Laboratori DAT / 8

Índex

1 Introducció

El conjunt d'exercicis d'aquest sessió de laboratori (molts d'ells solucionats) són una preparació per entrendre conceptes necessaris per a la propera pràctica. Estan dividits en tres parts principals: ús de threads, ús d'un canal de broadcast i un exemple del servei d'eco.

2 Threads

Aquest primer conjunt de funcions i accions monàdiques intenten mostrar com finalitzar threads i com es comporta un thread quan és interromput. No pretenem donar les solucions més adequades, ja que per això és necessari conèixer en detall el tractament de les excepcions en Haskell i aquest aspecte queda fora de l'abast del curs.

Per a simular una tasca farem servir les següents funcions:

```
-- retardar s segons
retardar :: Int -> IO ()
retardar s = threadDelay $ s*1000000

tascaSegons :: Int -> IO ()
tascaSegons s = do
            atomicPutStrLn  $ "Inici tasca retardar " ++ (show s) ++ " segons."
            retardar s
```

2.1 Ús de `finally`

La següent acció monàdica realitza una tasca a continuació d'una altra. Quina és la sortida per pantalla?

```
sequencials2 :: IO ()
sequencials2 = do
               tascaSegons 4
               putStrLn  "finalitzada tasca 4 segons"
               do
                   putStrLn  "10 segons mes"
                   tascaSegons 10
                   putStrLn  "finalitzada tasca 10 segons mes"
```

Ara observem el que passa en presencia d'excepcions. Quina és ara la sortida per pantalla?

```
tascaExcepcio ::  IO ()
tascaExcepcio  = do
               error "Excepcio MEVA!!"
               putStrLn $ "NO surt!"


sequencials2Excepcio :: IO ()
sequencials2Excepcio =  do
                           tascaExcepcio
                           do
                               putStrLn  "10 segons mes"
                               tascaSegons 10
                               putStrLn  "finalitzada tasca 10 segons mes"
```

En l'acció monàdica anterior `sequencials2Excepcio`, seria de molta utilitat que es realitzessin les accions de la segona tasca ja que això permetria potser tractar l'excepció que s'ha produït en la primera. Per aconseguir això es pot utilitzar la funció `finally` que té el següent tipus:

```
finally
    :: IO a  -- computation to run first
    -> IO b  -- computation to run afterward (even if an exception was raised)
    -> IO a
```

Fent servir `finally` l'acció monàdica `sequencials2` es pot expressar com:

```
finallySequencials2 :: IO ()
finallySequencials2 = finally   (do
                                    tascaSegons 4
                                    putStrLn  "finalitzada tasca 4 segons"
                                )
                                (do
                                    putStrLn "10 segons mes"
                                    tascaSegons 10
                                    putStrLn  "finalitzada tasca 10 segons mes"
                                )
```

Aparentment, les accions monàdiques `sequencials2` i `finallySequencials2` són equivalents. La diferència és pot veure quan hi ha excepcions.

```
finallySequencials2Excepcio :: IO ()
finallySequencials2Excepcio = finally   tascaExcepcio

                                        (do
                                            putStrLn  "Ara s'executa"
                                            putStrLn  "10 segons mes"
                                            tascaSegons 10
                                            putStrLn  "finalitzada tasca 10 segons mes"
                                        )
```

Com s'observa ara s'executa la segona tasca **però es propaga l'excepció**.

Si es vol capturar l'exepció, es pot fer servir la funció `forkFinally` que té el següent tipus:

```
forkFinally :: IO a -> (Either SomeException a -> IO ()) -> IO ThreadId
```

i que en la documentació té la següent descripció:

> Fork a thread and call the supplied function when the thread is about to terminate, with an exception or a returned value.

