Haskell im harten Alltag

Stefan Wehr (wehr@cp-med.com)
Twitter: @skogsbaer
factis research GmbH

20. Juni 2014

Wer bin ich? Was machen wir?

- Haskell Benutzer seit 2003
- Zunächst vor allem im akademischen Bereich
- Seit 2010: Anwendung von Haskell in der Industrie
- *factis research GmbH*, Freiburg im Breisgau
  - Softwareprodukte für den Medizin- und Pflegebereich
  - Serverseitige Software fast ausschließlich in Haskell geschrieben
  - Große Erfahrung mit komplexen mobilen Anwendungen

Warum funktional? Warum Haskell?

- Kontrollierter Umgang mit Seiteneffekten
- Hohe Modularität, klar abgegrenzte Komponenten
- Gute Testbarkeit
- Kurzer, prägnanter und lesbarer Code
- Einfaches Abstrahieren
- Hoher Wiederverwendungsgrad
- Ausdrucksstarkes Typsystem: Wenn das Programm kompiliert funktioniert es auch!
- "World's finest imperative programming language"
- Reichhaltige Palette an Ansätzen zur parallelen und nebenläufigen Programmierung
- "Schlaue Leute"

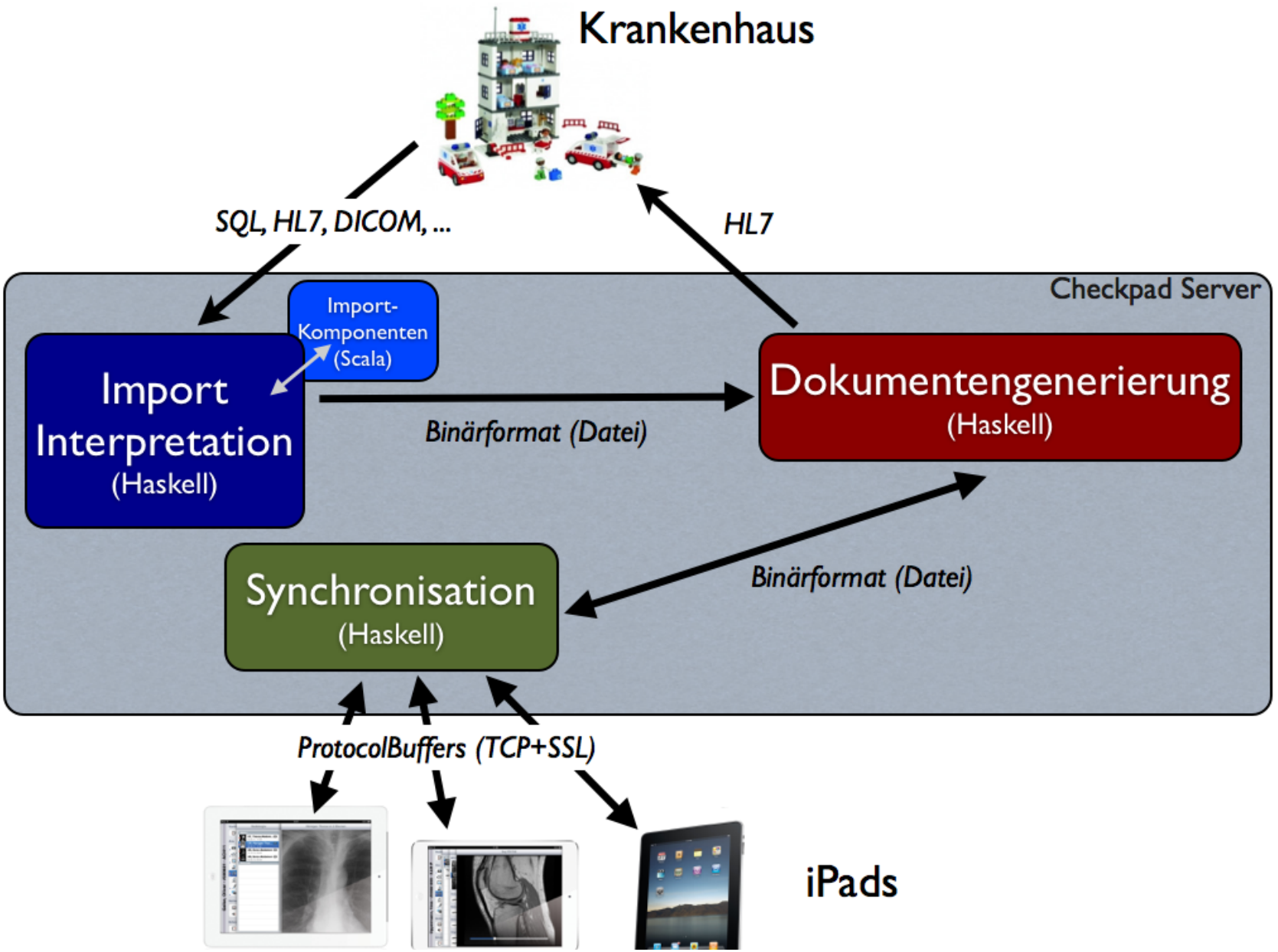
Was erwartet euch heute?

