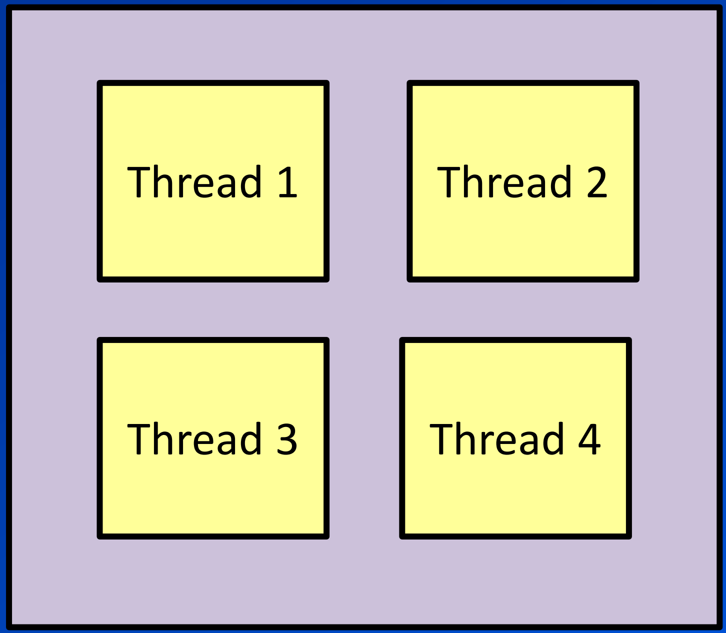
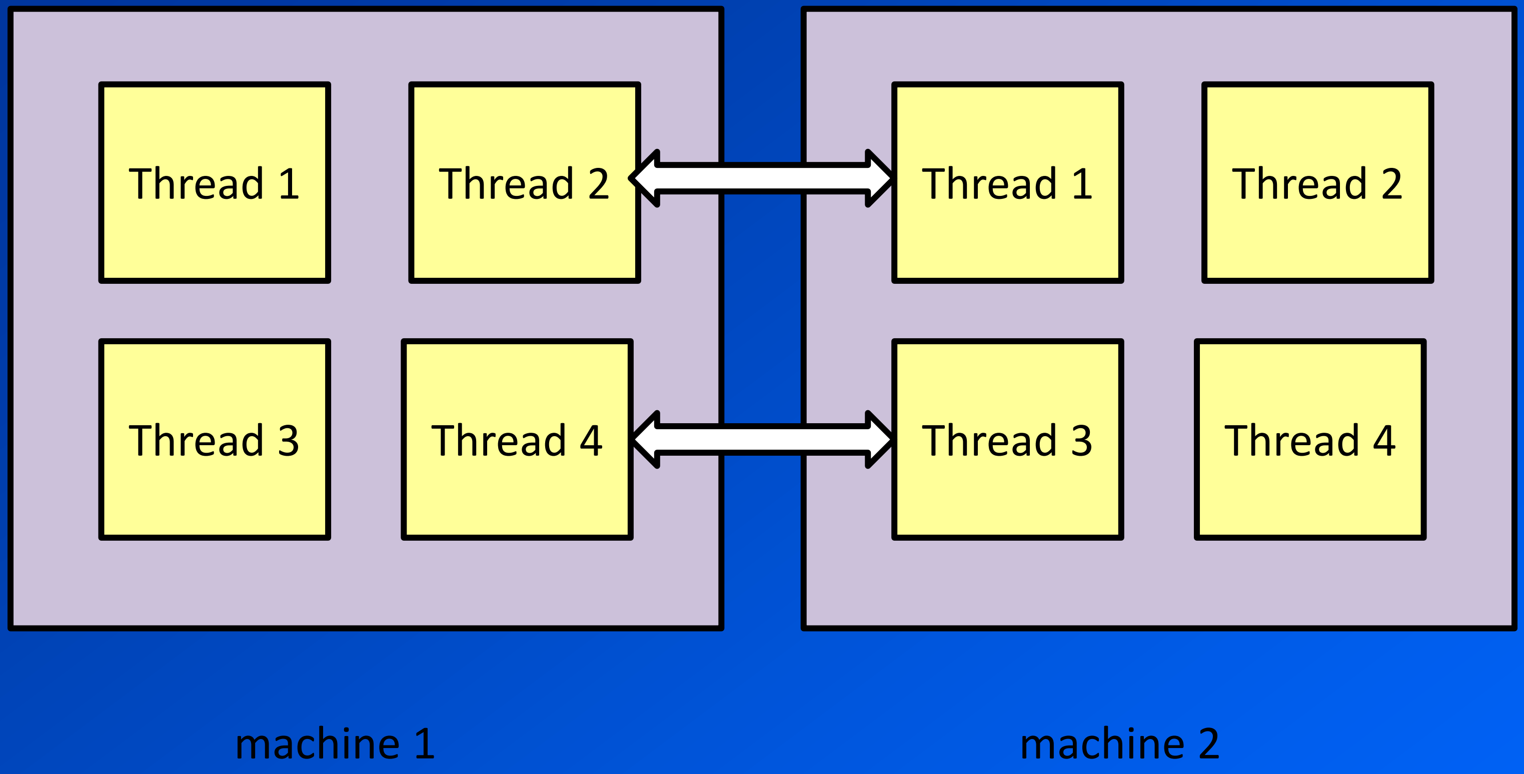
Parallel & Concurrent Haskell 6: Distributed programming with Cloud Haskell

