Laboratori SAD / 9

Índex

1 Introducció

Es proposa l'implementació d'un servidor de xat. La major part de la funcionalitat necessària ja l'heu implementat en el servidor d'eco. És per això, que esencialment el que s'ha de fer és afegir un canal de broadcast en el servidor d'eco, com es pot veure comparant la Figura 1 amb la figura del servidor d'eco de la pràctica anterior. Recordeu que el canal de broadcast també ha sigut estudiat en la pràctica previa. El tipus `Servidor` està definit com:

```
data Servidor = Servidor
        { connectats :: MVar (M.Map Nick Usuari)
        , canalEscriptura :: ChanW MissatgeUsuari
        }

crearServ :: IO Servidor
crearServ = do
    conectats <- newMVar  M.empty
    chanBroadcast <- newBroadcastChanW
    pure $ Servidor conectats chanBroadcast
```

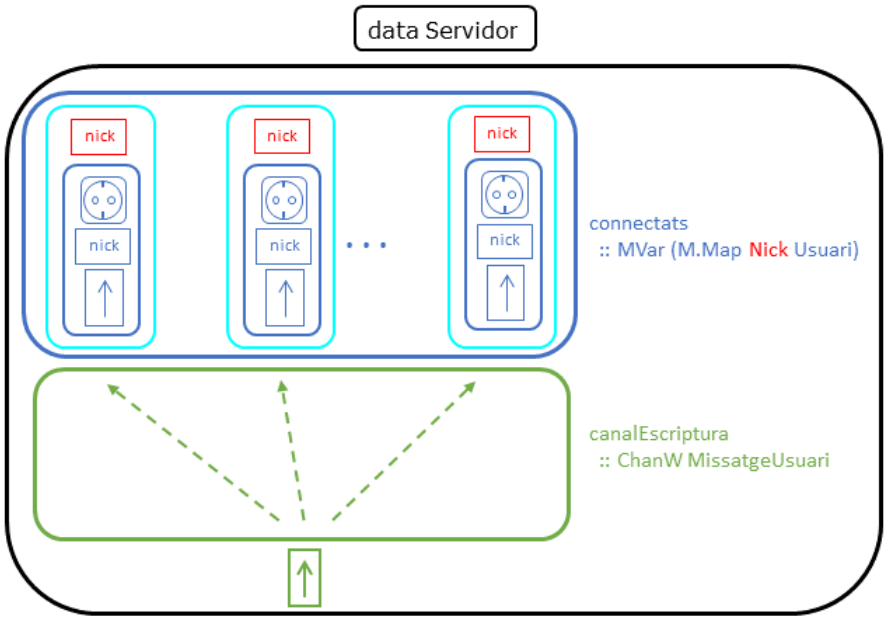

Figura 1: Tipus de dades Servidor (chat)

Com s'observa en la Figura 1, el tipus de dades `Usuari` també s'ha modificat respecte a la pràctica anterior i s'hi ha afegit un extrem de lectura del canal de broadcast. Recordeu que el tipus `Usuari` fa de "intermediari" entre el servidor i cada client remot.

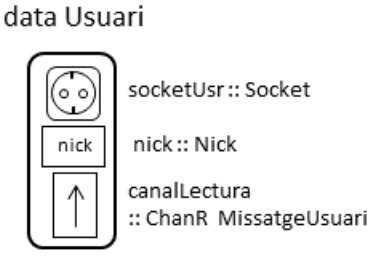

Figura 2: Tipus de dades Usuari

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

A més a més, es proposa d'utilitzar el següent tipus de dades pels missatges del servidor cap a al client remot:

```
-- Missatges del Servidor cap al client remot
data MissatgeUsuari
    = MissNotif String
    | MissChat Nick String
    deriving (Eq)

instance Show MissatgeUsuari where
  show (MissChat desde miss) = "<" <> desde <> ">: " <> miss
  show (MissNotif miss) = "*** " <> miss
```

2 Servei de Xat. Part del servidor.

2.1 Inicialització del servidor

També es donen fetes una funció per inicialitzar el servidor i una altre funció per acceptar connexions en un servidor multithread (com és el servidor de xat). Aquestes funcions són essencialment les mateixes funcions que en el servidor d'eco:

```
servidorChat :: IO ()
servidorChat = do
        let port = 3000 :: PortNumber
        srvrSckt <-  nouServerSocket port
        infoLog $ "Escoltant pel port " <> show port
        servidor <- crearServ
        bucleAccept srvrSckt (tascaChat servidor)

-- accept :: Socket -> IO (Socket, SockAddr)
bucleAccept :: Socket -> (Socket -> IO a) -> IO a
bucleAccept serverSocket tasca  = do
        (socket, adreRemota) <- accept serverSocket
        infoLog $ "Connexió acceptada desde " <> show adreRemota
        -- Crear un thread que executi la tascaChat i que en acabar
        -- la tascaChat tanqui el socket
        forkFinally (tasca socket)
                    (\_ -> do
                        infoLog $ "Connexió acabada desde " <> show adreRemota
                        tancarSocket socket
                    )
        bucleAccept serverSocket tasca
```

2.2 Gestió d'usuaris

Respecte al servidor d'eco que ja s'ha implementat i que no permet `nicks` repetits, s'hauran de fer modificacions en les funcions `afegirUsuari` i `treureUsuari`, tenint en compte que ara hi ha un canal de broadcast en el servidor.

```
afegirUsuari :: Nick -> Socket -> Servidor -> IO (Maybe Usuari)

treureUsuari :: Usuari -> Servidor -> IO ()
```

El mateix passa amb la funció:

```
obtenirNick :: Socket -> IO Usuari
```

ja que el tipus `Usuari` ha canviat respecte a la pràctica anterior.