Si volem evitar que l'excepció es propagui es pot utilitzar de la següent forma. Executa-la i intenta entendre el seu comportament:

```
forkFinallyExcepcio :: IO ()
forkFinallyExcepcio = do
                _ <- forkFinally    (tascaExcepcio)
                                    (\ _ -> do
                                        putStrLn $ "10 segons mes"
                                        tascaSegons 10
                                        putStrLn $ "finalitzada tasca 10 segons mes"
                                    )
                retardar 20
```

2.2 Finalització de threads

La següent acció monàdica mostra com finalitzar un thread des d'un altre thread. Intenta entendre bé les traces que es mostren a la pantalla un cop s'executa l'acció.

```
finalitzacioThreads :: IO ()
finalitzacioThreads = do
    -- Crear un thread
    idenThread1 <- forkIO $ do
                                tascaSegons 120
                                putStrLn "Fi tascaSegons 120 "
    -- quan acabi la seguent tasca
    -- fara finalitzar el thread idenThread1
    finally (do
```

```
        tascaSegons 10
        putStrLn "tascaSegons 10 acabada")
    (do
        putStrLn "finalitzar altre thread"
        killThread idenThread1
    )
```

2.3 Exercici

Implementar una acció monàdica amb dos Threads:

1. El primer thread demana entrar text per teclat i es bloqueja fins que l'usuari entra el text i després el mostra per pantalla.
2. El segon thread acaba el primer al cap de de 5 segons.

Executa el programa i intenta entendre la sortida.

3 Canal de broadcast

A continuació es mostra el funcionament d'un canal de broadcast basat en el tipus `Chan`. `Chan` és un tipus que representa un canal FIFO il·limitat.

El canal de broadcast disposa de les següents funcions que estan explicades en les figures.

```
type ChanW a = MVar [Chan a]
type ChanR a = Chan a

-- crea un nou canal de broadcast amb l'extrem d'escriptura
newBroadcastChanW :: IO (ChanW a)

-- afegeix un nou extrem de lectura
addChanR :: ChanW a -> IO (ChanR a)
-- treu un nou extrem de lectura
removeChanR :: ChanW a -> ChanR a -> IO ()

-- escriu en l'extrem d'escriptura
writeChanW :: ChanW a -> a -> IO ()
-- llegeix d'un extrem de lectura
readChanR :: ChanR a -> IO a
```

