

# SCALABLE AND RELIABLE VIDEO TRANSCODING IN HASKELL

---

Alfredo Di Napoli

Haskell Exchange 2015

Full story at: <http://goo.gl/qkKwKm>



```
launchMissile :: IO ()
```




**iris**  
Change. Doing it right.





*discover. develop. share.*

- ~ Present in over 1800 schools Worldwide (mostly UK, Europe, US & Australia)
- ~ Used by over 32000 teachers

# IRIS CONNECT (CONTD.)

 Home Reflections Forms Groups


Search  

New Reflection +

My Reflections >



Shared with me >


Invitations >



**Mandy's Lesson Starters**  
Demo User 02/15/15 11:14



Review


Details  



**History Lesson**  
Demo User 03/15/15 12:27



Review


Details  



**Dual View**  
Demo User 01/20/15 12:34



Review


Details  



**Cramlington Learning Village**  
Demo User 09/22/14 12:25



Review


Details  



**Maths Lesson**  
Demo User 09/09/14 14:23



Review


Details  



**History Lesson Cartoon**  
Demo User 06/20/15 12:34



Review


Details  



**Questioning Best Practice**  
Demo User 03/21/15 12:25



Review

Details  



**LiveView Sample - Math Lesson**  
Demo User 04/10/15 12:43

Review

Details  

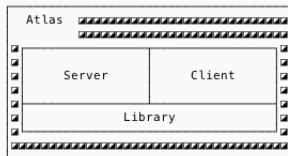
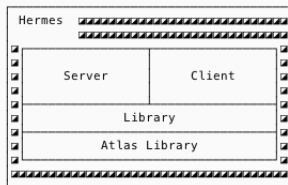
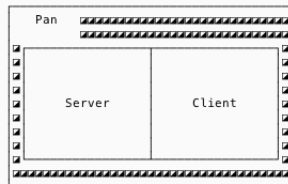
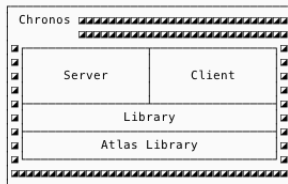
LiveView



Discovery Kit



# IRIS' GREEK ZOO



## October, 2013

```
hermes [74aeab2] ⚡ cloc src scripts
  3 text files.
  3 unique files.
  0 files ignored.

http://cloc.sourceforge.net v 1.60 T=0.24 s (12.3 files/s, 762.9 lines/s)
-----
Language          files          blank          comment          code
-----
Haskell              1              22              7              80
Bourne Shell         2              25              2              50
-----
SUM:                  3              47              9             130
-----
```

## October, 2015

```
hermes [master] cloc main server src test
 105 text files.
 105 unique files.
  0 files ignored.

http://cloc.sourceforge.net v 1.60 T=0.66 s (159.6 files/s, 26288.3 lines/s)
-----
Language          files          blank          comment          code
-----
Haskell           105           2332           2252           12712
-----
SUM:              105           2332           2252           12712
-----
```

Upon taking the lead on Hermes, I was asked for a couple of requirements to be fulfilled, the most important one being that the system needed to be deployed in a cluster, capable of scaling according to demand.

More specifically, we wanted a system with these desirable properties:

- ~ Scalable
- ~ Fault tolerant
- ~ Distributed



- ~ It's easy to see that what we want is a **cluster**, capable of scaling on demand
- ~ We need to transcode videos, which is a very stateful operation
- ~ A cluster typically implies machines talking to each other, which is also very stateful
- ~ **As good Haskell programmers**, we want to have components in our system to be **as stateless as possible**, and potentially treat videos as **persistent data structures**!

*A shared nothing architecture (SN) is a distributed computing architecture in which each node is independent and self-sufficient, and there is no single point of contention across the system.*

*"All problems in computer science can be solved by another level of indirection."* - Butler Lampson

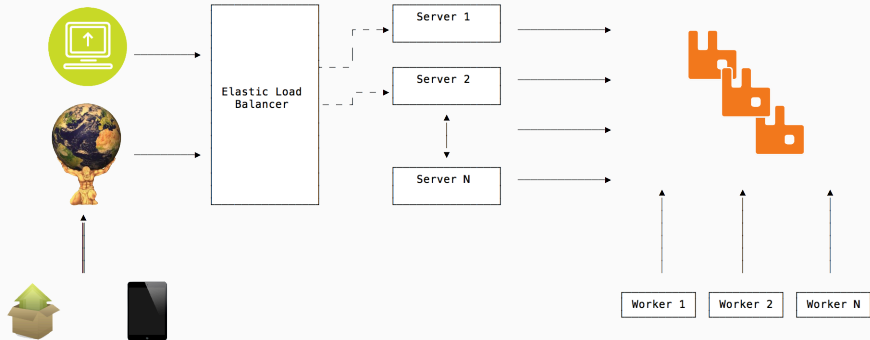
1. RabbitMQ was just the right tool for the job at hand:
  - ~ Easy to setup
  - ~ Can be configured to operate in a federation of nodes
  - ~ Extremely reliable
  - ~ Good Haskell bindings for it (*AMQP*)
2. A question genuinely arise: it seems extremely costly to shuffle video as binary blobs over the queues. Can we avoid that?