- Einblick in ein (größtenteils) in Haskell geschriebenes, kommerzielles Softwareprodukt
- Erfahrung von mehr als vier Jahren kommerzieller Softwareentwicklung mit Haskell
- Konkretes Beispiel: Messaging System
  - Serialisierung und Persistenz
  - Netzwerkprogrammierung
  - Nebenläufiges Programmieren
  - Testen
  - Logging
  - Buildsystem

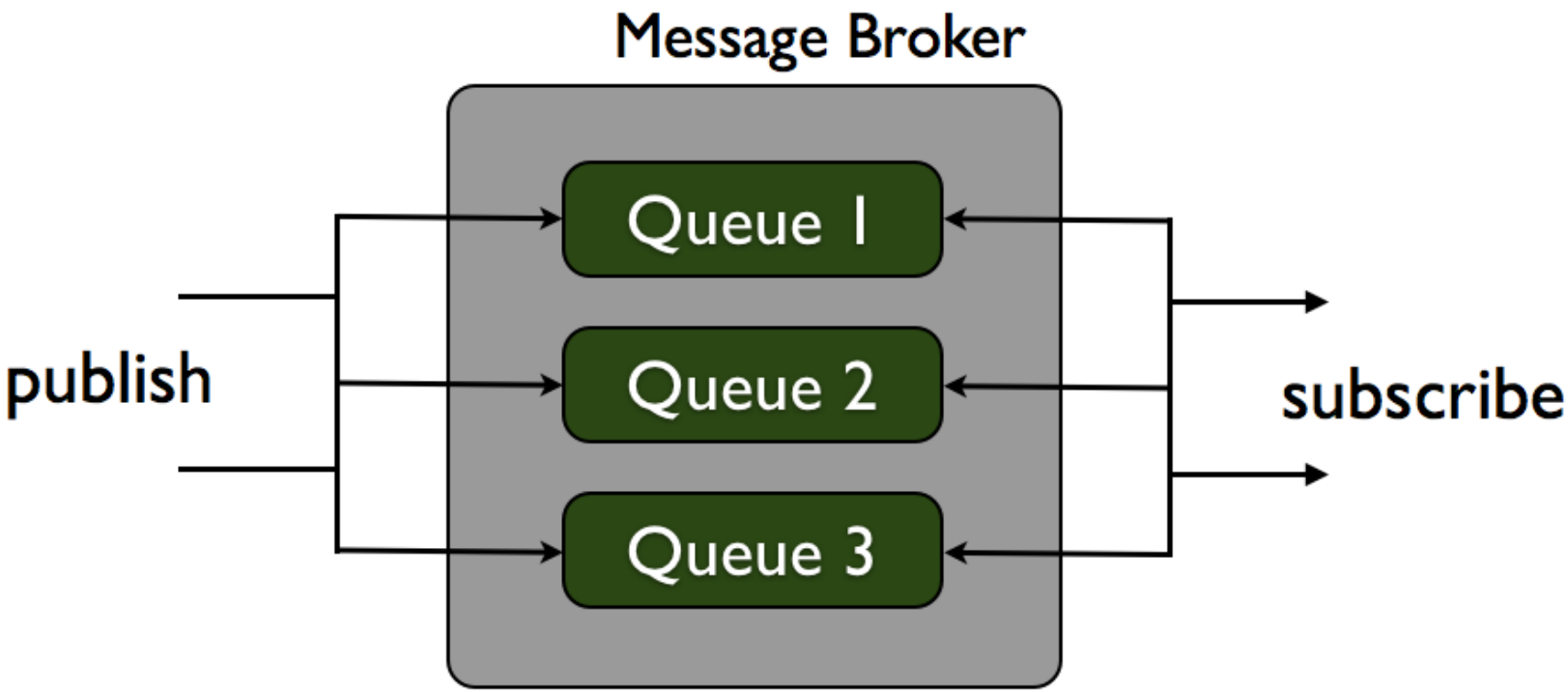
Kommerzielle Softwareentwicklung mit Haskell

