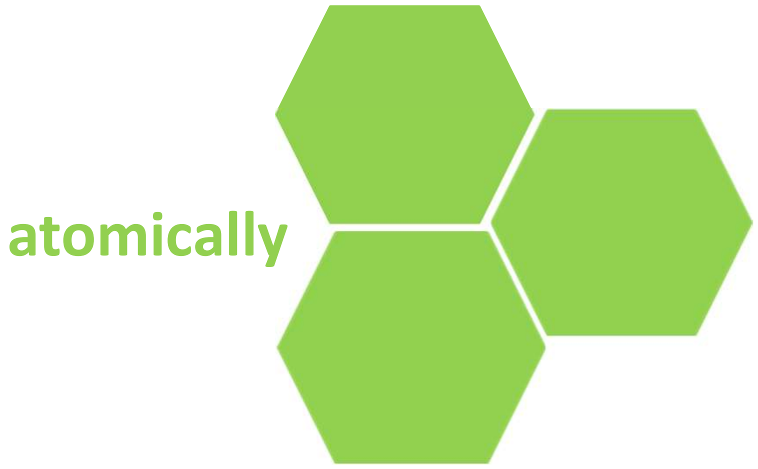
IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

Software Transactional Memory

Ioana Leustean

Simon Peyton Jones, Beautiful Concurrency

PCPH, Cap. 10
S.Marlow



Haskell

- Canale de comunicare: canale implementate cu MVar

readTChan
se blocheaza
cand canalul
este gol

writeTChan
nu se blocheaza
niciodata

```
import Control.Concurrent.STM

newTChan :: STM (TChan a)

writeTChan :: TChan a -> a -> STM ()

readTChan :: TChan  a -> STM a
```

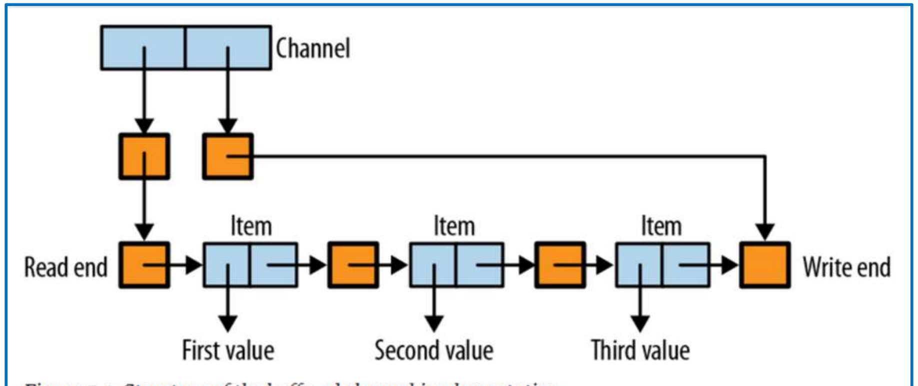


Figure 7-1. Structure of the buffered channel implementation

- in IO cu MVar

```
type Stream a = MVar (Item a)
data Item a   = Item a (Stream a)
data Chan a = Chan (MVar (Stream a)) (MVar (Stream a))
```

- Canal in STM

```
data TChan a = TChan (TVar (TVarList a))   (TVar (TVarList a))
type TVarList a = TVar (TList a)
data TList a = TNil | TCons a (TVarList a)
```

- Canal in STM

```
data TChan a = TChan  (TVar (TVarList a))    (TVar (TVarList a))
type TVarList a = TVar (TList a)
data TList a = TNil | TCons a (TVarList a)

newTChan :: STM (TChan a)
newTChan = do
            hole <- newTVar TNil
            read <- newTVar hole
            write <- newTVar hole
            return (TChan read write)
```

- Canal in STM

```
data TChan a = TChan (TVar (TVarList a))    (TVar (TVarList a))
type TVarList a = TVar (TList a)
data TList a = TNil | TCons a (TVarList a)

readTChan :: TChan a -> STM a
readTChan (TChan readVar _) = do
                               listHead <- readTVar readVar
                               head <- readTVar listHead
                               case head of
                                        TNil -> retry
                                        TCons val tail -> do
                                                         writeTVar readVar tail
                                                         return val
```

- Canal in STM

```
data TChan a = TChan (TVar (TVarList a))    (TVar (TVarList a))
type TVarList a = TVar (TList a)
data TList a = TNil | TCons a (TVarList a)


writeTChan :: TChan a -> a -> STM ()
writeTChan (TChan _ writeVar) a = do
                                        newListEnd <- newTVar TNil
                                        listEnd <- readTVar writeVar
                                        writeTVar writeVar newListEnd
                                        writeTVar listEnd (TCons a newListEnd)
```

- Canal in STM

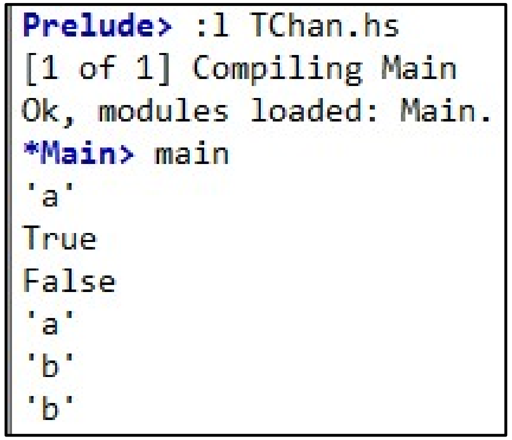
```
unGetTChan :: TChan a -> a -> STM ()
unGetTChan (TChan readVar _) a = do
                       listHead <- readTVar readVar
                       newHead <- newTVar (TCons a listHead)
                       writeTVar readVar newHead
```

unGetTChan este inversa lui readTChan; cand canalul este gol un thread poate chema unGetTChan pentru a debloca capatul de citire.

```
isEmptyTChan :: TChan a -> STM Bool
isEmptyTChan (TChan read _) = do
           listhead <- readTVar read
           head <- readTVar listhead
           case head of
                 TNil -> return True
                 TCons _ _ -> return False
```

- Canal in STM

```
main = do
          c <- atomically $ newTChan
          atomically $ writeTChan c 'a'
          atomically (readTChan c) >>= print
          atomically (isEmptyTChan c) >>= print
          atomically $ unGetTChan c 'a'
          atomically (isEmptyTChan c) >>= print
          atomically (readTChan c) >>= print
           c2 <- atomically $ dupTChan c
           atomically $ writeTChan c 'b'
           atomically (readTChan c) >>= print
           atomically (readTChan c2) >>= print
```



```
Prelude> :l TChan.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> main
'a'
True
False
'a'
'b'
'b'
```

TChan.hs ©2012, Simon Marlow

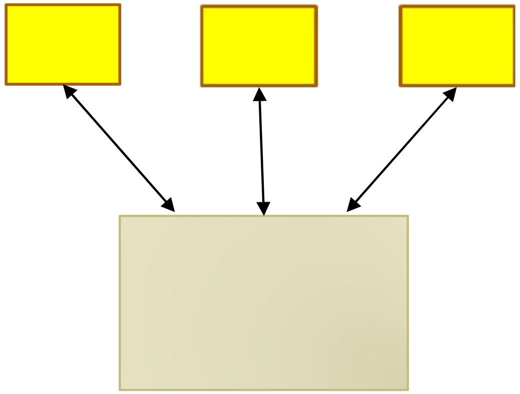
- Canale implementate ca liste
  - variabilele TVAr pot fi accesate fara blocarea
  - blocarea se poate face oricand este necesar folosind retry

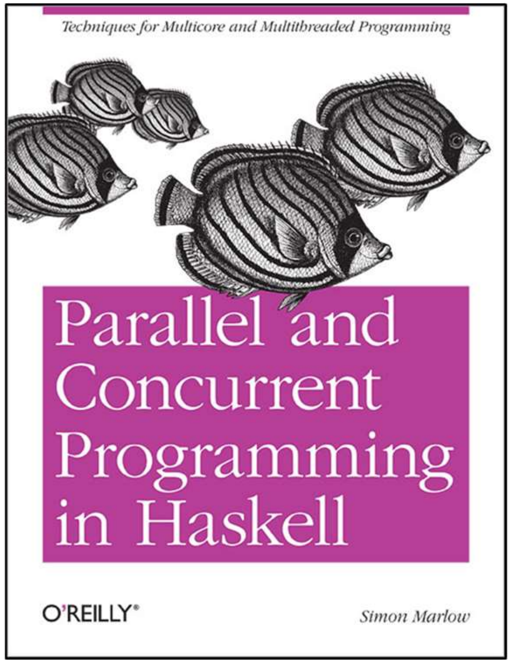
```
newtype TList a = TList (TVar [a])

newTList :: STM (TList a)
newTList = do
              v  <- newTVar []
             return (TList v)


writeTList :: TList a -> a -> STM ()
writeTList (TList v) a = do
               list <- readTVar v
               writeTVar v (list ++ [a])
```

```
isEmptyTList :: TBList a -> STM Bool
isEmptyTList (TBList cap v) = do
                                    list <- readTVar v
                                    return (null list)
```

```
readTList :: TList a -> STM a
readTList (TList v) = do
                        xs <- readTVar v
                        case xs of
                            []       -> retry
                            (x:xs') -> do
                                          writeTVar v xs'
                                          return x
```

Implementarea unei aplicatii de tip SERVER




Techniques for Multicore and Multithreaded Programming
Parallel and Concurrent Programming in Haskell
O'REILLY
Simon Marlow

[Part II. Concurrent Haskell
Cap. 12]

- System.IO

O data de tip Handle este o valoare extrasa dintr-o actiune IO asupra fisierului curent

data Handle

"Haskell defines operations to read and write characters from and to files,
represented by values of type Handle. Each value of this type is a handle:
a record used by the Haskell run-time system to manage I/O with file system objects.