# ABSTRACTION IS THE (MEDIA) KEY

```
root__m-stg-main-2014_10_29_13_27_26-videos-1-2333-vid-smc-oxz8dmdillx7fong
  ^      ^      ^      ^      ^      ^      ^      ^      ^      ^
  |      |      |      |      |      |      |      |      |      |
comment  |      |      |      |      |      |      |      |      |
host    ----+      |      |      |      |      |      |      |
database --- +      |      |      |      |      |      |      |
dataset version -----+      |      |      |      |      |      |
resource (video or image) -----+      |      |      |      |      |
user ID  -----+      |      |      |      |      |      |      |
video ID -----+      |      |      |      |      |      |      |
channel type -----+      |      |      |      |      |      |      |
video products -----+      |      |      |      |      |      |      |
MAC (avoids submission of bogus keys) -----+      |      |      |      |      |      |
```

To be fair, the media key abstraction was already present in Atlas when I choose RabbitMQ, but it was the perfect fit for it!

# THE ARCHITECTURE



## WHAT ABOUT DATA STORAGE?

Fine, but RabbitMQ doesn't give you data storage...

1. We use AWS' S3 for our storing needs

~ A media key **uniquely identifies** an S3 location (it's like an **IP address for videos!**)

2. Upon upload the original file from the user is synced over S3 and we call this generation-0 file the **master file**
3. Such master file is **immutable**, and each product we transcode generates a brand new binary on S3

**We are treating videos as immutable data structures!**

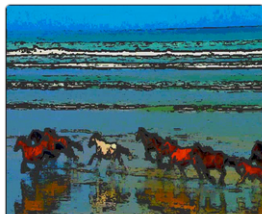
# THE IMPORTANCE OF BEING IMMUTABLE

Easy concurrency and parallelism!

`root__m-inplc-main-2015_09_07_12_31_16-videos-1-4-vid-s-4s7byyc9pkbkfv1v`



`root__m-inplc-main-2015_09_07_12_31_16-videos-1-4-vid-c-4mpvvngvajarzx0n`





## WHAT ABOUT SCALABILITY?

Fine, but RabbitMQ doesn't give you scalability...

1. We stood once again on the shoulder of giants - namely AWS' Auto Scaling Groups
2. Our very first naive scaling algorithm used AWS' builtin alarms and looked like:
  - ~ Scaling up: Based on CPU% over time
  - ~ Scaling down: Based on CPU% over time

It kept us going for a while...

1. Scaling up was too conservative and slow
  - ~ It could take up to 15 mins to spawn a new worker
2. Scaling down suffered similar problems
3. The result was unoptimal for customers (due to the slow turnaround time) and unoptimal for us (due to the additional costs incurring from poor scaling down)

- Scaling up is easy: all we care about is the total number of jobs in RabbitMQ's **Ready** state;

~ In RMQ jargon those are jobs waiting for a “transcoding slot”

Ready	Unacked	Total
0	0	0

- We periodically monitor those figures and kick off a “ScaleUP” action whenever `ready_jobs >= 0`

~ AWS's ASG allows us to put an upper bound to the total number of spawnable machines, to keep costs at bay.

1. More interesting is the scaling down. Ideally we want to kill a worker if the following is true:
  - ~ That worker didn't receive any new jobs within a `starvation_time` period (say, 5 minutes)
2. At the same time, we would like to optimise the time it stays around
3. Last but not least, it needs to commit suicide alone (I know it sounds sad..), taking a local decision, as it doesn't know its peers (SN architecture - remember?)

