Universidade Federal de Ouro Preto
Instituto de Ciências Exatas e Biológicas
Departamento de Computação
BCC362 - Sistemas Distribuídos

Um Sistema Distribuído Transparente Implementado em Haskell

Pedro Ribeiro Mendes Júnior
Prof.: Joubert de Castro Lima

O objetivo deste trabalho foi a implementação de um sistema distribuído transparente, utilizando a linguagem de programação Haskell, dividido nas partes cliente, servidor e *slaves*.

Ouro Preto - Minas Gerais - Brasil
16 de dezembro de 2011

Sumário

Lista de Figuras

Lista de Códigos

1 Introdução

Na realização deste trabalho, foram criados cinco programas principais: um cliente (ver Seção 3.2), dois servidores (ver Seções 4.2 e 4.3) e dois *slaves* (ver Seções 5.3 e 5.4). Para a criação destes programas foi necessário o desenvolvimento de módulos que capturam os padrões de computação de sistemas distribuídos e que implementam algumas transparências, pois um dos objetivos deste trabalho foi buscar em um sistema distribuído as seguintes transparências [3]:

- **Transparência de acesso:** abilita o acesso a recursos locais e remotos usando as mesmas operações;
- **Transparência de localização:** abilita o acesso a recursos sem conhecimento de suas localizações físicas e de rede, por exemplo, em qual país está localizado ou qual é seu endereço IP;
- **Transparência de migração:** abilita o cliente a continuar operar com o recurso, mesmo quando este mudar de localização e um cliente já estiver vinculado a ele;
- **Transparência de relocação:** abilita recursos que se movimentam entre invocações serem acessados sem que o cliente necessite possuir qualquer indicação de sua movimentação e de sua posição;
- **Transparência de replicação:** abilita várias instâncias de recursos serem usadas para aumentar a confiança e melhorar o tempo de execução sem o conhecimento das réplicas por parte dos usuários ou programadores das aplicações;
- **Transparência de concorrência:** abilita vários processos operarem concorrentemente usando recursos compartilhados sem a interferência entre eles;
- **Transparência à falha:** abilita a ocultação das falhas, permitindo usuários e programas de aplicações completarem suas tarefas apesar de falhas de *hardware* ou de componentes de *software*.

Além das transparências citadas acima, outro objetivo deste trabalho foi buscar um sistema distribuído assíncrono, ou seja, um sistema em que é possível delegar a tarefa a um dos *slaves* e continuar com a execução do lado do cliente enquando a computação é realizada do lado do *slave*. Na Seção 3.2.1 é mencionado como isso foi facilmente obtido com a implementação em Haskell [7].

Na Figura 1 está representado o desenho arquitetural do sistema, que contém como principais componentes os clientes, o servidor principal, o servidor de atualizações das informações utilizadas pelo servidor principal e os *slaves*.

O código fonte obtido com a realização deste trabalho encontra-se disponível em [2].

Este trabalho está organizado da seguinte maneira: na Seção 2 é explicado sobre a implementação de alguns módulos auxiliares utilizados para a implementação dos demais módulos deste trabalho. A Seção 3 apresenta a implementação do módulo `Client`, que é o módulo utilizado por qualquer programa cliente, e a implementação simples de um

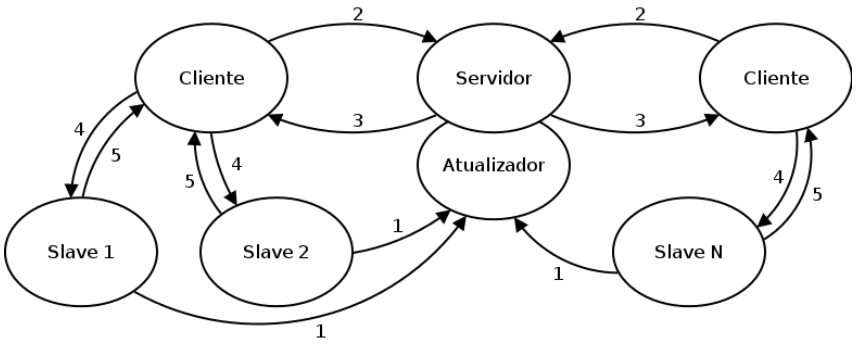

Figura 1: Desenho arquitetural do sistema. Os números da imagem informam o tipo de comunicação realizada entre os componentes: 1) Informar ao servidor quais funções são computadas pelo *slave*; 2) Requisitar a informação de quais *slaves* computam a função desejada; 3) Informar a localização dos *slaves*; 4) Requisitar computação da função; 5) Enviar o resultado da computação.

cliente que requisita a computação de algumas funções de maneira distribuída. O módulo que necessita ser utilizado para a implementação de um servidor é descrito na Seção 4. Além disso, nessa seção são apresentadas a implementação do servidor principal e do servidor de atualizações. Na Seção 5 é exibida a implementação do módulo `Slave` e dois exemplos de implementações de *slaves*. Na Seção 6 são apresentados aspectos práticos para a implementação de um sistema distribuído com base nas implementações realizadas neste trabalho. Por fim, a Seção 7 apresenta conclusões obtidas com a realização deste trabalho e possíveis trabalhos futuros.

2 Módulos auxiliares

Nesta seção são apresentadas as implementações de alguns módulos que foram utilizados por mais de um dos módulos que implementam o sistema distribuído apresentado neste trabalho

2.1 Módulo MaybeExceptions

Nesta seção é apresentado o módulo `MaybeExceptions`, que implementa algumas funções para tratar exceções. As funções apresentadas neste módulo são principalmente utilizadas pelo módulo `Client` (ver Seção 3.1), pois o módulo `Client` contém a implementação de funções que comunicam com o servidor principal e com os *slaves*, por isso pode ocorrer erros na comunicação.

O Código 1 mostra a implementação do módulo `MaybeExceptions`.

Código 1: Módulo `MaybeExceptions` que contém a implementação algumas funções para tratamento de exceções.

```
1
2  module MaybeExceptions
3          (
4            -- * Functions
5            catchMaybe,
6            catchMaybeJoin,
7            catchIO,
8            timeout,                    -- From "System.Timeout"
9          )
10         where
11
12 import Prelude hiding (catch)
13 import Control.Exception
14 import Control.Monad (join)
15 import System.Timeout (timeout)
16
17 catchMaybe :: IO a -> IO (Maybe a)
18 catchMaybe comp =
19   (comp >>= (return . return)) `catch` f
20   where f :: SomeException -> IO (Maybe a)
21         f e = do putStrLn $ "catchMaybe: " ++ show e -- verbose
22                  return Nothing
23
24 catchMaybeJoin :: IO (Maybe a) -> IO (Maybe a)
25 catchMaybeJoin comp = catchMaybe comp >>= return . join
26
27 catchIO :: IO () -> IO ()
28 catchIO comp =
29   comp `catch` f
```

```
30    where f :: SomeException -> IO ()
31          f e = putStrLn $ "catchIO: " ++ show e -- verbose
```

Antes de descrever cada uma das funções contidas no Código 1, é importante saber como é definido o tipo `Maybe`. O Código 2 apresenta a definição deste tipo.

Código 2: Definição do tipo `Maybe`.

```
1 data Maybe a = Just a | Nothing
```

O tipo `Maybe` é importante para representar falhas na computação. Quando é possível obter um resultado para a computação o retorno pode ser dado como `Just a`, onde `a` é o resultado da computação. Quando, por algum motivo, o resultado não pode ser obtido, o retorno pode ser dado como `Nothing`.

