Informática II
Práctica 3

GSyC
Departamento de Teoría de la Señal y Comunicaciones y Sistemas Telemáticos y Computación

Enero de 2021

1. Introducción

En esta práctica debes mejorar el chat realizado según el modelo cliente/servidor de prácticas anteriores en los siguientes aspectos:

- Mejora en el tratamiento de errores de comunicaciones con una versión diferente de `DatagramReader`.
- Establecimiento de un 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 de clientes, el servidor elegirá al cliente que lleva más tiempo sin escribir en el chat, y lo expulsará para dejar sitio al nuevo cliente.
- Mantenimiento de 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.
- Utilización de estructuras de datos más sofisticadas para almacenar tanto los clientes activos como los inactivos.

Cada uno de estos puntos se detalla en el resto de secciones de este documento.

2. Tratamiento de errores con `DatagramReader`

En esta práctica debe usarse la implementación de `DatagramReader` que se muestra en la figura 1, y cuyo código se proporciona como parte del enunciado. La diferencia fundamental respecto a la versión anterior puede observarse en la línea 25 de la figura, donde se muestra que la invocación al bloque que debe procesar cada lectura se realiza mediante un tipo nuevo `DatagramReaderResult`, que está definido en las líneas 9 a 12. Se trata de un tipo enumerado que puede tomar uno de dos valores:

- `success` si la lectura ha sido correcta. En este caso, el tipo enumerado tiene asociados los valores referentes a la lectura: el *buffer*, el número de bytes recibidos y la dirección de origen.
- `error` si se ha producido un error y no se ha podido completar la lectura. Los errores, a su vez, pueden ser de dos tipos diferentes:
  - `.timeout`, si ha pasado el plazo límite establecido para lecturas en la biblioteca BlueSocket.
  - `.datagramError`, en caso de haberse producido un error de comunicaciones. En este caso, el error indicado por BlueSocket se recibirá en el dato `socketError` asociado al enumerado `.datagramError`.

El **programa servidor** debe utilizar esta nueva implementación y actuar del siguiente modo:

- En caso de recibirse un error de tipo `.timeout`, se ignorará y continuará la ejecución. Ese error es en principio inocuo: sólo nos indica que no se ha recibido ningún mensaje en el tiempo máximo que hayamos definido.
- En caso de que `DatagramReader` envíe el error `.datagramError`, el programa debe terminar e indicar el tipo de error que se ha producido.

Para conseguir que el programa termine cuando se produce un error, puede seguirse la siguiente estrategia:

- El servidor tendrá una variable `serverError`, de tipo `ChatServerError?`, que inicialmente será `nil`.

```swift
import Foundation
import Socket

public enum DatagramReaderError : Error {
    case timeout
    case datagramError(socketError: Error)
}

public enum DatagramReaderResult {
    case success(buffer: Data, bytesRead: Int, from: Socket.Address?)
    case error(DatagramReaderError)
}

/// Simple helper class to read forever from a socket using a background queue.
public class DatagramReader {
    func readDatagram(from socket: Socket, into buffer: inout Data) throws -> (bytesRead: Int, address: Socket.Address?) {
        let (bytesRead, address) = try socket.readDatagram(into: &buffer)
        if bytesRead == 0 && errno == EAGAIN {
            throw DatagramReaderError.timeout
        }
        return (bytesRead, address)
    }

    /** Creates a DatagramReader and read datagrams forever in a loop. */
    public init(socket: Socket, capacity: Int, handler: @escaping (DatagramReaderResult) -> Void) {
        var buffer = Data(capacity: capacity)

        let queue = DispatchQueue.global(qos: .userInteractive)
        queue.async {
            repeat {
                buffer.removeAll()
                do {
                    let (bytesRead, address) = try self.readDatagram(from: socket, into: &buffer)
                    // TODO: main queue, buffer copy
                    handler(.success(buffer: buffer, bytesRead: bytesRead, from: address))
                } catch DatagramReaderError.timeout {
                    handler(.error(.timeout))
                } catch {
                    handler(.error(.datagramError(socketError: error)))
                }
            } while true
        }
    }

    /** Use this version to ignore all errors and just receive the buffer on success. */
    public convenience init(socket: Socket, capacity: Int, handler: @escaping (Data) -> Void) {
        self.init(socket: socket, capacity: capacity) { (result: DatagramReaderResult) in
            if case .success(let buffer, _, _) = result {
                handler(buffer)
            }
        }
    }
}

```

Figura 1: `DatagramReader`

- Cuando se produzca un error, se asignará a dicha variable el error correspondiente.
- El bucle principal de `ChatServer` debe comprobar si la variable que almacena el error tiene un valor diferente a `nil`, y en ese caso lanzará una excepción (con ese mismo error) para terminar el programa. Puesto que en la práctica anterior el bucle principal lo único que hacía es esperar a leer una línea del teclado, no podría comprobarse la existencia de error hasta que el usuario haya escrito algo. Para evitar este problema, lo que haremos será *ejecutar el bucle de lectura de teclado en una cola asíncrona*, y crearemos en su lugar un bucle infinito que lo único que haga sea comprobar la existencia de error.