## SCALING DOWN - THE REAPER

```
type HeartBeat = TQueue ()
type Toggle = TVar ()
type PoisonFlask = TVar ()

data Reaper = Reaper {
  _rr_timeout :: Maybe Int
  -- ^ The timeout to use for this `Reaper`. Setting this to
  -- Nothing means no timeout at all. This is useful for those transcoders
  -- not associated with jobs (e.g. dual-view, discovery-kit, notifications,..)
  , _rr_reapCondition :: IO Bool
  -- ^ If True, will trigger the reaping. If False, the Reaper will be
  -- permissive.
  , _rr_reapAction :: IO ()
  , _rr_heartbeat :: HeartBeat
  , _rr_toggle :: Toggle
  -- ^ When filled, inform the listeners they can carry on with their
  -- activities (i.e. fetch another job from the queue)
  , _rr_poisonFlask :: PoisonFlask
}
```

## SCALING DOWN - REAPER TIMEOUT

```
data ReaperPoisoned = ReaperPoisoned ()
type ReaperResponse = Either ReaperPoisoned (Maybe ())

-- | peek the next value from a TQueue or timeout
peekTQueueTimeout :: Maybe Int
                  -> HeartBeat
                  -> PoisonFlask
                  -> IO ReaperResponse
peekTQueueTimeout Nothing heartbeat fsk =
    atomically $ Right . Just <$> peekTQueue heartbeat <|>
        Left . ReaperPoisoned <$> takeTMVar fsk
peekTQueueTimeout (Just timeoutAfter) heartbeat fsk = do
    delay <- registerDelay timeoutAfter
    atomically $ (Right . Just <$> peekTQueue heartbeat) <|>
        (pure (Right Nothing) <*> untilTimeout delay) <|>
        (Left . ReaperPoisoned <$> takeTMVar fsk)
```

## SCALING DOWN, STM TO THE RESCUE

```
reap :: Reaper -> IO ReaperResponse
reap (Reaper t cond _ hb tgl pp) = do
  r <- peekTBQueueTimeout t hb pp
  case r of
    Left _ -> return r -- If we have been poisoned, honour the poisoning and die.
    v@(Right Nothing) -> do
      condT <- cond
      -- If the reaping condition is True, we need to die.
      -- If not, we simulate a state toggle to induce listeners to unlock
      -- and wait for the next batch of events.
      if condT then return v else toggle tgl >> return (Right $ Just ())
    Right s -> return . Right $ s
```

```
instance Alternative STM where
  empty = retry
  (<|>) = orElse

-- orElse: Compose two alternative STM actions. If the first action completes
-- without retrying then it forms the result of the orElse.
-- Otherwise, if the first action retries, then the second action is tried
-- in its place. If both actions retry then the orElse as a whole retries.
```

## SCALING DOWN, A TYPICAL TRANSCODER

```
newtype Transcoder a = Transcoder { transcode :: StateT TranscoderState IO a }  
    deriving (MonadState TranscoderState, Monad, Functor, Applicative, MonadIO)
```

```
transcoder :: Transcoder ()  
transcoder = do  
    ctx@TranscoderCtx{..} <- getContext  
    newTranscoder $ \_ hb tgl -> NewRabbitConsumer <$> do  
        consumeTranscodingJobsIO _tr_channelConfig $ \(msg, env) -> void $ forkIO $  
            withIncomingPacket msg $  
                \job@(PendingJob key HermesOptions{..} _ (RetryWindow _ _)) -> do  
                    sendHeartBeat hb -- puts a 'token' into the heartbeat queue  
                    -- Do here transcoding stuff...  
                    signalDone hb -- reads from the heartbeat queue  
                    toggle tgl      -- puts a token (unit) inside the toggle
```

```
newTranscoder :: (PoisonFlask -> HeartBeat -> Toggle -> IO NonBlockingAction)  
               -> Transcoder ()
```

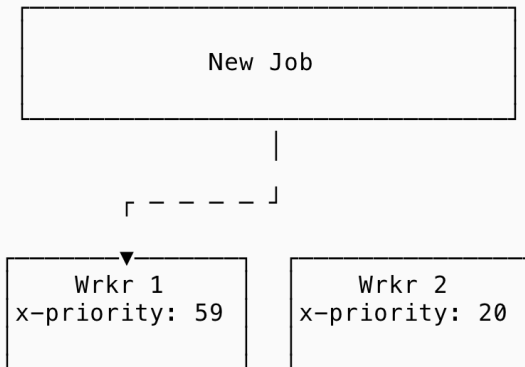
The purpose of that `NonBlockingAction` type will be clear soon.



- When we deployed the code, we didn't find a sensible difference in costs...
- ... reason being Amazon doesn't bill you for fractional hours
  - ~ Even if you spawn a machine 5 mins, you are billed the full hour!
- We needed workers to stay around as much as possible, maximising their billing hour, without crossing the next-hour mark, if possible!

*"Consumer priorities allow you to ensure that high priority consumers receive messages while they are active, with messages only going to lower priority consumers when the high priority consumers block."*

## RABBITMQ CONSUMER PRIORITIES (CNTD.)



- Each worker can transcode 1 job at time, before “blocking”
- The priority is set to be  $60 - (\text{uptime} \% 60)$
- A newly spawned machine gets max priority
- A machine close enough to the next billing hours (e.g. priority  $\leq 10$ ) **if starving**, gets evicted!