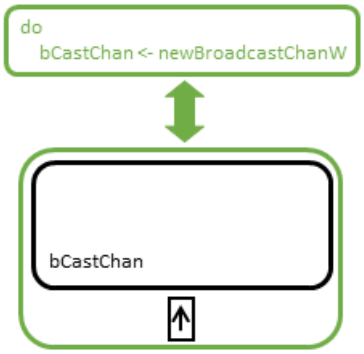

Figura 1: Crear Broadcast Chan

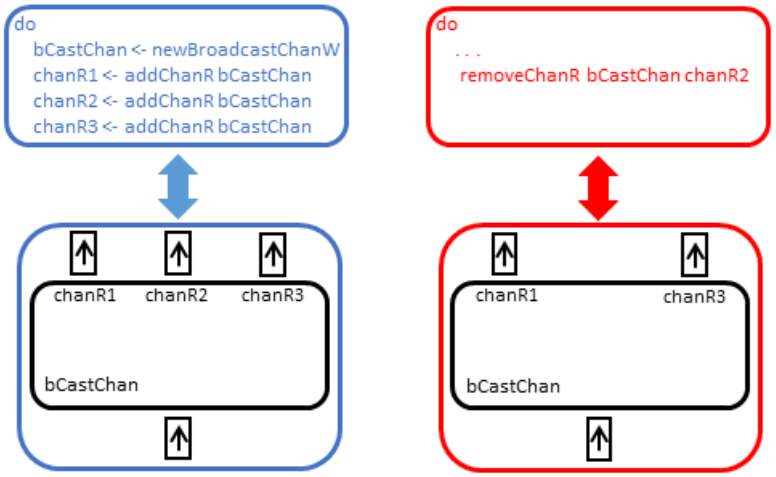

Figura 2: Afefgir i treure canals de lectura al canal de broadcast

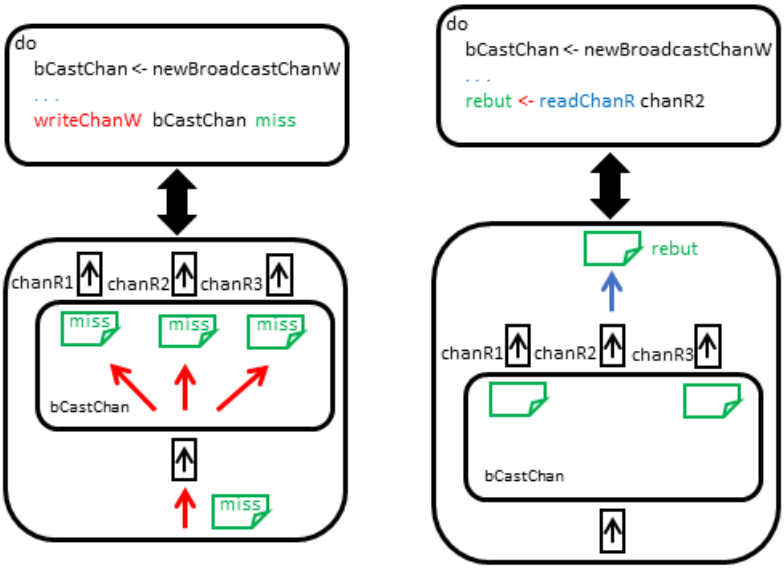

Figura 3: Enviar missatge com a broadcast i rebre missatge.

3.1 Exercici

A continuació es mostra una seqüència d'accions on dos emissors i dos receptors utilitzen el canal de broadcast. Cada caràcter enviat per un emissor serà rebut per tots dos receptors. Es dona fet el codi del programa principal i de l'emissor i es demana que es faci el del receptor.

```
provaChan :: IO ()
provaChan = do
    bCastChan <- newBroadcastChanW
    let dades = "abcde"
    chanR1 <- addChanR bCastChan
    forkIO $ receptor 0 (2*(length dades)) chanR1
    chanR2 <- addChanR bCastChan
    forkIO $ receptor 1 (2*(length dades)) chanR2
    retardar 10
    forkIO $ emissor 0 dades bCastChan
    forkIO $ emissor 1 dades bCastChan
    pure ()


emissor ::  (Show a) => Int -> [a] -> ChanW a  ->  IO ()
emissor  id [] chn    = pure ()
emissor  id (x:xs) chn    = do
                temps <- randomRIO (1::Int, 6)
                retardar temps
                writeChanW chn x
                atomicPutStrLn $ "Enviat < " ++ (show id) ++ " > : " ++ show x
                -- loop back
                emissor id xs chn


receptor :: (Show a) => Int -> Int -> ChanR a -> IO ()
-- completar
```

4 Servei Eco

Finalment es proposa implementar un servei d'eco seguint el model client-servidor. Per això es proporciona el següent codi pel servidor:

```
servidorEco :: IO ()
servidorEco = do
                let port = 3000 :: PortNumber
                srvrSckt <-  nouServerSocket port
                infoLog $ "Escoltant pel port " <> show port
                bucleAccept srvrSckt tascaEcoServidor

-- accept :: Socket -> IO (Socket, SockAddr)
bucleAccept :: Socket -> (Socket -> IO a) -> IO a
bucleAccept serverSocket tasca  = do
        (socket, adreRemota) <- accept serverSocket
        forkFinally (tasca socket)
                    (\_ -> do
                        infoLog $ "Connexió acabada desde " <> show adreRemota
                        tancarSocket socket
                    )
        bucleAccept serverSocket tasca


tascaEcoServidor :: Socket -> IO ()
tascaEcoServidor sc = do
        sendAll sc $ pack "Entra text: "
        miss <- recv sc 1024
        let missString = unpack miss
        infoLog $ "Rebut -> " ++ missString
        sendAll sc miss
        if missString == "/fi"  then do
                                        infoLog $ "Acabant connexio ... "
                                        pure ()
                                else do
                                        tascaEcoServidor sc
```

i el següent codi pel client:

```
tascaEcoClient :: Socket -> IO ()
tascaEcoClient sc = bucle
    where bucle = do
                    queFer <- recv sc 1024
                    putStrLn $ unpack queFer
                    txt <- getLine
                    sendAll sc (pack txt)
                    putStrLn ""
                    miss <- recv sc 1024
                    let missRebut = unpack miss
                    putStrLn $ "Rebut -> " ++ missRebut
                    if missRebut == "/fi" then do
                                               tancarSocket sc
                                               infoLog "Client finalitzat"
                                               pure ()
                                          else bucle
```