A handle has at least the following properties:
whether it manages input or output or both;
whether it is open, closed or semi-closed;
…

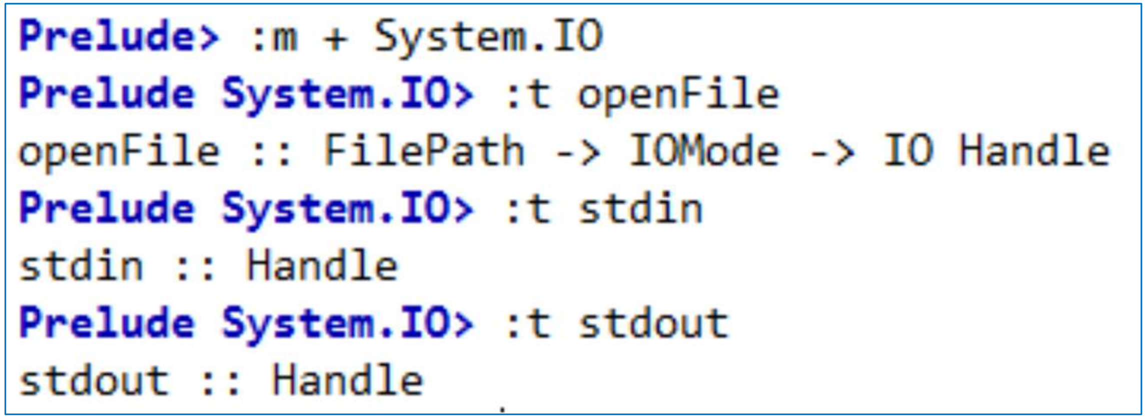
https://downloads.haskell.org/~ghc/6.2.1/docs/html/libraries/base/System.IO.html

https://hackage.haskell.org/package/base-4.18.0.0/docs/GHC-IO-Handle-FD.html

- System.IO

O data de tip Handle este o valoare extrasa dintr-o actiune IO asupra   fisierului curent

data Handle



```
Prelude> :m + System.IO
Prelude System.IO> :t openFile
openFile :: FilePath -> IOMode -> IO Handle
Prelude System.IO> :t stdin
stdin :: Handle
Prelude System.IO> :t stdout
stdout :: Handle
```

```
data Handle
type FilePath =  String
data IOMode   =  ReadMode| WriteMode|
                 AppendMode| ReadWriteMode
```

```
hdl <- openFile "fis.txt" ReadMode
hclose hdl
```

https://hackage.haskell.org/package/base-4.18.0.0/docs/GHC-IO-Handle-FD.html

- ➢ System.IO

```
import System.IO


exio1 = do
        hdl1 <- openFile "f1.txt" ReadMode
        hdl2 <- openFile "f2.txt" AppendMode
        s  <- hGetContents hdl1
        putStrLn s
        hPutStr hdl2 s
        hClose hdl1
        hClose hdl2
```

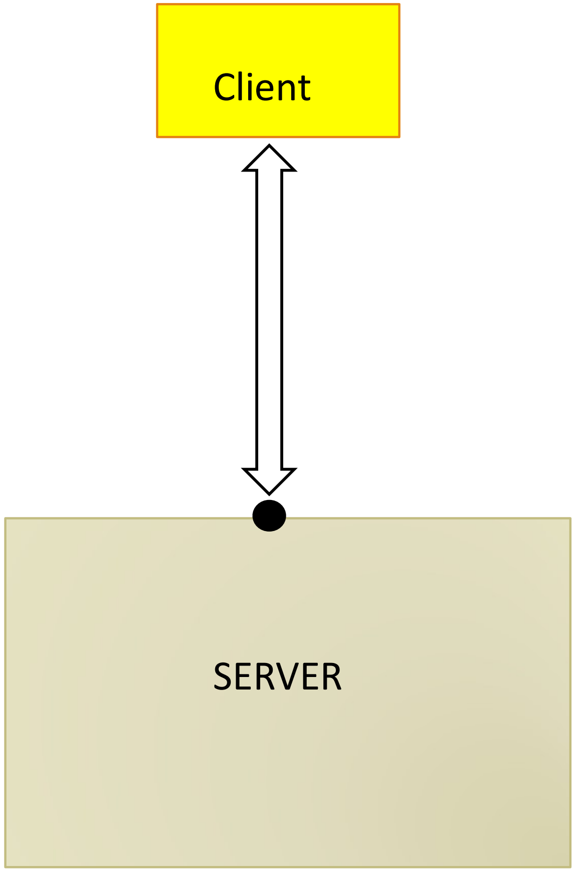
- System.IO

```
import System.IO


exio1 = do
      hdl1 <- openFile "f1.txt" ReadMode
      hdl2 <- openFile "f2.txt" AppendMode
      s  <- hGetContents hdl1
      putStrLn s
      hPutStr hdl2 s
      hClose hdl1
      hClose hdl2
```

```
exio2 = do
      s <- readFile "f1.txt"
      putStrLn s
      writeFile "f2.txt" s
```

```
readFile        :: FilePath -> IO String
readFile name   =  openFile name ReadMode >>= hGetContents
```

https://hackage.haskell.org/package/base-4.18.0.0/docs/src/System.IO.html#readFile



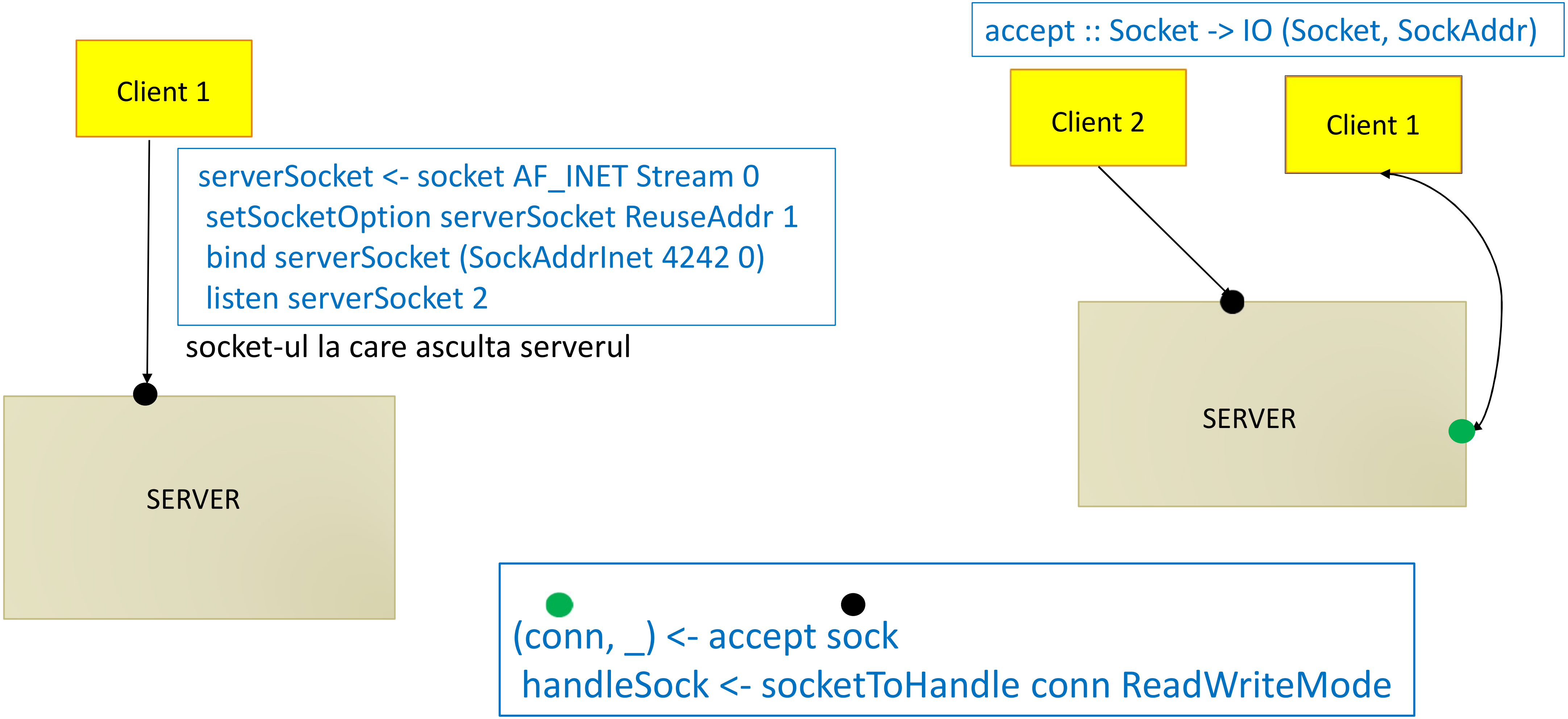
● socket

- un socket (soclu) este un punct final in comunicarea bidirectionala dintre doua programe din aceeasi retea
- un socket are asociat un port