El siguiente fragmento muestra la función principal del servidor, que puede tomarse como modelo para el desarrollo. Su funcionamiento es el siguiente:

- La función `readAndProcessMessages()` es la que se encarga de lanzar la lectura asíncrona utilizando `DatagramReader`. Esta función puede lanzar una excepción en caso de que la creación del socket (con `listen`) falle. El resto de errores que puedan producirse durante la lectura asíncrona se almacenarán en la variable `serverError` mencionada anteriormente.
- La función `keyboardLoop()` **utilizará otra cola asíncrona** para leer del teclado en un bucle infinito, y procesar las instrucciones recibidas por el usuario.
- El bucle `repeat` se ejecutará mientras no se produzca un error. La llamada a `sleep(1)` espera 1 segundo antes de proseguir, puesto que no necesitamos tener la CPU 100 % ocupada en verificar si se ha producido el error en cada instante de tiempo.
- Por último, en el momento en que la variable `serverError` cambie de `nil` a cualquier otra cosa, se lanzará una excepción que será capturada por el programa principal, terminando su ejecución.

```
func run() throws {
  try readAndProcessMessages()
  keyboardLoop()

  repeat {
      sleep(1)
  } while serverError == nil

  throw serverError!
}
```

En el caso del cliente, se permite ignorar todos los errores, excepto el timeout inicial cuando se está intentando establecer la conexión. El funcionamiento, por tanto, será igual que en la práctica anterior.

3. 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
==============
carlos (127.0.1.1:13311): 02-Ene-21 20:01:07
ana (127.0.1.1:10313): 02-Ene-21 20:00:10
pablo (127.0.1.1:23025): 02-Ene-21 20:00:01

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

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

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

3.1. Implementación

El servidor deberá guardar en *una tabla de símbolos* 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 tabla de símbolos de clientes activos, al menos, su *nickname*, dirección y la hora a la que envió el último mensaje.

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 tabla de símbolos* 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` al `Client_EP_Handler` de todos los clientes conocidos 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 tabla de clientes activos y almacenará los datos correspondientes en la tabla 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` ahora debe incorporar también el nick del usuario, como se describe en la sección 5.

4. Tablas de símbolos

Los clientes activos y los inactivos se almacenarán en sendas tablas de símbolos, cuya especificación debe cumplir el protocolo que se muestra en la figura 2. Esta definición de tabla de símbolos es similar a la vista en clase, pero incorpora una variable `maxCapacity` para forzar un número máximo de elementos que se permite almacenar. En caso de que la función `put` deba crear espacio para un nuevo elemento pero se haya alcanzado la capacidad máxima, lanzará la excepción `maxCapacityReached`. **Importante: Si `put` no necesita crear un nuevo elemento, no se lanzará la excepción.**

4.1. Tabla de clientes activos

Los clientes activos se almacenarán en una tabla de símbolos implementada con un **árbol de búsqueda binaria genérico**, similar a los vistos en clase. Debe probarse exhaustivamente el funcionamiento de todas las funciones del protocolo 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.

La clave de búsqueda (y de ordenación) para esta tabla de símbolos será el *nick* del usuario, que es el campo que tenemos que usar para localizar los usuarios cuando llegan mensajes `Init`, `Writer` o `Logout`.

4.2. Tabla de clientes inactivos

Los clientes inactivos se almacenarán en otra tabla de símbolos que debe cumplir el mismo protocolo, pero en este caso se implementará mediante un **array ordenado con búsqueda binaria**, como se verá en clase. Del mismo modo, es necesario realizar un programa de pruebas específico e incorporarlo como parte de la entrega.

4.3. Reutilización de clientes antiguos

La tabla de clientes antiguos debe consultarse cada vez que un nuevo cliente trata de conectarse al chat. El funcionamiento debe ser el siguiente:

- Si el nick del nuevo cliente ya está registrado en la lista de **clientes activos**, el servidor denegará la conexión.
- Si el nick del nuevo cliente no está registrado en la lista de **clientes activos** pero sí aparece en la tabla de **clientes inactivos**, debe eliminarse de la lista de clientes inactivos y volver a incluirse en la de clientes activos. Este caso se considera una **reconexión**, evento que debe comunicarse a los otros clientes con un mensaje Server como el siguiente:

```
server: ana rejoins the chat
```

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

public protocol SymbolTable {
    associatedtype Key
    associatedtype Value

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

    func get(key: Key) -> Value?
    mutating func put(key: Key, value: Value) throws
    mutating func remove(key: Key) -> Value?

    func forEach(_ body: (Key, Value) throws -> Void) rethrows
}

```

Figura 2: `SymbolTable`

Observa el uso de **rejoins** en lugar de **joins**. El primero se utilizará para reconexiones, el segundo para conexiones normales en las que el nick no se encuentra ni en la tabla de clientes activos ni en la de inactivos.

5. Formato de los mensajes

Los cinco tipos de mensajes que se necesitan para esta práctica son los mismos que en la práctica anterior. Sin embargo, el protocolo de comunicaciones es diferente, pues en este caso se envía también el nick en los mensajes de tipo `Writer` y `Logout`. Esto es así para facilitar la búsqueda e inserción de elementos en las estructuras de datos que mantiene el servidor.

En esta sección se describen todos los mensajes por completitud, pero se marcan especialmente aquéllos que han sido modificados respecto a la práctica anterior.

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, 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. 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. Entrega

La entrega de esta práctica se hará a través del aula virtual, con límite: **1 de febrero de 2021 a las 23:59h**.