Informática II
Práctica 1:
Mini-Chat en modo cliente/servidor

GSyC

Noviembre de 2021

1. Introducción

En esta práctica debes realizar dos programas en Swift que permitan implementar un sencillo sistema de chat entre usuarios, siguiendo el modelo cliente/servidor.

El programa cliente podrá comportarse de dos maneras: en modo escritor, o en modo lector. Si un cliente es escritor podrá enviar mensajes al servidor del chat. Si un cliente es lector podrá recibir los mensajes que envíe el servidor. Por limitaciones que iremos resolviendo en prácticas sucesivas, un mismo cliente no puede enviar y recibir mensajes al mismo tiempo.

El servidor se encargará de recibir los mensajes procedentes de los clientes escritores, y reenviárselos a los clientes lectores.

En una sesión de chat participará un programa servidor y varios programas clientes (lectores y escritores). Cada usuario del chat tiene que arrancar 2 programas cliente: uno escritor, que lee del teclado y envía mensajes, y uno lector, que recibe mensajes del servidor y los muestra en la pantalla.

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. Si el *nick* es `reader`, el cliente funcionará en modo lector. Con cualquier otro *nick* distinto a `reader` el cliente funcionará en modo escritor.

Una vez lanzado:

- Si el cliente se lanza en modo **escritor**, pedirá al usuario cadenas de caracteres, y se las irá enviando al servidor. Los clientes lanzados en modo escritor no reciben mensajes del servidor. El programa terminará cuando el usuario introduzca la cadena `.quit`
- Si el cliente se lanza en modo **lector**, esperará a recibir mensajes del servidor (conteniendo las cadenas enviadas por los clientes escritores del chat), y los mostrará en pantalla. En este modo el cliente nunca terminará su ejecución.

MUY IMPORTANTE: Si un cliente escritor intenta entrar al chat utilizando un *nick* que ya está siendo usado por otro cliente escritor que entró anteriormente, todos sus mensajes serán ignorados por el servidor.

Ejemplo de ejecución: Suponiendo que se ha lanzado previamente un programa servidor, se muestra a continuación cómo se lanza un cliente lector y varios clientes escritores, cada cliente en una ventana de terminal diferente. El lector es el primer cliente en lanzarse.

```
$ swift run chat_client f-l3204-pc02 9001 reader
server: ana joins the chat
ana: entro
server: carlos joins the chat
carlos: Hola
carlos: quién está ahí?
ana: estoy yo, soy ana
carlos: ana dime algo
ana: hola carlos
carlos: adios
ana: hasta luego
```

```
$ swift run chat_client f-l3204-pc02 9001 ana
Message: entro
Message: estoy yo, soy ana
Message: hola carlos
Message: hasta luego
Message: .quit
$
```

```
  $ swift run chat_client f-l3204-pc02 9001 carlos
  Message: Hola
  Message: quién está ahí?
  Message: ana dime algo
  Message: adios
  Message: .quit
  $
```

```
$ swift run chat_client f-l3204-pc02 9001 ana
Message: he entrado?
```

En el ejemplo puede verse cómo intenta entrar en el chat un segundo cliente escritor con *nick* `ana`, pero sus mensajes son ignorados por el servidor, por lo cual en el lector no aparece ni su intento de entrada ni su mensaje `he entrado?`.

Sólo en el primer cliente, lanzado como lector, pueden verse todos los mensajes que van enviando los clientes escritores carlos y ana.

2.2. Implementación

El programa cliente tendrá dos comportamientos diferentes según sea lanzado como lector (con el *nick* `reader`) o escritor (con cualquier otro *nick*).

Cuando es lanzado **como escritor**, enviará un mensaje `Init` al servidor para indicarle su *nick*. Tras enviar ese mensaje, el programa entrará en un bucle que pida al usuario cadenas de texto, que le irá enviando al servidor mediante mensajes `Writer`. El cliente escritor termina cuando el usuario introduce la cadena de texto `.quit` en el teclado. NOTA: cuando un cliente escritor termina no envía ningún mensaje, por lo que seguirá en la colección de clientes escritores que se almacena en el servidor.

Cuando es lanzado **como lector**, enviará un mensaje `Init` al servidor para indicarle su *nick* (que siempre será `reader`). Tras enviar este mensaje, el programa entrará en un bucle infinito esperando a recibir los mensajes `Server` enviados por el servidor, cuyo contenido mostrará en pantalla. El cliente lector nunca termina su ejecución[1].

