Informática II
Práctica 2:
Mini-Chat v2.0 en modo cliente/servidor

GSyC

Diciembre de 2021

1. Introducción

En esta práctica debes realizar de nuevo dos programas en Swift siguiendo el modelo cliente/servidor, para ofrecer un servicio de chat entre usuarios.

Mini-Chat v2.0 mejora la implementación y la interfaz de uso de la anterior versión de Mini-Chat. La programación asíncrona mediante el uso de manejadores o *handlers* para recibir mensajes permitirá que el programa cliente pueda realizar concurrentemente las funciones de lector y de escritor definidas en la anterior práctica, sin necesidad de arrancar un programa cliente en modo escritor para leer de la entrada estándar (teclado) y enviar mensajes al servidor; y otro programa cliente en modo lector para recibir mensajes del servidor y mostrarlos en la salida estándar (pantalla).

En una sesión de chat participará un programa servidor y varios programas cliente, uno por cada usuario:

- Los clientes leen de la entrada estándar cadenas de caracteres y las envían al servidor. Simultáneamente, los clientes van recibiendo las cadenas de caracteres que les envía el servidor procedentes de otros clientes, utilizando recepción asíncrona mediante manejadores o *handlers*.
- El servidor recibe los mensajes con cadenas de caracteres que le envían los clientes y las reenvía al resto de los clientes.

Mini-Chat v2.0 tendrá además la siguiente funcionalidad:

- El servidor se asegurará de que los apodos (*nicknames*) de los clientes no se repitan, rechazando a un cliente que pretenda utilizar un apodo que ya esté siendo usado por otro cliente, para lo cual enviará un mensaje al cliente informándole de que ha sido rechazado.
- El mandato `.quit` ejecutado por un cliente provocará que se desconecte el cliente del servidor.
- Existirá un número máximo de clientes activos en el servidor. Si en un momento dado quisiera entrar en el chat un cliente cuando ya se ha alcanzado el máximo, el servidor elegirá al cliente que lleva más tiempo sin escribir en el chat, y lo expulsará para dejar sitio al nuevo cliente.
- El servidor mantendrá una lista de clientes inactivos, que contendrá los datos de los clientes que hayan sido expulsados por inactividad, o que hayan abandonado el chat voluntariamente.
- Se utilizarán estructuras de datos genéricas para almacenar tanto los clientes activos como los inactivos.

Estos puntos se detallan en el resto de secciones de esta especificación.

2. Descripción del programa cliente `chat-client`

2.1. Interfaz de usuario

El programa cliente se lanzará pasándole 3 argumentos en la línea de comandos:

- Nombre de la máquina en la que está el servidor

- Número del puerto en el que escucha el servidor
- *Nickname* (apodo) del cliente del chat. Cualquier cadena de caracteres distinta de la cadena `server` será un apodo válido siempre que no exista ya otro cliente conocido por el servidor que haya usado el mismo apodo. El apodo `server` lo utiliza el servidor para enviar cadenas de caracteres a los clientes informando de la entrada de un nuevo cliente en el chat, o de la salida del chat de un cliente.

El cliente mostrará en pantalla una primera línea en la que informará de si ha sido aceptado o no por el servidor. Si no es aceptado el programa cliente termina.

En caso de ser aceptado el cliente irá pidiendo al usuario cadenas de caracteres por la entrada estándar, y se las enviará al servidor. Si la cadena de caracteres leída de la entrada estándar es `.quit`, el cliente terminará su ejecución tras enviar un mensaje al servidor informándole de su salida.

El cliente irá recibiendo del servidor las cadenas de caracteres enviadas por otros clientes del Mini-Chat v2.0, y las mostrará en pantalla.

En los siguientes cinco recuadros se muestra la salida de clientes que se arrancan en terminales distintos en el orden en el que aparecen los recuadros en el texto. Se supondrá que el servidor de Mini-Chat v2.0 que están usando los clientes está escuchando en el puerto 9001 en la máquina `f-l3204-pc02`, y que no hay ningún proceso escuchando en el puerto 10001 de la máquina `f-l3204-pc03`.