- *Checkpad MED*: elektronische Patientenakte auf dem iPad
  - Bringt alle Patientendaten zusammen
  - Unterstützt Krankenhausärzte bei Arbeitsabläufen
  - Unabhängig vom KIS (Krankenhausinformationssystem)
  - Demo
- Entwicklung seit 2010
- Heute: mehrere zahlende Kunden, Pilotbetrieb in mehreren großen Krankenhäusern

Architektur von Checkpad



Messaging System



- Quellcode: http://www.factisresearch.com/hal9/

Zentraler Datentyp: MessageBroker

```
data MessageBroker q
    = MessageBroker
      { mb_queues :: !(HashMap QueueName q)
      , mb_subscribeToQueue ::
          q -> Subscriber -> STM SubscriberId
      , mb_unsubscribeFromQueue ::
          q -> SubscriberId -> STM ()
      , mb_publishMessage ::
          q -> Message -> IO ()
      }
```

- Demo
- Codebeispiel, siehe `server/src/lib/Mgw/MessageQueue/Types.hs`

Nebenläufigkeit mit STM

- Beispiel: Überweisung von einem Bankkonto
  - `transfer acc1 acc2 amount`: überweise Betrag `amount` von Konto `acc1` auf `acc2`
  - `transfer` soll thread-safe sein
  - Im Beispiel: Konten sind nicht persistenz
- Probleme mit Threads
  - Deadlocks, Race conditions
  - Unmodular
- Idee von STM:
  - Markiere Codeblöcke als "atomar"
  - Atomare Blöcke werden entweder ganz oder gar nicht ausgeführt (wie Datenbanktransaktionen)
- Vorteile von Haskell:
  - Immutability
  - Lazy Auswertung

Kontoüberweisungen mit STM

```
transfer :: Account -> Account -> Int -> IO ()
transfer acc1 acc2 amount =
    atomically (do deposit acc2 amount
                   withdraw acc1 amount)
```

- `atomically :: STM a -> IO a`
  - Führt die übergebene Aktion atomar aus
  - Auszuführende Aktion wird auch *Transaktion* genannt
  - `STM a`: Typ einer Transaktion mit Ergebnis vom Typ `a`
  - `STM` ist eine Monade

Transaktionsvariablen

- STM-Aktionen kommunizieren über *Transaktionsvariablen*
  - `TVar a`: Transaktionsvariablen die Wert vom Typ `a` enthält
- Lesen: `readTVar :: TVar a -> STM a`
- Schreiben: `writeTVar :: TVar a -> a -> STM ()`
- Außerhalb von `atomically` können Transaktionsvariablen weder gelesen noch geschrieben werden.

```
type Account = TVar Int

deposit :: Account -> Int -> STM ()
deposit acc amount =
    do bal <- readTVar acc
       writeTVar acc (bal + amount)

withdraw :: Account -> Int -> STM ()
withdraw acc amount = deposit acc (- amount)
```

Blockieren mit STM

- Warten auf eine bestimmte Bedingung ist essential für nebenläufige Programme
- Beispiel: `limitedWithdraw` soll blockieren falls nicht genug Geld zum Abheben da ist

```
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount =
    do bal <- readTVar acc
       if amount > 0 && amount > bal
       then retry
       else writeTVar acc (bal - amount)
```

- `retry :: STM a`
  - Bricht die aktuelle Transaktion ab
  - Versucht zu einem späteren Zeitpunkt die Transaktion nochmals auszuführen