Simon Marlow


Thread 1
Thread 2
Thread 3
Thread 4



- Concurrency across multiple *machines*

Why?

- More *parallelism*
  - lots of machines = lots more processing power
  - think Amazon EC2, Google AppEngine, Microsoft Azure
  - or just a few machines on your local network
  - ... or even just a few processes on a single multicore machine
    - separate processes have separate heaps and can do independent GC
    - trade communication cost for better locality and maybe better scaling
- Spread your data around
  - a lot of machines can keep a large database in memory

Distributed programming

- Is it just like concurrent programming?
  - No: distribution is different.

	Communication cost	**Partial failure**	**Global consistency (e.g. STM, atomic operations)**
Concurrent programming	Only cache effects	No partial failure	Feasible, using shared memory
Distributed programming	Moving data is much more expensive, so we want to put it under programmer control	Individual nodes can go down, or there can be network outages	Not feasible

The basic idea

- All communication is done with *message passing*
- (just like Erlang)
- Primitive operations:
  - fork a *process*
  - send a message to another process
  - receive a message sent to the current process
  - (there are also channels)

How does message passing fit the platform constraints?

	Communication cost	Partial failure	Global consistency (e.g. STM, atomic operations)
Concurrent programming	Only cache effects	No partial failure	Feasible, using shared memory
Distributed programming	Moving data is much more expensive, so we want to put it under programmer control	Individual nodes can go down, or there can be network outages	Not feasible
Message passing	Send and receive are explicit	Special messages are sent to indicate failure	Message passing does not allow a globally consistent view

Using Cloud Haskell

- It is "just a library", implemented using Concurrent Haskell and the network package
- Install the remote package:

```
$ cabal install remote derive
```

- The main module is "Remote"

