Parallel & Concurrent Haskell 5: Server Applications

Simon Marlow
Feedback please!

• If you tried the exercises so far, I would love to hear feedback about:
  – what you found hard, easy, or surprising
  – what you would have liked to see covered in the lectures
  – If you tried more than one of Eval, Strategies, and Par:
    • Which did you find easier?
    • Which gave better results?

• email me: marlowsd@gmail.com
• Feedback will be useful for my book – you’ll get a mention in the acknowledgements!
Overview

• Chapter 14 in the notes
• Building a simple multi-client network server
  – no shared state
  – one thread per client
• Error handling
• More complex example: a chat server
  – shared state
  – managing multiple threads per client
Setup

e.g.

• web server
• mail server
• database server
• ...

How do we write the server?
Background:
Threads vs. Events
Without concurrency

One big server with an event loop. Client code uses callbacks.

• e.g. node.js
• Advantages
  – performance (?)
  – simplicity: no mutable shared state
• Disadvantages
  – coding with callbacks is painful
  – clients can accidentally block each other
  – No parallelism
Concurrency

- One thread per client
- Advantages:
  - the interaction with a single client is straight-line code
  - clients cannot accidentally block each other
  - automatically uses multiple cores
- Disadvantages
  - performance?
  - shared state?

[Diagram showing client threads and server application]
Disadvantages?

• **Performance:**
  – In GHC threads are very lightweight
  – When many threads are blocked on I/O, a single thread handles the interaction with the OS, using epoll(). (the “IO manager thread”)
  – Typical for a single core using 1 thread per client:
    • 100K+ simultaneous connections
    • 10K+ requests/sec

• **Mutable shared state:**
  – shared state is less of a problem in Haskell: e.g. use STM
Threads or events?

- Long running debate
- See e.g. “Why Events Are A Bad Idea (for high-concurrency servers)” Rob von Behren, Jeremy Condit and Eric Brewer (HotOS IX, 2003)
- In Haskell we think we provide a nice point in the design space:
  - threads at the programmer level, for ease of programming and abstraction
  - implemented using event-based techniques (epoll() etc.), which eliminates many of the traditional disadvantages of threads
End of background!
Let’s write some code.
Simple example

• Sample: server.hs
• Server accepts connections on port 44444
• Loop:
  – Client sends an integer \( n \)
  – Server replies with \( 2n \)
• If client sends “end”, the connection is terminated
• Code for the interaction with a single client

talk :: Handle -> IO ()
talk h = do
  hSetBuffering h LineBuffering
  loop
  where
    loop = do
      line <- hGetLine h
      if line == "end"
        then putStrLn h ('Thank you for using the " ++
                        "Haskell doubling service."')
        else do putStrLn h (show (2 * (read line :: Integer)))
                loop

• Handle is bound to the network socket
The main program

```haskell
main :: IO ()
main = withSocketsDo $ do
  sock <- listenOn (PortNumber (fromIntegral port))
  printf "Listening on port %d\n" port
  forever $ do
    (handle, host, port) <- accept sock
    printf "Accepted connection from %s: %s\n" host (show port)
    forkIO (talk handle `finally` hClose handle)

port :: Int
port = 44444
```

finally :: IO a -> IO b -> IO b
Demo
• Note 1: making the server concurrent required zero changes to the `talk` function that interacts with a single client.
• Note 2: the server will use multiple cores if we compile with `-threaded`.

• Q: what happens when an error occurs?
  1. if the user types a non-number?
  2. if the user closes the connection?

• Q: if we wanted to handle the case of the user typing a non-number, what should we do?
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
$
Behaviour

- On client connection, the server first asks for the user’s name before entering the command loop

- Commands:
  - `/tell <name> <msg>`
    - message a specific user
  - `/kick <name>`
    - forcibly disconnect another user
  - `/quit`
    - disconnect the current user
  - `<anything else>`
    - broadcast to all users

- All clients are informed of connections/disconnections
Why is this harder?

- **We have some shared state:**
  - the clients that are currently connected
- **We need to think about consistency**
  - what happens if two users try to kick each other?
  - our choice: exactly one succeeds
  - what happens if two users simultaneously log in with the same name?
  - again: exactly one succeeds
- **The client interaction cannot be programmed in a single thread**
  - we need to receive network input *and* events from other clients (broadcast, tell, kick, client connected or disconnected)
  - no way to do both in one thread
**Design**

- Must have only one thread that sends information back to the client, otherwise some locking would be needed to prevent interleaving.