En el apartado 4 se explica el formato de los mensajes.

[1] Para interrumpir el programa habrá que pulsar CTRL-C en la ventana de terminal en la que está lanzado

3. Descripción del programa servidor `chat-server`

3.1. Interfaz de usuario

El programa servidor se lanzará pasándole 1 argumento en la línea de comandos:

- Número del puerto en el que escucha el servidor

Una vez lanzado, el servidor recibirá mensajes procedentes de clientes:

- Si recibe un mensaje `Init`, añadirá el cliente a la colección de clientes lectores o a la de escritores, y, además, cuando el cliente sea un nuevo escritor, enviará **a todos los lectores** un mensaje `Server` notificando la entrada del nuevo usuario en el chat.
- Si recibe un mensaje `Writer`, comprobará si pertenece a un cliente conocido y, en caso afirmativo, enviará **a todos los lectores** un mensaje `Server` conteniendo el *nick* del cliente escritor y el texto que contenía el mensaje `Writer` recibido.

El servidor nunca terminará su ejecución[2].
El servidor irá mostrando en pantalla los mensajes que vaya recibiendo para permitir comprobar su funcionamiento.
Ejemplo de ejecución que se corresponde con los mensajes enviados por los clientes del ejemplo del apartado anterior:

```
$ ./chat_server 9001
INIT received from reader
INIT received from ana
INIT received form carlos
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
INIT received from ana. IGNORED, nick already used
WRITER received from carlos: ana dime algo
WRITER received from ana: hola carlos
WRITER received from unknown client. IGNORED
WRITER received from carlos: adios
WRITER received from ana: hasta luego
```

En el ejemplo puede verse cómo es ignorado el mensaje inicial del segundo cliente escritor que quiere usar el *nick* `ana`, y cómo posteriormente su mensaje de escritor es también ignorado al venir de un cliente desconocido.

3.2. Implementación

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 argumento. Entrará en un bucle infinito en el que recibirá datagramas UDP procedentes de los clientes.

El servidor deberá almacenar en **dos colecciones** los clientes conocidos. **Cada colección será una variable de tipo `ClientCollectionArray`, que debe implementarse siguiendo los requisitos del Tipo Abstracto de Datos definido en el protocolo `ClientCollection`**.

El protocolo `ClientCollection` se suministra como parte del código inicial propuesto para la práctica. Su implementación con un array debe completarse escribiendo el código correspondiente en `ClientCollectionArray`.

Para cada cliente el servidor almacenará tanto su dirección (de tipo `Socket.Address`) como su `nick`.

Cuando el servidor reciba un mensaje `Init` de un cliente, si es de un lector lo añadirá a la colección de clientes lectores conocidos. Si es de un escritor y su *nick* aún no está siendo usado, lo añadirá a su colección de clientes escritores conocidos y enviará un mensaje `Server` a todos los clientes lectores indicando el *nick* del nuevo usuario que entra en el chat. Si el

[2]Para interrumpir el programa habrá que pulsar CTRL-C en la ventana de terminal en la que está lanzado

nick recibido en un mensaje `Init` de un escritor ya está siendo utilizado por otro cliente, el servidor ignorará ese mensaje inicial, y no enviará ningún mensaje a los lectores notificando la entrada.

Cuando el servidor reciba un mensaje `Writer` de un cliente, comprobará si el mensaje pertenece a un cliente que esté en su colección de clientes escritores conocidos. Si es así, enviará un mensaje `Server` a todos los clientes de la colección de lectores indicando el *nick* del cliente escritor y el texto contenido en el mensaje recibido.

En el apartado 4 se explica el formato de los mensajes. Como puede verse en ese apartado, el mensaje `Writer` que envía un cliente sólo contiene el texto del mensaje y no el *nick* del usuario. Por ello el servidor deberá buscar en su colección de clientes escritores conocidos la dirección del cliente y así podrá obtener el *nick* asociado al mismo. Lo necesita para poder componer el mensaje `Server` que enviará a todos los clientes de la colección de clientes lectores, pues este mensaje lleva el *nick* del cliente escritor que lo ha enviado. Si la dirección buscada no aparece en la colección de escritores, es que el mensaje `Writer` procede de un cliente escritor desconocido y debe ser ignorado. En este caso no se enviará un mensaje `Server` a los clientes de la colección de lectores.

