Network servers

Haskell and Cryptocurrencies

Dr. Andres Löh, Well-Typed LLP
Dr. Lars Brünjes, IOHK
Dr. Polina Vinogradova, IOHK

2019-01-29

Goals

- Revisit servers.
- `MVar`s,
- Servers with state.
- More about `Async` and `STM`.

Credits

This lecture follows parts of Chapters 10 and 12 of Simon Marlow's book "Parallel and Concurrent Programming in Haskell" rather closely.

All errors are of course our own.

Shouting server, again

```
main :: IO ()
main = do
  s <- listenOn (PortNumber 8765)
  forever $ do
    (h, _, _) <- accept s
    forkIO $ handleClient h
handleClient :: Handle -> IO ()
handleClient h = do
  hSetBuffering h LineBuffering
  forever $ do
    line <- hGetLine h
    hPutStrLn h (map toUpper line)
```

Observation

- Every client process is completely independent of each other.
- Therefore, no state is needed in the server.

Adding state

A server that counts the clients

Goal

Server accepts connections and reports on request the number of currently connected clients.

We need to maintain the current number of clients as state.

Options for maintaining state

- A `TVar` with atomic access.

Options for maintaining state

- A `TVar` with atomic access.
- An `MVar` with synchronized access.

Options for maintaining state

- A `TVar` with atomic access.
- An `MVar` with synchronized access.
- An `IORef`, as long as we update it atomically.

`MVar`s

- An `MVar a` is mutable location that is either empty or contains a value of type `a`.
- It has two fundamental operations:
  - `putMVar :: MVar a -> a -> IO ()` which fills an `MVar` if it is empty and blocks otherwise, and
  - `takeMVar :: MVar a -> IO a` which empties an `MVar` if it is full and blocks otherwise.

`MVar`s (contd.)

They can be used in multiple different ways:

- as synchronized mutable variables,
- as channels, with `takeMVar` and `putMVar` as receive and send, and
- as a binary semaphore `MVar ()`, with `takeMVar` and `putMVar` as wait and signal.

Applicability of `MVar`s

- `MVar`s offer more flexibility than `IORef`s, but less flexibility than `TVar`s.
- They are appropriate for building synchronization primitives and performing simple interthread communication;
- however they are very simple and susceptible to race conditions, deadlocks or uncaught exceptions.
- Do not use them if you need to perform larger atomic operations such as reading from multiple variables: Use `TVar`s instead.

`MVar` fairness guarantee

- No thread can be blocked indefinitely on an `MVar` unless another thread holds that `MVar` indefinitely.
- One usual implementation of this fairness guarantee is that threads blocked on an `MVar` are served in a first-in-first-out fashion, but this is not guaranteed in the semantics.
- `TVar`s do *not* give the same guarantee.

`MVar` API (excerpt)

```haskell
newEmptyMVar :: IO (MVar a)
newMVar :: a -> IO (MVar a)
```

```haskell
putMVar :: MVar a -> a -> IO ()
takeMVar :: MVar a -> IO a
readMVar :: MVar a -> IO a
```

```haskell
tryPutMVar :: MVar a -> a -> IO Bool
tryTakeMVar :: MVar a -> IO (Maybe a)
tryReadMVar :: MVar a -> IO (Maybe a)
```

```haskell
modifyMVar :: MVar a -> (a -> IO (a, b)) -> IO b
withMVar :: MVar a -> (a -> IO b) -> IO b
```

Creating and maintaining a TVar

```haskell
main :: IO ()
main = do
  s <- listenOn (PortNumber 8765)
  conns <- newTVarIO 0  -- new
  forever $ do
    (h, _, _) <- accept s
    forkFinally   -- changed
      (handleClient h conns)
      (const $ removeClient h conns)
```

`forkFinally`

From `Control.Concurrent`:

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

Executes the second argument once the thread is about to finish, whether normally or via an exception.

Cleaning up a client

```haskell
removeClient :: Handle -> TVar Int -> IO ()
removeClient h conns = do
  atomically $ modifyTVar' conns (\ x -> x - 1)
  hClose h
```

We can also use this to explicitly close the handle, which is better than relying on garbage collection.