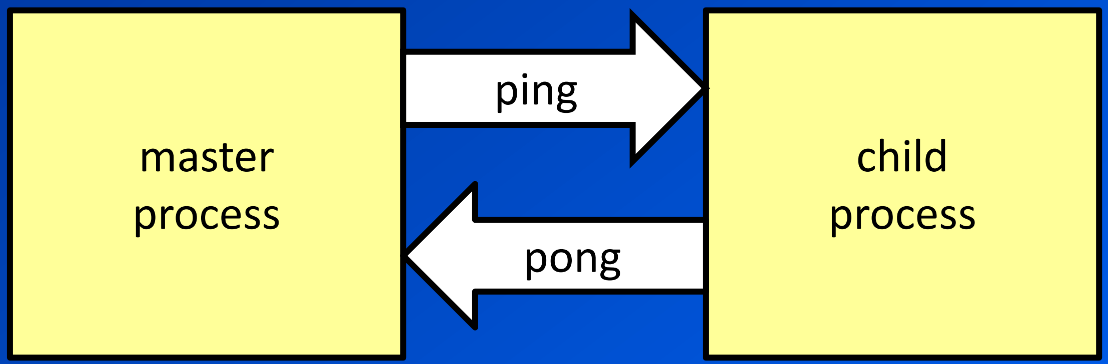
```
import Remote
```

- (remote is really just a prototype, but it works. There is a rewrite in progress.)

Overview

- This lecture: I will cover all the pieces that you will need to solve the exercise
  - spawning processes and simple message-passing
  - running a distributed program
  - typed channels
  - handling failure

First example: ping-pong



- The master process creates the child process
- master: send a ping message to the child
- child: receive the ping
- child: reply with pong

- First, define the type of messages

```
data Message = Ping ProcessId
             | Pong ProcessId
  deriving Typeable

$( derive makeBinary ''Message )
```

Typeable is required for messages

Binary is also required for messages: derive uses TH to generate the instance automatically

- ProcessId: process identifiers
  - processes are like threads, except they can be created on another machine (*node*), and can communicate using messages
- Why do messages contain ProcessId?
  - So that we know where to send the response

- Shorthand: Serializable

```
class    (Binary a, Typeable a) => Serializable a
instance (Binary a, Typeable a) => Serializable a
```

- Sending a message

```
send   :: Serializable a => ProcessId -> a -> ProcessM ()
```

Destination process

Any Serializable message

- What is ProcessM?

```
data ProcessM  -- instance Monad, MonadIO
```

  – A layer over the IO monad
  – All distributed operations are in ProcessM
  – use ‘liftIO’ to perform an IO operation

- Receiving a message

```
expect :: Serializable a => ProcessM a
```

returned message

- a process can receive messages of any type, so at any time the queue may contain messages of multiple different types
- expect returns the first message in the queue of type a
- How does it know which type you want?
  - by the context.
  - just like ‘read’, e.g. read “3” :: Int

- The "ping server" (child process)

```
pingServer :: ProcessM ()
pingServer = do
  Ping from <- expect
  mypid <- getSelfPid
  send from (Pong mypid)
```

```
data ProcessM  -- instance of Monad, MonadIO

data NodeId    -- instance of Eq, Ord, Typeable, Binary
data ProcessId -- instance of Eq, Ord, Typeable, Binary

getSelfPid  :: ProcessM ProcessId

send   :: Serializable a => ProcessId -> a -> ProcessM ()
expect :: Serializable a => ProcessM a
```

- A bit of boilerplate...

```
$( remotable ['pingServer] )
```

- this is a bit of TH magic that makes it possible to execute pingServer on a remote machine
- it generates:

```
pingServer__closure :: Closure (ProcessM ())
```

- which we use on the next slide...

- The master process

```
master :: ProcessM ()
master = do
  node <- getSelfNode

  say $ printf "spawning on %s" (show node)
  pid <- spawn node pingServer__closure

  mypid <- getSelfPid
  say $ printf "pinging %s" (show pid)
  send pid (Ping mypid)

  Pong _ <- expect
  say "pong."
  terminate
```

```
getSelfNode :: ProcessM NodeId

spawn   :: NodeId
        -> Closure (ProcessM ())
        -> ProcessM ProcessId

say     :: String -> ProcessM ()
```

- Starting it all up

```
main = remoteInit (Just "config") [Main.__remoteCallMetaData] $
         \_ -> master
```

This also came from the $(remotable ...) earlier

```
cfgRole MASTER
cfgHostName localhost
cfgKnownHosts localhost
```

This config file will be enough for our examples

Run the program...