- A consumer priority cannot be updated dynamically
- The easiest way we found was - after a successful heartbeat - to simply cancel the old consumer and create a new one, with updated priority

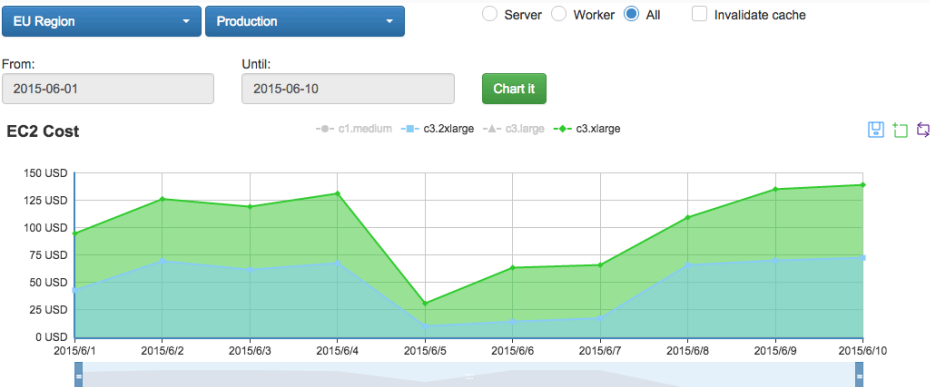
```
data NonBlockingAction = VoidAction ()  
                        | NewRabbitConsumer ConsumerTag
```

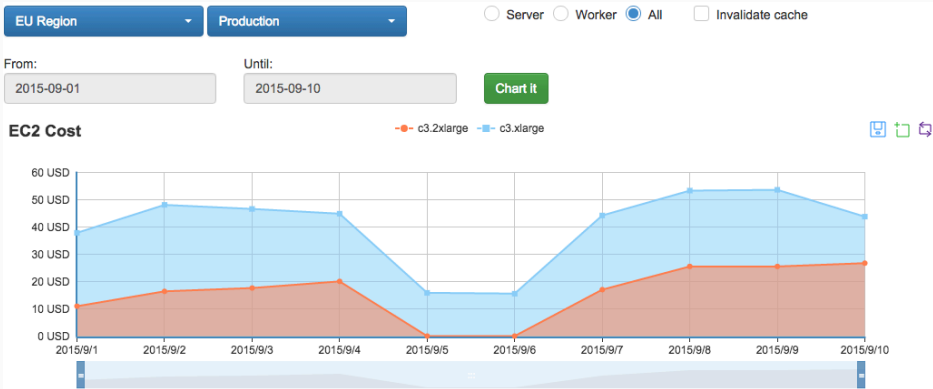
- We used the `ConsumerTag` returned by `NewRabbitConsumer` to cancel the old one, compute the updated priority and start a new transcoder

## PUTTING EVERYTHING TOGETHER

```
newTranscoderState :: HeartBeat
                    -> TranscoderType
                    -> TranscoderCtx
                    -> IO TranscoderState
newTranscoderState hb ttype tctx = do
  let config = tctx ^. tr_config
  pp <- newPoisonFlask
  rpr <- case ttype of
    JobTranscoder (Production _) ->
      newReaper pp hb (Just twoMinutes) closeToNextBillingHour (reapFromAWS config)
    JobTranscoder Devel ->
      newReaper pp hb (Just oneMinute) (return True) reapLocally
    JobTranscoder _ ->
      newReaper pp hb Nothing (return True) (return ())
    AuxiliaryTranscoder ->
      newReaper pp hb Nothing (return True) (return ())
  return TranscoderState {
    _ts_transitions = singleton 50 WaitingForJob
    , _ts_poisonFlask = pp
    , _ts_reaper = rpr
    , _ts_ctx = tctx
  }
```

# THE NEW SCALING ALGORITHM: RESULTS





**We shaved almost 50% of the costs!**

# THE ELEPHANT IN THE ROOM

**What's the elephant in the room?**





**Why not use Cloud Haskell?**

1. CH encourages Erlang-style (i.e. actor based) communication, so nodes should know each other

We do not want that!

2. Peer discovery would have been tricky in a dynamic environment where new machines born and die frequently
3. It wasn't mature enough in 2013, if not for a handful of companies using it

Thank you!

Questions?

### **My road to Haskell**

<http://www.alfredodinapoli.com/posts/2014-04-27-my-road-to-haskell.html>