https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html

```
serverSocket <- socket AF_INET Stream 0
setSocketOption serverSocket ReuseAddr 1
bind serverSocket (SockAddrInet 4242 0)
listen serverSocket 2
```

https://hackage.haskell.org/package/network-3.1.2.9/docs/Network-Socket.html



```
serverSocket <- socket AF_INET Stream 0
 setSocketOption serverSocket ReuseAddr 1
 bind serverSocket (SockAddrInet 4242 0)
 listen serverSocket 2
```

socket-ul la care asculta serverul

```
accept :: Socket -> IO (Socket, SockAddr)
```

● ●

```
(conn, _) <- accept sock
 handleSock <- socketToHandle conn ReadWriteMode
```

clientul va scrie/citi folosind `handleSock :: Handle`

SERVER socket

```
main = do
  sock <- socket AF_INET Stream 0
  setSocketOption sock ReuseAddr 1
  bind sock (SockAddrInet 4242 0)
  listen sock 2
  putStrLn "Listening on port 4242..."
  loopForever sock
```

stabilirea conexiunilor CLIENT

```
loopForever :: Socket -> IO ()
loopForever sock = do
            (conn, _) <- accept sock
            handleSock <- socketToHandle conn  ReadWriteMode
            line <- hGetLine handleSock
            putStrLn $ "Request received: " ++ line
            hPutStrLn handleSock $ "Hey, client!"
            hClose handleSock
            loopForever sock
```

https://dev.to/leandronsp/a-crud-journey-in-haskell-part-ii-socket-programming-2po1

```
main = do
  sock <- socket AF_INET Stream 0
  setSocketOption sock ReuseAddr 1
  bind sock (SockAddrInet 4242 0)
  listen sock 2
  putStrLn "Listening on port 4242..."
  loopForever sock
```

```
loopForever :: Socket -> IO ()
loopForever sock = do
              (conn, _) <- accept sock
              handleSock <- socketToHandle conn  ReadWriteMode
              line <- hGetLine handleSock
              putStrLn $ "Request received: " ++ line
              hPutStrLn handleSock $ "Hey, client!"
              hClose handleSock
              loopForever sock
```

Server

```
*Main> main
Listening on port 4242...
Request received: Ioana
Request received: Ana
```

Client 1

```
C:\Users\igleu\nc>nc64 localhost 4242
Ioana
Hey, client!
```

Client 2

```
C:\Users\igleu>telnet localhost 4242
Ana
Hey, client!
```

https://dev.to/leandronsp/a-crud-journey-in-haskell-part-ii-socket-programming-2po1

```
main = do
  sock <- socket AF_INET Stream 0
  setSocketOption sock ReuseAddr 1
  bind sock (SockAddrInet 4242 0)
  listen sock 2
  putStrLn "Listening on port 4242..."
  loopForever sock
```

```
loopForever :: Socket -> IO ()
loopForever sock = do
          (conn, _) <- accept sock
          handleSock <- socketToHandle conn  ReadWriteMode
          line <- hGetLine handleSock
          loopClient handleSock   -- interactiunea cu clientul
          loopForever sock
```

Handle pentru client ←

```
main = do
  sock <- socket AF_INET Stream 0
  setSocketOption sock ReuseAddr 1
  bind sock (SockAddrInet 4242 0)
  listen sock 2
  putStrLn "Listening on port 4242..."
  loopForever sock
```

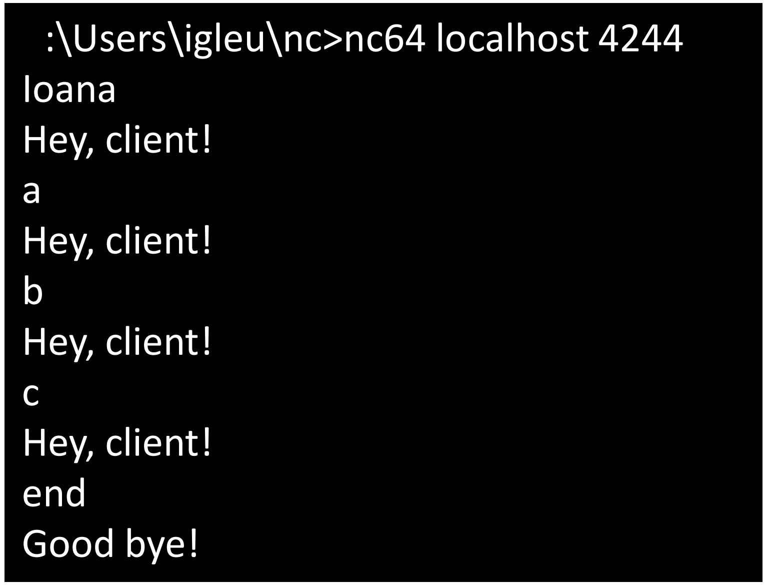
```
loopForever :: Socket -> IO ()
loopForever sock = do
            (conn, _) <- accept sock
            handleSock <- socketToHandle conn  ReadWriteMode
            line <- hGetLine handleSock
            loopClient handleSock   -- interactiunea cu clientul
            loopForever sock
```

```
loopclient handleSock = do
                    line <- hGetLine handleSock
                    if line == "end"
                         then do
                              hPutStrLn handleSock ("Good bye!")
                              hClose handleSock
                         else do
                              putStrLn $ "Request received from: " ++ line
                              hPutStrLn handleSock $ "Hey, client!"
                              loopclient handleSock
```

```
main = do
  sock <- socket AF_INET Stream 0
  setSocketOption sock ReuseAddr 1
  bind sock (SockAddrInet 4244 0)
  listen sock 2
  putStrLn "Listening on port 4244..."
  loopForever sock
```

```
loopForever :: Socket -> IO ()
loopForever sock = do
              (conn, _) <- accept sock
              handleSock <- socketToHandle conn  ReadWriteMode
              line <- hGetLine handleSock
              loopClient handleSock   -- interactiunea cu clientul
              loopForever sock
```

```
loopclient handleSock = do
               line <- hGetLine handleSock
               if line == "end"
                     then do
                           hPutStrLn handleSock ("Good bye!")
                           hClose handleSock
                     else do
                             putStrLn $ "Request received from: " ++ line
                             hPutStrLn handleSock $ "Hey, client!"
                             loopclient handleSock
```



```
  :\Users\igleu\nc>nc64 localhost 4244
Ioana
Hey, client!
a
Hey, client!
b
Hey, client!
c
Hey, client!
end
Good bye!
```

Clientii sunt serviti secvential!

Clientului 2 i se raspunde numai dupa ce Clientul 1 a trimis "end".

```
main = do
  sock <- socket AF_INET Stream 0
  setSocketOption sock ReuseAddr 1
  bind sock (SockAddrInet 4246 0)
  listen sock 2
  putStrLn "Listening on port 4246..."
  loopForever sock
```

```
loopForever :: Socket -> IO ()
loopForever sock = do
         (conn, _) <- accept sock
         handleSock <- socketToHandle conn  ReadWriteMode
         line <- hGetLine handleSock
         forkIO $ loopClient handleSock
         loopForever sock
```

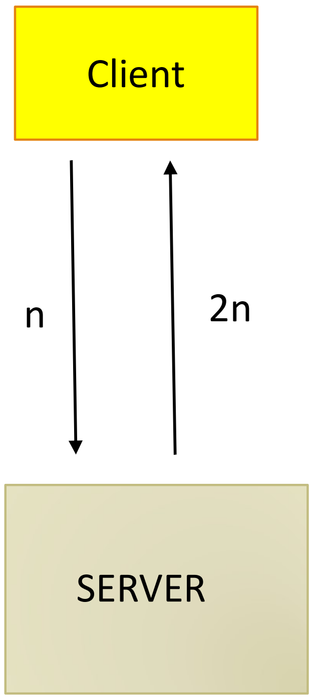
Clientii sunt serviti concurrent, pe thread-uri separate