Primer cliente que se arranca:

```
$ swift run chat-client f-l3204-pc02 9001 carlos
Mini-Chat v2.0: Welcome carlos
>>
server: ana joins the chat
>>
server: pablo joins the chat
>>
ana: entro
>> Hola
>> quién está ahí?
>>
ana: estoy yo, soy ana
>> ana dime algo
>>
ana: hola carlos
>> adios
>> .quit
$
```

Segundo cliente que se arranca:

```
$ swift run chat-client f-l3204-pc02 9001 ana
Mini-Chat v2.0: Welcome ana
>>
server: pablo joins the chat
>> entro
>>
carlos: Hola
>>
carlos: quién está ahí?
>> estoy yo, soy ana
>>
carlos: ana dime algo
>> hola carlos
>>
carlos: adios
```

```
>>
server: carlos leaves the chat
>> hasta luego chico, ¡vaya modales! Yo también me voy.
>> .quit
$
```

Tercer cliente que se arranca:

```
$ swift run chat-client f-l3204-pc02 9001 pablo
Mini-Chat v2.0: Welcome pablo
>>
ana: entro
>>
carlos: Hola
>>
carlos: quién está ahí?
>>
ana: estoy yo, soy ana
>>
carlos: ana dime algo
>>
ana: hola carlos
>>
carlos: adios
>>
server: carlos leaves the chat
>>
ana: hasta luego chico, ¡vaya modales! Yo también me voy.
>>
server: ana leaves the chat
>>
```

Cuarto cliente que se arranca:

```
$ swift run chat-client f-l3204-pc02 9001 pablo
Mini-Chat v2.0: IGNORED new user pablo, nick already used
$
```

Quinto cliente que se arranca:

```
$ swift run chat-client f-l3204-pc03 10001 pepe
Server unreachable
$
```

2.2. Implementación

El cliente deberá enviar al arrancar un mensaje `Init` al servidor de Mini-Chat v2.0 con su *nickname* para solicitar ser aceptado como nuevo cliente.

El servidor le enviará como respuesta un mensaje `Welcome`, informándole de si ha sido aceptado o no. Además, en caso de que lo acepte, el servidor enviará un mensaje `Server` **al resto de clientes** informándoles de la entrada de un nuevo usuario.

Un cliente será rechazado por pretender utilizar un apodo que ya está usándose por otro cliente del chat.

Tras enviar un cliente su mensaje `Init`, pueden presentarse las siguientes situaciones:

- Si transcurridos 10 segundos el cliente no ha recibido el mensaje `Welcome` del servidor, el cliente terminará, mostrando en pantalla el mensaje `Server unreachable`. Puede verse un ejemplo en el último recuadro de la sección anterior.

- Si el cliente recibe el mensaje `Welcome` indicándole que es rechazado terminará su ejecución, mostrando en pantalla un mensaje explicativo. Puede verse un ejemplo en el penúltimo recuadro de la sección anterior.
- Si el cliente recibe el mensaje `Welcome` indicándole que es aceptado:
  - El cliente entrará en un bucle en el que pedirá al usuario cadenas de caracteres que le irá enviando al servidor mediante mensajes `Writer`. Si la cadena de caracteres leída es `.quit` el cliente envía un mensaje `Logout` al servidor y a continuación termina su ejecución.
  - Simultáneamente, el cliente irá recibiendo mensajes `Server`, utilizando para ello la recepción asíncrona mediante *handlers* que se ha visto en clase. El cliente debe mostrar en pantalla el contenido de los mensajes recibidos.

    El mensaje `Server` recibido en un cliente puede contener mensajes de texto de otros clientes que han sido recibidos por el servidor, pero también puede contener mensajes generados por el propio servidor para informar de que ha entrado un nuevo cliente al chat, o de que un cliente ha abandonado el chat.

Mientras se está esperando a recibir el mensaje `Welcome` del servidor, el cliente no hará nada más: ni leerá mensajes tecleados por el usuario ni mostrará posibles mensajes `Server` que lleguen de la red. Esto es así porque en este intervalo de tiempo el cliente **todavía no está conectado al chat**.

La espera a recibir el mensaje `Welcome` se hará, por tanto, de forma **síncrona, en modo bloqueante**. Una vez el cliente ha sido aceptado, se iniciará el proceso de lectura asíncrono.

En el apartado 5 se explica el formato de los mensajes mencionados.

3. Servidor: Clientes activos e inactivos

En esta práctica, el programa servidor se lanzará pasándole 2 argumentos en la línea de comandos:

- Número del puerto en el que debe escuchar el servidor.
- Número máximo de clientes que acepta, que estará comprendido entre 2 y 50.

En caso de que no se pasen los dos parámetros o alguno sea incorrecto, el programa debe mostrar un error de ejecución. Si los dos parámetros son correctos, escuchará indefinidamente en el puerto indicado.

El servidor mostrará la información de los clientes activos cuando se pulse la tecla `l` o la tecla `L`, mostrando para cada uno de ellos su *nick*, su *dirección* y la hora de última actualización.