Com es pot observar, per a cada petició de connexió el servidor crea un nou thread per atendre sol·licituds d'eco. Aquest thread executa una tasca que crida la funció `recv` que té semàntica bloquejant i espera llegir d'un socket. Un cop la funció `recv` completa, el servidor reenvia el que ha rebut mitjançant la funció `sendAll`.

Els missatges que comencen pel caràcter `'/'` s'interpreten com a missatges de control. En el nostre cas el missatege `/fi` serveix per finalitzar la connexió.

4.1 Exercici multifil

El client proporcionat utilitza un sol thread d'execució. Es demana canviar el disseny a un client amb dos threads tal i com es mostra a la Figura 4. S'observa que en els threads s'executen les següents tasques:

1. `tascaXarxa`: Espera a llegir del socket i mostra el que ha llegit per pantalla.
2. `tascaTeclat`: Espera a llegir del teclat i envia el que s'ha escrit pel socket.

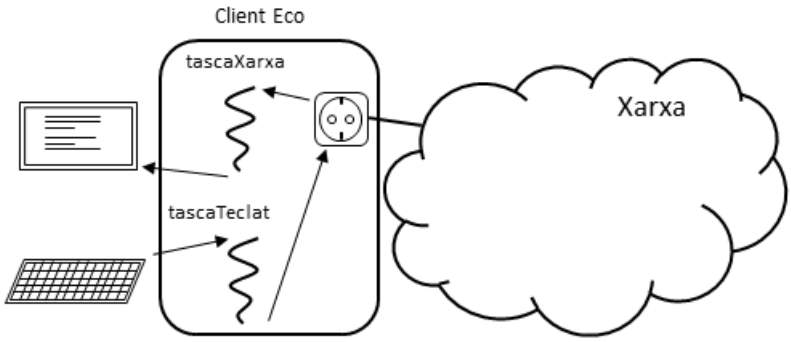

Figura 4: Client Eco

4.2 Exercici protocol de tancament

En una aplicació multithread la finalització dels threads no és trivial. Per tancar els threads que executen `tascaXarxa` i `tascaTeclat`, de manera ordenada es proposa el protocol de tancament de la Figura 5 que haureu d'implementar. Consisteix en els següents passos:

1. L'usuari escriu `/fi` des del teclat.
2. La `tascaTeclat`, en llegir `/fi`, l'envia pel socket
3. i acaba.
4. La `tascaXarxa` rep `/fi` des del socket,
5. mostra `/fi` per pantalla,
6. tanca el socket
7. i acaba.