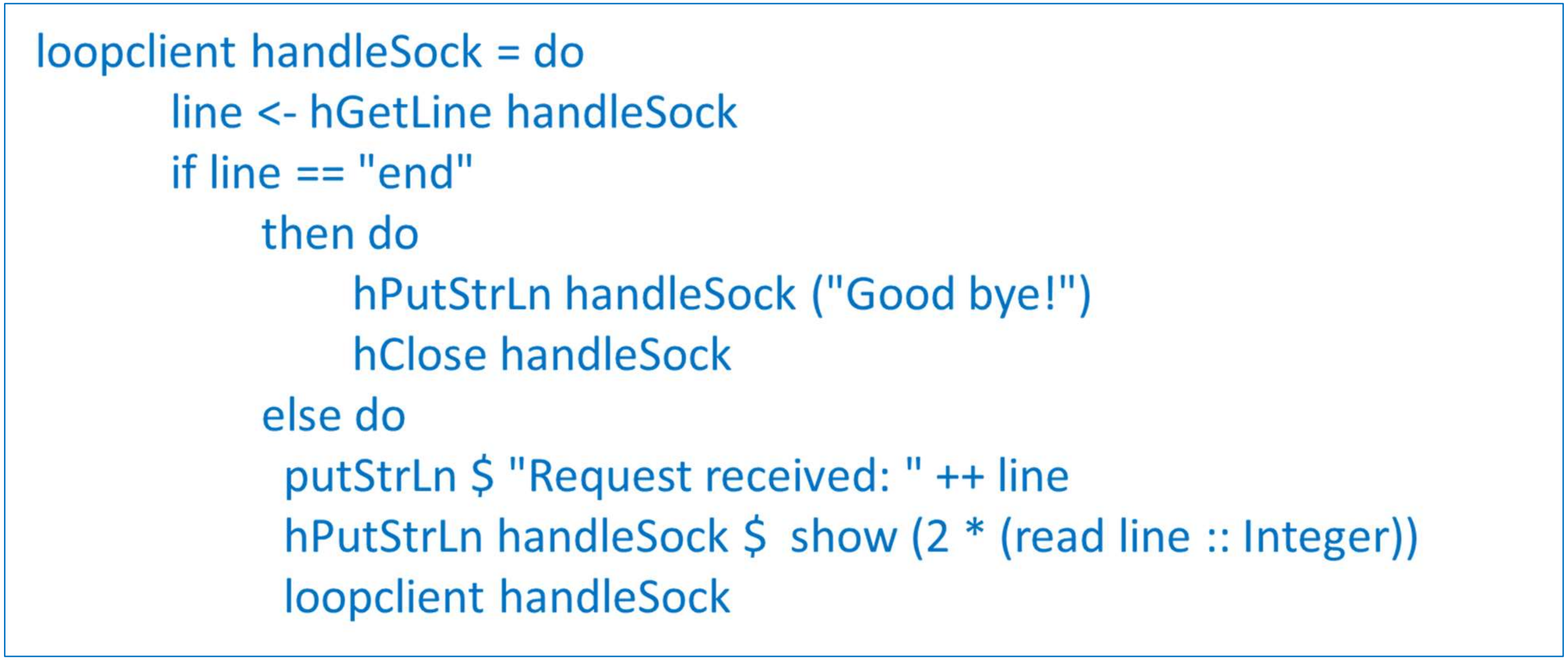
```
loopForever :: Socket -> IO ()
loopForever sock = do
             (conn, _) <- accept sock
             handleSock <- socketToHandle conn  ReadWriteMode
             line <- hGetLine handleSock
             forkIO $ loopClient handleSock
             loopForever sock
```

```
*Main> main
Listening on port 4246...
Request received from: Ioana
Request received from: a
Request received from: Ana
Request received from: 10
Request received from: 3
```

```
C:\Users\igleu\nc>nc64 localhost 4246
Ioana
Hey, client!
a
Hey, client!
```

```
C:\Users\igleu\nc>nc64 localhost 4246
Ana
Hey, client!
10
Hey, client!
3
```





```
loopclient handleSock = do
          line <- hGetLine handleSock
          if line == "end"
                then do
                      hPutStrLn handleSock ("Good bye!")
                      hClose handleSock
                else do
                  putStrLn $ "Request received: " ++ line
                  hPutStrLn handleSock $  show (2 * (read line :: Integer))
                  loopclient handleSock
```

[Parallel and Concurrent Programming in Haskell · Simon Marlow]

```
loopForever :: Socket -> IO ()
loopForever sock = do
  (conn, _) <- accept sock
  handleSock <- socketToHandle conn  ReadWriteMode
  line <- hGetLine handleSock
  forkIO $ loopClient handleSock
  loopForever sock
```

```
loopclient handleSock = do
        line <- hGetLine handleSock
        if line == "end"
             then do
                  hPutStrLn handleSock ("Good bye!")
                  hClose handleSock
             else do
               putStrLn $ "Request received: " ++ line
               hPutStrLn handleSock $  show (2 * (read line :: Integer))
               loopclient handleSock
```

```
*Main> main
Listening on port 4250...
Request received: 2
Request received: 3
Request received: 3
Request received: 5
Request received: 6
Request received: a
<interactive>: Prelude.read: no parse
Request received: 8
```

exceptie

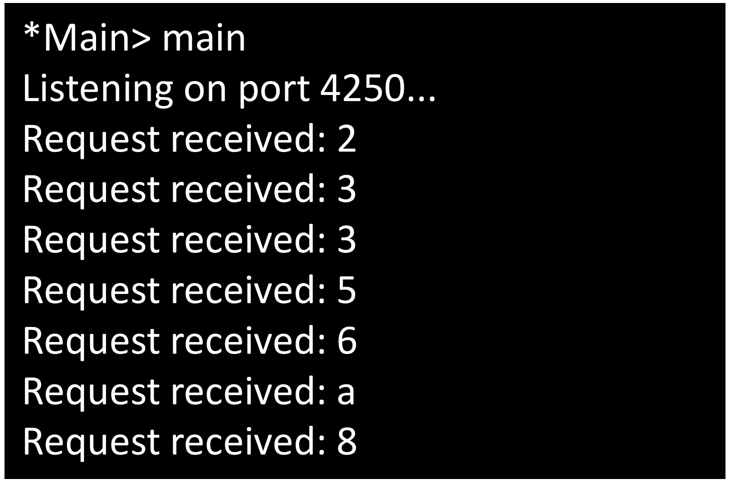
```
C:\Users\igleu\nc>nc64 localhost 4250
2
4
3
6
3
6
8
16
```

```
C:\Users\igleu\nc>nc64 localhost 4250
5
10
6
12
a

C:\Users\igleu\nc>
```

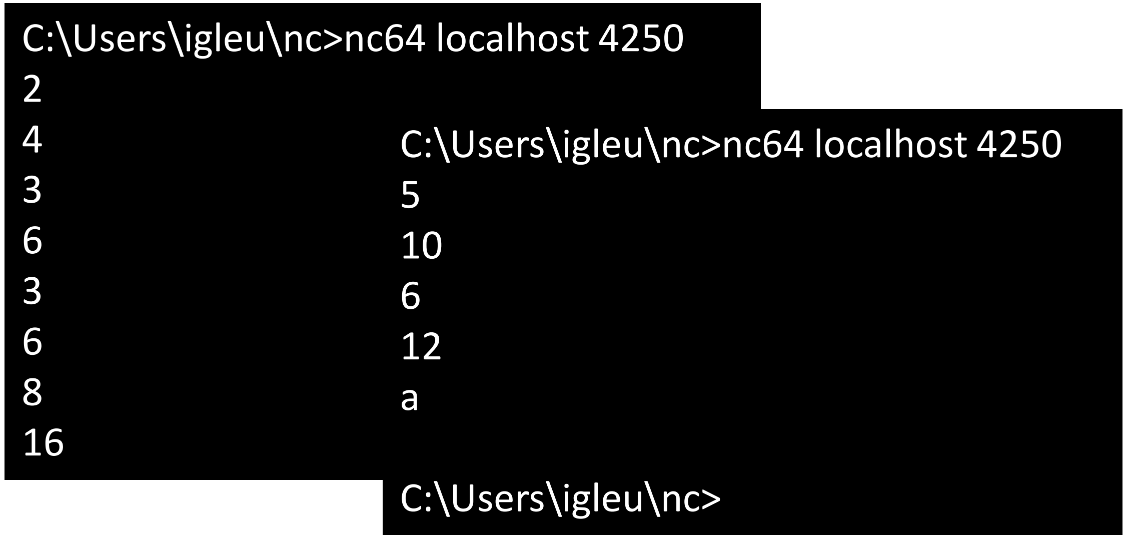
Este posibil ca handle-ul sa ramana deschis

```
loopForever :: Socket -> IO ()
loopForever sock = do
  (conn, _) <- accept sock
  handleSock <- socketToHandle conn  ReadWriteMode
  line <- hGetLine handleSock
  forkFinally  (loopClient handleSock)   (\_ -> hClose handleSock)
  loopForever sock
```