- Summary:
  - processes are in the ProcessM monad
  - each process has a message queue
  - send to a process with send (any Serializable type)
  - receive a message with expect (type depends on context)
  - create a process with spawn (function to spawn must be declared remotable)
- There is some boilerplate:
  - $(derive mkBinary ...), $(remotable ...)
  - don’t worry about it, just follow the examples
  - some of this will go away in the future

- The ping example is as simple as it gets
  - only one node so far
  - no failure handling
- Let’s extend it to multiple nodes next
  - a *node* is basically another instance of the program running, either on the same machine or on a different machine

- The master process

```haskell
master :: ProcessM ()
master = do
  peers <- getPeers

  let workers = findPeerByRole peers "WORKER"

  ps <- forM workers $ \nid -> do
          say $ printf "spawning on %s" (show nid)
          spawn nid pingServer__closure

  mypid <- getSelfPid

  forM_ ps $ \pid -> do
    say $ printf "pinging %s" (show pid)
    send pid (Ping mypid)

  waitForPongs ps
  terminate
```

```haskell
getPeers :: ProcessM [NodeId]
findPeerByRole :: [NodeId] -> String -> [NodeId]
```

```haskell
waitForPongs :: [ProcessId] -> ProcessM ()
waitForPongs [] = return ()
waitForPongs ps = do
  m <- expect
  case m of
    Pong p -> waitForPongs (filter (/= p) ps)
    _  -> say "MASTER received ping" >> terminate
```

- main is a little different:

```
main = remoteInit
          (Just "config")
          [Main.__remoteCallMetaData]
          initialProcess

initialProcess :: String -> ProcessM ()
initialProcess "WORKER" = receiveWait []
initialProcess "MASTER" = master
```

- The intial process on each node has to distinguish between the node *roles* and do something different
  - MASTER node: start the master process
  - WORKER nodes: just wait

How does a node know its role?

- Remember the config file?

```
cfgRole MASTER
cfgHostName localhost
cfgKnownHosts localhost
```

Here is the role

- Might not be convenient to have different config files if we're starting multiple nodes on the same machine. Alternative:

```
$ ./prog -cfgRole=WORKER
```

How do the nodes find each other?

- Magic ☺
- (actually, there is “automatic node discovery” that works by sending a UDP broadcast to port 38813)
- Start the worker nodes first, then the master node
  - so that when the master process starts up it can see all the workers.
- You can do manual node discovery using the config file (we won’t cover this – see the docs)

Run the program...

Typed Channels

- So far we have been sending messages directly to a process
  - This is the Erlang way
  - It is a bit unHaskellish, because the processes message queue has messages of multiple types, and we have to do dynamic type checking
  - the alternative: typed channels

- The Typed Channel API:

```
data SendPort a        -- instance of Typeable, Binary
data ReceivePort a

newChannel        :: Serializable a
                  => ProcessM (SendPort a, ReceivePort a)

sendChannel       :: Serializable a
                  => SendPort a -> a -> ProcessM ()

receiveChannel    :: Serializable a
                  => ReceivePort a -> ProcessM a
```

- Note: SendPort is serialisable, but ReceivePort is not!
  - the destination for a message cannot change or be duplicated, because we would have to tell all the senders

- Ping-pong with typed channels
  - basic idea: Ping message contains the channel to respond on
  - no need for a Pong constructor; the pong message is just the ProcessId sent down the channel

```
data Message = Ping (SendPort ProcessId)
  deriving Typeable

$( derive makeBinary ''Message )
```

- Modifying pingServer is straightforward:

```
pingServer :: ProcessM ()
pingServer = do
  Ping chan <- expect
  mypid <- getSelfPid
  sendChannel chan mypid
```

- Modifying pingServer is straightforward:

```
master = do
  peers <- getPeers

  let workers = findPeerByRole peers "WORKER"

  ps <- forM workers $ \nid -> do
          say $ printf "spawning on %s" (show nid)
          spawn nid pingServer__closure

  mypid <- getSelfPid

  ports <- forM ps $ \pid -> do
    say $ printf "pinging %s" (show pid)
    (sendport,recvport) <- newChannel
    send pid (Ping sendport)
    return recvport

  forM_ ports $ \port -> do
     p <- receiveChannel port
     return ()
```

just receive on each channel; simpler than previous waitForPongs loop

- Hang on a minute. We’re only using a channel for the pong. What about the ping?
- Let’s try:

```
do
   (s1,r1) <- newChannel
   spawn nid (pingServer__closure r1)

   (s2,r2) <- newChannel
   sendChannel s1 (Ping s2)

   receiveChannel r2
```