- Simplest approach is to have two threads:
  - receive forwards the network traffic from the client to a channel (only)
  - send does everything else, including sending data back to the client
• How should the **send** thread wait for events? Two alternatives:
  – using MVars: all events must go down a single Chan, and the **send** thread processes events one at a time from the Chan
  – using STM: we can have multiple event sources, and the **send** thread uses `orElse` to wait for an event on any of them

• commands from the client
• broadcast messages
• `/kick` commands

![Diagram](image-url)
• Using MVar/Chan would make it difficult to guarantee consistency for multiple kicks, because there could be multiple kick messages in-flight simultaneously

• Using STM, we can guarantee atomic access to the shared state

• So we’ll use STM...
• **Client types**

```haskell

-- type

type ClientName = String

data Client = Client
  { clientName :: ClientName
   , clientHandle :: Handle
   , clientKicked :: TVar (Maybe String)
   , clientSendChan :: TChan Message
  }

-- data

data Message = Notice String
  | Tell ClientName String
  | Broadcast ClientName String
  | Command String

-- newClient

newClient :: ClientName -> Handle -> STM Client
```

• We could use two channels to send messages to the client, but one is simpler.
• Sending a message to a client:

```plaintext
sendMessage :: Client -> Message -> STM ()
sendMessage Client{..} msg =
    writeTChan clientSendChan msg
```

• Client{..} is a *record wild card*. We could have written:

```plaintext
sendMessage Client{ clientName   = clientName
                 , clientHandle = clientHandle
                 , clientKicked = clientKicked
                 , clientSendChan = clientSendChan
                 } msg =
    writeTChan clientSendChan msg
```
• **Server state**

```haskell
data Server = Server
  { clients :: TVar (Map ClientName Client) }

newServer :: IO Server
newServer = do
  c <- newTVarIO Map.empty
  return Server { clients = c }
```

• All the clients will have access to the Server state, because they need to be able to send messages to each other.
Broadcast a message to all clients

```haskell
broadcast :: Server -> Message -> STM ()
broadcast Server{..} msg = do
    clientmap <- readTVar clients
    mapM_ (\client -> sendMessage client msg)
    (Map.elems clientmap)
```
So far...

type ClientName = String

data Client

data Server

data Message

newServer :: IO Server
newClient :: ClientName -> Handle -> STM Client

sendMessage :: Client -> Message -> STM ()
broadcast :: Server -> Message -> STM ()
• Now go top-down, write the rest of the code

```haskell
main :: IO ()
main = withSocketsDo $ do
    server <- newServer
    sock <- listenOn (PortNumber (fromIntegral port))
    printf "Listening on port %d\n" port
    forever $ do
        (handle, host, port) <- accept sock
        printf "Accepted connection from %s: %s\n" host (show port)
        fork (talk server handle `finally` hClose handle)

port :: Int
port = 44444
```
Defining talk

• Overall plan:
  – ask the user for their name
  – if the name already exists, ask again
  – otherwise, create a Client and insert it into the Server state
    • NB. make sure the Client is removed from the state if the connection is closed, or an error occurs
  – Notify other clients of the new connection
  – Set up the threads and start processing
talk :: Server -> Handle -> IO ()
talk server@Server{..} handle = do
  hSetNewlineMode handle universalNewlineMode
  hSetBuffering handle LineBuffering
readName
  where
    readName = do
      hPutStrLn handle "What is your name?"
      name <- hGetLine handle
      clientmap <- atomically $ readTVar clients
      if Map.member name clientmap
        then do
          hPrintf handle "The name %s is in use" name
          readName
        else do
          client <- atomically $ newClient name handle
          atomically $
            writeTVar clients
            (Map.insert name client clientmap)
  ...

Not enough atomicity here, let’s try again
checkAddClient :: Server -> ClientName -> Handle -> IO (Maybe Client)
checkAddClient server@Server{..} name handle = atomically $ do
  clientmap <- readTVar clients
  if Map.member name clientmap
    then return Nothing
    else do
        client <- newClient name handle
        writeTVar clients (Map.insert name client clientmap)
        broadcast server $ Notice $ name ++ " has connected"
        return (Just client)

• Checks for and adds the new client atomically
  – returns (Just client) if successful
  – or Nothing otherwise
Back to talk...

talk :: Server -> Handle -> IO ()
talk server@Server{..} handle = do
  hSetNewlineMode handle universalNewlineMode
  hSetBuffering handle LineBuffering
  readName
where
  readName = do
    hPutStrLn handle "What is your name?"
    name <- hGetLine handle
    m <- checkAddClient server name handle
    case m of
      Nothing -> do
        hPrintf handle "The name %s is in use" name
        readName
      Just client -> do
        runClient server client
        `finally` removeClient server name

Strictly speaking we should plug the hole between checkAddClient and finally (see the notes...)
removeClient is quite straightforward:

```haskell
removeClient :: Server -> ClientName -> IO ()
removeClient server@Server{..} name = atomically $ do
  clientmap <- readTVar clients
  writeTVar clientmap (Map.delete name m)
  broadcast server (Notice (name ++ " has disconnected"))
```
Now to start up the client threads.

runClient :: Server -> Client -> IO ()

