CREATING RELIABLE, DISTRIBUTED APPLICATIONS

CLOUD HASKELL

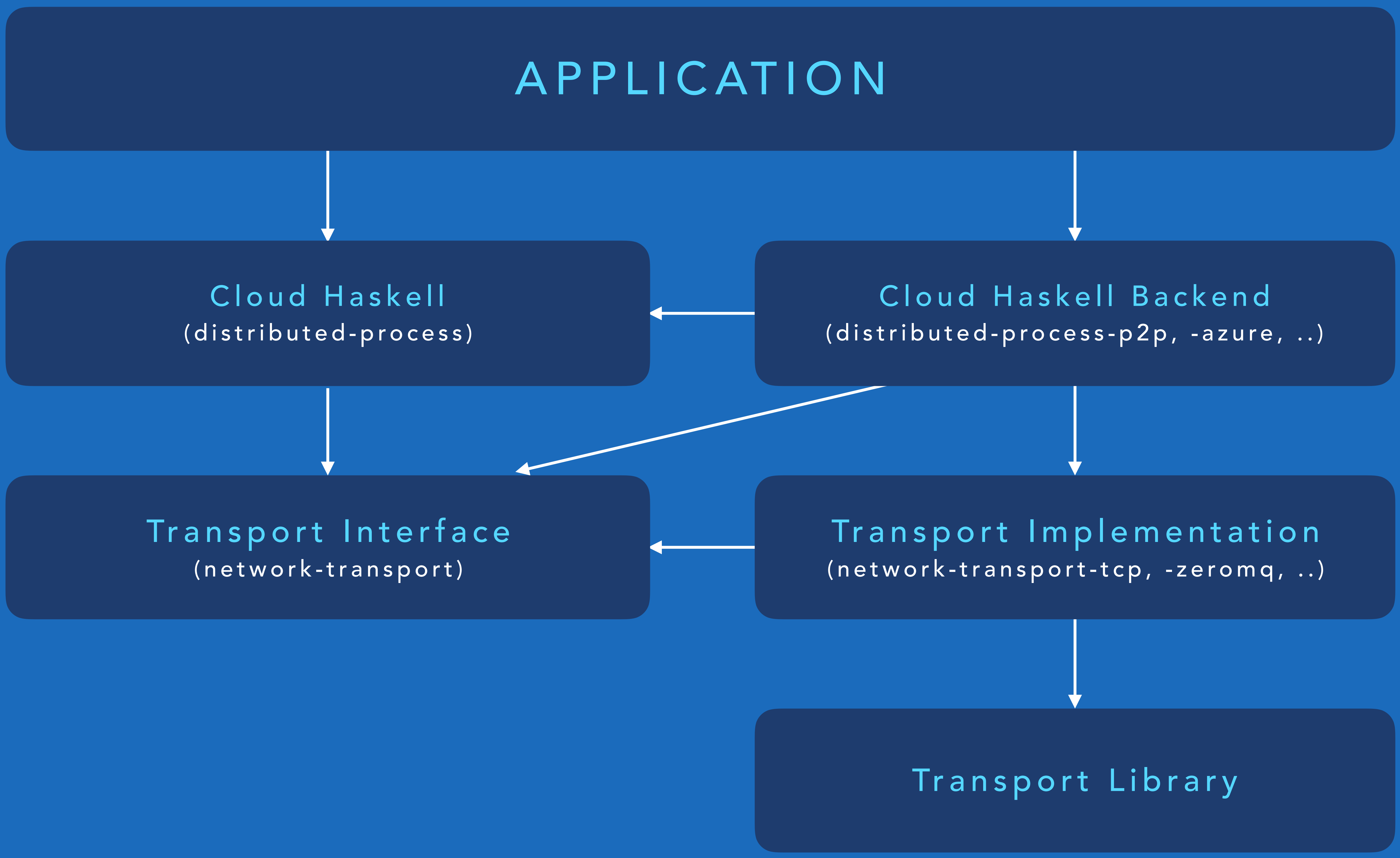
```
module AboutMe where
import Data.Human

theSpeaker :: Human
theSpeaker = Human
  { names  = [ mkName “sn” “不动点帕琪”
             , mkName “en” “Tyler Ling”
             , mkName “zh” “凌辉” ]
  , email  = “tylerblugandersen@gmail.com”
  , qq     = “503228590”
  , langs  = [ “Chinese”, “English”
             , “Haskell”, “C++”, “Lua” ]
  }
```

