Desarrollo de un inyector de carga
Práctica 1

1 Objetivo

Esta práctica tiene como objetivo que el alumno aprenda a hacer inyectores de carga y, al mismo tiempo, se familiarice con los conceptos explicados en el tema *La carga de trabajo de un sistema*. Además, el trabajo desarrollado servirá como base a futuras prácticas.

2 Descripción

2.1 Esquema general

Al alumno se le plantea un problema de evaluación de prestaciones de un sistema transaccional (ver Fig. 1). El sistema está compuesto por un servidor de transacciones y un conjunto de clientes usuarios que solicitan servicios al servidor. El servidor será un PC, con sistema operativo Windows Server 2012, y ejecutando una aplicación servidora que realiza un tipo de transacción. Esta aplicación será proporcionada al alumno y se explica en el siguiente punto.

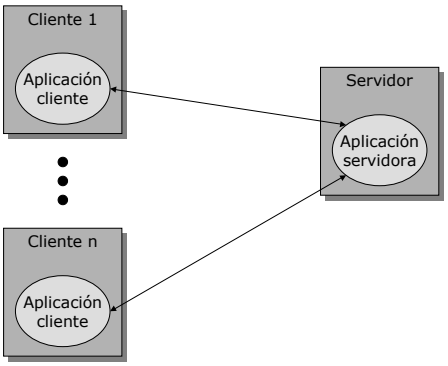

Fig. 1 Esquema del sistema a evaluar

Para realizar el estudio se ha decidido utilizar la técnica de medición. Ante la imposibilidad de disponer de un RTE (Emulador de Terminales Remotas) el alumno debe desarrollar en esta práctica un inyector de carga que permita suplir la función del RTE. El inyector a desarrollar emulará clientes realizando peticiones al servidor según un tiempo de reflexión que será configurable. En el apartado 2.3 se explica cómo debe funcionar este inyector.

2.2 Descripción del servidor

Desde la página web de la asignatura, el alumno dispondrá de un ejecutable llamado *servidor_sces* (servidor de CES) que será la aplicación servidora que ejecuta el servidor a evaluar. Para lanzar el servidor, se invocará desde una ventana de comandos y habrá que pasarle estos 7 parámetros en el orden con el que se explican a continuación:

- Consumo de CPU: entero que configura la cantidad de instrucciones ejecutadas como respuesta a cada petición.
- Consumo de lectura de disco: entero que configura la cantidad de datos leídos.
- Consumo de escritura de disco: entero que configura la cantidad de datos escritos.
- Consumo de memoria: entero que configura la cantidad de memoria utilizada.

- Número de visitas: entero que configura el número de visitas a cada componente del sistema por cada petición.
- Número de hilos: entero que configura el número de hilos servidores que se lanzarán de la aplicación. El número de clientes conectados al servidor no puede ser mayor que este número.
- Directorio base: cadena que representa el directorio base en donde se encuentran los directorios *Lect* (donde estarán los ficheros a leer) y *Escr* (donde se escribirá). Si el programa *servidor_sces* se invoca desde el mismo directorio donde están los directorios de escritura y lectura (*Lect* y *Escr*) se pondrá el carácter (.).

Además, se puede pasar un último parámetro opcional con valor **–v** para que el servidor muestre información de las peticiones que va sirviendo. **Este parámetro sólo se debe usar en fase de depuración y NUNCA en las prácticas posteriores que requieran realizar mediciones.**

Cuando se ejecute *servidor_sces* se crearán tantos hilos como indique el parámetro correspondiente. Estos hilos se quedarán escuchando en un socket TCP. El primer hilo escuchará en el puerto 57000, el segundo en el 57001 y así sucesivamente. Las peticiones al servidor deben hacerse a través del envío de una cadena 1250 bytes a este puerto (el contenido de la cadena es indiferente). Como respuesta a esta petición, el servidor realizará un servicio que consumirá recursos según los valores configurados. Cuando finalice el servicio, el servidor enviará a través de la conexión una respuesta al cliente, que será de nuevo una cadena de 1250 bytes.