Handling a client

```haskell
handleClient :: Handle -> TVar Int -> IO ()
handleClient h conns = do
  atomically $ modifyTVar' conns (\ x -> x + 1)
  hSetBuffering h LineBuffering
  forever $ do
    line <- hGetLine h
    count <- readTVarIO conns
    hPrint h count
```

Reporting changes asynchronously

Goal

Rather than reporting the number of connected clients on request, the server should asynchronously report the number of clients whenever it changes.

Reporting changes asynchronously

Goal

Rather than reporting the number of connected clients on request, the server should asynchronously report the number of clients whenever it changes.

One option is to maintain the list of handles rather than the number of clients.

Maintaining a list of handles

```haskell
main :: IO ()
main = do
  s <- listenOn (PortNumber 8765)
  conns <- newTVarIO []  -- changed type
  forkIO (monitor 0 conns)  -- new
  forever $ do
    (h, _, _) <- accept s
    forkFinally
      (handleClient h conns)
      (const $ removeClient h conns)
```

Registering and removing a client

```haskell
removeClient :: Handle -> TVar [Handle] -> IO ()
removeClient h conns = do
  atomically $ modifyTVar' conns (delete h)
  hClose h
```

```haskell
handleClient :: Handle -> TVar [Handle] -> IO ()
handleClient h conns = do
  hSetBuffering h LineBuffering
  atomically $ modifyTVar conns (h:)
  forever $ hGetLine h  -- hack
```

Monitoring changes

```haskell
monitor :: Int -> TVar [Handle] -> IO ()
monitor count conns = do
  (handles, newcount) <- atomically $ do
    handles <- readTVar conns
    let newcount = length handles
    when (count == newcount) retry
    return (handles, newcount)
  mapM_ (\ h -> hPrint h newcount) handles
  monitor newcount conns
```

Distributing handles is problematic

- We potentially have to deal with disappearing handles / exceptions in several places.
- If multiple parts of the program access the same handle, outputs and inputs could become interleaved in unexpected ways.

Distributing handles is problematic

- We potentially have to deal with disappearing handles / exceptions in several places.
- If multiple parts of the program access the same handle, outputs and inputs could become interleaved in unexpected ways.

A better solution:

- Let every client handler deal with communication alone.
- Use a (broadcast) channel for the messages.

Channels

Channels (STM)

A thread-safe FIFO queue.

```haskell
data TChan a  -- abstract

newTChan            :: STM (TChan a)
newBroadcastTChan :: STM (TChan a)  -- write-only

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

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

On channels

- Items written into a channel do not get lost.
- Items will be read in the order they have been written (first-in first-out, FIFO).
- Items can be read only once (even if accessed concurrently).

On channels

- Items written into a channel do not get lost.
- Items will be read in the order they have been written (first-in first-out, FIFO).
- Items can be read only once (even if accessed concurrently).

Using `dupTChan`:

- We create a new empty channel.
- Items written to the new or original channel will be available on both.
- So in this case, any item written can be read twice.

The main program

```haskell
main :: IO ()
main = do
  s <- listenOn (PortNumber 8765)
  conns <- newTVarIO 0  -- back to integers
  bchan <- newBroadcastTChanIO  -- new
  forkIO (monitor 0 conns bchan)
  forever $ do
    (h, _, _) <- accept s
    forkFinally
      (handleClient h conns bchan)   -- changed
      (const $ removeClient h conns)
```

Handling new clients

```
handleClient ::
  Handle -> TVar Int -> TChan String -> IO ()
handleClient h conns bchan = do
  chan <- atomically $ dupTChan bchan
  atomically $ modifyTVar' conns (\ x -> x + 1)
  hSetBuffering h LineBuffering
  void $ input `race` output chan
  where
    input       = forever $ hGetLine h
    output chan = forever $ do
      line <- atomically $ readTChan chan
      hPutStrLn h line
```

Another `Async` function

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

- Runs two operations concurrently.
- Returns the one that finishes first.
- Cancels the other.
- Propagates possible exceptions.

Removing a client (as earlier)

```haskell
removeClient :: Handle -> TVar Int -> IO ()
removeClient h conns = do
  atomically $ modifyTVar' conns (\ x -> x - 1)
  hClose h
```

Monitoring

```haskell
monitor :: Int -> TVar Int -> TChan String -> IO ()
monitor count conns bchan = do
  newcount <- atomically $ do
    newcount <- readTVar conns
    when (count == newcount) retry
    return newcount
  atomically $ writeTChan bchan (show newcount)
  monitor newcount conns bchan
```

Implementing channels

Looking into `TChan`s

It turns out that `TChan`s are built on top of other STM primitives.