El servidor mostrará la información de antiguos clientes que ya no están activos cuando se pulse la tecla `o` o la tecla `O`. Para cada cliente inactivo, se mostrará su *nick* y *el momento en que dejaron de estar activos*.

Puede verse un ejemplo de la salida que muestra el servidor en el siguiente recuadro:

```
$ swift run chat-server 9001 5
INIT received from carlos: ACCEPTED
INIT received from ana: ACCEPTED
INIT received form pablo: ACCEPTED
INIT received form pablo: IGNORED. Nick already used
WRITER received from ana: entro
WRITER received from carlos: Hola
WRITER received from carlos: quién está ahí?
WRITER received from ana: estoy yo, soy ana
WRITER received from unknown client. IGNORED
WRITER received from carlos: ana dime algo
WRITER received from ana: hola carlos
WRITER received from carlos: adios
l
ACTIVE CLIENTS
==============
pablo (127.0.1.1:23025): 02-Ene-21 20:00:01
ana (127.0.1.1:10313): 02-Ene-21 20:00:10
carlos (127.0.1.1:13311): 02-Ene-21 20:01:07
```

```
LOGOUT received from carlos
o
OLD CLIENTS
===========
carlos: 02-Ene-21 20:05:00

l
ACTIVE CLIENTS
==============
pablo (127.0.1.1:23025): 02-Ene-21 20:00:01
ana (127.0.1.1:10313): 02-Ene-21 20:00:10

WRITER received from ana: hasta luego chico, ¡vaya modales! Yo también me voy.
LOGOUT received from ana
o
OLD CLIENTS
===========
ana: 02-Ene-21 20:07:50
carlos: 02-Ene-21 20:05:00
```

3.1. Implementación

El servidor deberá guardar en una cola (*queue*) genérica la información relativa a los **clientes activos**. Se añadirán clientes cuando se reciban mensajes `Init`. De cada cliente activo el servidor deberá almacenar en la cola su *nickname*, dirección y la fecha y hora de última actualización.

Un cliente deja de estar activo en el servidor de chat cuando se recibe un mensaje `Logout` de ese cliente, o cuando el servidor decide expulsarlo. El servidor deberá guardar en *otra colección, que se implementará como un* `Stack` la información relativa a **clientes antiguos que ya no están activos**. De cada cliente no activo el servidor deberá almacenar solamente su nickname y la hora a la que abandonó el chat (bien sea porque hizo *logout* o porque fue expulsado).

El servidor debe ponerse a escuchar en la dirección IP de la máquina en la que se ejecuta y el puerto que se le pasa como primer argumento al lanzar el programa.

A continuación, entrará en un bucle infinito recibiendo mensajes de clientes:

- Cuando el servidor reciba un mensaje `Init` de un cliente, lo añadirá si el *nick* no está siendo ya utilizado por otro cliente, guardando la hora en la que se recibió el mensaje `Init`. A continuación le enviará un mensaje `Welcome` a ese mismo cliente, informándole de si ha sido aceptado o rechazado.

  Si no hubiera huecos libres para almacenar la información del nuevo cliente, el servidor generará un hueco eliminando la entrada correspondiente al cliente que hace más tiempo que envió su último mensaje `Writer`. Si un cliente no ha enviado ningún mensaje `Writer` se considerará la hora a la que envió su mensaje de inicio para decidir si se le expulsa. El servidor enviará un mensaje `Server` a todos los clientes, incluyendo al que se expulsa, informándoles de que el cliente ha sido expulsado del chat. Este mensaje llevará en el campo *nickname* la cadena `server`, y en el campo `text` la cadena `<nickname> banned for being idle too long`, siendo `<nickname>` el apodo del cliente expulsado.

  Si el cliente es aceptado, el servidor enviará un **mensaje de servidor** a todos los clientes, salvo al que ha sido aceptado, informándoles de que un nuevo cliente ha sido aceptado en el chat. Este mensaje llevará en el campo *nickname* la cadena `server`, y en el campo `text` la cadena
  `<nickname> joins the chat`, siendo `<nickname>` el apodo del cliente que ha sido aceptado.

- Cuando el servidor reciba un mensaje `Writer` de un cliente, buscará el *nick* entre los de los clientes activos, y si lo encuentra, comprobará que la dirección almacenada para ese usuario coincide con la recibida en el mensaje. Si es así, enviará un mensaje `Server` a todos los clientes conocidos, **salvo al que le ha enviado el mensaje**.

  Observa que el mensaje `Writer` ahora debe incorporar también el nick del usuario, como se describe en la sección 5.