Kombinieren von Transaktionen

- Beispiel: Abheben mittels `limitedWithdraw` von mehreren Konto solange bis ein Konto genug Geld hat

```
limitedWithdrawMany :: [Account] -> Int -> STM ()
limitedWithdrawMany [] _ = retry
limitedWithdrawMany (acc:rest) amount =
    limitedWithdraw acc amount `orElse`
    limitedWithdrawMany rest amount
```

- `orElse :: STM a -> STM a -> STM a`
  - `orElse t1 t2` führt zuerst `t1` aus
  - Falls `t1` erfolgreich, so auch `orElse t1 t2`
  - Falls `t1` abbricht (durch Aufruf von `retry`) wird `t2` ausgeführt
  - Falls `t2` abbricht bricht auch die gesamte Transaktion ab, d.h. sie wird zu einem späteren Zeitpunkt nochmals ausgeführt

Zusammenfassung der STM API

```
atomically :: STM a -> IO a

retry  :: STM a
orElse :: STM a -> STM a -> STM a

newTVar   :: a -> STM (TVar a)
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

Ergänzungen

- Vermeide Thunks in `TVars` oder force die Thunks nach der Transaktion
  - Sonst gibt's Memory-Leaks.
- Benutze `runTx` statt `atomically`
  - Warnt dich, wenn Transaktionen häufig fehlschlagen
  - Auf hackage: `stm-stats` (dort als `trackSTM`)

Ein lokaler MessageBroker

```
createLocalBroker :: LogId
                  -> Maybe FilePath
                  -> [(QueueName, QueueOpts)]
                  -> IO (MessageBroker LocalQueue)

data QueuePersistence
    = PersistentQueue | TransientQueue
      deriving (Eq, Show)

data QueueOpts
  = QueueOpts { qo_persistence :: QueuePersistence }
    deriving (Eq, Show)
```

- Code:
  `server/src/lib/Mgw/MessageQueue/LocalBroker.hs`

Testen mit HTF

- Paket `HTF` auf Hackage
- Automatisches Aufsammeln von Unit-Tests und QuickCheck-Properties
- Dateiename und Zeilennummer in Fehlermeldungen
- Diff beim Fehlschlagen von Gleichheits-Assertions
- Möglichkeit zum Replay von fehlgeschlagenen QuickCheck-Properties
- Paralles Ausführen von Tests
- Maschinenlesbare Ausgabe (auf Wunsch)

Serialisierung mit SafeCopy

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

deriveSafeCopy 1 'base ''MessageId
deriveSafeCopy 1 'base ''Message

safeDecode :: (Monad m, SafeCopy a)
           => BS.ByteString -> m a
safeEncode :: SafeCopy a => a -> BS.ByteString
```

Migration mit SafeCopy

```
data MessageV1 = MessageV1
       { msgV1_id :: !MessageId
       , msgV1_payload :: !BS.ByteString }
data Message = Message
       { msg_id :: !MessageId
       , msg_time :: !(Option ClockTime)
       , msg_payload :: !BS.ByteString }
deriveSafeCopy 1 'base ''MessageId
deriveSafeCopy 1 'base ''MessageV1
deriveSafeCopy 2 'extension ''Message
instance Migrate Message where
    type MigrateFrom Message = MessageV1
    migrate msg =
      Message { msg_id = msgV1_id msg
              , msg_payload = msgV1_payload msg
              , msg_time = None }
```

Lazyness und Speicherlecks

- Unausgewertete Datenstrukturen können zu Speicherlecks führen
- Schwierig zu finden
- Kleines Rätsel: Ist das mit folgendem Code der Fall?
  - Annahme: an `storeMD5` übergebene Map bleibt lange im Speicher

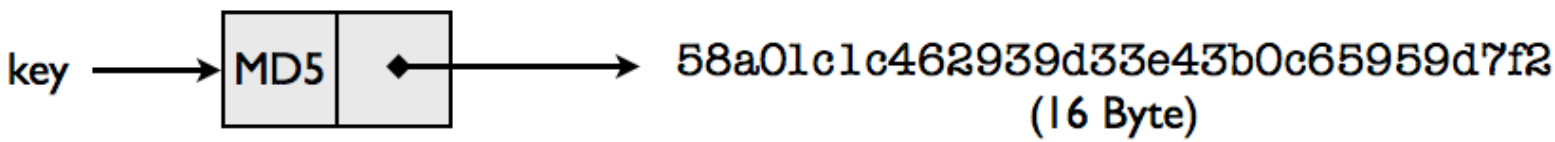
```
import qualified Data.ByteString as BS

md5 :: BS.ByteString -> MD5
md5 bs = MD5 (md5' bs)
    where
      md5' :: BS.ByteString -> BS.ByteString
      md5' = ...
storeMD5 :: Key -> BS.ByteString -> Map.Map Key MD5
         -> Map.Map Key MD5
storeMD5 key bs = Map.insert key (md5 bs)
```