WHY HASKELL

- 纯函数：引用透明，不可变量
- 强类型：保障代码和被传递数据的正确性
- Monad：表达、控制副作用

分布式编程的挑战

- 可扩展性需求
  - 通讯后端多样（TCP/IP，UDP，ZeroMQ，In-memory，Pipeline…）
  - 节点种类丰富（物理机器，虚拟机，云服务器…）
  - 组织形式灵活（Master-slave，P2P，…）
- 节点间的交互
  - 消息的发送与接收
  - 启动远程进程
- 容错机制
  - 错误检测
  - 错误恢复

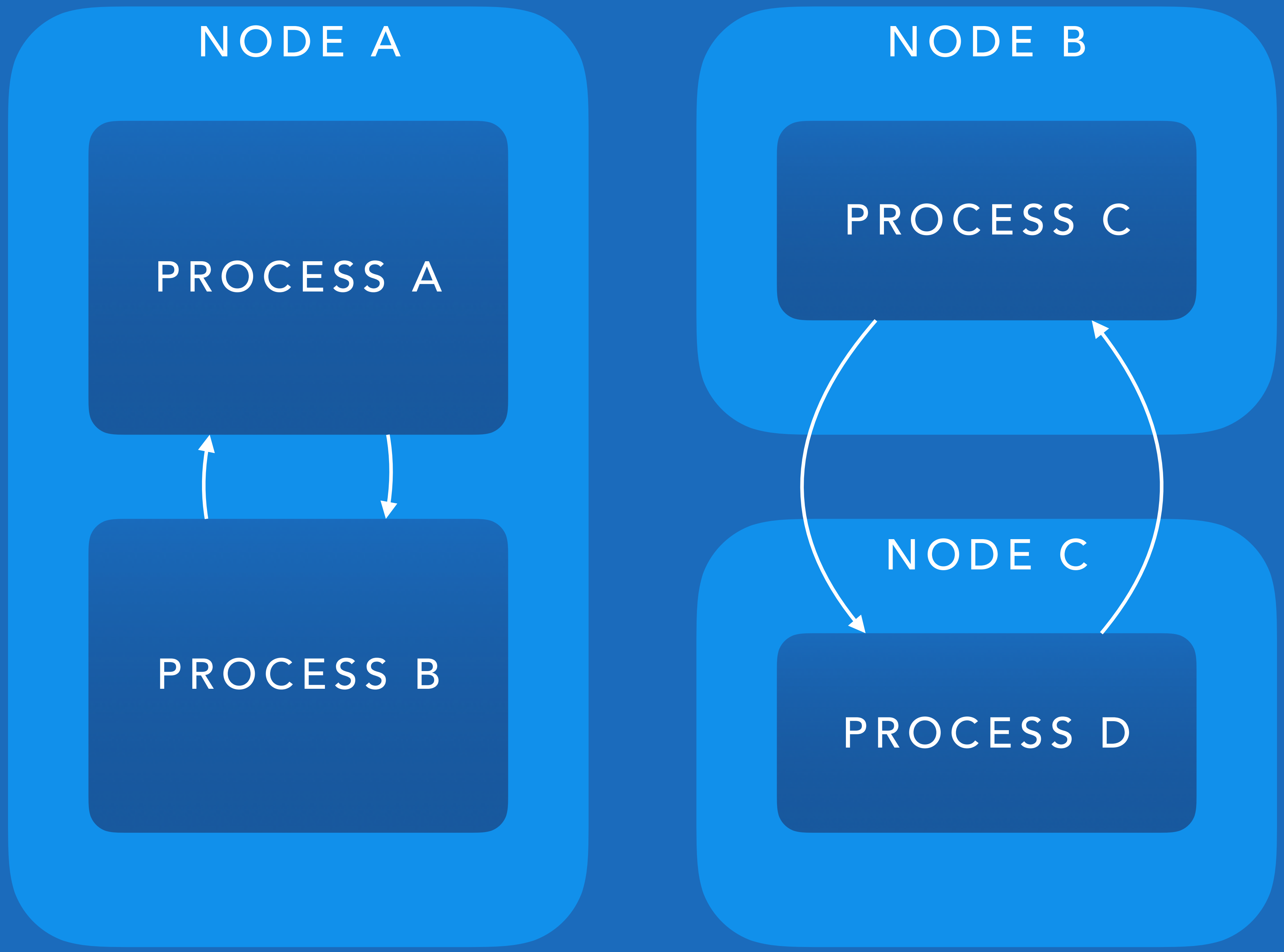

APPLICATION
Cloud Haskell
(distributed-process)
Cloud Haskell Backend
(distributed-process-p2p, -azure, ..)
Transport Interface
(network-transport)
Transport Implementation
(network-transport-tcp, -zeromq, ..)
Transport Library

"Cloud Haskell is a set of libraries that bring Erlang-style concurrency and distribution to Haskell programs."

CLOUD HASKELL/DOCUMENTATION

抽象

Actor Model



抽象

- Node
  - `newLocalNode :: Transport -> RemoteTable -> IO LocalNode`
  - `forkProcess :: LocalNode -> Process () -> IO ProcessId`
  - `runProcess  :: LocalNode -> Process () -> IO ()`
- Process
  - 轻量级
  - 无共享资源
  - 异步消息传递（send & receive）
  - 启动本地、远程进程（spawnLocal & spawn）
  - ……

PROCESS LAYER

- Message
  - 有限的可序列化数据结构。
  - 类型一致性检查。

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

- Serializable
  - Int, Char, etc.
  - (Serializable a) => [a], etc.
- Non-serializable
  - MVar, TVar, etc.
  - LocalNode, etc.

PROCESS LAYER

- Process
    - It's **`Monad`**! (also **`MonadIO`**)
    - ProcessId：用于接收数据的唯一的身份标识符。

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

- send
    - 异步
    - 绝不失败：但不保证接收端能否收到
- expect
    - 同步：没有收到预期类型的消息前堵塞线程
    - 其他不符合的消息将继续留在消息信箱中。

PROCESS LAYER

```haskell
data Ping = Ping ProcessId
          deriving (Typeable, Generic, Binary)
data Pong = Pong ProcessId