As Seções seguintes apresentam explicações sobre as funções mostradas no Código 1.

2.1.1 Função catchMaybe

A função `catchMaybe` recebe uma computação que obterá um valor como resultado. Caso a computação seja realizada com sucesso, o valor resultante desta computação é colocado na mônada `Maybe`. Caso ocorra alguma exceção durante a computação, esta exceção é capturada e o valor retornado é `Nothing`

2.1.2 Função catchMaybeJoin

Uma vez que uma computação do tipo `IO (Maybe a)` também pode causar exceções, foi implementada a função `catchMaybeJoin` que recebe uma computação do tipo `IO (Maybe a)` e caso essa computação é realizada com sucesso, o seu resultado é simplemente retornado sem alterações, ou seja, quando não há exceções na computação a seguinte propriedade é válida:

`catchMaybeJoin comp` $\equiv$ `comp`,

onde `comp` é a computação a ser realizada. Mas quando acontece alguma exceção a função `catchMaybeJoin` capturará a exceção e a computação `catchMaybeJoin comp` terá resultado `IO Nothing`.

Observe também que o resultado `IO Nothing` também será obtido quando a computação recebida como parâmetro pela função `catchMaybeJoin` obter resultado `IO Nothing`.

2.1.3 Função catchIO

A função `catchIO` não informará se a computação recebida como argumento foi realizada com sucesso ou não, ou seja, se houve exceções ou não. Esta função tem como objetivo simplesmente capturar as exceções que venham a ocorrer para que a execução do programa não seja comprometida.

2.1.4 Função timeout

Apesar da função `timeout` não ser uma função que lida com exceções, foi decidido exportá-la pelo módulo `MaybeExceptions`, pois neste trabalho sua utilização normalmente se dá quando também são utilizadas as demais funções exportadas por este módulo e por seu tipo de retorno também ser do tipo `Maybe`. Em Haskell, esta função é implementada no módulo `System.Timeout` e seu tipo é `Int -> IO a -> IO (Maybe a)`.

O primeiro argumento da função (argumento de tipo `Int`) é um tempo em microsegundos que especifica o tempo máximo que a computação do segundo argumento executará. Caso a computação termine dentro do tempo especificado o resultado é obtido dentro da mônada Maybe. Caso o tempo estoure e a computação não foi terminada, a computação é cancelada e o retorno é dado como `Nothing`.

2.2 Módulo BookLocations

Antes de ser apresentada a implementação do módulo `BookLocations` contida no Código 3, é apresentado como é especificado o arquivo que contém as informações que o servidor utiliza para informar ao cliente quais são os *slaves* que computam a função desejada pelo cliente. Neste trabalho, este arquivo é chamado de `BookLocations`, pois o tipo `BookLocations` definido na linha 17 do Código 3 é utilizado para representar tal arquivo.

2.2.1 Arquivo BookLocations

Como mostrado na Figura 1, o servidor principal (ver Seção 4.2) e o servidor de atualizações do arquivo `BookLocations` (ver Seção 4.3; referenciado como `Atualizador` na Figura 1) devem ser executados na mesma máquina, pois ambos compartilharão do mesmo arquivo: enquanto um apenas lê, o outro lê e o atualiza.

Este arquivo mantém as informações de quais *slaves* computam uma dada função. Ele é organizado da seguinte maneira:

- As linhas ímpares contém o nome de alguma função;
- Cada linha par contém as informações de quais máquinas computam a função da linha imediatamente anterior;
- Cada uma dessas informações de localização das máquinas estão separadas por espaço;
- Cada informação de localização é armazenada como `host:port`, onde `host` é o endereço da máquina e `port` é a porta de comunicação.

Como é mostrado na linha 20 do Código 3, o arquivo deve estar localizado no mesmo diretório que o programa executável que trabalha sobre o arquivo `BookLocations`. Além disso, esse arquivo estará armazenado com o nome `BookLocations.dat`.

2.2.2 Implementação do módulo BookLocations

O Código 3 apresenta a implementação do módulo `BookLocations`.

Código 3: Módulo `BookLocations` que contém a implementação de algumas funções que trabalham sobre o arquivo `BookLocations`.

```
1
2  module BookLocations
3          (
4            readBookLocations,
5            writeBookLocations,
6            getLocations,
7            updateLocations,
8          )
9          where
10
11 import System.IO
12   (hGetLine, hClose, hPutStrLn, hSetBuffering, BufferMode(..),
13    Handle, stdout, IOMode(..), openFile, hIsEOF)
14 import Data.Map as Map
15 import Data.List as List (intercalate, union)
16
17 type BookLocations = Map String [String]
18
19 book_locations :: FilePath
20 book_locations = "BookLocations.dat"
21
22 readBookLocations :: IO BookLocations
23 readBookLocations = do
24   h <- openFile book_locations ReadMode
25   l <- readLocationsFile h
26   hClose h
27   return $ Map.fromList l
28
29 readLocationsFile :: Handle -> IO [(String, [String])]
30 readLocationsFile h = hIsEOF h >>= \eof ->
31   if eof then return [] else do
32     fn <- hGetLine h
33     line_las <- hGetLine h
34     let las = words line_las
35     remaining <- readLocationsFile h
36     return $ (fn, las) : remaining
37
38 writeBookLocations :: BookLocations -> IO ()
39 writeBookLocations bl = let
40   fns  = Map.keys bl
41   llns = Map.elems bl
42   in do
43     h <- openFile book_locations WriteMode
```

```
44      writeLocationsFile h fns llns
45      hClose h
46
47  writeLocationsFile :: Handle -> [String] -> [[String]] -> IO ()
48  writeLocationsFile h (fn:fns) (lln:llns) = do
49    hPutStrLn h fn
50    hPutStrLn h $ intercalate "␣" lln
51    writeLocationsFile h fns llns
52  writeLocationsFile _ _ _                  = return ()
53
54  getLocations :: BookLocations -> String -> Maybe [String]
55  getLocations bl fn = Map.lookup fn bl
56
57  updateLocations ::
58    String -> BookLocations -> String -> BookLocations
59  updateLocations adrs bl fn =  case Map.lookup fn bl of
60    Nothing    -> Map.insert fn [adrs] bl
61    Just adrss -> Map.insert fn (List.union adrss [adrs]) bl
```

Como dito anteriormente, o tipo definido na linha 17 do Código 3 foi definido para representar o conteúdo do arquivo `BookLocations`. O arquivo é simplesmente representado como um mapeamento de uma `String` (nome da função) a uma lista de `String`s (lista de informações das localizações dos *slaves*).

As funções exportadas pelo módulo são facilmente compreensíveis, pois elas só são utilizadas para trabalhar sobre o tipo `BookLocations`.

2.2.3 Função readBookLocations

A função `readBookLocations` é intuitivamente compreendida pois seu objetivo é simplesmente obter um `BookLocations` a partir de um arquivo `BookLocations` localizado no mesmo diretório que o executável e com o nome `BookLocations.dat`.

2.2.4 Função writeBookLocations

Esta função tem o sentido inverso da função mostrada anteriormente, pois no caso da `writeBookLocations`, um `BookLocations` é recebido como parâmetro e é atualizado o arquivo `BookLocations`.