- Hang on a minute. We're only using a channel for the pong. What about the ping?
- Let's try:

```
do
    (s1,r1) <- newChannel
    spawn nid (pingServer__closure r1)

    (s2,r2) <- newChannel
    sendChannel s1 (Ping s2)

    receiveChannel r2
```

- This requires serialising the ReceivePort, which is not allowed

- The fix is a bit ugly:

```
do
   (s,r) <- newChannel  -- throw-away channel
   spawn nid (pingServer__closure s)
   ping <- receiveChannel r

   (sendpong,recvpong) <- newChannel
   sendChannel ping (Ping sendpong)

   receiveChannel recvpong
```

Typed channels: conclusion

- Useful when you are sending a message that needs a response
  - the code that receives the response knows where it came from
  - sometimes allows message types to be simplified
  - should be faster – no type tagging required (but current implementation is slower)
- Not so useful when you need to spawn a process and then send a message to it
  - because we can't send the ReceivePort
- Not covered here: ReceivePorts can be merged, so you can listen on several simultaneously.

Handling Failure

- One of the main benefits of using remote is that it helps us manage failure:
  - network or node failure
  - process failure (exceptions)

The Erlang Philosophy

Let it crash!

- Don't try to program fine-grained error handling
  - it is hard to get right, and hard to test
  - anyway, we have to handle the case when a node goes down completely
  - just treat every error the same way: let the process crash
  - when a process crashes, a supervisor restarts it in a known-good state
  - know where your state is, and how to recover a known-good state

- Recall the code for pingServer (the old version, not using channels):

```
pingServer :: ProcessM ()
pingServer = do
   Ping from <- expect
   mypid <- getSelfPid
   send from (Pong mypid)
```

```
data Message = Ping ProcessId
             | Pong ProcessId
```

This pattern match can fail!

- If the pattern match fails, an exception will be raised, which causes the process to crash

- Catching the crash

```
withMonitor :: ProcessId -> ProcessM a -> ProcessM a
```

Process that we want to monitor for failure

Monitoring lasts for the duration of this action

- While a process is being monitored, failures result in a ProcessMonitorException message being sent to the monitoring process

```
data ProcessMonitorException
  = ProcessMonitorException ProcessId SignalReason
```

- But how do we receive the ProcessMonitorException when we are waiting for the Pong message at the same time?

```
receiveWait
   [ match $ \p -> do ...
   , match $ \q -> do ...
   ]
```

```
receiveWait :: [MatchM q ()] -> ProcessM q

match :: Serializable a => (a -> ProcessM q) -> MatchM q ()
```