```
data TChan a =
  TChan
    (TVar (TVarList a))   -- channel head
    (TVar (TVarList a))   -- channel end
```

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

Looking into `TChan`s

It turns out that `TChan`s are built on top of other STM primitives.

```
data TChan a =
  TChan
    (TVar (TVarList a))   -- channel head
    (TVar (TVarList a))   -- channel end
```

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

- Two pointers.
- `TNil` means no element there.
- Channel end always points at `TNil`.

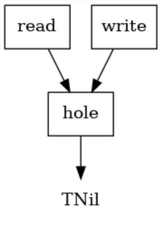
Creating a new channel

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

Creating a new channel

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

- In a new channel, head and end point to the same `TVar`.



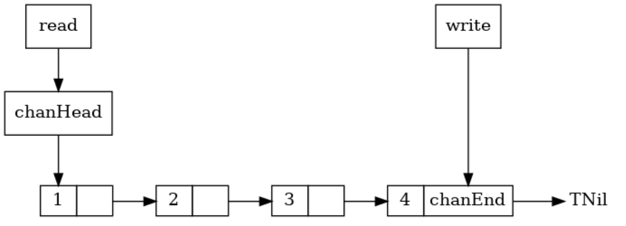
Reading from a channel

```
readTChan :: TChan a -> STM a
readTChan (TChan read _write) = do
  chanHead <- readTVar read
  items    <- readTVar chanHead
  case items of
    TNil          -> retry
    TCons a rest -> do
      writeTVar read rest
      return a
```

- Only the head (read) pointer is used.
- If nothing there, `TNil`, we `retry`.
- Otherwise, we move the pointer to the right.

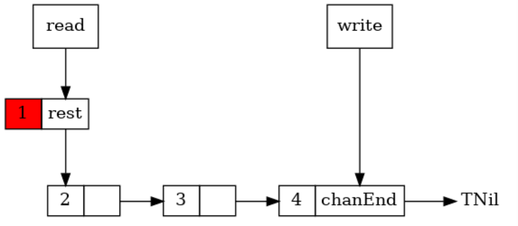
Reading from a channel

```haskell
readTChan :: TChan a -> STM a
readTChan (TChan read _write) = do
  chanHead <- readTVar read
  items    <- readTVar chanHead
  case items of
    TNil           -> retry
    TCons a rest -> do
      writeTVar read rest
      return a
```



Reading from a channel

```
readTChan :: TChan a -> STM a
readTChan (TChan read _write) = do
  chanHead <- readTVar read
  items    <- readTVar chanHead
  case items of
    TNil          -> retry
    TCons a rest -> do
      writeTVar read rest
      return a
```



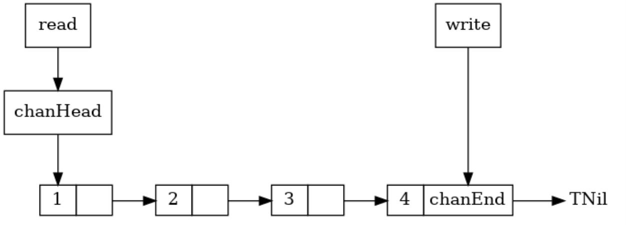
Writing to a channel

```haskell
writeTChan :: TChan a -> a -> STM ()
writeTChan (TChan _read write) a = do
  chanEnd    <- readTVar write
  newChanEnd <- newTVar TNil
  writeTVar chanEnd (TCons a newChanEnd)
  writeTVar write newChanEnd
```

- Only the tail (write) pointer is used.
- Write pointer always points at `TNil`.
- We write the new item and move the pointer to the right.

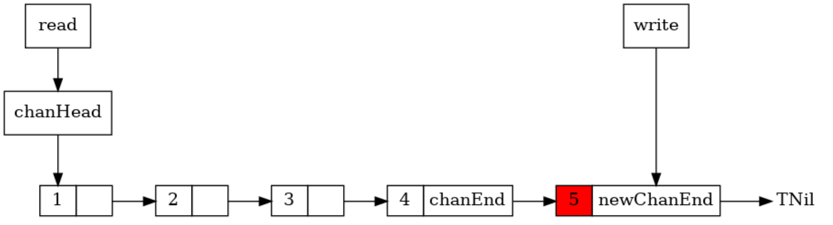
Writing to a channel