2.2.5 Função getLocations

A função `getLocations` simplesmente retorna uma lista de `String`s, onde esses `String`s são as informações de localização das máquinas que computam a função desejada.

2.2.6 Função updateLocations

O primeiro argumento da função `updateLocations` é a informação de localização de uma máquina, o segundo um `BookLocations` e o terceiro o nome da função que é computada pela máquina do primeiro argumento. O retorno desta função é um `BookLocations` atualizado.

2.3 Módulo ConfigFile

Antes de se fazer referência à implementação do módulo `ConfigFile` (ver Código 4), nesta seção é tratado de como é organizado os arquivos de configuração de cada uma das partes do sistema distribuído. Neste trabalho, esse arquivo de configuração é chamado de arquivo `ConfigFile`, pois o tipo `ConfigFile` definido na linha 20 do Código 4 é utilizado para representar tal arquivo.

2.3.1 Arquivo ConfigFile

Cada programa executável (cliente, servidores e *slaves*) do sistema distribuído necessita de um arquivo de configuração. Este arquivo deve estar localizado no mesmo diretório do programa executável e deve ser criado com o nome `ConfigFile.dat`.

Cada linha do arquivo `ConfigFile` está organizado como `keyword:value`, onde `keyword` é o nome de um atributo que pode ser necessário pelo programa executável de alguma das partes do sistema distribuído e `value` especifica o valor do atributo. Nas Seções 6.2, 6.4 e 6.5 são especificados os atributos necessários de serem definidos para cada uma das partes.

O Código 4 mostra a implementação deste módulo e a seguir são dadas algumas explicações de sua implementação.

Código 4: Módulo `ConfigFile` que contém a implementação de algumas funções que trabalham sobre os arquivos de configurações.

```
1
2  module ConfigFile
3         (
4           readConfigFile,
5           configLookup,
6           -- * Read Functions
7           readIP,
8           readPort,
9           readTimeout,
10        )
11        where
12
13 import System.IO
14   (openFile, IOMode(..), Handle, hIsEOF, hGetLine, hClose)
15 import Data.Map as Map
```

```
16    (Map, empty, insert, lookup)
17  import Network
18    (PortNumber)
19
20  type ConfigFile = Map String String
21
22  config_file :: FilePath
23  config_file = "ConfigFile.dat"
24
25  readConfigFile :: IO ConfigFile
26  readConfigFile = do
27    h <- openFile config_file ReadMode
28    readConfigFile' h empty
29
30  readConfigFile' :: Handle -> ConfigFile -> IO ConfigFile
31  readConfigFile' h cf = hIsEOF h >>= \eof ->
32    if eof then hClose h >> return cf else do
33      line <- hGetLine h
34      let (keyword, value) = break' ':' line
35      readConfigFile' h (insert keyword value cf)
36
37  break' :: (Eq a) => a -> [a] -> ([a], [a])
38  break' c as = case break (== c) as of
39    (l, r) -> if null r then (l, r) else (l, tail r)
40
41  configLookup :: ConfigFile -> String -> (String -> a) -> a
42  configLookup cf key f = maybe undefined f $ Map.lookup key cf
43
44  readIP :: String -> String
45  readIP = id
46
47  readPort :: String -> PortNumber
48  readPort = fromIntegral . read
49
50  readTimeout :: String -> Int
51  readTimeout = read
```

2.3.2 Função readConfigFile

A função `readConfigFile` procura pelo arquivo `ConfigFile.dat` localizado no mesmo diretório do programa executável e retorna o conteúdo deste arquivo represetado pelo tipo `ConfigFile`, definido na linha 20 do Código 4.

2.3.3 Função configLookup

Como dito no início da Seção 2.3 cada linha do arquivo `ConfigFile` contém as informações organizadas como `keyword:value`. A função `configLookup` recebe um `ConfigFile`, um `keyword` e uma função que recebe um `value` (armazenado como `String`) e retorna o `value` lido a partir do `String`. Seu resultado é o `value` lido.

Na Seção 2.3.4 são apresentadas algumas dessas funções de conversão.

2.3.4 Demais funções do módulo

As demais funções exportadas pelo módulo tem como objetivo serem utilizadas juntamente com a função `configLookup`, pois todas essas funções são funções que lêem o valor requerido a partir de um `String`.

3 Parte cliente

Nesta seção é apresentado o módulo `Client`, que é o módulo utilizado por qualquer implementação de programa cliente que for utilizar o sistema distribuído criado neste trabalho.

Além disso, nesta seção é apresentada a implementação de um programa cliente. Esse programa cliente requer a computação de algumas funções de maneira distribuída. Com isso, obtém-se um exemplo da utilização dos módulos implementados com a realização deste trabalho para a criação de programas clientes que farão uso da computação distribuída.

3.1 Módulo Client

Como pode-se ver no Código 5, as únicas funções exportadas pelo módulo `Client` são as funções `compute` e `getResult`, mas observe também que os operadores `$/` e `$\` são exportados pelo módulo e são equivalentes às funções `compute` e `getResult`, respectivamente, podendo ser usados ao invés das funções. A primeira é a função encarregada de conectar ao servidor, obter a informação de quais *slaves* computam a função requerida, delegar a tarefa a um dos *slaves* e em seguida retornar um `Maybe Ticket` que será depois utilizado pela segunda função para que o usuário possa obter o resultado da computação quando desejado.

Observe que o tipo `Ticket` definido na linha 18 deste módulo não é exportado pelo módulo, ou seja, o usuário não poderá trabalhar sobre esse tipo, pois as únicas funções capazes de trabalhar sobre ele são as funções definidas dentro do módulo. Com isso, o retorno da função `compute` só será utilizado para obter o resultado da computação posteriormente.

A função `compute` tenta conectar a um dos *slaves* informados pelo servidor. Quando a conexão é realizada e a tarefa é delegada ao *slave* sem problemas, um `Ticket` é retornado. Uma vez que o retorno da função `compute` tem o tipo `Maybe`, este retorno poderá ser utilizado pelo usuário para verificar se a tarefa pôde ser delegada com sucesso.