El servidor necesita disponer de unos ficheros de lectura que contendrán datos que lee para poder elaborar la respuesta. Estos ficheros deben cumplir una serie de requisitos. Para poder generarlos se proporciona en el directorio de la asignatura la aplicación *CreaFich* que recibe los siguientes parámetros:

- Directorio: Lugar donde se crearán los ficheros. Para que el servidor funcione correctamente, este directorio debe ser *Directorio base\Lect*, es decir, un subdirectorio llamado *Lect* dentro del directorio base.
- Número de ficheros a crear: Para que el servidor funcione correctamente este número debe ser 2000.
- Tamaño de cada fichero: Para que el servidor funcione correctamente este número debe ser 2048.

En la partición de Windows Server 2012 de las máquinas del aula de prácticas se crearán, si no estuvieran creados ya, por este procedimiento los ficheros en carpeta C*:\trabajo\Lect*. La herramienta *CreaFich* se proporciona también para el caso de que los alumnos deseen hacer pruebas de desarrollo en otras máquinas.

Al atender peticiones el servidor va creando ficheros en el directorio *Directorio base\Escr*. **La versión actual del servidor realiza el borrado de los archivos, pero la carpeta debe existir. En caso de que hubiera ficheros en esta carpeta podrían llegar a ocupar mucho espacio, por lo que se recomienda borrarlos tras cada batería de pruebas**. Es conveniente hacerlo desde el interfaz de comandos (con *del* C*:\trabajo\Escr*.**, por ejemplo) ya que desde el explorador de Windows tarda mucho más.

Para finalizar la ejecución del servidor se debe pulsar Ctrl-C.

2.3 Descripción del inyector

El inyector que el alumno debe desarrollar en esta práctica deberá simular varios clientes realizando peticiones al servidor. Para ello deberá desarrollar en Visual C un programa mulitihilo. Cada hilo emulará a un cliente que tendrá el siguiente *pseudocódigo*:

```
Para i = 0 hasta NumPeticiones hacer
  Conectarse al servidor;
  Enviar una cadena de petición (1250 bytes);
  Esperar por la respuesta del servidor (1250 bytes);
  Cerrar la conexión;
  Dormirse para simular el tiempo de reflexión;
fPara
```

Cada cliente calculará el puerto del servidor al que debe conectarse como *57000 + número de hilo*.

Dormirse representa el tiempo que el usuario "reflexiona" antes de lanzar una nueva petición, es por tanto un tiempo de espera. Para dormir el hilo se utilizará la función *Sleep* del API de Win32. Esta función recibe un valor de tipo *unsigned int* que representa el tiempo expresado en **milisegundos** que el hilo debe permanecer inactivo.

Tanto el número de peticiones a realizar como el tiempo de reflexión deben ser parámetros del inyector. El parámetro *tiempo de reflexión* representará el valor medio de una distribución exponencial para generar el tiempo de reflexión tras cada petición.

En el apartado 4 de esta práctica se explican los conceptos relativos a hilos, números aleatorios y sockets que son necesarios para desarrollar el inyector.

2.4 Pruebas a realizar

El correcto desarrollo del inyector es fundamental. Un error en él traerá como consecuencia que las mediciones que se hagan posteriormente no sean válidas, lo que a su vez invalida los modelos que se desarrollen. Por esta razón conviene probar exhaustivamente el inyector. Como de momento no se medirán tiempos con objeto de obtener las prestaciones del servidor, se pueden realizar las pruebas con el servidor y el cliente en la misma máquina.

Deben probarse dos aspectos con especial atención: la generación de números aleatorios y la realización del número de peticiones correcto. Para ello se proponen las siguientes pruebas:

Prueba 1: Lanzar el servidor con los siguientes parámetros:

120000 40 15 300 5 50 C:\trabajo.