          deriving (Typeable, Generic, Binary)

ping :: Process ()
ping = do
  self <- getSelfPid
  Pong pid <- expect
  send pid (Ping self)
  ping
```

```haskell
{-# LANGUAGE DeriveDataTypeable #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE DeriveAnyClass #-}
```

PROCESS LAYER

- Multiple Choices

```
data Ping = Ping ProcessId
data Pong = Pong ProcessId

pingAndPong :: Process ()
pingAndPong = do
  self <- getSelfPid
  ..?
```

PROCESS LAYER

- Multiple Choices

```
pingAndPong :: Process ()
pingAndPong = do
  self <- getSelfPid
  receiveWait
    Ping pid -> send pid (Pong self)
    Pong pid -> send pid (Ping self)
```

PROCESS LAYER

- Multiple Choices
  - `match :: Serializable a => (a -> Process b) -> Match b`
  - `receiveWait :: [Match b] -> Process b`

```
pingAndPong :: Process ()
pingAndPong = do
  self <- getSelfPid
  receiveWait
    [ match $ \(Ping pid) -> send pid (Pong self)
    , match $ \(Pong pid) -> seld pid (Pong self) ]
```

PROCESS LAYER

- Multiple Choices
  - `match :: Serializable a => (a -> Process b) -> Match b`
  - `receiveWait :: [Match b] -> Process b`
  - `receiveTimeout :: Int -> [Match b] -> Process (Maybe b)`
  - `matchIf :: Serializable a => (a -> Bool) -> (a -> Process b) -> Match b`
  - `matchAny :: (Message -> Process b) -> Match b`
  - ……

```
expect :: Serializable a => Process a
expect = receiveWait [ match return ]
```

PROCESS LAYER

- What if…

```
Process 1> send pid2 $ length [1, 2, 3]
Process 2> v <- except
           send pid3 (v + 2 :: Double)

Process 3> Excuse me?
```

"Let there be type!"