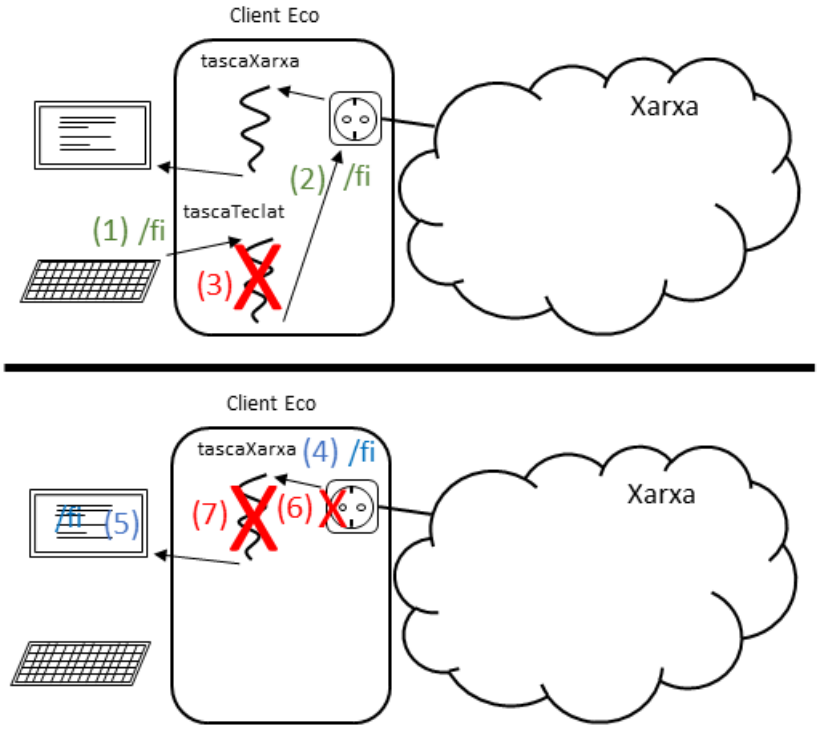

Figura 5: Tancament Client

4.3 Exercici `nick`

En aquest exercici es proposa que abans de començar el servei d'eco, el servidor demani al client que entri un `nick`. Per aquest motiu s'introdueix el tipus de dades `Usuari`, definit de la següent manera:

```
-- Dades Usuari

data Usuari = Usuari
        { nick :: Nick
        , socketUsr :: Socket
        }
```

i representat en la Figura 6

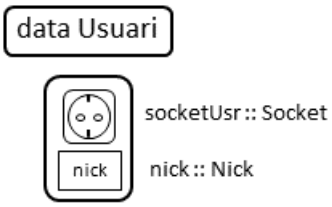

Figura 6: Usuari

A partir d'aquest moment, en els intercanvis entre el client i el servidor, el servidor es dirigirà al client mitjançant el seu `nick`, tal i com es pot veure en la següent captura de pantalla del client:

```
> Entra nick:
Cervantes
> Cervantes entra text:
En un lugar de la Mancha ...
> En un lugar de la Mancha ...
> Cervantes entra text:
```

```
/fi
> /fi
DEBUG: Socket tancat
INFO: Teclat tancat
INFO: Client finalitzat
```

El `nick` també pot servir per tenir traces més entenedores, com s'observa en la següent captura de pantalla (del servidor):

```
INFO: Escoltant pel port 3000
INFO: Nick rebut ->  < Cervantes >
INFO: Rebut de < Cervantes > : En un lugar de la Mancha ...
INFO: Rebut de < Cervantes > : /fi
INFO: Acabant connexio amb Cervantes
INFO: Connexió acabada desde 127.0.0.1:57428
DEBUG: Socket tancat
```

Com es pot veure en el codi proporcionat (que es mostra a continuació), la tasca del servidor d'eco es divideix en una tasca per obtenir el `nick` i una altra tasca que fa eco repetidament.

```
tascaEcoServidorNick :: Socket -> IO ()
tascaEcoServidorNick sc = do
    usuari <- obtenirNick sc
    bucleEco usuari

obtenirNick :: Socket -> IO Usuari
obtenirNick sc = undefined

bucleEco :: Usuari -> IO ()
bucleEco usuari = undefined
```

4.4 Exercici `nick` únic

Ara es proposa que el `nick` sigui únic, és a dir, no hi poden haver dos o més clients remots amb el mateix `nick` utilitzant el servei d'eco. El comportament ha de com es mostra en les següents traces d'execució. Les traces d'aquest client

```
ghci> clientGeneral
> Entra nick:
Cervantes
> Cervantes entra text:
En un lugar de la Mancha ..
> En un lugar de la Mancha ..
> Cervantes entra text:
```

i aquest altre client

```
ghci> clientGeneral
> Entra nick:
Cervantes
> El nick Cervantes ja es fa servir, tria'n un altre
> Entra nick:
Lope de Vega
> Lope de Vega entra text:
El amor tiene facil la entrada y dificil la salida.
> El amor tiene facil la entrada y dificil la salida.
> Lope de Vega entra text:
```