Realizar una prueba con 50 clientes, cada uno de ellos haciendo 10 peticiones y con un tiempo de reflexión de media 1 segundo.

Prueba 2: Igual a la anterior pero los clientes harán 100 peticiones cada uno.

Prueba 3: Igual a la *prueba 1* pero con tiempo de reflexión 5 segundos.

Prueba 4: Igual a la *prueba 2* pero con tiempo de reflexión 5 segundos.

En las pruebas se debe comprobar que la media de los tiempos de reflexión y el número de peticiones realizado por cada hilo son correctos. Se exige como requisito que se vuelquen estos datos a un fichero para que quede registro y poder comprobar con una hoja de cálculo la corrección de los datos. Para llevar esto a cabo, se recomienda utilizar un vector global que tenga una posición para cada hilo cliente. En cada posición habrá dos campos:

- Un contador del número de peticiones que lleva realizadas el hilo.
- Un vector que en cada posición guardará el tiempo de reflexión tras cada una de las peticiones realizadas por el hilo.

El hilo principal, tras la finalización de todos los hilos cliente, volcará el contenido del vector a disco en un formato adecuado para la hoja de cálculo Excel (con los campos separados por tabulaciones u otro carácter adecuado: coma, punto y coma, etc).

3 Material a entregar

Se debe entregar una pequeña memoria que contenga:

- Esta tabla rellenada:

Prueba	Concepto	Valor esperado	Valor obtenido
1	Media del tiempo de reflexión		
	Número de peticiones por hilo		
2	Media del tiempo de reflexión		
	Número de peticiones por hilo		
3	Media del tiempo de reflexión		
	Número de peticiones por hilo		
4	Media del tiempo de reflexión		
	Número de peticiones por hilo		

- El código fuente (fichero *.cpp*) del inyector. Es obligatorio que la letra del código fuente sea monoespaciada (por ejemplo, *Courier new*) y tenga un tamaño conveniente para que entren las líneas de 80 caracteres (por ejemplo, 8 puntos). Se debe utilizar espacio simple y no tener espacio ni antes ni después del párrafo.
- La respuesta a las preguntas planteadas en el anexo a la práctica 1.

4 Información de interés

4.1 Creación de la aplicación

Microsoft Visual Studio 2019 permite hacer muchos tipos de aplicaciones. Para la creación de la aplicación de la práctica se debe escoger *Archivo->Nuevo ->Proyecto*. En la ventana que aparece se elegirá *C++* y dentro de los tipos posibles, *Proyecto vacío*. Se le da un nombre al proyecto y pulsamos en *Aceptar.*

Posteriormente nos aparecerá el explorador de soluciones donde estará creado nuestro proyecto. Debemos seleccionar *archivos de código fuente* y pulsar el botón derecho del ratón. En el menú que aparece elegiremos la opción *Agregar ->Nuevo elemento*, dentro de las opciones posibles se elige *Archivo C++ (.cpp)*. En la pantalla aparecerá una ventana en blanco donde iremos introduciendo el código necesario – **NOTA,** si la ventana aparece con código pre-escrito indicará que se ha cometido un error en algún paso previo, cierra el proyecto y créalo de nuevo.

4.2 Lanzamiento de hilos

La forma más sencilla de simular varios usuarios accediendo a la vez al servidor desde un solo programa cliente es utilizar hilos (*threads*). Los hilos son flujos de ejecución dentro de un proceso. Todos los hilos comparten la misma memoria global, pero cada uno puede estar ejecutando una función distinta y tener su propia pila. La programación con hilos plantea problemas muy complejos que no son el objetivo de esta asignatura. Aquí sólo veremos los conceptos básicos para Win32, evitando en lo posible la sincronización entre hilos, que es la mayor fuente de complejidades.