Código 5: Módulo `Client` que necessita ser importado pelo programa que executará como cliente.

```
1
2  module Client
3         (
4           compute,
5           ($/),
6           getResult,
7           ($\),
8         )
9         where
10
11 import Network
12 import System.IO (hGetLine, hClose, hPutStrLn, hSetBuffering,
13                    BufferMode(..), Handle, stdout)
14
```

```
15 import MaybeExceptions
16 import ConfigFile
17
18 data Ticket = Ticket String String Handle
19             deriving (Eq, Show)
20
21 askForHostName :: Handle -> String -> IO [(String, String)]
22 askForHostName h fn = do
23   hPutStrLn h fn
24   lhn_maybe <- hGetLine h
25   case read lhn_maybe :: Maybe [String] of
26     Just lhn -> return $ map (break ':') lhn
27     Nothing -> return []
28
29 askForComputation :: Handle -> String -> String -> IO ()
30 askForComputation h fn param = do
31   hPutStrLn h fn
32   hPutStrLn h param
33
34 infix 1 $/
35 ($/) :: (Show a) => String -> a -> IO (Maybe Ticket)
36 ($/) = compute
37
38 compute :: (Show a) => String -> a -> IO (Maybe Ticket)
39 compute fn param = withSocketsDo $ catchMaybeJoin $ do
40   cf <- readConfigFile
41   let ($>$) :: String -> (String -> a) -> a
42       ($>$) s f = configLookup cf s f
43
44       server_port = "serverPort" $>$ readPort
45       server_ip = "serverIP" $>$ readIP
46
47   hSetBuffering stdout LineBuffering
48   h_server <- connectTo server_ip (PortNumber server_port)
49   hSetBuffering h_server LineBuffering
50
51   l <- askForHostName h_server fn
52   print l                          -- verbose
53   hClose h_server
54
55   compute' l
56   where
57     compute' :: [(String, String)] -> IO (Maybe Ticket)
58     compute' ((slave_ip, slave_port_str):xs) = let
59       slave_port :: PortNumber
60       slave_port = fromIntegral (read slave_port_str :: Integer)
61       in
62        catchMaybe
```

```
63          (connectTo slave_ip (PortNumber slave_port)) >>= maybe
64          (compute' xs)
65          (\h -> do
66              let show_param = show param
67              hSetBuffering h LineBuffering
68              (catchMaybe $ askForComputation h fn show_param) >>=
69                maybe
70                (compute' xs)
71                (const $ return $ return (Ticket fn show_param h)))
72      compute' [] = return Nothing
73
74  infix 1 $\
75  ($\) :: Maybe Ticket -> (String -> a) -> IO (Maybe a)
76  ($\) = getResult
77
78  getResult :: Maybe Ticket -> (String -> a) -> IO (Maybe a)
79  getResult Nothing _                    = return Nothing
80  getResult (Just (Ticket fn param h)) f = do
81    cf <- readConfigFile
82    let ($>$) :: String -> (String -> a) -> a
83        ($>$) s f = configLookup cf s f
84
85        server_port = "serverPort" $>$ readPort
86        server_ip   = "serverIP"   $>$ readIP
87        tout        = "timeout"    $>$ readTimeout
88
89        compute' = withSocketsDo $ do
90          hSetBuffering stdout LineBuffering
91          putStrLn "getResult: original Ticket fail" -- verbose
92          --
93          catchMaybe $ hClose h
94          --
95          h_server <- connectTo server_ip (PortNumber server_port)
96          hSetBuffering h_server LineBuffering
97          --
98          l <- askForHostName h_server fn
99          print l                  -- verbose
100         hClose h_server
101         --
102         compute'' l
103
104       compute'' ((slave_ip, slave_port_str):xs) = let
105         slave_port :: PortNumber
106         slave_port =
107           fromIntegral (read slave_port_str :: Integer)
108         in catchMaybe
109            (connectTo slave_ip (PortNumber slave_port)) >>= maybe
110            (compute'' xs)
```

```
111             (\h -> do
112                 hSetBuffering h LineBuffering
113                 catchMaybe $ askForComputation h fn param
114                 getResult' tout h f >>= maybe
115                   (do catchMaybe $ hClose h
116                       if null xs
117                         then putStrLn "getResult: last fail"
118                         else putStrLn "getResult: attempt fail"
119                       compute'' xs)
120                   (return . return))
121       compute'' [] = return Nothing
122
123   catchMaybeJoin $
124     getResult' tout h f >>= maybe compute' (return . return)
125
126 getResult' :: Int -> Handle -> (String -> a) -> IO (Maybe a)
127 getResult' tout h f = catchMaybeJoin $
128                       timeout tout $
129                       hGetLine h >>= return . f
130
131 break' :: (Eq a) => a -> [a] -> ([a], [a])
132 break' c as = case break (== c) as of
133   (l, r) -> if null r then (l, r) else (l, tail r)
```

No momento de obter o resultado com `getResult`, o `Maybe Ticket` retornado pela função `compute` é dado como parâmetro. Além disso, é dado como parâmetro à função `getResult` uma função do tipo `String -> a`, onde `a` é o resultado desejado pela computação.

Observe que no pior caso o resultado não poderá ser obtido. Por isso o tipo de retorno da função `getResult` é do tipo `Maybe a`.

A função `getResult` primeiro tenta obter o resultado do *slave* que recebeu a tarefa delegada pela função `compute`. Caso o tempo em aguardo estoure o tempo indicado no `ConfigFile` do cliente (ver Seção 6.2) ou a conexão com o *slave* se perca, a função `getResult` delegará a tarefa novamente a outros *slaves* e aguardará em cada caso o tempo máximo especificado pelo `timeout` no `ConfigFile` do cliente.

Especificamente, a função `getResult'` é a responsável por obter o resultado do *slave*. Observe que ela utiliza a função `timeout` mencionada na Seção 2.1 para limitar o tempo de espera pelo resultado.

3.1.1 Obtenção das transparências de acesso e localização

A função `compute` implementa as transparências de acesso e de localização, pois o usuário que utilizará esta função, a utilizará como se a computação fosse realizada localmente e não se preocupando com a localização da máquina que realmente realizará a computação.

Isso será visto na Seção 3.2, pois o Código 6 dessa seção apresenta um exemplo de implementação de um programa cliente que utiliza a função `compute` para requerer a com-

putação de funções de maneira distribuída.

3.1.2 Obtenção da transparência à falha

A transparência à falha é obtida neste módulo, pois qualquer eventual falha que venha a ocorrer ao longo da computação de alguma função por algum *slave* será capturada e uma nova tentativa de computação será realizada.

Observe o uso das funções do módulo `MaybeExceptions`, descritas na Seção 2.1, ao longo da implementação das funções `compute` e `getResult`. Essas funções são utilizadas extensivamente justamente para capturar as exceções que podem vir a ocorrer o quanto antes, para que não se propagarem e para que novas tentativas para realizar a computação requerida ocorram.

Observe também que mesmo quando não é possível realizar a computação requerida, o programa do cliente não está comprometido, pois o retorno da função `getResult` é do tipo `Maybe`, portanto obter o retorno `Nothing` significa que a computação não pôde ser realizada e decisões podem ser tomadas no programa cliente baseando-se nesta informação.

3.2 Programa principal do cliente

O Código 6 contém a implementação de um programa cliente que pede pela computação de algumas funções de maneira transparente.

Código 6: Módulo `Main` que implementa um programa cliente.

```
1
2  module Main
3          (
4            main,
5          )
6          where
7
8  import Client
9  import Data.Array
10 import Computations.Tree
11
12 treeExample :: Tree Char
13 treeExample =
14   Node
15   (Node (Node (Node
16                 (Leaf '!')
17                 (Leaf '!'))
18           (Node
19            (Leaf '!')
20            (Leaf '!')))
21    (Node (Node
22            (Leaf '!')
```

```
23              (Leaf 'd'))
24       (Node
25        (Leaf 'l')
26        (Leaf 'r'))))
27    (Node (Node (Node
28                 (Leaf 'o')
29                 (Leaf 'W'))
30            (Node
31             (Leaf ' ')
32             (Leaf 'o')))
33     (Node (Node
34             (Leaf 'l')
35             (Leaf 'l'))
36      (Node
37       (Leaf 'e')
38       (Leaf 'H'))))
39
40 main :: IO ()
41 main = do
42   t1 <- "fibonacci" $/ 30
43   t2 <- "treeCharToInt" $/ treeExample
44   t3 <- "sudoku" $/
45         ".......21" ++
46         "43......." ++
47         "6........" ++
48         "2.15....." ++
49         ".....637." ++
50         "........." ++
51         ".68...4.." ++
52         "...23...." ++
53         "....7...."
54
55   result1 <- t1 $\ (read :: String -> Int)
56   result2 <- t2 $\ (read :: String -> Tree Int)
57   result3 <- t3 $\ (read :: String
58                             -> Maybe (Array (Char, Char) [Char]))
59
60   putStr "Result1: "
61   print result1
62   putStr "Result2: "
63   print result2
64   putStr "Result3: "
65   print result3
```

Como pode ser observado nas linhas 42, 43 e 44 a maneira de requerer a computação de qualquer que seja a função é sempre a mesma, não importando onde a máquina que realmente irá realizar a computação esteja localizada, ou seja, observa-se nesse caso a existência das transparências de acesso e de localização.