- Overall plan:
  - create a receive thread to forward network traffic to the `clientSendChan`
  - create a send thread to watch `clientSendChan` and `clientKicked`
  - if *either* of these threads dies or returns, we want to close the connection and clean up (i.e. `runClient` should return or throw an exception)
  - there should be no possibility that a thread is left running after the client has disconnected.
• Let’s package up this behaviour behind a new abstraction:

\[
\text{concurrently} :: \text{IO } () \rightarrow \text{IO } () \rightarrow \text{IO } ()
\]

• this runs both IO actions concurrently, such that
  – if either one returns, the other is killed, and concurrently then returns
  – if either one throws an exception, then the other is killed and concurrently re-throws the exception in the current thread.

• Not hard to build (later...)

• Note: actually we have three threads per client!
runClient :: Server -> Client -> IO ()
runClient server@Server{..} client@Client{..} = concurrently send receive
where
  send = join $ atomically $ do
    k <- readTVar clientKicked
    case k of
      Just reason -> return $ hPutStrLn clientHandle $ "You have been kicked: " ++ reason
      Nothing -> do
        msg <- readTChan clientSendChan
        return $ do
          continue <- handleMessage server client msg
          when continue $ send

  receive = forever $ do
    msg <- hGetLine clientHandle
    atomically $ sendMessage client $ Command msg

Note we check clientKicked first – cannot process any other commands if we are kicked.
handleMessage :: Server -> Client -> Message -> IO Bool
handleChange server client@Client{..} message =
  case message of
    Notice msg        -> output $ "*** " ++ msg
    Tell name msg     -> output $ "*" ++ name ++ "*: " ++ msg
    Broadcast name msg -> output $ "<" ++ name ++ ">: " ++ msg
    Command msg ->
      case words msg of
        ["/kick", who] -> do
          join $ atomically $ kick server client who
          return True
        "/tell" : who : what -> do
          atomically $ tell server clientName who (unwords what)
          return True
        ["/quit"] ->
          return False
        ("/":_):_ -> do
          hPutStrLn clientHandle $ "Unrecognised command: " ++ msg
          return True
        _ -> do
          atomically $ broadcast server $ Broadcast clientName msg
          return True
  where
    output s = do hPutStrLn clientHandle s; return True
tell :: Server -> ClientName -> ClientName -> String -> STM ()
tell Server{..} from who msg = do
  clientmap <- readTVar clients
  case Map.lookup who clientmap of
    Nothing -> return ()
    Just client -> sendMessage client $ Tell from msg

kick :: Server -> Client -> ClientName -> STM (IO ()
kick Server{..} client@Client{clientHandle=handle} who = do
  clientmap <- readTVar clients
  case Map.lookup who clientmap of
    Nothing ->
      return $ hPutStrLn handle (who ++ " is not connected")
    Just victim -> do
      writeTVar (clientKicked victim) $
      Just ("by " ++ clientName client)
      return $ hPutStrLn handle ("you kicked " ++ who)
Demo
Defining ‘concurrently’
(the nice way, using STM;
for the hard way see the notes)
data Async a = Async ThreadId (TMVar (Either SomeException a))

forkFinally :: IO a -> (Either SomeException a -> IO ()) -> IO ThreadId
forkFinally action and_then =
  mask $ \restore ->
  forkIO $ try (restore action) >>= and_then

async :: IO a -> IO (Async a)
async action = do
  var <- newEmptyTMVarIO
  t <- forkFinally action (\r -> atomically $ putTMVar var r)
  return (Async t var)

waitSTMThrow :: Async a -> STM a
waitSTMThrow (Async _ var) = do
  r <- readTMVar var
  case r of
    Left a -> return a
    Right e -> throwSTM e

cancel :: Async a -> IO ()
cancel (Async t _) = throwTo t ThreadKilled
withAsync :: IO a -> (Async a -> IO b) -> IO b
withAsync action inner = bracket (async action) cancel inner

concurrently :: IO a -> IO b -> IO ()
concurrently left right =
    withAsync left $ \a ->
    withAsync right $ \b ->
    atomically $
    (void $ waitSTMThrow a)
    `orElse`
    (void $ waitSTMThrow b)
Dealing with clients that must respond both to network and local events:

- use two threads
  - forward messages from the network to a local channel
  - then use STM with `orElse` to multiplex events from the channel and other event sources (`clientKicked` in our case)

- to manage the two threads, we used the `concurrently` abstraction:

```
concurrently :: IO a -> IO b -> IO ()
```

- some similarity with Erlang’s linked processes
Summary (cont)

• Global consistency guarantees are easier to manage with STM – just do atomic operations on the state.
  – No need to worry about fine-grained vs. coarse-grained locking

• Take particular care with atomicity at startup/shutdown
  – no loopholes where a thread might be left behind if something goes wrong