Cuando se lanza una aplicación, se crea un proceso y un hilo, que se denominará hilo principal y que empieza a ejecutar la función `main`. Si suponemos que el comportamiento de un usuario está implementado en una función `Usuario` como la siguiente:

```
DWORD WINAPI Usuario (LPVOID parametro) {
  DWORD dwResult = 0;
  int numHilo = *((int*) parametro);

  //... Resto de cosas comunes para cada usuario

  return dwResult;
}
```

para lanzar un nuevo hilo dentro del mismo proceso por cada usuario se debe hacer lo siguiente:

```
    HANDLE handleThread[numUsuarios];
    int parametro[numUsuarios];
    for (i = 0; i < numUsuarios; i++) {
     parametro[i] = i;
      handleThread[i] = CreateThread (NULL, 0, Usuario, &parametro[i], 0, NULL);
      if (handleThread[i] == NULL) {
        cerr << "Error al lanzar el hilo" << endl;
        exit (EXIT_FAILURE);
      }
    }

    // Hacer que el thread principal espere por sus hijos
    for (i = 0; i < numUsuarios; i++)
      WaitForSingleObject (handleThread[i], INFINITE);
```

La función `CreateThread` lanza un nuevo hilo, que ejecutará la función `Usuario`. Como parámetro al hilo se le pasa el valor `i`, es decir, el número de hilo. La función devuelve un `handle` al hilo lanzado. **Nota:** el valor de **`numUsuarios`**, en el array de `handleThread` debe ser un valor constante, por tanto, o bien se define al inicio del programa o como una variable de tipo `const`.

Después de lanzar los hilos, es necesario hacer que el hilo principal espere a que hayan acabado, ya que, si no esperase y acabase, toda la aplicación finalizaría. Para hacer la espera se utiliza la función `WaitForSingleObject`, indicando que espere hasta que acabe cada uno de los hilos lanzados.

Si ninguna de las funciones que se llaman desde la función `Usuario` escribe en una zona común de memoria (por ejemplo, una variable global), no es necesaria ninguna sincronización entre hilos. Por ello, con evitar las variables globales se evitarán los problemas con los hilos. Sin embargo, debe haber una variable global a la que los hilos accedan: aquella en la que dejarán los tiempos de reflexión y el número de peticiones. Como se explica en el apartado 2.4, para evitar conflictos entre hilos, esta variable se definirá como un vector. Cada hilo accederá a una posición distinta de este vector y, por lo tanto, ninguno modificará la misma zona de memoria global.

Cuando se usan hilos es necesario utilizar las opciones de compilación y enlazado de multihilo. Para ello se debe abrir el *Explorador de soluciones*, pulsar con el botón derecho sobre el nombre del proyecto y seleccionar la opción *Propiedades* en el menú contextual que aparece. Se abrirá un cuadro de diálogo. Seleccionar a la izquierda *Propiedades de la configuración->C/C++->Generación de código*. Aparecerá a la derecha un *combo-box* llamado *Biblioteca de tiempo de ejecución* donde se debe escoger la opción *MTd* (*Multi-thread debug*, depuración multihilo).

4.3 Generación de números aleatorios

Para la generación de números aleatorios, Win32 proporciona la función `rand`, que genera números distribuidos uniformemente entre 0 y RAND_MAX. En esta práctica se necesita generar una distribución exponencial para el tiempo de reflexión, lo que se puede lograr usando las siguientes funciones:

```
float NumeroAleatorio (float limiteInferior, float limiteSuperior) {
  float num = (float) rand ();
  num = num * (limiteSuperior - limiteInferior) / RAND MAX;
  num += limiteInferior;
  return num;
}

float DistribucionExponencial (float media) {
  float numAleatorio = NumeroAleatorio(0, 1);
// El while evita los valores 0 y 1 que en la función de logaritmos
// darían tiempos fuera de rango
  while (numAleatorio == 0 || numAleatorio == 1)
    numAleatorio = NumeroAleatorio(0, 1);
  return (-media) * logf(numAleatorio);
}
```