Exceptia este tratata folosind forkFinally



```
*Main> main
Listening on port 4250...
Request received: 2
Request received: 3
Request received: 3
Request received: 5
Request received: 6
Request received: a
Request received: 8
```



```
C:\Users\igleu\nc>nc64 localhost 4250
2
4
3
6
3
6
8
16
```

```
C:\Users\igleu\nc>nc64 localhost 4250
5
10
6
12
a

C:\Users\igleu\nc>
```

- Server cu stare partajata

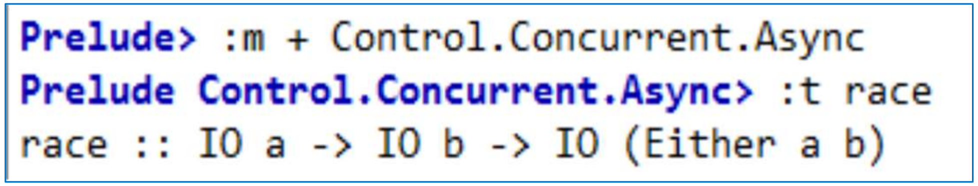
> "The new behavior is as follows: instead of multiplying each number by two, the server will multiply each number by the current factor. Any connected client can change the current factor by sending the command *N, where N is an integer. When a client changes the factor, the server sends a message to all the other connected clients informing them of the change.
>
> While this seems like a small change in behavior, it introduces some interesting new challenges in designing the server.
>
> - There is a shared state—the current factor—so we must decide how to store it and how it is accessed and modified.
> - When one server thread changes the state in response to its client issuing the *N command, we must arrange to send a message to all the connected clients."

Parallel and Concurrent Programming in Haskell · Simon Marlow

➢ Server cu stare partajata

Detalii de implementare:

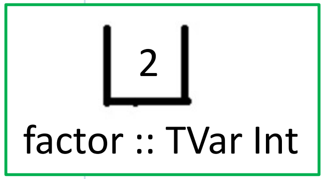
- Pentru fiecare conexiune (client) se creaza un thread nou in care se executa functia loopClient.
- Functia loopClient creaza un canal de comunicare si executa in paralel o functiile server si receive.
- Functia receive citeste comenzile clientului si le introduce in canalul de comunicare, de unde sunt citite si prelucrate de functia server.
- Functia server implementeaza actiunile serverului: citeste factorul initial, citeste si executa comenzile clientu
  comanda *N a clientului poate modifica valoarea factorului.
- Pentru executarea in paralel a functiilor server si receive se foloseste functia race.
  Functia race executa doua actiuni in parallel si o intoarce pe prima care se termina.



```
Prelude> :m + Control.Concurrent.Async
Prelude Control.Concurrent.Async> :t race
race :: IO a -> IO b -> IO (Either a b)
```

Memoria partajata este implementata in **STM**
TVar, TChan

memoria accesibila tuturor clientilor



factor :: TVar Int

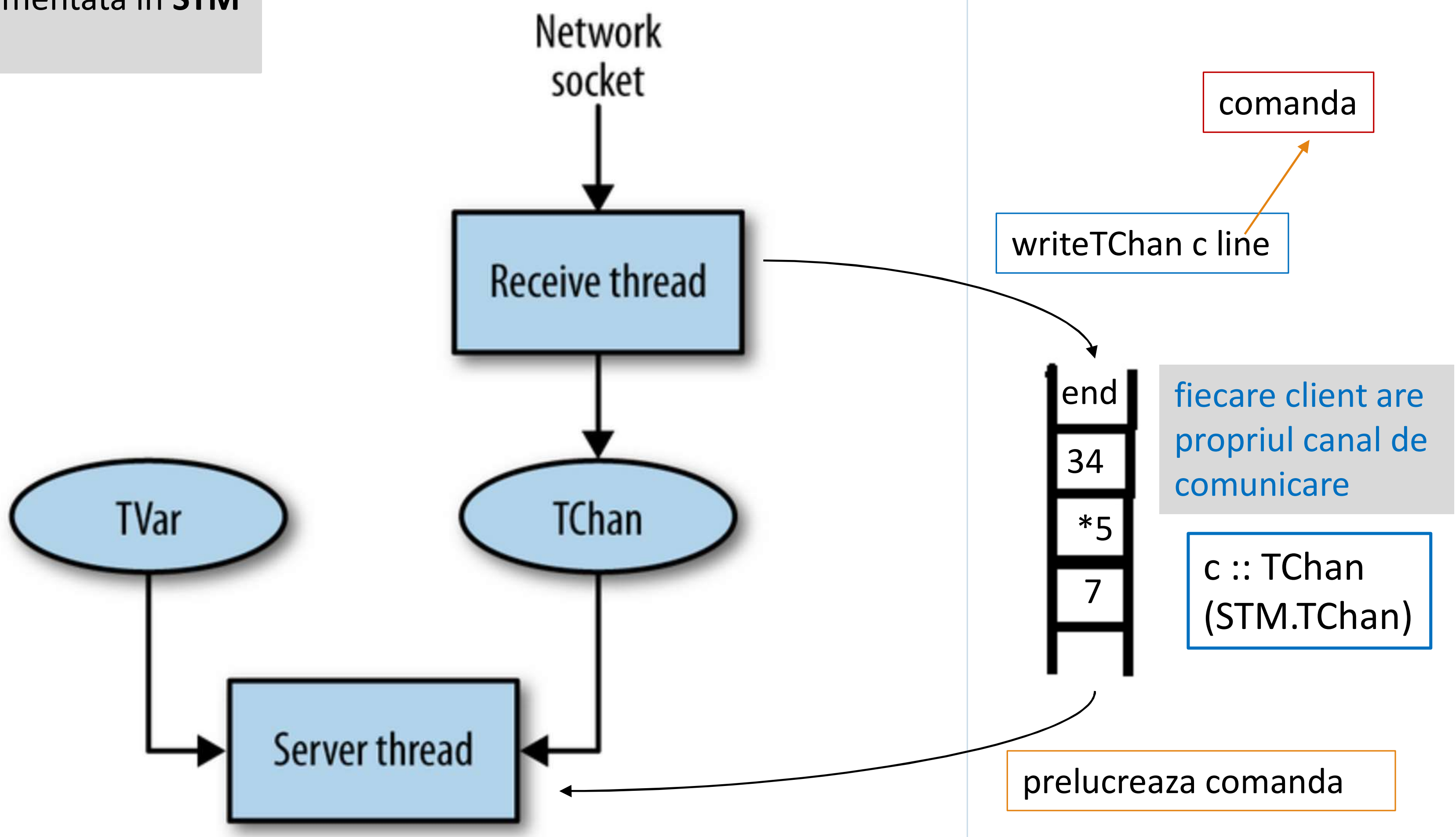


Figure 12-2. Server structure with STM

comanda

writeTChan c line

fiecare client are propriul canal de comunicare

c :: TChan (STM.TChan)

prelucreaza comanda

```
main = do
  sock <- socket AF_INET Stream 0
  setSocketOption sock ReuseAddr 1
  bind sock (SockAddrInet 4000 0)
  listen sock 2
  putStrLn "Listening on port 4000..."
  factor <- atomically $ newTVar 2
  loopForever sock   factor
```

```
import Network.Socket
import System.IO
import Control.Monad
import Control.Concurrent
import Control.Concurrent.STM
import Control.Concurrent.Async
```

```
loopForever :: Socket -> TVar Integer -> IO ()
loopForever sock  factor = do
  (conn, _) <- accept sock
  handleSock <- socketToHandle conn ReadWriteMode
  forkIO $ loopClient handleSock  factor
  loopForever sock   factor
```

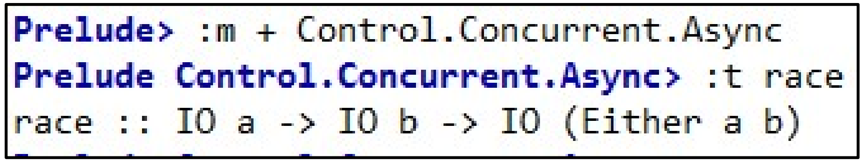
server2.hs

```
loopClient h  f = do
          c <- atomically newTChan
          race_ (server h f c) (receive h c)
```

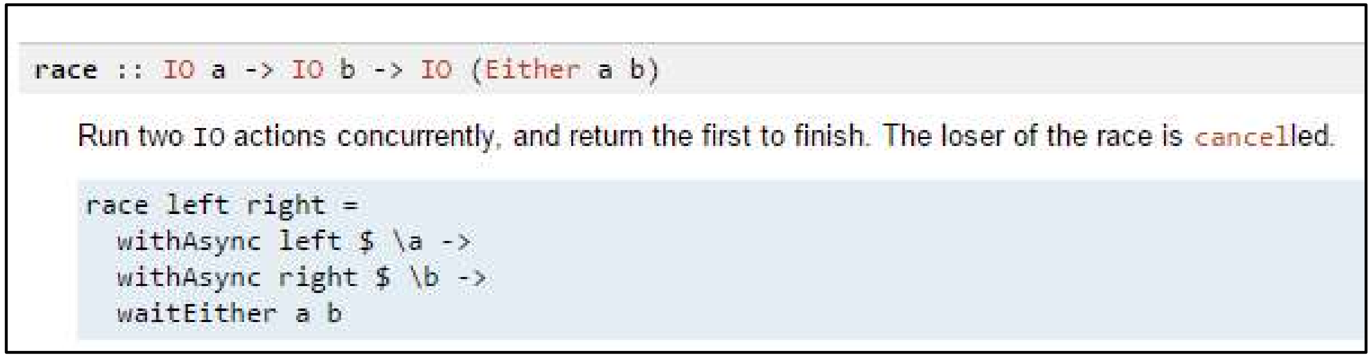
```
h :: Handle          --propriu clientului
f :: TVar Integer  -- factorul comun
c :: TChan String -- propriu clientului
```