Para requerer a computação da função é simplesmente necessário utilizar a função `compute` (ou o operador `$/`), passando como primeiro argumento o nome da função requerida (um `String`) e como segundo argumento o parâmetro da função.

Confirmando o que foi dito na Seção 3.1.2, é bom mencionar que no caso de falhas, pela função `compute` ou pela função `getResult`, o valor obtido por elas será `Nothing`, não fazendo com que a execução do programa cliente seja comprometida.

3.2.1 Obtenção de um sistema assíncrono

Pela maneira que foi implementado o sistema, obteve-se um sistema assíncrono, pois o retorno da função `compute` é do tipo `Ticket`. Observe na linha 18 do Código 5 que este tipo é definido contendo um campo com um valor do tipo `Handle`. Isso implica que o programa do cliente só esperará o término da computação da função quando a função `getResult` requisitar o resultado por meio do `Handle` contido no `Ticket`.

4 Parte servidor

Nesta seção é apresentado o módulo `Server`, que é o módulo utilizado por qualquer programa que implementa um servidor, inclusive os *slaves* que são apresentados na Seção 5.

Além disso, é apresentado a implementação de um servidor, cujo propósito é atender às requisições dos clientes para informar os *slaves* passíveis de computar funções desejadas pelos clientes.

Outra implementação de servidor apresentada nesta seção é o servidor de atualizações, que tem como alvo receber as informações dos *slaves* de quais são as funções computadas por eles e atualizar o arquivo `BookLocations` para que o servidor mencionado no parágrafo anterior faça uso dessas informações.

4.1 Módulo Server

De acordo com o Código 7, que contém o módulo que implementa a função básica de qualquer servidor, que é fazer com que o servidor execute indefinidamente atendendo às requisições, pode-se observar que a única função exportada pelo módulo é `serverMain`.

Código 7: Módulo `Server` que necessita ser importado pelo programa que executará como servidor.

```
1
2  module Server
3          (
4            module Network,
5            serverMain,
6          )
7          where
8
9  import System.IO
10   (hGetLine, hClose, hPutStrLn, hSetBuffering,
11    BufferMode(..), Handle, stdout, IOMode(..), openFile, hIsEOF)
12 import Network
13 import Control.Concurrent
14   (forkIO)
15
16 import MaybeExceptions
17 import ConfigFile
18
19 serverMain ::
20   (Handle -> HostName -> PortNumber -> IO ()) -> IO ()
21 serverMain fcr = withSocketsDo $ do
22   cf <- readConfigFile
23   let ($>$) :: String -> (String -> a) -> a
24       ($>$) s f = configLookup cf s f
25
```

```
26        server_port = "serverPort" $>$ readPort
27
28    hSetBuffering stdout LineBuffering
29    sock <- listenOn (PortNumber server_port)
30    let serverMain' :: IO ()
31        serverMain' = withSocketsDo $ do
32          putStrLn "Awaiting connection..." -- verbose
33          (h, host, port) <- accept sock
34          putStrLn $
35            "Received connection from " ++ (prepareHost host) ++
36            ":" ++ show port -- verbose
37
38          forkIO $
39            catchIO $ fcr h (prepareHost host) port
40          serverMain'
41    serverMain'
42
43  prepareHost :: String -> String
44  prepareHost s = prepareHost' s s
45    where prepareHost' (x:xs) res =
46            prepareHost' xs (if x == ':' then xs else res)
47          prepareHost' [] res     = res
```

Uma vez que uma conexão é aceita por um servidor, como observado na linha 33 do Código 7, são obtidos um `Handle` de comunicação, o nome da máquina e a porta por onde está sendo realizada a comunicação com a máquina que requisitou a comunicação com o servidor. Desse modo, a função `serverMain` foi implementada para receber uma função que tem como argumentos esses três tipos (`Handle`, `HostName` e `PortNumber`) e que realiza a tarefa de atender uma requisição, ou seja, essa função recebida como primeiro argumento da função `serverMain` tem como objetivo atender às requisições feitas ao servidor.

A função deste módulo será utilizada por qualquer tipo de servidor, inclusive os *slaves*, que também ficarão executando indefinidamente atendendo às requisições. Observe que a computação definida por `serverMain'` fica executando indefinidamente, recebendo requisições e atendendo-as.

4.2 Programa principal do servidor

De acordo com o mencionado na Seção 4.1, ocorre que para a implementação de um servidor é necessário a implementação de uma função do tipo `Handle -> HostName -> PortNumber -> IO ()`. A implementação desta função para a construção do servidor principal é apresentada no Código 8.

Código 8: Módulo `Main` que implementa um programa servidor.

```
1
2  module Main
3          (
4            main,
```

```
5           )
6           where
7
8   import Server
9   import BookLocations
10
11  import System.IO
12    (hGetLine, hClose, hPutStrLn, hSetBuffering, BufferMode(..),
13     Handle)
14
15  computeRequisition :: Handle -> HostName -> PortNumber -> IO ()
16  computeRequisition h _ _ = do
17    hSetBuffering h LineBuffering
18
19    bl <- readBookLocations
20
21    putStr "Receiving function name... " -- verbose
22    fn <- hGetLine h
23    putStrLn fn                                -- verbose
24    putStr "Sending location addresses... " -- verbose
25    let la = getLocations bl fn
26    print la                          -- verbose
27    hPutStrLn h $ show la
28
29    hClose h
30
31  main :: IO ()
32  main = serverMain computeRequisition
```

Para a implementação do servidor principal apresentado nesta seção, a função que atende às requisições (`computeRequisitions`) é implementada de modo a receber do cliente, via o `Handle` do primeiro argumento, o nome de uma função e simplesmente informar os *slaves* capazes de computar a função desejada, de acordo com as informações contidas no arquivo `BookLocations`.

Observe na linha 32 do Código 8 que o programa do servidor principal implementado por este módulo simplesmente utiliza a função `serverMain`, informando uma função para atender às requisições.

4.3 Programa principal do servidor de atualizações

O módulo descrito nesta seção implementa um servidor para atender às requisições dos *slaves*. Para atender a tais requisições este servidor simplesmente recebe as informações de cada um dos *slaves* de quais são as funções que eles computam e as insere no arquivo `BookLocations` que armazena tais informações.

Da mesma maneira que o servidor descrito na Seção 4.2 implementa uma função para atender às requisições, este módulo também implementa tal função, como pode ser visto

no Código 9.