Cada usuario debe generar una secuencia de números aleatorios distinta, ya que si no las peticiones de los distintos usuarios estarían correlacionadas y los resultados no serían representativos de la realidad. Por ello, al principio de cada hilo se debe llamar a la función `srand`, que fija la semilla, pasándole como parámetro un número distinto a cada hilo y en la medida de lo posible diferente, por ejemplo: `71 + numHilo * 3`. De esta forma se conseguiría que cada hilo tenga una semilla distinta.

4.4 Programación de sockets en Win32

Windows sigue prácticamente el estándar de los sockets de Berkeley y sólo hay pequeñas desviaciones. La principal está en que el fichero a incluir es *winsock2.h* (o *windows.h*) y en alguno de los tipos básicos y nombres de función que se utilizan. Además, se debe incluir la librería **ws2_32.lib** entre las opciones de enlazado. Para esto se deben abrir las propiedades del proyecto (ver final del apartado 4.2), escoger a la izquierda *Vinculador->Línea de comandos* y en la caja de edición llamada *Opciones adicionales*, añadir *ws2_32.lib*. En este apartado se explica cómo realizar un cliente de sockets TCP en Win32.

4.4.1 Esquema general del cliente

1. Inicializar el uso de sockets llamando a la función *WSAStartUp*.
2. Crear el socket. Para ello se usa la función *socket*.
3. Conectarse con el servidor, para ello se utiliza la función *connect*.
4. Si la conexión tiene éxito, se produce el intercambio de datos, que se realiza mediante las funciones *send* y *recv*.
5. Cuando se finaliza la conexión se debe cerrar el socket, lo que se hace con la función *closesocket*.
6. Finalizar el uso de sockets con la función *WSACleanUp*.

4.4.2 Funciones

4.4.2.1 La función *WSAStartup*

Esta función debe ser la primera relacionada con sockets que haga cualquier aplicación. Permite especificar el tipo de sockets de Windows que se va a usar y obtener detalles de la implementación que hay de ellos. Tiene el siguiente prototipo:

```
int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSAData);
```

El primer parámetro indica la versión de sockets de Windows deseada. El segundo parámetro apunta a una estructura que la función rellenará con detalles de la implementación de sockets pedida. La forma habitual de utilizarla es:

```
WORD wVersionDeseada = MAKEWORD (2, 0);
WSAData wsaData;
if (WSAStartup (wVersionDeseada, &wsaData) != 0) {
  // Tratar el error
}
```

En una aplicación multihilo, se debe llamar sólo una vez. Lo más fácil es llamarla en el hilo principal antes de crear otros hilos.

4.4.2.2 La función *socket*

Esta función crea un socket. Tiene el siguiente prototipo:

```
SOCKET socket (int af, int type, int protocol);
```

El primer parámetro, *af*, especifica una familia de direcciones. Para las direcciones IP se debe usar el identificador AF_INET.

El segundo parámetro, *type*, especifica el tipo de socket. Puede ser:

- SOCK_STREAM: Para sockets con conexión.
- SOCK_DGRAM: Para sockets sin conexión.

El tercer parámetro, *protocol*, especifica el protocolo. Si se pone 0, la función escoge el adecuado.

La función devuelve un socket. Si hay un error, devuelve el valor INVALID_SOCKET.