FIXED-POINT PATCHY

PROCESS LAYER

- Typed Channel
  - SendPort (Serializable)：发送端。
  - ReceivePort (Non-serializable)：接收端。

```
newChan :: Serializable a => Process (SendPort a, ReceivePort a)
sendChan :: Serializable a => SendPort a -> a -> Process ()
receiveChan :: Serializable a => ReceivePort a -> Process a
receiveChanTimeout :: Serializable a => Int -> ReceivePort a -> Process (Maybe a)

mergePortsBiased :: Serializable a => [ReceivePort a] -> Process (ReceivePort a)
mergePortsRR :: Serializable a => [ReceivePort a] -> Process (ReceivePort a)
```

PROCESS LAYER

```
example :: Process ()
example = do
  (sp, rp) <- newChan
  -- spawnLocal :: Process () -> Process ProcessId
  spawnLocal $ sendChan sp “Hello world”
  “Hello world” <- receiveChan rp
```

But…

spawnLocal

PROCESS LAYER

- 创建远程进程

```
sender :: SendPort String -> Process ()
sender sp = sendChan sp “Hello world”

example :: NodeId -> Process ()
example nid = do
  (sp, rp) <- newChan
  spawn ???
```

PROCESS LAYER

- 创建远程进程

```
spawn :: NodeId -> Closure (Process ()) -> Proecss ProcessId
```

Where to run

What to run

PROCESS LAYER

- 创建远程进程
  - What might be Closure?
    - 函数本体
    - 调用环境（自由变量，引用库……）

```
data Closure a = Closure function environment
```

PROCESS LAYER

- 创建远程进程
  - What might be Closure?
    - 函数本体
    - 调用环境（自由变量，引用库……）
  - 这样的 Closure 能被序列化吗?

```
instance Binary (a -> b) where
  put x = ___???
  get   = ___???
```

PROCESS LAYER

- 创建远程进程
  - What might be Closure?
    - 函数本体
    - 调用环境（自由变量，引用库……）
  - 这样的 Closure 能被序列化吗?
    - × ~~函数本体~~
    - × ~~调用环境~~

PROCESS LAYER

- 创建远程进程
    - 解决方案
        - 扩展运行时
            - 致使代码失去对序列化的精准控制。
            - 误将序列化无意义的数据与函数一同传至其他节点。
            - 一些数据经过人为处理后传输更有效率。
        - **monolithic + static value**

PROCESS LAYER

- 创建远程进程
  - 哪些函数最容易在其他节点上引用?
    - 其他节点也存在同样的代码。
    - 顶级（Top-level）。
    - 无自由变量，或者自由变量也是顶级的。

```
instance Serializable (Static a)

static :: 对于满足条件的 a => a -> Static a
unstatic :: Static a -> a
```

PROCESS LAYER

- 创建远程进程
  - 实现 Static value
    - RemoteTable (`Map String Dynamic`)
    - Since GHC 7.10：GHC.StaticPtr

```
newtype Static a = Static StaticLabel
data StaticLabel =
     StaticLabel String
   | StaticApply !StaticLabel !StaticLabel
 #if __GLASGOW_HASKELL__ >= 710
   | StaticPtr SDynamic
     -- data SDynamic = SDynamic TypeRep (StaticPtr GHC.Any)
 #endif
```

PROCESS LAYER

```
data Closure a where
  Closure :: Serializable env => Static (env -> a) -> env -> Closure a

instance Binary (Closure a) where
  put (Closure f env) = put f >> put env
  get = ????
```

Which deserializer to use?