- Cuando el servidor reciba un mensaje `Logout` de un cliente buscará el *nick* entre los de los clientes activos, y si lo encuentra, comprobará que la dirección almacenada para ese usuario coincide con la recibida en el mensaje. De ser así, eliminará su registro de la colección de clientes activos y almacenará los datos correspondientes en la lista de clientes inactivos. Además, enviará un mensaje `Server` al resto de los clientes, informándoles de que el cliente ha abandonado el chat. Este mensaje llevará en el campo *nickname* la cadena `server` y en el campo `text` la cadena `<nickname> leaves the chat`, siendo `<nickname>` el apodo del cliente que abandona el chat.

  Observa que el mensaje `Logout` debe incorporar también el nick del usuario, como se describe en la sección 5.

4. Colecciones

Los clientes activos y los inactivos se almacenarán, respectivamente, en un `Queue` y un `Stack` genéricos, cuyo interfaz público se proporciona en sendos protocolos que forman parte del código de inicio que se suministra con el enunciado de la práctica.

Las colecciones deben implementarse dentro del módulo `Collections`, que será importado como dependencia desde el programa `chat-server`. En este módulo reside la definición de los protocolos genéricos `Queue` y `Stack`, y deben añadirse también las implementaciones de `ArrayQueue` y `ArrayStack` necesarias para completar la práctica.

4.1. Lista de clientes activos

Los clientes activos, como se acaba de indicar, se almacenarán en un `Queue` genérico cuya definición es similar a la vista en clase. Se incorpora, además, una variable `maxCapacity` para forzar el número máximo de elementos que se permite almacenar. Así, el método `enqueue` lanzará la excepción `maxCapacityReached` si se pretende almacenar un nuevo elemento una vez se ha alcanzado la capacidad máxima definida.

Asimismo, se han incorporado funciones necesarias para la iteración, búsqueda y eliminación de elementos. Como puede verse en la figura 1, a las funciones de búsqueda y eliminación debe suministrárseles como parámetro un bloque de código que expresa la condición que debe cumplirse para encontrar o eliminar el elemento deseado.

La implementación del protocolo `Queue`, por simplicidad, debe realizarse mediante una estructura genérica basada en un Array. Este tipo se denominará `ArrayQueue`. Debe probarse exhaustivamente el funcionamiento de todas sus funciones antes de incorporarlo en la práctica; en caso contrario, no sabremos si los errores se deben a la estructura de datos o al programa servidor. Para ello, lo mejor es realizar un programa de pruebas específico, que debe incorporarse también como parte de la entrega.

El hecho de elegir una cola como la estructura de datos donde almacenar los clientes activos no es casual, pues nos facilita mantener a los clientes ordenados por fecha de última actualización. Una estrategia válida para conseguirlo puede ser la siguiente:

- Cuando el servidor recibe un mensaje `Writer`, verifica que existe un cliente almacenado cuyo nick y dirección coinciden con la del origen del mensaje.
- De ser así, **el servidor eliminará temporalmente** de la lista ese cliente.
- A continuación, el servidor utilizará `enqueue` para almacenarlo de nuevo inmediatamente, de modo que el cliente se guardará en la última posición de la cola. En este punto se actualizará también la fecha de actualización para que coincida con el momento en que se ha recibido el mensaje.
- Cuando el servidor necesite eliminar un cliente por inactividad, puede utilizar `dequeue` para eliminar el cliente más antiguo. Si seguimos el proceso anterior cuidadosamente (los clientes sobre los que se reciben novedades se reincorporan al final de la cola), podemos garantizar que la cola mantiene los datos ordenados por su fecha de última actualización.

4.2. Lista de clientes inactivos

Los clientes inactivos se almacenarán en otra colección, que en este caso se implementará mediante un **`Stack` genérico**, que se implementará utilizando un array como almacenamiento.

Del mismo modo, es necesario realizar un programa de pruebas específico e incorporarlo como parte de la entrega.

```swift
public enum CollectionsError : Error {
    case maxCapacityReached
}

/// A simple generic Queue protocol with a maximum capacity.
public protocol Queue {
    associatedtype Element

    var count: Int { get }
    var maxCapacity: Int { get }

    mutating func enqueue(_ value: Element) throws
    mutating func dequeue() -> Element?

    func forEach(_ body: (Element) throws -> Void) rethrows

    func contains(where predicate: (Element) -> Bool) -> Bool
    func findFirst(where predicate: (Element) -> Bool) -> Element?

    mutating func remove(where predicate: (Element) -> Bool)
}
```

Figura 1: `Queue`

5. Formato de los mensajes

En esta sección se describen todos los mensajes necesarios para implementar el protocolo de comunicaciones deseado. Algunos son similares a los de la práctica anterior, pero debe prestarse atención a las modificaciones, pues ahora el nick se incluye en los mensajes `Writer`.