4.4.2.3 La función *connect*

Esta función es sólo usada por el cliente de una comunicación usando sockets con conexión. Hace que el cliente se conecte a un servidor. Tiene el siguiente prototipo:

```
int connect (SOCKET s, const struct sockaddr FAR* name, int namelen);
```

El primer parámetro, *s*, es el socket del cliente. El segundo parámetro, *name*, es una estructura que contendrá la dirección y el puerto del servidor. El tercer parámetro, *namelen*, es la longitud del segundo parámetro. Se usa de la siguiente forma:

```
sockaddr in serv;          // Estructura para la dirección del servidor
int codigo;                // Para el valor de retorno de connect
serv.sin_family = AF_INET;
serv.sin_addr.s_addr = inet_addr("156.35.103.10");       ← CAMBIAR por dir. IP del servidor
serv.sin_port = htons(57000);                            // 57000 es el puerto
cout << "Direccion IP del servidor: " << inet ntoa(serv.sin addr) << endl;
codigo = connect(soc, (struct sockaddr *) &serv, sizeof(serv));
if (codigo == SOCKET ERROR)       {
  // Tratar el error
}
```

4.4.2.4 La función *send*

Esta función sirve para enviar datos a través de un socket. Tiene el siguiente prototipo:

```
int send (SOCKET s, const char FAR * buf, int len, int flags);
```

El primer parámetro, *s*, es el socket que se utiliza para la comunicación. El segundo parámetro, *buf*, es el buffer de bytes a enviar. El tercer parámetro, *len*, es el número de bytes del buffer anterior a enviar. El último parámetro, *flags*, permite indicar opciones alternativas que no se suelen usar habitualmente. Por lo tanto, el valor habitual para este parámetro será un cero.

La función devuelve el número de bytes enviados si no hubo error. Si hubo error, devuelve SOCKET_ERROR.

4.4.2.5 La función *recv*

Esta función sirve para recibir datos a través de un socket. Tiene el siguiente prototipo:

```
int recv (SOCKET s, char FAR* buf, int len, int flags);
```

El primer parámetro, *s*, es el socket que se usa para la comunicación. El segundo parámetro, *buf*, es un buffer donde se almacenarán los bytes recibidos. El tercer parámetro, *len*, es la longitud del buffer anterior. El último parámetro permite especificar ciertas opciones y será habitualmente cero.

La función devuelve el número de bytes leídos si tuvo éxito. Si el otro extremo ha cerrado la comunicación, devuelve cero. Si se produce un error, devuelve SOCKET_ERROR.

4.4.2.6 La función *closesocket*

Esta función cierra un socket. Si el socket está asociado a una conexión, cierra la conexión. Tiene el siguiente prototipo:

```
int closesocket (SOCKET s);
```

Devuelve cero si no ocurre ningún error. En otro caso, devuelve SOCKET_ERROR.

4.4.2.7 La función *WSACleanup*

Libera los recursos asociados a los sockets. Se debe llamar cuando ya no se desee trabajar más con sockets. Tiene el siguiente prototipo:

```
int WSACleanup(void);
```

Devuelve cero si no ocurre ningún error. En otro caso, devuelve SOCKET_ERROR.

4.4.2.8 La función *WSAGetLastError*

Esta función devuelve un código de error cuando ha ocurrido un error relativo a sockets. Tiene el siguiente prototipo:

```
int WSAGetLastError(void);
```

Cada función que trabaja con sockets tiene asociados ciertos códigos de error que se pueden consultar en su documentación en la ayuda en línea de Visual C++.

4.4.2.9 Función *gethostbyname*

Esta función obtiene información sobre una máquina a partir de su nombre. Tiene el siguiente prototipo:

```
struct HOSTENT FAR * gethostbyname (const char FAR * name);
```

La función retorna una estructura HOSTENT. La memoria para esta estructura es obtenida dinámicamente por la propia función, así que no es necesario reservar memoria antes. El miembro que interesa habitualmente de la estructura HOSTENT es *h_addr*, que será el necesario para el campo *sin_addr.s_addr* de las estructuras *sockaddr_in*.

4.5 Fuentes de información adicional

- FAQ sobre WinSock: http://tangentsoft.net/wskfaq/
- Sobre generación de números aleatorios: Capítulo 7 de Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis***.** Ed. McGraw-Hill.