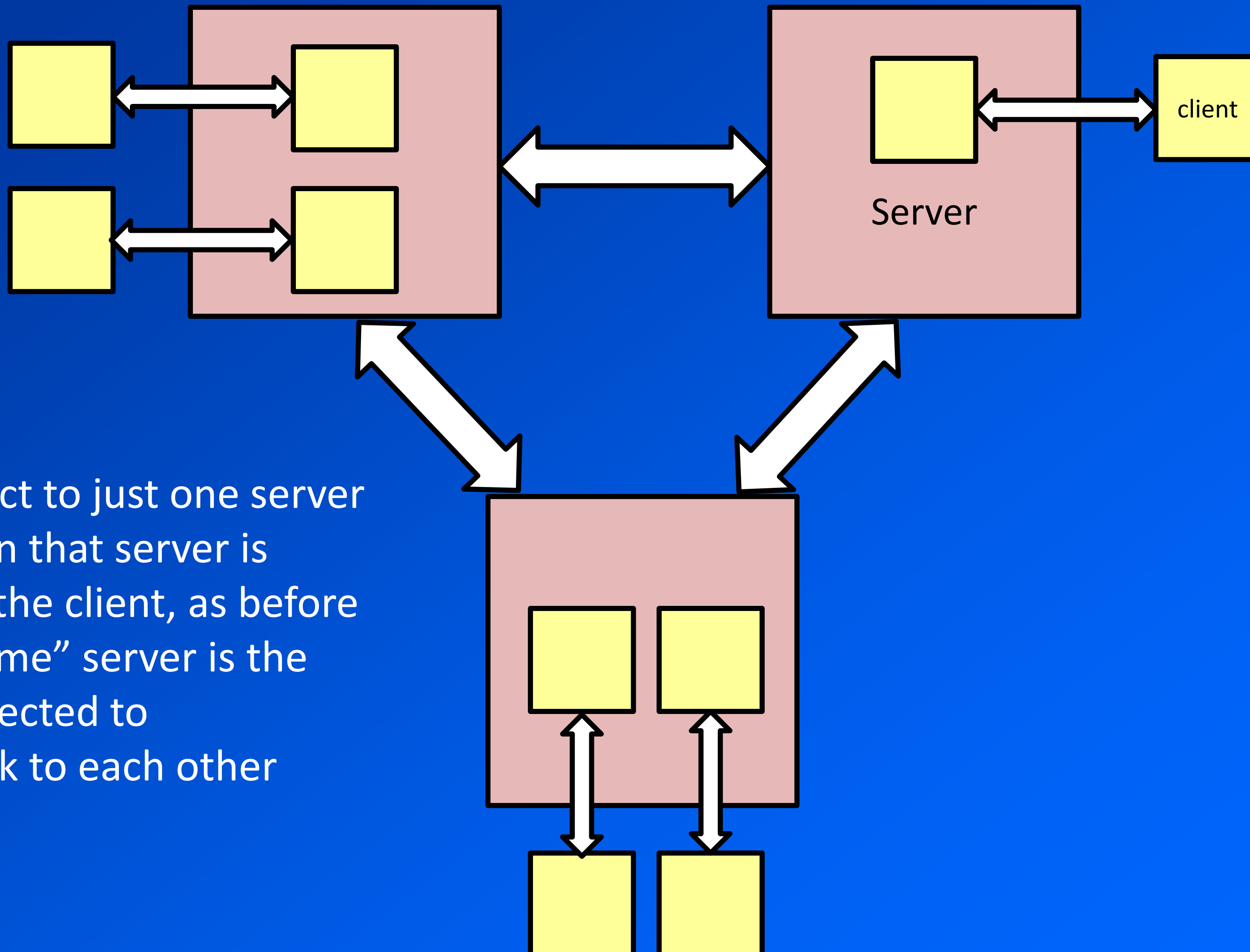
- Demonstrate catching the failure:

```
withMonitor pid $ do
   send pid (Pong mypid)
   receiveWait
     [ match $ \(Pong _) -> do
        say "pong."
        terminate
     , match $ \(ProcessMonitorException pid reason) -> do
        say $ printf "process %s died: %s" (show pid)
                                           (show reason)
        terminate
     ]
```

Run the program...

A distributed chat server

- In the previous lecture we made a concurrent chat server, out of forkIO and STM
- Now we'll make it distributed
- Should we replace all the threads with processes, and use ProcessM throughout?
  - We could, but there is no need.
  - We can continue to use the concurrent server mostly as-is, but wrap it in some distributed logic
  - This will be a mixed concurrent/distributed app
- Code is in remote-chat/chat.hs



- Clients connect to just one server
- One thread on that server is dedicated to the client, as before
- A client's "home" server is the one it is connected to
- All servers talk to each other