Nein, kein Speicherleck!

```haskell
import qualified Data.Map.Strict as Map

data MD5 = MD5 { unMD5 :: !BS.ByteString }
```

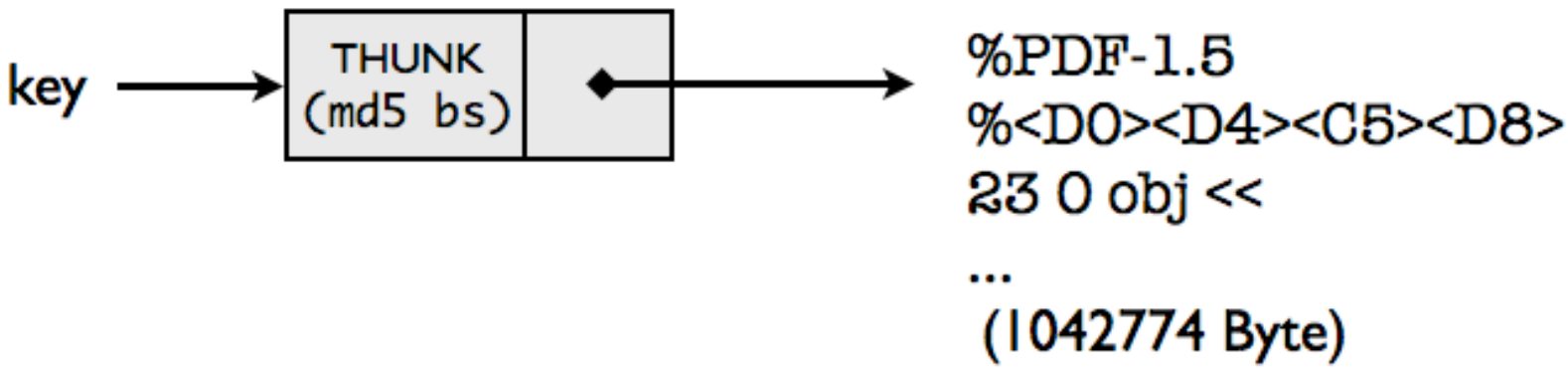



```haskell
storeMD5 :: Key -> BS.ByteString -> Map.Map Key MD5
         -> Map.Map Key MD5
storeMD5 key bs = Map.insert key (md5 bs)
```

Doch, Speicherleck!

```haskell
import qualified Data.Map.Lazy as Map

data MD5 = MD5 { unMD5 :: !BS.ByteString }
```

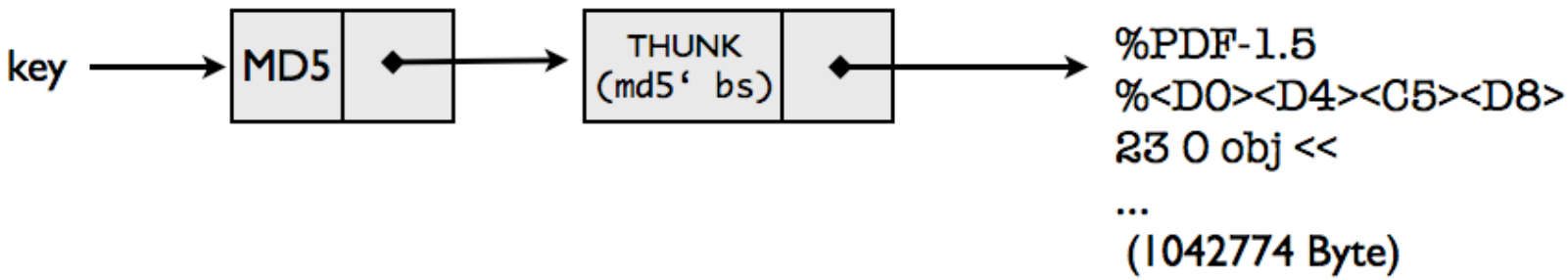



```haskell
storeMD5 :: Key -> BS.ByteString -> Map.Map Key MD5
         -> Map.Map Key MD5
storeMD5 key bs = Map.insert key (md5 bs)
```