Mensaje `Init`

Es el que envía un cliente al servidor al arrancar. Formato:

`Init`	`Nick`

en donde:

- `Init`: valor del tipo `ChatMessage` que identifica el tipo de mensaje.
- `Nick`: `String` con el *nick* del cliente. Debe enviarse en formato C; es decir, como caracteres UTF8 terminados con el carácter nulo (0).

Mensaje `Welcome`

Es el que envía un servidor a un cliente tras recibir un mensaje `Init` para indicar al cliente si ha sido aceptado o rechazado en Mini-Chat v2.0. Formato:

`Welcome`	`Accepted`

en donde:

- `Welcome`: `ChatMessage` que identifica el tipo de mensaje.
- `Accepted`: `Bool` que tomará el valor `false` si el cliente ha sido rechazado y `true` si el cliente ha sido aceptado.

Mensaje `Writer`

Es el que envía un cliente al servidor con una cadena de caracteres introducida por el usuario. Formato:

`Writer`	`Nick`	`Text`

en donde:

- `Writer`: valor del tipo `ChatMessage` que identifica el tipo de mensaje.
- **`Nick`: nickname del usuario que envía el mensaje.**
- `Text`: `String` con la cadena de caracteres introducida por el usuario. Debe enviarse en formato C, con longitud variable.

Nótese que, a diferencia de la práctica anterior, el *nick* ahora sí viaja en estos mensajes. El servidor, tras recibir un mensaje de este tipo, deberá buscar en su colección de clientes activos el *nick* del cliente, y verificar que la dirección almacenada en la colección coincide con la dirección de origen del mensaje.

Mensaje `Server`

Es el que envía un servidor a cada cliente lector, tras haber recibido un mensaje `Writer` procedente de un cliente escritor conteniendo un texto escrito por un usuario. El servidor envía el mensaje `Server` para comunicar a todos los lectores dicho texto. También se utiliza este mensaje para informar a los clientes de los usuarios que entran o salen del chat.

Formato:

`Server`	`Nick`	`Text`

en donde:

- `Server`: valor del tipo `ChatMessage` que identifica el tipo de mensaje.
- `Nick`: `String` (en formato C) con el *nick* del cliente que escribió el texto. Si el mensaje es para informar de que un nuevo cliente ha entrado en el chat o lo ha abandonado, este campo tendrá el valor `server`.
- `Texto`: `String` (en formato C) con la cadena de caracteres introducida por el usuario. Si el mensaje es para informar de que un nuevo cliente ha entrado o salido del chat, este campo tendrá el valor del *nick* del usuario concatenado con la cadena `joins the chat` o `leaves the chat`, respectivamente.

Mensaje `Logout`

Es el que envía un cliente al servidor para informarle de que abandona el servicio de chat. Formato:

Server	Nick

en donde:

- `Logout`: `ChatMessage` que identifica el tipo de mensaje.
- `Nick`: nickname del usuario que envía el mensaje.

6. Fechas en Swift

Para poder manejar fechas en Swift debe usarse el tipo `Date`. Al crear una variable tipo `Date` sin parámetros, obtendremos un valor que representa la **fecha y hora actual del sistema**. Puede comprobarse utilizando las siguientes líneas de código:

```
let fechaDeAhora = Date()
print(fechaDeAhora)       // 2020-12-07 10:35:06 +0000
```

La descripción de fechas realizada por `print()` no suele ser adecuada para mostrar a los usuarios. La forma más sencilla de *formatear* la fecha para obtener un `String` conforme a nuestras necesidades es mediante el tipo `DateFormatter`, que incorpora muchos mecanismos para especificar el resultado que deseamos. Por ejemplo, las siguientes líneas configuran un `DateFormatter` para mostrar una fecha (y hora) formateada para mostrar **primero el año, después el nombre corto del mes y día, seguidos de horas y minutos**:

```
let df = DateFormatter()
df.dateFormat = "yy-MMM-dd HH:mm"
print(df.string(from: fechaDeAhora))    // 20-Dec-07 11:35
```

Al objeto `df`, de tipo `DateFormatter`, se le indica el formato deseado para fechas utilizando `dateFormat`, que es un `String` que indica de forma simbólica dónde debe situarse cada campo. Al invocar la función `df.string(from: fecha)`, el `DateFormatter` coloca los campos en el string como le hemos indicado.

7. Entrega

El límite para la entrega de esta práctica es: **Viernes, 14 de Enero a las 10:00h**