.html
Otro ejemplo:http://www.cplusplus.com/forum/beginner/52630/