```
writeTChan :: TChan a -> a -> STM ()
writeTChan (TChan _read write) a = do
  chanEnd     <- readTVar write
  newChanEnd <- newTVar TNil
  writeTVar chanEnd (TCons a newChanEnd)
  writeTVar write newChanEnd
```



Writing to a channel

```
writeTChan :: TChan a -> a -> STM ()
writeTChan (TChan _read write) a = do
  chanEnd     <- readTVar write
  newChanEnd <- newTVar TNil
  writeTVar chanEnd (TCons a newChanEnd)
  writeTVar write newChanEnd
```



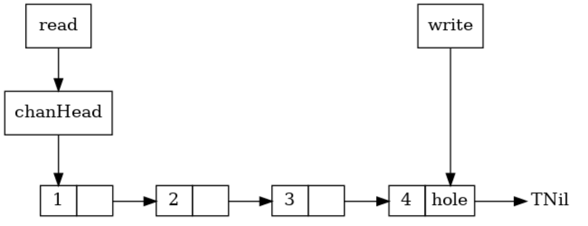
Duplicating a channel

```
dupTChan :: TChan a -> STM (TChan a)
dupTChan (TChan _read write) = do
  hole <- readTVar write
  newChanHead <- newTVar hole
  return (TChan newChanHead write)
```

- A duped channel starts out empty.
- We duplicate the hole at the write end of the channel …
- … and turn that into a new read pointer.

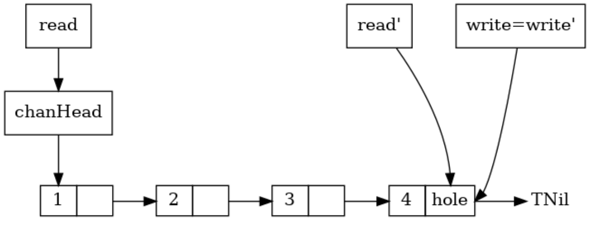
Duplicating a channel

```haskell
dupTChan :: TChan a -> STM (TChan a)
dupTChan (TChan _read write) = do
  hole <- readTVar write
  newChanHead <- newTVar hole
  return (TChan newChanHead write)
```



Duplicating a channel

```
dupTChan :: TChan a -> STM (TChan a)
dupTChan (TChan _read write) = do
  hole <- readTVar write
  newChanHead <- newTVar hole
  return (TChan newChanHead write)
```



Broadcasting channels

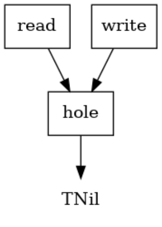
```haskell
newTChan :: STM (TChan a)
newTChan = do
  hole  <- newTVar TNil
  read  <- newTVar hole
  write <- newTVar hole
  return (TChan read write)
```

This is the old version:

- If a channel is written to but never read from, the items cannot be garbage collected and stay in memory.

Broadcasting channels

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



Broadcasting channels

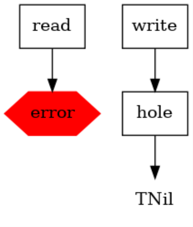
```
newBroadcastTChan :: STM (TChan a)
newBroadcastTChan = do
  hole  <- newTVar TNil
  read  <- newTVar (error "must use dupTChan")
  write <- newTVar hole
  return (TChan read write)
```

This is the new version:

- This is a write-only channel.
- Readable channels can still be created via `dupTChan`.

Broadcasting channels

```
newBroadcastTChan :: STM (TChan a)
newBroadcastTChan = do
  hole  <- newTVar TNil
  read  <- newTVar (error "must use dupTChan")
  write <- newTVar hole
  return (TChan read write)
```



A chat server

Informal specification

- When client connects, server requests nickname.
  Nickname must be fresh, otherwise server will ask again.
- Each line from the client is one of the following:

  ```
  /tell <name> <message>
  /kick <name>
  /quit
  <message>
  ```
- Broadcast notifications on (dis)connects.

Demo