es corresponen amb les següents traces en el servidor

```
ghci> servidorEcoNickUnic
INFO: Escoltant pel port 3000
INFO: Nick rebut ->  < Cervantes >
DEBUG: Usuari Cervantes afegit
INFO: Nick rebut ->  < Cervantes >
DEBUG: Usuari Cervantes ja existeix
INFO: Nick rebut ->  < Lope de Vega >
DEBUG: Usuari Lope de Vega afegit
INFO: Rebut de < Cervantes > : En un lugar de la Mancha ...
INFO: Rebut de < Lope de Vega > : El amor tiene facil la entrada y dificil la salida.
```

El fet de no permetre nicks repetits serveix per veure com es pot introduir *estat* en el servidor. Degut a que el servidor és multifil i que per tant l'estat serà accedit per varis threads alhora, s'hauran d'utilitzar mecanismes de concurrència, que en aquest curs són variables `MVars`. Observeu el següent tipus de dades:

```
-- Dades Servidor

type Nick = String

data Servidor = Servidor { connectats :: MVar (M.Map Nick Usuari) }

crearServ :: IO Servidor
crearServ = do
    connectats <- newMVar M.empty
    pure $ Servidor connectats
```

La Figura 7 mostra una representació gràfica del tipus de dades `Servidor`.

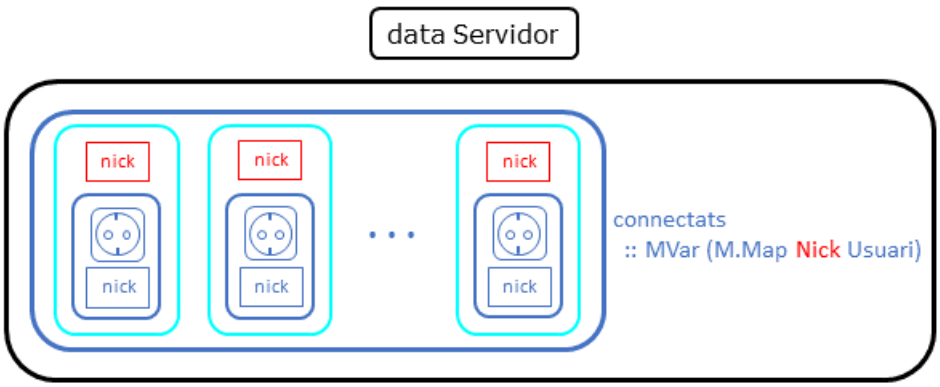

Figura 7: Servidor

Com es pot veure l'estat del servidor es manté en una llista de parelles `Nick`, `Usuari`. On `Usuari` està definit com en la Figura 6 de la Secció 4.3.

Potser pot semblar que s'utilitza el tipus `Nick` de manera redundant, però fer-ho d'aquesta manera aplana el camí cap a transformar el servidor d'eco a un servidor de xat.

Segurament també serà d'utilitat implementar les següents funcions:

```
-- intenta afegir un nou usuari al mapa del servidor
-- La funció té com a resultat una acció que produeix:
--  - Just usuari, si no hi cap usuari amb nick 'nom'
--  - Nothing, si ja hi ha un usuari amb nick 'nom'
afegirUsuari :: Nick -> Socket -> Servidor -> IO (Maybe Usuari)
afegirUsuari nom socket servidor = undefined


-- treu l'usuari del mapa del servidor
treureUsuari :: Usuari -> Servidor -> IO ()
treureUsuari usuari servidor = undefined
```

L'estructura de codi que es proposa és una continuació del de la Secció 4.3:

```
obtenirNickUnic :: Servidor -> Socket -> IO Usuari
obtenirNickUnic servidor sc = undefined

tascaEcoServidorNickUnic ::  Servidor -> Socket ->IO ()
tascaEcoServidorNickUnic servidor sc  = undefined
```

D'aquesta manera en la funció `tascaEcoServidorNickUnic` podeu reutilitzar la funció `bucleEco` de la Secció 4.3.