- Two kinds of client:

```haskell
data Client
  = ClientLocal   LocalClient
  | ClientRemote  RemoteClient
```

new

```haskell
data RemoteClient = RemoteClient
       { remoteName :: ClientName
       , clientHome :: ProcessId
       }
```

new

```haskell
data LocalClient = LocalClient
       { localName      :: ClientName
       , clientHandle   :: Handle
       , clientKicked   :: TVar (Maybe String)
       , clientSendChan :: TChan Message
       }
```

- Server type has more fields:

```
data Server = Server
  { clients   :: TVar (Map ClientName Client)
  , proxychan :: TChan (ProcessM ())
  , servers   :: TVar [ProcessId]
  , spid      :: ProcessId
  }
```

as before

see below

all the other servers

current server's pid

- what's this proxychan?
  – ordinary forkIO threads cannot perform ProcessM operations
  – but we're using forkIO threads for our clients
  – the proxychan lets forkIO threads send ProcessM operations to a process for execution.

- sending messages

```
sendLocal :: LocalClient -> Message -> STM ()
sendLocal LocalClient{..} msg =
   writeTChan clientSendChan msg

sendRemote :: Server -> ProcessId -> PMessage -> STM ()
sendRemote Server{..} pid pmsg =
   writeTChan proxychan (send pid pmsg)

data PMessage
  = MsgNewClient          ClientName ProcessId
  | MsgClientDisconnected ClientName ProcessId
  | MsgKick               ClientName ClientName
  | MsgBroadcast          Message
  | MsgSend               ClientName Message
  | MsgServers            [ProcessId]
```

new

- more sending messages

```
sendToClient :: Server -> Client -> Message -> STM ()
sendToClient server (ClientLocal client) msg =
    sendLocal client msg
sendToClient server (ClientRemote client) msg =
    sendRemote server (clientHome client)
         (MsgSend (remoteName client) msg)


sendRemoteAll :: Server -> PMessage -> STM ()
sendRemoteAll server@Server{..} pmsg = do
    pids <- readTVar servers
    mapM_ (\pid -> sendRemote server pid pmsg) pids
```

- broadcast

```
broadcast :: Server -> Message -> STM ()
broadcast server@Server{..} msg = do
    sendRemoteAll server (MsgBroadcast msg)
    broadcastLocal server msg

broadcastLocal :: Server -> Message -> STM ()
broadcastLocal server@Server{..} msg = do
    clientmap <- readTVar clients
    mapM_ sendIfLocal (Map.elems clientmap)
  where
    sendIfLocal (ClientLocal c)  = sendLocal c msg
    sendIfLocal (ClientRemote _) = return ()
```

- other changes are similarly straightforward

- chatServer process
  - this is the process that will be started on each node

```
chatServer :: Int -> [ProcessId] -> ProcessM ()
chatServer port pids = do
  server <- newServer pids
  liftIO $ forkIO (socketListener server port)
  spawnLocal (proxy server)
  forever (handleRemoteMessage server)

proxy :: Server -> ProcessM ()
proxy Server{..} =
   forever $ do
      action <- liftIO $ atomically $ readTChan proxychan
      action
```

```haskell
handleRemoteMessage :: Server -> ProcessM ()
handleRemoteMessage server@Server{..} = do
  m <- expect
  liftIO $ atomically $
    case m of
      MsgServers pids -> writeTVar servers (filter (/= spid) pids)

      MsgNewClient name pid -> do
          ok <- checkAddClient server (ClientRemote (RemoteClient name pid))
          when (not ok) $
              sendRemote server pid (MsgKick name "SYSTEM")

      MsgClientDisconnected name pid -> do
           clientmap <- readTVar clients
           case Map.lookup name clientmap of
              Nothing -> return ()
              Just (ClientRemote (RemoteClient _ pid')) | pid == pid' ->
                deleteClient server name
              Just _ ->
                return ()

      MsgBroadcast msg -> broadcastLocal server msg
      MsgSend name msg -> void $ sendToName server name msg
      MsgKick who by   -> kick server who by
```

Final thoughts

- We have only scratched the surface of remote
  - lots more in the documentation
  - http://hackage.haskell.org/packages/archive/remote/0.1.1/doc/html/Remote.html