PROCESS LAYER

```
data Closure a = Closure (Static (ByteString -> a)) ByteString
```

Deserializer

Environment

```
unclosure :: Typeable a => RemoteTable -> Closure a -> Either String a
unclosure rtable (Closure dec env) = do
  f <- unstatic rtable dec
  return (f env)
```

PROCESS LAYER

```
sender :: SendPort String -> Process ()
sender sp = sendChan sp “Hello world”

senderStatic :: Static (SendPort String -> Process ())
senderStatic = staticLabel “$sender”

decodeSendPortStatic :: Static (ByteString -> SendPort String)
decodeSendPortStatic = staticLabel “$decodeSendPort”

senderClosure :: SendPort String -> Closure (Process ())
senderClosure sp = closure decoder (encode sp)
  where decoder :: Static (ByteString -> Process ())
        decoder = senderStatic `staticCompose` decodeSendPortStatic
```

PROCESS LAYER

```
rtable :: RemoteTable
rtable =
    registerStatic "$sender" (toDynamic sender)
  . registerStatic “$decodeSendPort"
      (toDynamic (decode :: ByteString -> SendPort String))
  $ initRemoteTable
```

```
newLocalNode :: Transport -> RemoteTable -> IO LocalNode
```

“Template Haskell!”

FIXED-POINT PATCHY

PROCESS LAYER

```haskell
{-# LANGUAGE TemplateHaskell #-}

import Control.Distributed.Process
import Control.Distributed.Process.Closure
import Control.Distributed.Process.Node
import Network.Transport.TCP (createTransport, defaultTCPParameters)

sender :: SendPort String -> Process ()
sender sp = sendChan sp “Hello world”

remotable [‘sender]
-- __remoteTable :: RemoteTable -> RemoteTable

remoteTable :: RemoteTable
remoteTable = Main.__remoteTable initRemoteTable

main :: IO ()
main = do
  Right transport <- createTransport “127.0.0.1” “10001” defaultTCPParameters
  node <- newLocalNode transport remoteTable
  runProcess node $ do
    snid <- getSelfNode
    (sp, rp) <- newChan
    spawn snid ($(mkClosure ‘sender) sp)
```

PROCESS LAYER

- 创建远程进程
  - Polymorphic?
    - Data.Rank1Dynamic (toDynamic)
    - Data.Rank1Typeable (Typeable, ANY, ANY1, ANY2, ANY3, ANY4)

```
rtable :: RemoteTable
rtable =
   registerStatic “$decode"
       (toDynamic (decode :: ByteString -> ANY))  -- Really?
   $ initRemoteTable
```

PROCESS LAYER

```
data SerializableDict a where
  SerializableDict :: Serializable a => SerializableDict a
  deriving (Typeable)
```

```
decodeDict :: SerializableDict a -> ByteString -> a
decodeDict SerializableDict = decode

rtable :: RemoteTable
rtable =
   registerStatic “$decodeDict"
       (toDynamic (decodeDict :: SerializableDict ANY -> ByteString -> ANY))
   $ initRemoteTable

staticDecode :: Typeable a => Static (SerializableDict a) -> Static (ByteString -> a)
staticDecode dict = decodeDictStatic `staticApply` dict
  where
    decodeDictStatic :: Typeable a => Static (SerializableDict a -> ByteString -> a)
    decodeDictStatic = staticLabel “$decodeDict”
```

PROCESS LAYER

```
sdictT :: SerializableDict T
sdictT = SerializableDict
$(mkStatic 'sdictT) :: Static (SerializableDict T)
```

```
remotable [‘f] -- f :: T1 -> T2
$(functionSDict 'f) :: Static (SerializableDict T1)
-- if f :: T1 -> Process T2
$(functionTDict 'f) :: Static (SerializableDict T2)
```