- se termina odata cu   primul dintre server si receive

- in aceasta implementare, server se termina cand primeste comanda end, iar receive este o actiune definita cu forever



```
Prelude> :m + Control.Concurrent.Async
Prelude Control.Concurrent.Async> :t race
race :: IO a -> IO b -> IO (Either a b)
```



```
race :: IO a -> IO b -> IO (Either a b)
```

Run two IO actions concurrently, and return the first to finish. The loser of the race is cancelled.

```
race left right =
  withAsync left $ \a ->
  withAsync right $ \b ->
  waitEither a b
```

https://www.haskell.org/hoogle/

```
loopClient h  f = do
          c <- atomically newTChan
          race_ (server h f c) (receive h c)
```

```
receive :: Handle -> TChan String -> IO ()

receive h c = forever $ do
      line <- hGetLine h
      atomically $ writeTChan c line
```

citeste datele/comenzile introduse de client si le scrie pe canal

```
server :: Handle -> TVar Integer -> TChan String -> IO ()
server :: Handle -> TVar Integer -> TChan String -> IO ()
server h  f c = do
          fval <- atomically $ readTVar f                   --valoarea curenta a factorului
          hPutStrLn h $ "Current factor is " ++ show fval  --la inceputul interactiunii cu clientul
          loop h fval f c

loop h fval f c  = do
             action <- atomically $ do
                  newval <- readTVar f
                  if (fval /= newval)
                   then return (newfactor h newval f c)
                   else do
                      line <- readTChan c
                      return (command h fval f c line)
             action
```

newfactor -anunta clientului
modificarea factorului

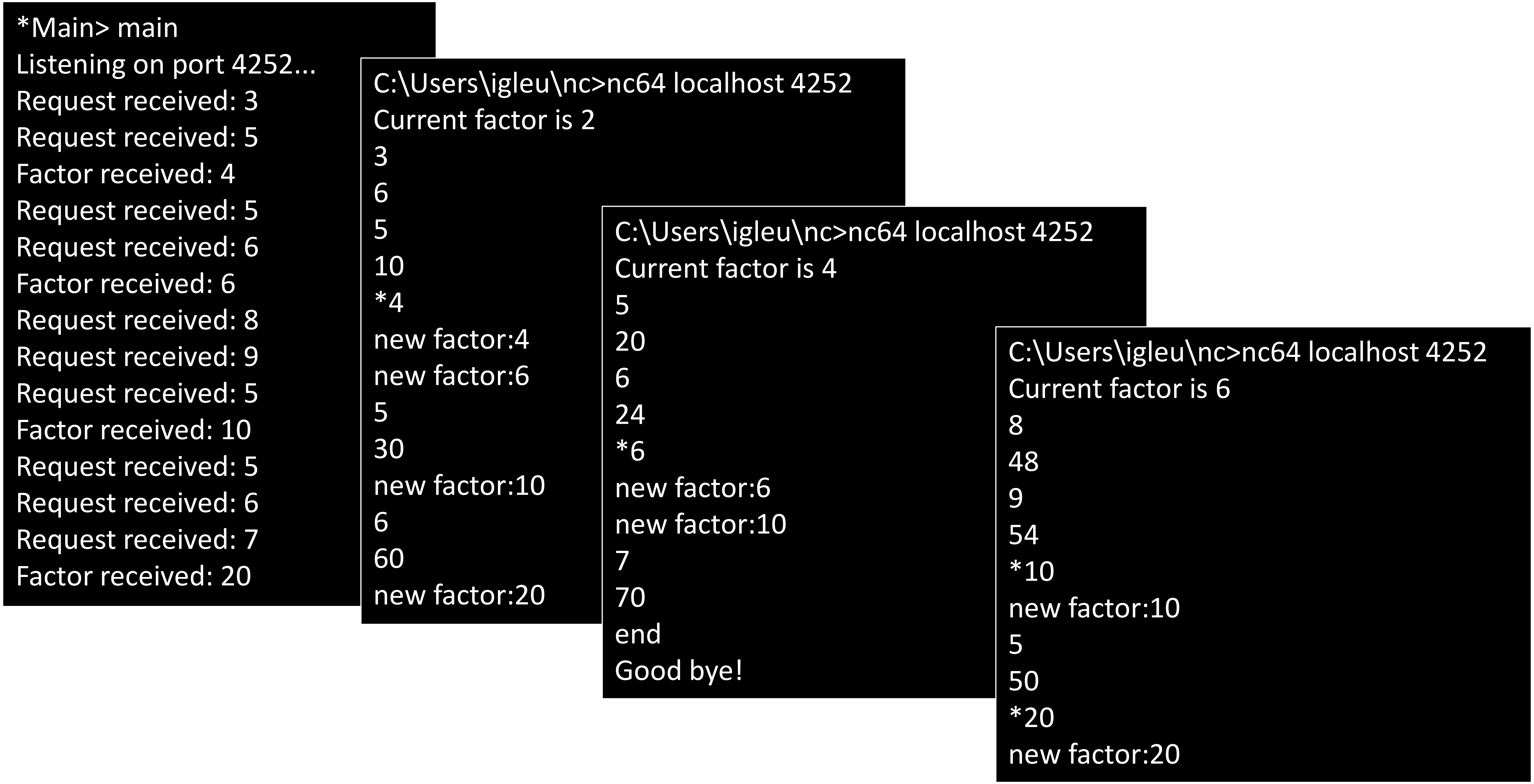
command -executa comanda citita
de pe canalul clientului

ambele apeleaza recursiv loop

```
loop h fval f c  = do
            action <- atomically $ do
                newval <- readTVar f
                if (fval /= newval)
                 then return (newfactor h newval f c)
                 else do
                    line <- readTChan c
                    return (command h fval f c line)
            action
```

```
newfactor h newval f c = do
               hPutStrLn h $ "new factor:" ++ show newval
               loop  h newval f c
```

```
command h fval f c line = case line of
              "end" -> do
                    hPutStrLn h ("Good bye!")
                    hClose h
              '*':s  -> do
                     putStrLn $ "Factor received: " ++ s
                     atomically $ writeTVar f (read s :: Integer)
                     loop h fval f c
              _    ->   do
                      putStrLn $ "Request received: " ++ line
                      hPutStrLn h $  show (fval * (read line :: Integer))
                      loop h fval f c
```



```
*Main> main
Listening on port 4252...
Request received: 3
Request received: 5
Factor received: 4
Request received: 5
Request received: 6
Factor received: 6
Request received: 8
Request received: 9
Request received: 5
Factor received: 10
Request received: 5
Request received: 6
Request received: 7
Factor received: 20
```

```
C:\Users\igleu\nc>nc64 localhost 4252
Current factor is 2
3
6
5
10
*4
new factor:4
new factor:6
5
30
new factor:10
6
60
new factor:20
```

```
C:\Users\igleu\nc>nc64 localhost 4252
Current factor is 4
5
20
6
24
*6
new factor:6
new factor:10
7
70
end
Good bye!
```

```
C:\Users\igleu\nc>nc64 localhost 4252
Current factor is 6
8
48
9
54
*10
new factor:10
5
50
*20
new factor:20
```

➢ Server cu stare partajata si tip de data pentru clienti

Detalii de implementare:

client={nume, handle, canal}
server =[client]

- Pentru fiecare conexiune se creaza un thread nou in care se executa functia createClient.
- Functia createClient creaza un client nou, reprezentat printr-o structura {nume, handle, canal} si apeleaza functia loopClient.
- **Fiecare client nou este anuntat celorlalti clienti.**
- Functia loopClient executa in parallel functiile server si receive (folosind race).
- Functia receive citeste comenzile clientului si le introduce in canalul de comunicare, de unde sunt citite si prelucrate de functia server.
- Functia server implementeaza actiunile serverului: citeste factorul curent, citeste si executa comenzile clientului; comanda *N a clientului poate modifica valoarea factorului .
- Pentru executarea in paralel a functiilor server si receive se foloseste functia race.
- Functia race executa doua actiuni in parallel si o intoarce pe prima care se termina