4. Formato de los mensajes

Los tres tipos de mensajes que se necesitan para esta práctica se distinguen por el primer campo, que podrá adoptar los valores del siguiente tipo enumerado, que se suministra como parte del código inicial de la práctica dentro del fichero `ChatMessage.swift`.

```
public enum ChatMessage {
  case Init
  case Writer
  case Server
}
```

MUY IMPORTANTE: Es imprescindible mantener el orden mostrado en los valores de este tipo, a fin de que sean compatibles las implementaciones de clientes y servidores realizadas por distintos alumnos.

Tal como se especifica en el apartado 6, el código en `ChatMessage.swift` se compilará como un módulo Swift y podrá ser utilizado tanto por el servidor como por el cliente, utilizando la sentencia `import ChatMessage` en los ficheros que lo necesiten.

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). Si el *nick* es `reader`, el cliente estará en modo lector, en caso contrario estará en modo escritor.

Mensaje `Writer`

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

`Writer`	`Text`

en donde:

- `Writer`: valor del tipo `ChatMessage` que identifica el tipo de mensaje.
- `Text`: `String` con la cadena de caracteres introducida por el usuario. Debe enviarse en formato C, como en el caso anterior.

Nótese que el *nick* no viaja en estos mensajes: sólo se puede identificar al cliente que envía un mensaje `Writer` a partir de su dirección de origen. El servidor, tras recibir este mensaje, deberá buscar en su colección de clientes escritores el *nick* asociado a la dirección del cliente. Una vez encontrado el *nick*, podrá utilizarlo para componer el mensaje de servidor que reenviará a los clientes lectores.

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. Formato:

`Server`	`Nick`	`Texto`

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, 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 en el chat, este campo tendrá el valor del *nick* del usuario concatenado con la cadena `joins the chat`.

5. Condiciones obligatorias de funcionamiento

1. Se supondrá que nunca se pierden los mensajes enviados ni por el servidor ni por los clientes.

2. Los programas deberán ser robustos, comportándose de manera adecuada cuando no se arranquen con las opciones adecuadas en línea de comandos.

3. El servidor debe almacenar los clientes lectores conocidos y los clientes escritores conocidos en dos variables distintas. Estas variables serán del mismo **tipo abstracto de datos representado en `ClientCollection.swift`, que se suministra como parte del código inicial de la práctica.**. Este fichero (que se reproduce en la figura 1) define un `protocol` que debe ser implementado en el fichero `ClientCollectionArray`, utilizando un *array* como estructura donde almacenar los datos.

4. La variable donde se almacenen los clientes **lectores** se creará como un `ClientCollectionArray` en el que **`uniqueNicks`** será **`false`** (puesto que todos los nicks de lectores son siempre el mismo: `reader`).

5. La variable donde se almacenen los clientes **escritores** se creará como un `ClientCollectionArray` en el que **`uniqueNicks`** será **`true`**, puesto que no se aceptan escritores con el mismo *nick*.

6. Comportamiento esperado de las funciones de `ClientCollection`:

   - `addClient`:
     - Si el parámetro `nick` NO está en la lista, añade un nuevo elemento a la misma con su dirección (que debe ser del tipo `Socket.Address`) y su `nick`.
     - Si el parámetro `nick` ya está en la lista y la colección se ha inicializado con `uniqueNicks` a `true`, lanzará la excepción `ClientCollectionError.repeatedClient`.
   - `removeClient`: Si el `nick` está en la lista, lo elimina de la misma. En caso contrario, lanza la excepción `ClientCollectionError.noSuchClient`.
   - `searchClient`: Busca el cliente a partir de la dirección desde donde se ha conectado. Si dicha dirección está en la lista, devuelve el `nick` asociado. En caso contrario, devuelve `nil`.
   - `forEach`: Esta función recibe como parámetro otra función (o un *closure*) a ejecutar para cada uno de los elementos de la colección. El *closure* recibirá como parámetros tanto la dirección de conexión como el *nick* de cada uno de los clientes.

     Este mecanismo puede utilizarse para enviar un mensaje a todos los clientes escritores. Dentro del bloque de código suministrado a `forEach` se enviará el contenido de un *buffer* previamente construido a la dirección del cliente.

     Obsérvese que el *closure* o función suministrado a `forEach` puede lanzar excepciones. Por este motivo `forEach` se declara con `rethrows`, que tiene el siguiente significado:
     - Si la función lanza excepciones, `forEach` también lo hará.
     - Si la función no lanza excepciones, `forEach` tampoco lo hará.