Código 9: Módulo `Main` que implementa o servidor responsável por receber as atualizações do arquivo `BookLocations`.

```
1
2  module Main
3          (
4            main,
5          )
6          where
7
8  import Server
9  import BookLocations
10
11 import System.IO
12   (hGetLine, hClose, hPutStrLn, hSetBuffering, BufferMode(..),
13    Handle, stdout)
14 import qualified Data.Map as Map
15 import qualified Data.List as List (union)
16
17 computeRequisition :: Handle -> HostName -> PortNumber -> IO ()
18 computeRequisition h host _ = do
19   hSetBuffering h LineBuffering
20
21   bl <- readBookLocations
22
23   putStr "Receiving␣Slave␣address...␣" -- verbose
24   port <- hGetLine h
25   let adrs = host ++ ":" ++ (read port :: String)
26   putStrLn adrs                              -- verbose
27   putStr "Receiving␣function␣names...␣" -- verbose
28   fns_str <- hGetLine h
29   putStrLn fns_str                  -- verbose
30   let fns = read fns_str :: [String]
31       new_bl = foldl (updateLocations adrs) bl fns
32   writeBookLocations new_bl
33
34   hClose h
35
36 main :: IO ()
37 main = serverMain computeRequisition
```

Como representado pela Figura 1, este servidor deve executar na mesma máquina que o servidor principal descrito na Seção 4.2, pois eles compartilham do mesmo arquivo `BookLocations`.

4.3.1 Obtenção da transparência de replicação

A transparência de replicação é obtida uma vez que o servidor atende uma requisição do cliente informando todas as possíveis máquinas capazes de computar a função desejada pelo cliente, com base no arquivo de atualizações. Então qualquer uma das máquinas replicadas serão informadas e uma política de decisão poderá ser implementada para decidir entre elas.

5 Parte dos slaves

Nesta seção, são apresentadas algumas implementações de funções que são computadas pelos *slaves*. Estas funções foram utizadas durante a realização deste trabalho para testes e são apresentas aqui simplesmente para exemplificar o restante deste trabalho.

Além disso, nesta seção são apresentados a implementação do módulo `Slave` (módulo utilizado pelas implementações dos programas *slaves*) e dois exemplos de implementações de programas *slaves*.

5.1 Algumas computações

Por questão de padronização e para facilitar a possível extensão deste trabalho (ver Seção 7 sobre trabalhos futuros) e também para facilitar a implementação dos *slaves* (ver Seção 6.3), cada computação passível de ser aceita pelos *slaves* deve ser implementada em um módulo à parte e deve implementar uma função com nome `compute`. Esta função `compute` deve ser do tipo `String -> String`, em que o primeiro `String` representa o argumento da computação a ser realizada e o segundo `String` representa a resposta obtida, ou seja, a função `compute` deverá ler o argumento a partir de um `String` e transformar o resultado para um `String` quando obtê-lo.

Para realização deste trabalho, foram implementadas três funções para serem utilizadas pelos *slaves*. Como será observados nos códigos que se seguem, a implementação da função compute normalmente utilizará das funções `read :: String -> a` para obter o valor a ser computado a partir de um `String` e `show :: a -> String` para transformar o resultado para um `String`.

5.1.1 Fibonacci

A função implementada no Código 10 recebe como argumento um inteiro `n` e tem como retorno o `n`-ésimo número da sequência de Fibonacci [4].

Código 10: Módulo `Fibonacci` que implementa uma computação aceita pelos *slaves*.

```
1
2  module Computations.Fibonacci
3          (
4            compute,
5          )
6          where
7
8  compute :: String -> String
9  compute = show . fibonacci . read
10
11 fibonacci :: Int -> Int
12 fibonacci 0 = 1
13 fibonacci 1 = 1
14 fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

Para a realização dos testes, foi implementado o algoritmo exponencial de Fibonacci para que houvesse um certo impacto no momento da computação.

5.1.2 Conversão de Tree Char para Tree Int

No Código 11 é apresentada a implementação de uma função que simplesmente recebe uma árvore de caracteres e retorna uma árvore de inteiros, em que os inteiros são os códigos dos caracteres originais na tabela ASCII [1].

Código 11: Módulo `Tree` que implementa uma computação aceita pelos *slaves*.

```
1
2  module Computations.Tree
3          (
4            Tree(..),
5            compute,
6          )
7          where
8
9  import Data.Char (ord)
10
11 data Tree a = Node (Tree a) (Tree a)
12             | Leaf a
13             deriving (Eq, Ord, Show, Read)
14
15 compute :: String -> String
16 compute = show . treeCharToInt . read
17
18 treeCharToInt :: Tree Char -> Tree Int
19 treeCharToInt (Node l r) =
20   Node (treeCharToInt l) (treeCharToInt r)
21 treeCharToInt (Leaf x)   = Leaf (ord x)
```

5.1.3 Sudoku

No Código 12 é apresentada a implementação de uma função que recebe como argumento um `String` que representa um tabuleiro de Sudoku [8] e retorna um `Maybe (Array (Char, Char) [Char])`.

Código 12: Módulo `Tree` que implementa uma computação aceita pelos *slaves*.

```
1 -- |
2 -- Sudoku solver using constraint propagation. The algorithm is
3 -- by Peter Norvig <http://norvig.com/sudoku.html>; the Haskell
4 -- implementation is by Manu and Daniel Fischer, and can be found
5 -- on the Haskell Wiki
6 -- <http://www.haskell.org/haskellwiki/Sudoku>
7 --
```

```
 8 -- The Haskell wiki license applies to this code:
 9 --
10 -- Permission is hereby granted, free of charge, to any person
11 -- obtaining this work (the \"Work\"), to deal in the Work
12 -- without restriction, including without limitation the rights
13 -- to use, copy, modify, merge, publish, distribute, sublicense,
14 -- and/or sell copies of the Work, and to permit persons to whom
15 -- the Work is furnished to do so.
16 --
17 -- THE WORK IS PROVIDED \"AS IS\", WITHOUT WARRANTY OF ANY KIND,
18 -- EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
19 -- WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
20 -- PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
21 -- COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
22 -- LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
23 -- OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE WORK
24 -- OR THE USE OR OTHER DEALINGS IN THE WORK.
25
26 module Computations.Sudoku
27         (
28           solve,
29           printGrid,
30           compute,
31         )
32         where
33
34 import Data.List hiding (lookup)
35 import Data.Array
36 import Control.Monad
37 import Data.Maybe
38
39 -- Types
40 type Digit  = Char
41 type Square = (Char,Char)
42 type Unit   = [Square]
43
44 -- We represent our grid as an array
45 type Grid = Array Square [Digit]
46
47
48 -- Setting Up the Problem
49 rows = "ABCDEFGHI"
50 cols = "123456789"
51 digits = "123456789"
52 box = (('A','1'),('I','9'))
53
54 cross :: String -> String -> [Square]
55 cross rows cols = [ (r,c) | r <- rows, c <- cols ]
```