- Server cu stare partajata si tip de date client

```
data Client = Client {cName :: String
                     ,cHandle :: Handle
                     ,cChan :: TChan String}
type Server = TVar [Client]
```

```
main = do
  sock <- socket AF_INET Stream 0
  setSocketOption sock ReuseAddr 1
  bind sock (SockAddrInet 44445 0)
  listen sock 2
  putStrLn "Listening on port 44445..."
  factor <- atomically $ newTVar 2
  sv <- atomically $ newTVar []      -- serverul este lista clientilor, initial este []
  loopForever sv sock factor
loopForever :: Server -> Socket -> TVar Integer -> IO ()
loopForever sv sock  factor = do
  (conn, _) <- accept sock
  handleSock <- socketToHandle conn ReadWriteMode
  forkFinally (createClient sv handleSock  factor) (\_ -> hClose handleSock)
  loopForever sv sock  factor
```

```
createClient sv h f = do
                          hPutStrLn h "Name"
                          name <- hGetLine h
                          c <- atomically newTChan
                          putStrLn $ "New client: " ++ name
                          hPutStrLn h $ "Wellcome " ++ name
                          addClient sv name h c
                          loopClient h f c sv
```

```
loopClient h  f c sv =  race_ (server h f c sv) (receive h c)
```

```
addClient sv name h c = atomically $ do
                svclients <- readTVar sv
                writeTVar sv (svclients ++ [(Client name h c)])
                broadcast svclients ("@"++ name)


broadcast [] msg = return ()
broadcast ((Client _ _ c):lcl) msg = do
                writeTChan c msg
                broadcast lcl msg
```

- pentru a transmite celorlalti client numele noului client,

pe canalul fiecarui client deja existent este pusa o comanda speciala

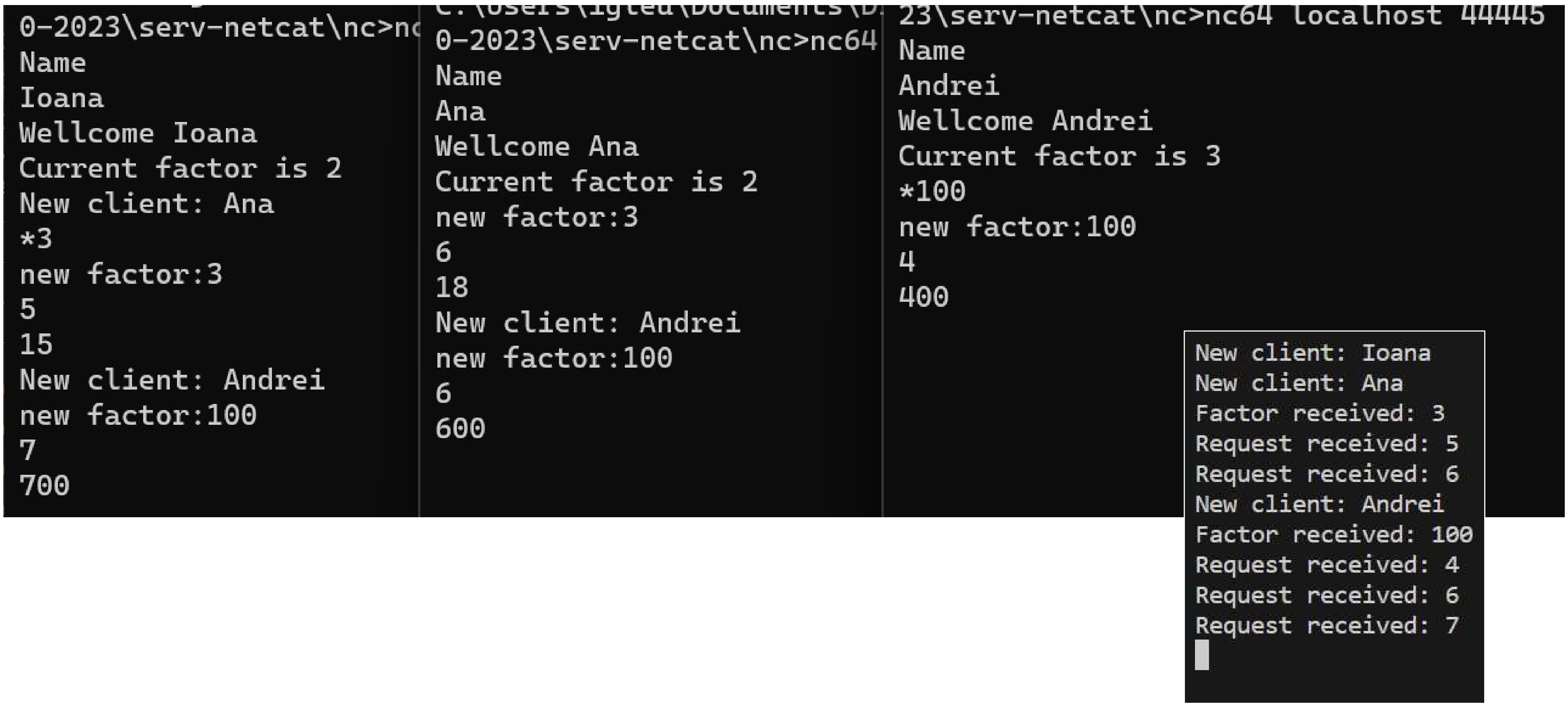
- aceasta comanda va fi prelucrata in functia command

```
loopClient h  f c sv =  race_ (server h f c sv) (receive h c)
```

```
server :: Handle -> TVar Integer -> TChan String -> Server -> IO ()
server h  f c sv = do
          fval <- atomically $ readTVar f
          hPutStrLn h $ "Current factor is " ++ show fval
          loop h fval f c sv
```

```
receive :: Handle -> TChan String -> IO ()
receive h c = forever $ do
               line <- hGetLine h
               atomically $ writeTChan c line
```

```
loop h fval f c sv  = ...

newfactor h newval f c sv = ...

command h fval f c line sv = ...
command h fval f c line sv = case line of
                "end" -> do
                      hPutStrLn h ("Good bye!")
                      hClose h
                '*':s -> do
                       putStrLn $ "Factor received: " ++ s
                       atomically $ writeTVar f (read s :: Integer)
                       loop h fval f c sv
                '@':s -> do
                       hPutStrLn h $ "New client: " ++ s
                       loop h fval f c sv
                _    ->   do
                        putStrLn $ "Request received: " ++ line
                        hPutStrLn h $  show (fval * (read line :: Integer))
                        loop h fval f c sv
```



```
0-2023\serv-netcat\nc>nc
Name
Ioana
Wellcome Ioana
Current factor is 2
New client: Ana
*3
new factor:3
5
15
New client: Andrei
new factor:100
7
700
```

```
0-2023\serv-netcat\nc>nc64
Name
Ana
Wellcome Ana
Current factor is 2
new factor:3
6
18
New client: Andrei
new factor:100
6
600
```

```
23\serv-netcat\nc>nc64 localhost 44445
Name
Andrei
Wellcome Andrei
Current factor is 3
*100
new factor:100
4
400
```

```
New client: Ioana
New client: Ana
Factor received: 3
Request received: 5
Request received: 6
New client: Andrei
Factor received: 100
Request received: 4
Request received: 6
Request received: 7
```

- Pentru fiecare client adaugam o comanda ">nume" care transmite "Hugs!" clientului "nume"

```
command h fval f c line sv = case line of
                    "end" -> do
                            hPutStrLn h ("Good bye!")
                            hClose h
                    '*':s -> do
                             putStrLn $ "Factor received: " ++ s
                             atomically $ writeTVar f (read s :: Integer)
                             loop h fval f c sv
                    '@':s -> do
                             hPutStrLn h $ "New client: " ++ s
                             loop h fval f c sv
                    '>':s -> do
                            sendmessageto s sv
                            loop h fval f c sv
                    "<" -> do
                            hPutStrLn h  "Hugs!"
                            loop h fval f c sv
                    _    ->   do
```