PROCESS LAYER

- See also…
  - Control.Distributed.Process.Closure (distributed-process)

```
type CP a b = Closure (a -> Process b)

idCP :: Typeable a => CP a a
returnCP :: Serializable a => Static (SerializableDict a) -> a -> Closure (Process a)
bindCP :: (Typeable a, Typeable b) => Closure (Process a) -> CP a b -> Closure (Process b)
seqCP :: (Typeable a, Typeable b) => Closure (Process a) -> Closure (Process b) -> Closure (Process b)

cpLink :: ProcessId -> Closure (Process ())
cpSend :: Typeable a => Static (SerializableDict a) -> ProcessId -> CP a ()
cpExpect :: Typeable a => Static (SerializableDict a) -> Closure (Process a)
```

PROCESS LAYER

- 容错
  - Link
    - 单向：(A) Process -> (B) Process / Node / Channel
    - 当 B 正常/异常结束、或失去联系时，A 会产生一个异步异常（ProcessLinkException），导致 A 也被终止。
    - ProcessLinkException 没有被 Cloud Haskell 导出。

```haskell
data ProcessLinkException = ProcessLinkException ProcessId DiedReason
link :: ProcessId -> Process ()
unlink :: ProcessId -> Process ()

-- located in distributed-process-extras, implemented by moniter
linkOnFailure :: ProcessId -> Process ()
```

PROCESS LAYER

- 容错
  - Monitor
    - 单向
    - 当被监控的 Process/Node/Port 结束时，进行监控的进程将会收到类型为 Process-/Node-/PortMonitorNotification 的消息。
    - 每次对 monitor 的调用产生新的 MonitorRef，需要分别 unmonitor。

```
data ProcessMonitorNotification =
    ProcessMonitorNotification MonitorRef ProcessId DiedReason
monitor :: ProcessId -> Process MonitorRef
unmonitor :: MonitorRef -> ProcessId
withMonitor :: ProcessId -> Process a -> Process a
```

PROCESS LAYER

```haskell
linkOnFailure :: ProcessId -> Process ()
linkOnFailure them = do
  us <- getSelfPid
  tid <- liftIO $ myThreadId
  void $ spawnLocal $ do
    callerRef <- P.monitor us
    calleeRef <- P.monitor them
    reason <- receiveWait [
             matchIf (\(ProcessMonitorNotification mRef _ _) ->
                       mRef == callerRef) -- nothing left to do
                     (\_ -> return DiedNormal)
           , matchIf (\(ProcessMonitorNotification mRef' _ _) ->
                       mRef' == calleeRef)
                     (\(ProcessMonitorNotification _ _ r') -> return r')
         ]
    case reason of
      DiedNormal -> return ()
      _ -> liftIO $ throwTo tid (ProcessLinkException us reason)
```

PROCESS LAYER

```
spawnLink :: NodeId -> Closure (Process ()) -> Process ProcessId
spawnMonitor :: NodeId -> Closure (Process ()) -> Process (ProcessId, MonitorRef)

spawnSupervised :: NodeId -> Closure (Process ()) -> Process (ProcessId, MonitorRef)
```

PROCESS LAYER

- See also…
  - Control.Distributed.Process.Supervisor (distributed-process-supervisor)
    - Supervision tree: Hierarchical process structure.
    - Restart Strategies…

What’s the behavior of …

```
link pid                                <- Async!
send pid “Hello world”                  <- Async!
unlink pid                              <- Async!
```

```
link pid
send pid “Hello world”
reply <- expect
unlink pid
```

PROCESS LAYER

- Process Layer 的特点：底层
  - 类型安全保障弱。
  - 手动错误侦测与恢复。
  - 对逻辑的表达能力弱。

TASKレヤの消失

- 在很久很久以前……（Cloud Haskell 只有一个叫 remote 的库的时候）
  - Promise/future (inspired by Skywriting/CIEL)
    - 代表一个完成或没有完成的计算结果，Serializable。
    - 对值的提取操作会让该 Promise 在得到结果前堵塞。
    - 计算在 TaskM 中完成。
  - TaskM：不能执行任何 IO 操作的 Monad
    - 最大限度减少错误的可能。
    - 一个出错终止的 TaskM 可以很容易地自动重启。

```
newPromise :: Serializable a => Closure (TaskM a) -> TaskM (Promise a)
readPromise :: Serializable a => Promise a -> TaskM a
runTask :: Serializable => TaskM a -> Process a
```