```
56
57 squares :: [Square]
58 squares = cross rows cols  -- [('A','1'),('A','2'),('A','3'),...]
59
60 peers :: Array Square [Square]
61 peers = array box [(s, set (units!s)) | s <- squares ]
62       where
63         set = nub . concat
64
65 unitlist :: [Unit]
66 unitlist = [ cross rows [c] | c <- cols ] ++
67            [ cross [r] cols | r <- rows ] ++
68            [ cross rs cs | rs <- ["ABC","DEF","GHI"],
69                            cs <- ["123","456","789"]]
70
71 -- this could still be done more efficiently, but what the
72 -- heck...
73 units :: Array Square [Unit]
74 units = array box
75         [(s, [filter (/= s) u | u <- unitlist, s `elem` u ]) |
76          s <- squares]
77
78
79 allPossibilities :: Grid
80 allPossibilities = array box [ (s,digits) | s <- squares ]
81
82 -- Parsing a grid into an Array
83 parsegrid      :: String -> Maybe Grid
84 parsegrid g    = do regularGrid g
85                     foldM assign allPossibilities (zip squares g)
86
87    where  regularGrid    :: String -> Maybe String
88           regularGrid g  = if all (`elem` "0.-123456789") g
89                            then Just g
90                            else Nothing
91
92 -- Propagating Constraints
93 assign         :: Grid -> (Square, Digit) -> Maybe Grid
94 assign g (s,d) = if d `elem` digits
95                     -- check that we are assigning a digit and
96                     -- not a '.'
97                  then do
98                    let ds = g ! s
99                        toDump = delete d ds
100                   foldM eliminate g (zip (repeat s) toDump)
101                 else return g
102
103 eliminate      ::  Grid -> (Square, Digit) -> Maybe Grid
```

```
104 eliminate g (s,d) =
105   let cell = g ! s in
106   if d `notElem` cell then return g -- already eliminated
107   -- else d is deleted from s' values
108   else do let newCell = delete d cell
109               newV = g // [(s,newCell)]
110           newV2 <- case newCell of
111             -- contradiction : Nothing terminates the computation
112             []    -> Nothing
113             -- if there is only one value left in s, remove it
114             -- from peers
115             [d'] -> do
116               let peersOfS = peers ! s
117               foldM eliminate newV (zip peersOfS (repeat d'))
118             -- else : return the new grid
119             _     -> return newV
120             -- Now check the places where d appears in the peers
121             -- of s
122           foldM (locate d) newV2 (units ! s)
123
124 locate :: Digit -> Grid -> Unit -> Maybe Grid
125 locate d g u = case filter ((d `elem`) . (g !)) u of
126                  []  -> Nothing
127                  [s] -> assign g (s,d)
128                  _   -> return g
129
130 -- Search
131 search :: Grid -> Maybe Grid
132 search g =
133   case [(l,(s,xs)) | (s,xs) <- assocs g,
134         let l = length xs, l /= 1] of
135     [] -> return g
136     ls -> do let (_,(s,ds)) = minimum ls
137              msum [assign g (s,d) >>= search | d <- ds]
138
139 solve :: String -> Maybe Grid
140 solve str = do
141     grd <- parsegrid str
142     search grd
143
144 -- Display solved grid
145 printGrid :: Grid -> IO ()
146 printGrid = putStrLn . gridToString
147
148 gridToString :: Grid -> String
149 gridToString g =
150   let l0 = elems g
151       -- [("1537"),("4"),...]
```

```
152         l1 = (map (\s -> "␣" ++ s ++ "␣")) l0
153         -- ["1  "," 2  ",...]
154         l2 = (map concat . sublist 3) l1
155         -- ["1  2  3 "," 4  5  6 ", ...]
156         l3 = (sublist 3) l2
157         -- [["1  2  3 "," 4  5  6 "," 7  8  9 "],...]
158         l4 = (map (concat . intersperse "|")) l3
159         -- ["1  2  3 | 4  5  6 | 7  8  9 ",...]
160         l5 = (concat . intersperse [line] . sublist 3) l4
161     in unlines l5
162         where sublist n [] = []
163               sublist n xs = ys : sublist n zs
164                 where (ys,zs) = splitAt n xs
165               line = hyphens ++ "+" ++ hyphens ++ "+" ++ hyphens
166               hyphens = replicate 9 '-'
167
168 compute :: String -> String
169 compute = show . solve . read
```

Caso não haja uma solução é retornado `Nothing` e caso haja é retornado `Just array`, em que `array` representa uma matriz de lista de caracteres e a lista de caracteres contém a solução para cada posição.

5.2 Módulo Slave

O Módulo `Slave` exporta somente uma função, que será utilizada por qualquer programa que implemente um *slave*. De modo semelhante ao módulo `Server` apresentado na Seção 4.1, este módulo implementa uma função que deverá ser chamada na função `main` do *slave*.

A função `slaveMain` exportada pelo módulo recebe como parâmetro uma lista de tuplas. Cada tupla tem tipo `(String, String -> String)`, ou seja, o primeiro elemento da tupla é um `String` e o segundo uma função do tipo `String -> String`. O primeiro elemento de cada tupla deverá conter o nome da função aceita pelo *slave* e o segundo a função que realiza a computação.

Código 13: Módulo `Slave` que necessita ser importado pelo programa que executará como *slave*.

```
1
2  module Slave
3         (
4           module Network,
5           slaveMain,
6         )
7         where
8
9  import Network
10 import System.IO
11   (hGetLine, Handle, hSetBuffering, BufferMode(..), stdout,
```

```
12      hPutStrLn, hClose)
13
14  import Server
15  import MaybeExceptions
16  import ConfigFile
17
18  computeRequisition :: [(String, String -> String)]
19    -> Handle -> HostName -> PortNumber -> IO ()
20  computeRequisition l h _ _ = do
21    hSetBuffering h LineBuffering
22
23    fn <- hGetLine h
24    param <- hGetLine h
25    maybe (sendResult h "") (\f -> sendResult h $ f param)
26      (lookup fn l)
27    hClose h
28    putStrLn "computeRequisition: Handle closed."
29
30  sendResult :: Handle -> String -> IO ()
31  sendResult h result = do
32    hPutStrLn h result
33    putStrLn "sendResult: result sent." -- verbose
34
35  -- | Unused for now.
36  compute :: (Read a, Show b) => (a -> b) -> String -> String
37  compute f = show . f . read
38
39  slaveMain :: [(String, String -> String)] -> IO ()
40  slaveMain arg = do
41    cf <- readConfigFile
42    let ($>$) :: String -> (String -> a) -> a
43        ($>$) s f = configLookup cf s f
44
45        server_port  = "serverPort"  $>$ readPort
46        updater_ip   = "updaterIP"   $>$ readIP
47        updater_port = "updaterPort" $>$ readPort
48        computations = map fst arg
49
50        notifyServer :: IO ()
51        notifyServer = do
52          hSetBuffering stdout LineBuffering
53
54          h <- connectTo updater_ip (PortNumber updater_port)
55          hSetBuffering h LineBuffering
56
57          putStrLn "Sending port information..." -- vervose
58          hPutStrLn h $ show $ show server_port
59          putStrLn "Sending list of functions..." -- verbose
```

```
60          hPutStrLn h $ show computations
61
62          hClose h
63
64    catchIO $ notifyServer
65    serverMain (computeRequisition arg)
```

Como visto na linha 65 do Código 13, a função `slaveMain` cria, a partir da lista recebida, uma função para atender às requisições e utiliza da função `serverMain` (ver Seção 4.1) para inicializar a execução do servidor *slave*.

Antes do *slave* começar a atender às requisições, observe que a computação `notifyServer` informa ao servidor quais são as computações aceitas pelo *slave* com base na lista recebida como argumento pela função `slaveMain`.

5.2.1 Obtenção da transparência de relocação

Quando utilizada a função `notifyServer` a transparência de relocação passa a fazer parte do sistema, pois sempre que um *slave* alterar sua localização, este notificará ao servidor de quais funções são computadas por ele, uma vez que é iniciada a execução nesta nova localização e consequentemente a função `notifyServer` será chamada, ou seja, o cliente não necessita de ser informado sobre qualquer mudança de localização que eventualmente venha a ocorrer com o *slave*.

5.3 Programa principal do Slave 1

Nesta seção é apresentado a implementação de um *slave* que atende à requisições de computação de algumas funções. Tem-se como objetivo com esta seção e com a seguinte mostrar os padrões para implementação dos *slaves*, uma vez que ainda não é possível realizar somente a implementação de um *slave* e atribuir as computações aceitas por ele em tempo de execução, ficando isso como trabalho futuro (ver Seção 7).

No Código 14, que implementa o primeiro dos *slaves*, a lista de tuplas mencionada na Seção 5.2 é descrita de modo a especificar que o *slave* é capaz de computar as funções `fibonacci`, `treeCharToInt` e `sudoku`.

Código 14: Módulo `Main` que implementa o primeiro programa *slave*.