2.3 Canal de Broadcast

Per a l'acció de broadcast, serà d'utilitat implementar les següents funcions :

```
-- escriure missatge pel canal de broadcast
broadcast :: MissatgeUsuari -> Servidor -> IO ()

-- llegir missatge de l'extrem de lectura de l'usuari
esperarMissatge :: Usuari -> IO MissatgeUsuari
```

2.4 Tasques i threads

En la solució que es proposa, associats a cada element de tipus `Usuari`, hi ha dos threads que executen dues tasques de manera concurrent (veure Figura 3). Les tasques són:

- `tascaMissAltres`: Tasca per esperar a llegir missatges que ALTRES usuaris han posat al canal de broadcast i quan arriba un missatge enviar-lo al client remot, i així succesivament.
- `tascaMissPropi`: Tasca que espera rebre missatges del client remot associat a l'usuari i un cop rebut el missatge en fa broadcast i així succesivament fins que es rep "`/fi`" i la tasca acaba.

Observar que un cop acaba la `tascaMissPropi` *també* ha d'acabar la `tascaMissAltres`.

Aquestes tasques s'executen dins d'una tasca més general que s'anomena `tascaChat`.

```
tascaChat ::  Servidor -> Socket ->IO ()
```

En la `tascaChat` primer s'ha d'obtenir un nick (que serà únic) del client remot i després executar `tascaBroadcast`. Quan la `tascaBroadcast` acaba s'ha de treure l'usuari de la llista de connectats.

Per obtenir el nick, podeu reutilitzat la funció

```
obtenirNickUnic :: Servidor -> Socket -> IO Usuari
```

que ja es va implementar pel servidor d'eco.

2.4.1 Lògica de l'aplicació

En la Figura 3 es mostra el flux d'execució d'una acció de broadcast. L'acció de broadcast l'implementen concurrentment les tasques `tascaMissPropi` i `tascaMissAltres`:

1. Arriba un missatge d'un client remot.
2. Aquest missatge es llegit en la `tascaMissPropi` mitjançant el socket de l'`Usuari` associat al client remot.
3. Desde de la `tascaMissPropi` s'utilitza la funció `broadcast` per posar en missatge en l'extrem d'escriptura del canal de broadcast.
4. El canal de broadcast fa arribar el missatge a cada extrem de lectura.
5. Quan el missatge està disponible en l'extrem de lectura, es desbloquejen les `tascaMissAltres` i es llegeix el missatge en cada una d'elles.
6. Les `tascaMissAltres` utilitzen el socket del seu `Usuari` per
7. enviar el missatge a cada client remot.

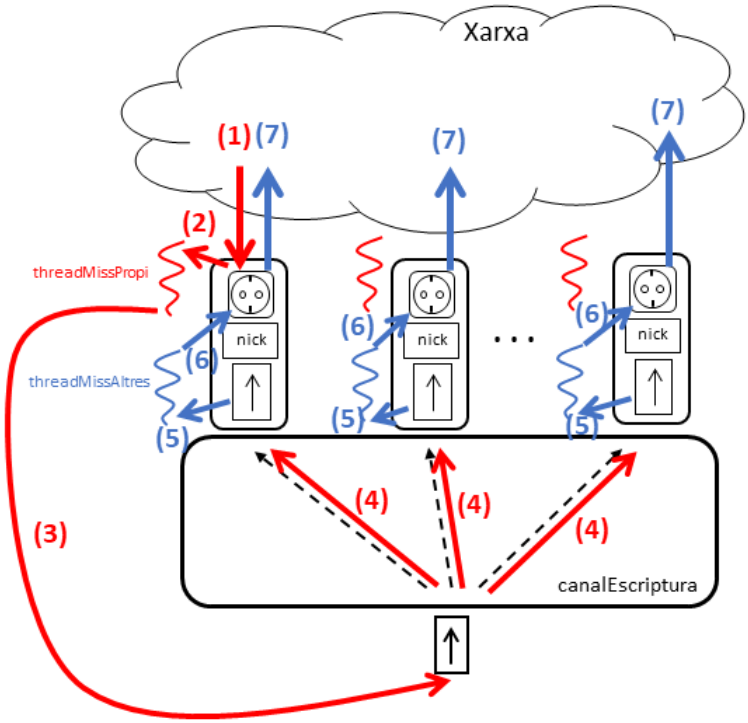

Figura 3: Lògica de l'aplicació chat