Doch, Speicherleck!

```haskell
import qualified Data.Map.Strict as Map

data MD5 = MD5 { unMD5 :: BS.ByteString }
```

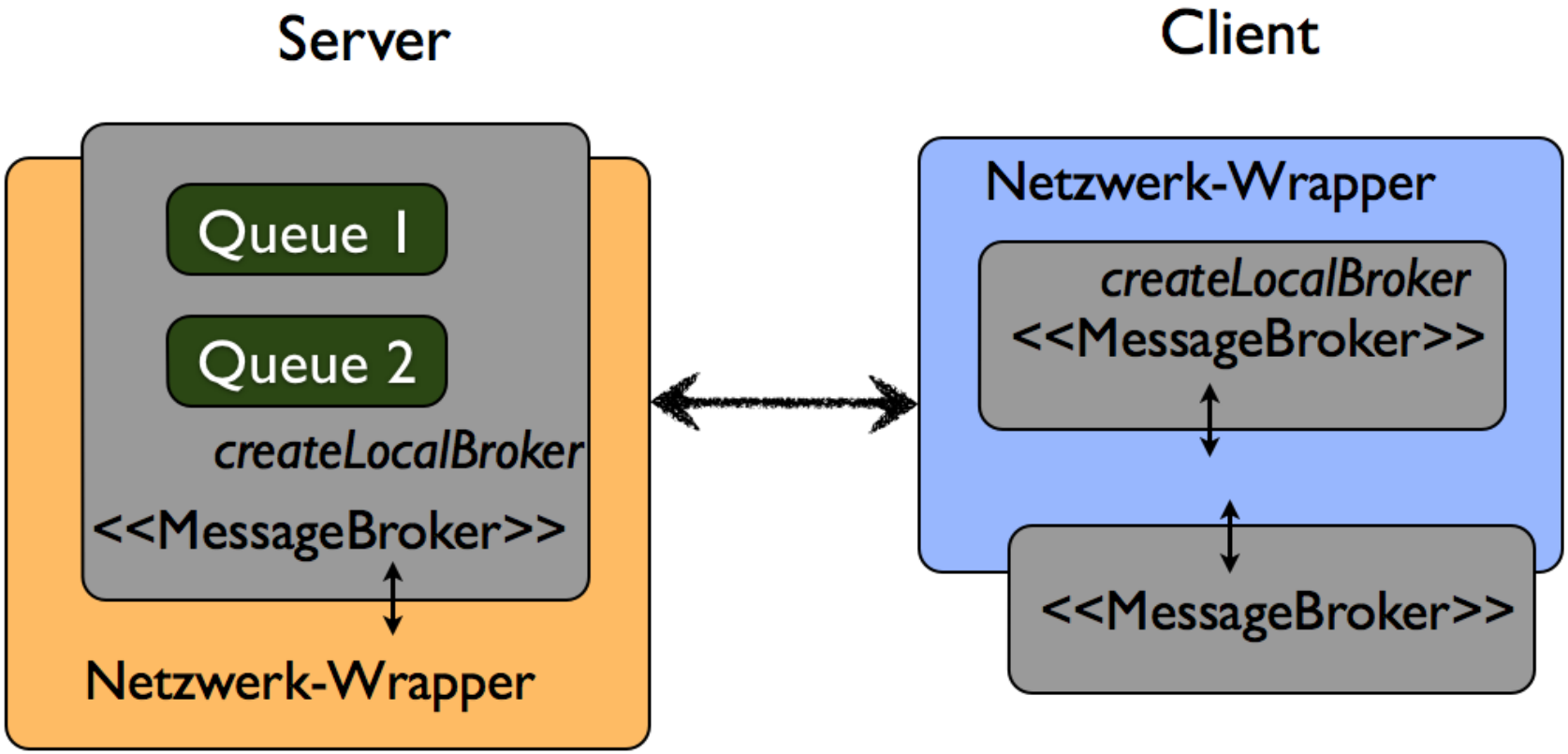



```haskell
storeMD5 :: Key -> BS.ByteString -> Map.Map Key MD5
         -> Map.Map Key MD5
storeMD5 key bs = Map.insert key (md5 bs)
```