transmiterea mesajului catre clientul cu numele s

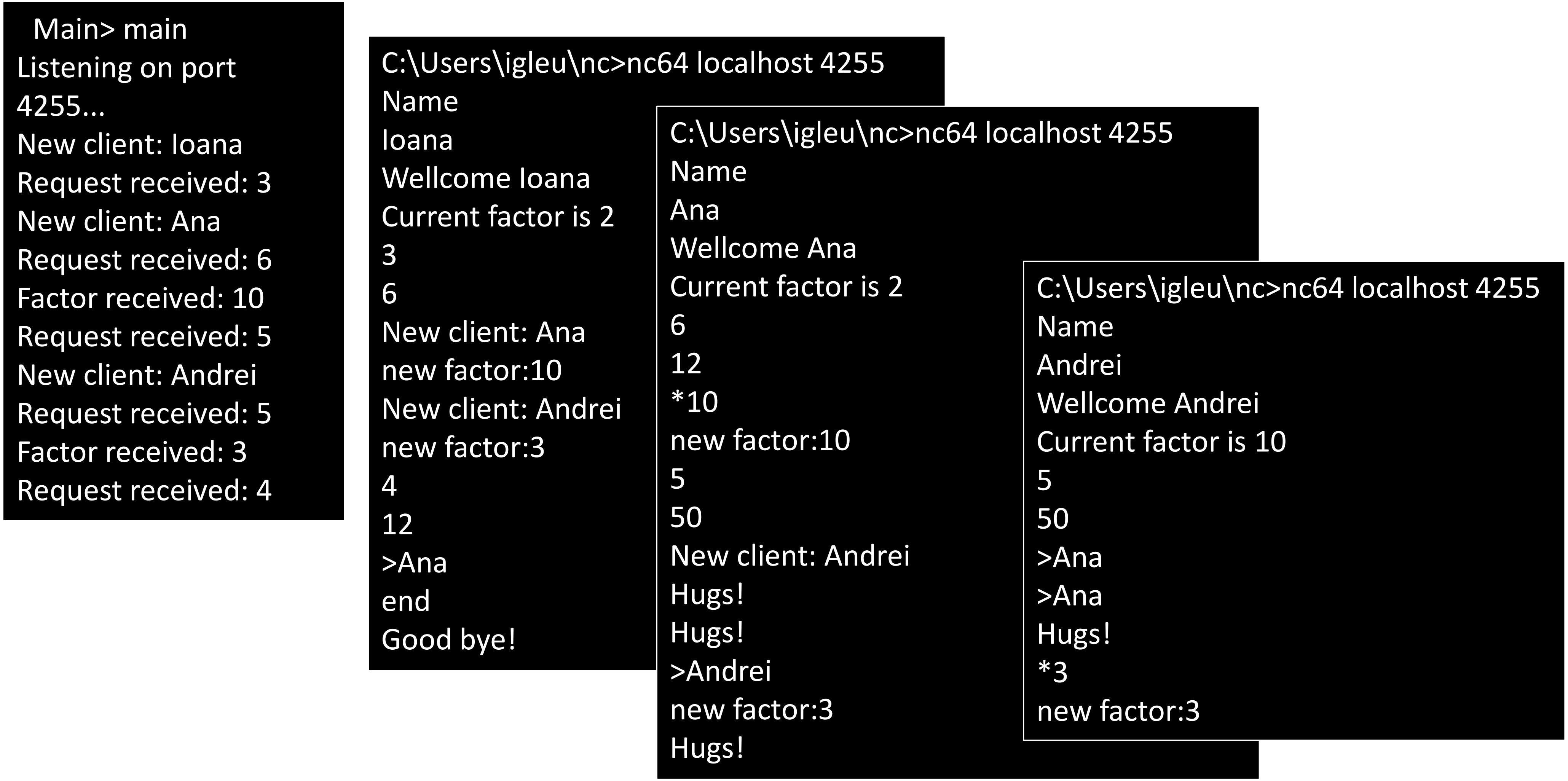
prelucrarea mesajului primit de la alt client

- Pentru fiecare client adaugam o comanda ">nume" care transmite "Hugs!" clientului "nume"

```
command h fval f c line sv = case line of
                  "end" -> do
                         hPutStrLn h ("Good bye!")
                         hClose h
                  '*':s -> do
                          putStrLn $ "Factor received: " ++ s
                          atomically $ writeTVar f (read s :: Integer)
                          loop h fval f c sv
                  '@':s -> do
                          hPutStrLn h $ "New client: " ++ s
                          loop h fval f c sv
                  '>':s -> do
                         sendmessageto s sv
                         loop h fval f c sv
                  "<" -> do
                         hPutStrLn h  "Hugs!"
                         loop h fval f c sv
                  _    ->   do
```

transmiterea mesajului catre clientul cu numele s

```
sendmessageto s sv = atomically $ do
                svclients <- readTVar sv
                let  c = channelof s svclients
                writeTChan c "<"

channelof s ((Client n _ c):lcl) =  if s==n
                                             then c
                                             else (channelof s lcl)
```

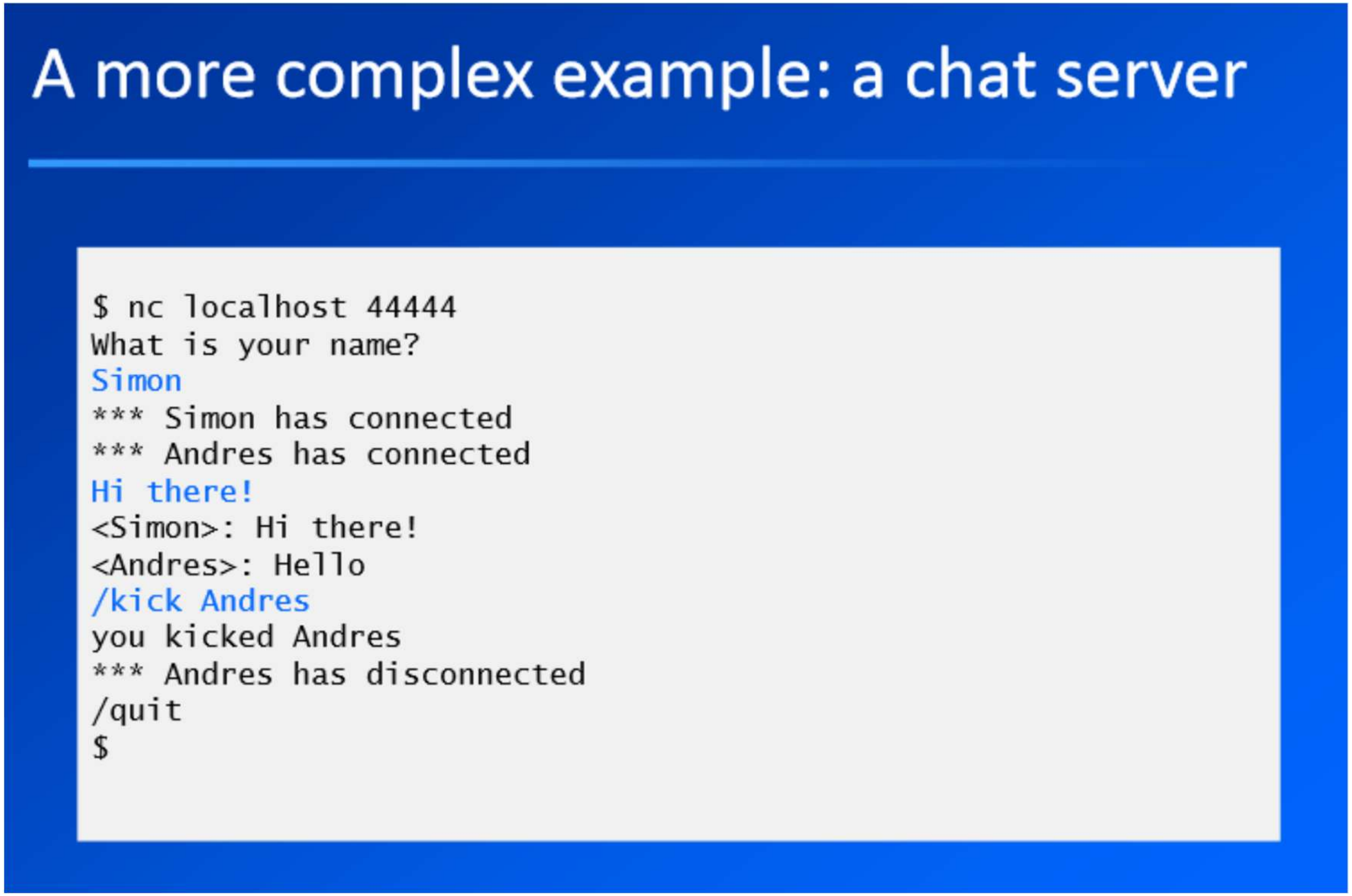


```
  Main> main
Listening on port
4255...
New client: Ioana
Request received: 3
New client: Ana
Request received: 6
Factor received: 10
Request received: 5
New client: Andrei
Request received: 5
Factor received: 3
Request received: 4
```

```
C:\Users\igleu\nc>nc64 localhost 4255
Name
Ioana
Wellcome Ioana
Current factor is 2
3
6
New client: Ana
new factor:10
New client: Andrei
new factor:3
4
12
>Ana
end
Good bye!
```

```
C:\Users\igleu\nc>nc64 localhost 4255
Name
Ana
Wellcome Ana
Current factor is 2
6
12
*10
new factor:10
5
50
New client: Andrei
Hugs!
Hugs!
>Andrei
new factor:3
Hugs!
```

```
C:\Users\igleu\nc>nc64 localhost 4255
Name
Andrei
Wellcome Andrei
Current factor is 10
5
50
>Ana
>Ana
Hugs!
*3
new factor:3
```



A more complex example: a chat server

```
$ nc localhost 44444
What is your name?
Simon
*** Simon has connected
*** Andres has connected
Hi there!
<Simon>: Hi there!
<Andres>: Hello
/kick Andres
you kicked Andres
*** Andres has disconnected
/quit
$
```

http://community.haskell.org/~simonmar/slides/cadarache2012/5%20-%20server%20apps.pdf