TASKレヤの消失

```
avg :: [Integer] -> TaskM Integer
avg xs = return $ sum xs `div` fromIntegral (length xs)

diff :: Promise Integer -> Promise Integer -> TaskM Integer
diff pa pb = do
  a <- readPromise pa
  b <- readPromise pb
  return $ (a + b) / 2

$(remotable [‘avg, ‘diff])

process :: Process ()
process = do
  res <- runTask $ do
    p1 <- newPromise ($(mkClosure ‘avg) [0..50])
    p2 <- newPromise ($(mkClosure ‘avg) [50..100])
    p3 <- newPromise ($(mkClosure ‘diff) p1 p2)
    readPromise p3
  say $ “Result: ” ++ show res
```

TASKレヤの消失

- 消失的原因
  - 不可扩展：Master-slave
  - 不够健壮：Master 节点一旦崩溃整个集群必须重启。
  - 节点分配的算法实用性低：round-robin
    - 需要硬件资源监控和负载均衡的技术。

Maybe it will come back…

TASKレヤの消失

- Control.Distributed.Process.Async!
  - 分布式版本的 Control.Concurrent.Async。
  - 同时支持堵塞等待与非堵塞式查询。
  - 不限制 IO 的使用。

```
async :: Serializable a => AsyncTask a -> Process (Async a)
asyncLinked :: Serializable a => AsyncTask a -> Process (Async a)

wait :: Async a -> Process (AsyncResult a)
poll :: Serializable a => Async a -> Process (AsyncResult a)
```

TASKレヤの消失

```
data AsyncTask a =
    AsyncTask {
        asyncTask :: Process a
      }
  | AsyncRemoteTask {
        asyncTaskDict :: Static (SerializableDict a)
      , asyncTaskNode :: NodeId
      , asyncTaskProc :: Closure (Process a)
      }

task :: Process a -> AsyncTask a
remoteTask :: Static (SerializableDict a) -> NodeId -> Closure (Process a) -> AsyncTask a
```

MANAGED PROCESS

- Client/Server 模式 (distributed-process-client-server)
- 自动消息解码、分发以及错误处理
  - Mailbox -> 用户定义的 handlers（根据消息的类型以及用户提供的 predicates）
- 两种可选的通讯方法
  - cast：客户端异步发送消息，服务端不回复。
  - call：远程过程调用（Remote Procedure Call），客户端等待服务端传回结果。

```
cast :: (Addressable a, Serializable m) => a -> m -> Process ()
call :: (Addressable s, Serializable a, Serializable b) => s -> a -> Process b
```

MANAGED PROCESS

```haskell
import Data.Time.LocalTime

data TimeType = TtUTC | TtLocal
  deriving (Eq, Typeable, Generic, Binary)
data GetTime = GetTime TimeType
  deriving (Typeable, Generic, Binary)

getTime :: (Addressable a) => a -> TimeType -> Process String
getTime a tt = call pid $ GetTime tt

timeServer :: Process ProcessId
timeServer =
  let server = statelessProcess {
    apiHandlers =
        [ handleCallIf_ (input $ \(GetTime tt) -> tt == TtUTC)
                        (\_ -> fmap show (liftIO getZonedTime))
        , handleCallIf_ (input $ \(GetTime tt) -> tt == TtLocal)
                        (\_ -> fmap (show . zonedTimeToLocalTime) $ liftIO getZonedTime)]
    , unhandledMessagePolicy = Drop }
  in spawnLocal serve () (statelessInit Infinity) server
```

参考

- Functional programming for the data centre, June 2011
    - Jeffrey Epstein
- Towards Haskell in the Cloud, September 2011
    - Jeff Epstein
    - Andrew P. Black
    - Simon Peyton-Jones
- Cloud Haskel Documentation, Tutorials
    - http://haskell-distributed.github.io

Thanks!