7. La salida del programa deberá ser **exactamente igual** a la que se muestra en los ejemplos de ejecución, con el mismo formato.

8. Un cliente programado por un alumno deberá poder funcionar con un servidor programado por cualquier otro alumno, y viceversa. Es conveniente que pruebes tus programas con los de otros alumnos. Por ello es imprescindible respetar el protocolo de comunicación entre cliente y servidor y, especialmente, el formato de los mensajes.

9. Cualquier cuestión no especificada en este enunciado puede resolverse e implementarse como se desee.

```swift
//
//  ClientCollection.swift
//

import Foundation
import Socket

public enum ClientCollectionError: Error {
    /** Thrown by addClient, if attempting to add the same nick and `uniqueNicks` is true. */
    case repeatedClient

    /** Thrown by removeClient, if the client does not exist. */
    case noSuchClient
}

public protocol ClientCollection {
    /** Indicates whether nicknames need to be unique or can be repeated. */
    var uniqueNicks: Bool { get }

    /**
     Add a new client.
     Throws `ClientCollectionError.repeatedClient` if the nick exists and `uniqueNicks` is true.
     */
    mutating func addClient(address: Socket.Address, nick: String) throws

    /**
     Remove the client(s) specified by the nick.
     Throws `ClientCollectionError.noSuchClient` if the client does not exist.
     */
    mutating func removeClient(nick: String) throws

    /**
     Search by address. Returns the nickname, or `nil` if the address was not found in the list.
     */
    func searchClient(address: Socket.Address) -> String?

    /**
     Runs `body` closure for each element in the list.
     `rethrows` means that `forEach` will throw if the closure `throws`.
     */
    func forEach(_ body: (Socket.Address, String) throws -> Void) rethrows
}

```

Figura 1: `ClientCollection`

```swift
import PackageDescription

let package = Package(
    name: "p1",
    dependencies: [
        .package(name: "Socket", url: "https://github.com/Kitura/BlueSocket.git", from: "1.0.0"),
    ],
    targets: [
        .target(
            name: "ChatMessage",
            dependencies: ["Socket"]),
        .target(
            name: "chat-server",
            dependencies: ["Socket", "ChatMessage"]),
        .target(
            name: "chat-client",
            dependencies: ["Socket", "ChatMessage"]),
    ]
)
```

Figura 2: `Package.swift`

6. Estructura del código suministrado

Junto con el enunciado de esta práctica se proporciona también un *esqueleto* de código en formato Swift Package Manager que debe utilizarse para organizar el código de la práctica.

La estructura del paquete se define en el fichero **`Package.swift`**, que se reproduce en la figura 2. Consta de los siguientes componentes:

- La dependencia **`BlueSocket`**, que utilizarán tanto cliente como servidor para enviar y recibir datagramas UDP, tal como se ha visto en clase.
- Un *target* no ejecutable llamado **`ChatMessage`**, que incorpora la definición del tipo `ChatMessage` utilizado tanto por servidor como por cliente. Si se desea, puede añadirse funcionalidad adicional a este módulo. Por ejemplo, puede ser interesante añadir funciones para enviar y recibir los mensajes de comunicaciones descritos y utilizar esas funciones auxiliares tanto en el servidor como en el cliente.
- Un ejecutable llamado **`chat-server`**, donde se implementará el servidor descrito en esta práctica. El servidor necesita para funcionar dos dependencias: el paquete `BlueSocket` y el paquete `ChatMessage`. Ambos módulos deben importarse (utilizando `import`) en los puntos en los que sea necesario.
- Un ejecutable llamado **`chat-client`**, donde se implementará el cliente descrito en esta práctica. Al igual que en el caso del servidor, el cliente necesita los módulos `BlueSocket` y `ChatMessage`.

El código se proporciona como base a partir de la cual pueden construirse el servidor y el cliente. Ten en cuenta que **no compila** tal como está: es necesario, como mínimo, implementar las funciones del protocolo `ClientCollection` dentro del fichero `ClientCollectionArray`. Es necesario también implementar el código de comunicaciones necesario para enviar y recibir los mensajes que se describen en este enunciado.

7. Entrega

El límite para la entrega de esta práctica es: **Martes, 30 de Noviembre a las 23:55**. La entrega se realizará a través del aula virtual.