Unsere Konvention

- Nur strikte Datenstrukturen dürfen längere Zeit im Speicher bleiben
- Erfordert Disziplin

Server für MessageBroker



Nachrichten zwischen Client und Server

```
data ServerMessage
    = ServerQueues !(V.Vector QueueName)
    | ServerPublishMessage !QueueName !Message
    deriving (Show, Eq)

data ClientMessage
    = ClientSubscribe !QueueName
    | ClientPublishMessage !QueueName !Message
    deriving (Show, Eq)
```

Netzwerkprogrammierung mit Conduits

- Streaming-API (Analogie: Unix-Pipes)
- Resourcen (wie z.B. File-Deskriptoren) werden zeitnah freigegeben


```
type Source m o = ConduitM () o m ()
type Conduit i m o = ConduitM i o m ()
type Sink i m r = ConduitM i Void m r

type Producer m o = forall i. ConduitM i o m ()
type Consumer i m r = forall o. ConduitM i o m r
```

Einfaches Beispiel mit Conduits

```haskell
import Data.Conduit
import qualified Data.Conduit.List as CL

source :: Source IO Int
source = CL.sourceList [1..4]

sink :: Sink String IO ()
sink = CL.mapM_ putStrLn

conduit :: Conduit Int IO String
conduit = CL.map show

main = do
    source $$ conduit =$ sink
```

Funktion auf Conduits

```
($$) :: Monad m => Source m a -> Sink a m b -> m b
($=) :: Monad m => Source m a -> Conduit a m b
                -> Source m b
(=$) :: Monad m => Conduit a m b -> Sink b m c
                -> Sink a m c
(=$=) :: Monad m => Conduit a m b -> ConduitM b c m r
                 -> ConduitM a c m r
await :: Monad m => Consumer i m (Maybe i)
yield :: Monad m => o -> ConduitM i o m ()
finallyP :: MonadResource m => ConduitM i o m r -> IO ()
                            -> ConduitM i o m r
runResourceT :: MonadBaseControl IO m
             => ResourceT m a -> m a
```

Noch ein Beispiel: Implementierung von CL.mapM

```
module Data.Conduit.List
import Control.Monad.Trans.Class (lift)

mapM :: Monad m => (a -> m b) -> Conduit a m b
mapM f = loop
    where
      loop =
        do mx <- await
           case mx of
             Nothing -> return ()
             Just x ->
                 do x' <- lift (f x)
                    yield x'
                    loop
```

Server für MessageBroker: Abstraktion über die Netzwerkschicht

```
createClientHandler ::
    (MonadResource m, MonadIO m)
    => LogId
    -> MessageBroker q
    -> IO (Source m ServerMessage
          ,Sink ClientMessage m ())
```

- Code:
  `server/src/lib/Mgw/MessageQueue/BrokerServer.hs`

Client für den MessageBroker

```
createBrokerStub ::
    LogId
    -> (TBMChan ServerMessage
        -> TBMChan ClientMessage -> IO ())
    -> IO (MessageBroker LocalQueue)
```

- `TBMChan`: bounded, closable Channels
- Über die Channels kann mit dem Netzwerk kommuniziert werden
- Client kann mit Verbindungsabbrüchen umgehen
- Code: `server/src/lib/Mgw/MessageQueue/BrokerStub.hs`

Datenaustausch mit ProtocolBuffers

Google ProtocolBuffers

- Effizientes Serialisierungsformat
- Sprachneutral
- Ermöglicht Rück- und Vorwärtskompatibilität
- Pakete auf Hackage: `hprotoc`, `protocol-buffers`, `protocol-buffers-descriptor`

Im Code

- Definitionen: `protocols/protos/MessageQueue.proto`
- Konvertierungen: `server/src/lib/Mgw/Message/Protocol.hs`
- generierter Code: `server/build/gen-hs/Com/Factisresearch/Checkpad/Protos`

Build-System mit shake

- Haskell-Bibliothek zum Schreiben eines Build-Systems
  - keine vorgefertigten Build-Regeln
  - Regeln werden als Haskell-Code formuliert
- Dynamische Abhängigkeiten
  - Abhängigkeiten entstehen während ein Build läuft
  - anders als z.B. make
- Wenn dein Projekt mit Cabal funktioniert: bleib' bei Cabal
- Mehr Infos zu shake:
  http://community.haskell.org/~ndm/downloads/paper-shake_before_building-10_sep_2012.pdf

shake in Aktion (1)

- `Action`-Monad: trackt Abhängigkeiten
  - Bsp: `readFile' f` führt eine Abhängigkeit auf Datei `f` ein

```
cIncludes :: FilePath -> Action [FilePath]
cIncludes x =
    do s <- readFile' x
       return $ mapMaybe parseInclude (lines s)
    where
      parseInclude line =
          do rest <- List.stripPrefix "#include \"" line
             return $ takeWhile (/= '"') rest
```

shake in Aktion (2)

- `Rules`-Monad: ermöglicht Definition von Build-Regeln

```
(*>) :: FilePattern -> (FilePath -> Action ()) -> Rules ()

rules :: Rules ()
rules =
    "*.o" *> \out ->
        do let c = replaceExtension out "c"
           need (cIncludes c)
           system' "gcc" ["-o", out, "-c", c]
```

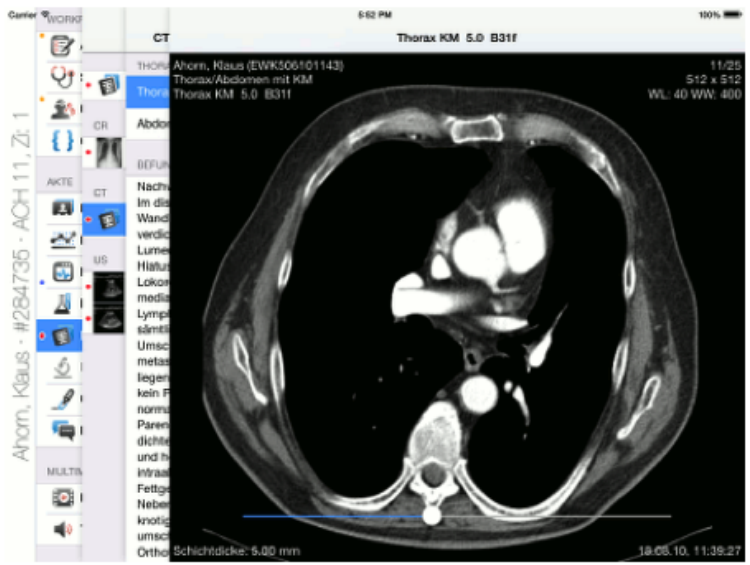
Abschluss: mögliche Erweiterungen des Messaging-Systems

- Deregistrierung von Subscribern am Server
  - Bisher deregistriert ein Client gegenüber dem Server seine Subscriber nie, auch wenn es keine lokalen Subscriber mehr gibt.
- Timestamp für Nachrichten
  - Neben einer ID und der Payload sollen Nachrichten auch eine Zeitstempel bekommen (Zeitpunkt der Erstellung).
- Synchrone Schnittstelle
  - Die bisherige Schnittstelle über das Netzwerk ist asynchron
  - Zusätzlich: synchrone Schnittstelle mit Bestätigungen für Registrierung und Publishing.
  - Kein `sleepTimeSpan` mehr in den Tests nötig!
- Dynamisches Erstellen von Queues
  - Bisher werden Queues ganz am Anfang erzeugt und es gibt keine Möglichkeit, neue Queues später hinzuzufügen oder alte zu entfernen.

Fazit & Zusammenfassung

- Mit Haskell wird kommerzielle Software entwickelt!
- Haskell punktet auch in der Industrie:
  - Hohe Produktivität
  - Korrektheit
  - Sicherheit
  - Hohe Wiederverwendbarkeit
- Haskell ist “World’s finest imperative programming language”
- Haskell ist performant
- Blog: http://funktionale-programmierung.de

Jobs



- ✓Software Engineer (Haskell)
- ✓Integration Engineer (Haskell)
- ✓Support Engineer
- ✓Software QA Engineer
- ✓Software Engineer (iOS)
- ✓Software Engineer (Android)
- ✓DevOps Engineer
- ✓Unix-Admin
- ✓Praktikum