```
1
2  module Main
3          (
4             main,
5          )
6          where
7
8  import Slave
9  import qualified Computations.Fibonacci as Fib
10 import qualified Computations.Tree as Tree
```

```
11 import qualified Computations.Sudoku as Sudoku
12
13 main :: IO ()
14 main = slaveMain [("fibonacci", Fib.compute),
15                   ("treeCharToInt", Tree.compute),
16                   ("sudoku", Sudoku.compute)]
```

Como foi visto na Seção 5.2, a tarefa de informar ao servidor quais são as computações aceitas já foi implementada no módulo `Slave` pela função `slaveMain`, fazendo com que o programador de um *slave* não tenha que se preocupar com isso.

5.4 Programa principal do Slave 2

O módulo que implementa o segundo *slave* é semelhante ao módulo da Seção 5.3, diferenciando somente nas funções aceitas por este *slave*, que neste caso é somente a função `fibonacci`, como visto na linha 12 do Código 15.

Código 15: Módulo `Main` que implementa o segundo programa *slave*.

```
1
2  module Main
3          (
4            main,
5          )
6          where
7
8  import Slave
9  import qualified Computations.Fibonacci as Fib
10
11 main :: IO ()
12 main = slaveMain [("fibonacci", Fib.compute)]
```

6 Utilização pelo programador

Nesta seção é apresentado de forma prática como utilizar deste trabalho para a implementação de um sistema distribuído.

Primeiramente, deve ser mencionado que os dados trafegados pela rede, sendo parâmetros de funções a serem computadas ou resposta de alguma computação realizada devem ser instanciados pelas classes `Show` e `Read`. Sendo assim, funções não poderão ser transmitidas pela rede, pois não é possível definir as funções `showsPrec` e `readsPrec` (funções pertencentes às classes `Show` e `Read`, respectivamente) sobre funções.

6.1 Implementação do programa cliente

Para a implementação de um programa cliente, é simplesmente necessário importar o módulo `Client` mostrado na Seção 3.1 e utilizar os operadores `$/` e `$\` para requisitar uma computação e obter o resultado da computação, respectivamente.

Por exemplo, para requisitar a computação de uma função `f` sobre uma lista de `Double`s, é simplesmente necessário aplicar o operador `$/`:

```
"f" $/ [1.0, 1.1, 1.2]
```

Em todos os casos, o resultado da utilização do operador `$/` é `IO (Maybe Ticket)`, pois seu tipo é `(Show a) => String -> a -> IO (Maybe Ticket)`, onde o `Maybe Ticket` obtido será posteriormente utilizado para obter o resultado da computação utilizando o operador `$\`. Um exemplo pode ser visto na linha 42 do Código 6.

O operador `$\` tem tipo `Maybe Ticket -> (String -> a) -> IO (Maybe a)`, portanto o valor do tipo `Maybe Ticket` obtido com o operador `$/` é utilizado com o operador `$\`, juntamente com uma função do tipo `String -> a` (função que lê o resultado desejado a partir de um `String`), para obter o resultado `Maybe a`, onde `a` é o resultado obtido pela computação da função de maneira distribuída.

Observe a maneira intuitiva de se programar utilizando esses operadores, pois são semelhantes ao operador `($) :: (a -> b) -> a -> b` frequentemente utilizado em programas Haskell. Com a utilização desses operadores, a sintaxe obtida ao escrever um código que será computado de maneira distribuída é bastante semelhante a sintaxe utilizada para descrever computações locais.

Portanto, para escrever um programa cliente basta que se saiba utilizar esses dois operadores, obtendo assim uma grande facilidade para programar de maneira distribuída. Uma outra questão a se verificar do lado cliente é quanto ao arquivo de configurações, referenciado neste trabalho como arquivo `ConfigFile`. A configuração do cliente é tratada na Seção 6.2.

6.2 Configuração do cliente

Como foi dito na Seção 2.3.1, cada linha do arquivo `ConfigFile` está organizada como `keyword:value`. Nesta seção é mostrado quais são os `keywords` que devem ser especifi-

cados do lado cliente para o correto funcionamento do sistema distribuído.

Uma vez que não é necessário o cliente conhecer a localização dos *slaves*, mas somente a localização do servidor que as conhece, é necessário a especificação da localização do servidor. Essa especificação é realizada com a informação do endereço do servidor e a porta por onde ele atende às requisições. No arquivo `ConfigFile`, essas duas informações são especificadas com os `keywords` `serverIP` e `serverPort`, respectivamente.

Outra configuração requerida pelo programa cliente é o tempo limite de espera pelo resultado de uma computação delegada de maneira distribuída. Essa informação pode ser incorporada no arquivo `ConfigFile` pelo `keyword` `timeout`. O tempo especificado com esse `keyword` é especificado em microsegundos, ou seja, um valor de 1000000 equivale a 1 segundo.

Veja abaixo um exemplo do conteúdo de um arquivo `ConfigFile` para configurar um cliente:

```
serverIP:127.0.0.1
serverPort:8000
timeout:60000000
```

6.3 Implementação de um slave

De acordo com a implementação deste trabalho, para se obter *slaves* diferentes (que computam conjuntos diferentes de funções), necessariamente deve-se ter programas diferentes, pois ainda não foi obtida uma maneira de carregar as funções a serem computadas em tempo de execução. Como é mostrado na Seção 7, este ponto é considerado como um ponto a ser trabalhado futuramente.

Para facilitar a tarefa de implementar programas *slaves*, durante a realização deste trabalho foi realizado um esforço para se obter uma maneira simples de realizar tais implementações. Nesta seção é descrito como um programa *slave* pode ser implementado.

Observando-se as definições da função `main` nos Códigos 14 e 15, é possível verificar a facilidade para se contruir um programa *slave*. A função `main` do programa *slave* deve simplesmente utilizar a função `slaveMain` definida no módulo `Slave` passando como parâmetro um lista de tuplas (ver Seção 5.2).

Uma vez que a função `compute :: String -> String` é definida em cada módulo que implementa uma computação passível de ser aceita pelos *slaves*, é simplesmente necessário definir a lista mencionada anteriormente com tuplas, onde a primeira posição é o `String` que identifica a computação aceita e a segunda posição é a função `compute` do módulo que implementa tal computação.

6.4 Configuração do slave

Todo *slave* ao iniciar sua execução, informa ao servidor de atualizações (ver Seção 4.3) quais são as computações aceitas por ele. Portanto, é necessário que o *slave* conheça sua

localização. Essa localização é conhecida pelo *slave* por meio de seu arquivo `ConfigFile` com a definição dos `keyword`s `updaterIP` e `updaterPort`, que definem o endereço e a porta, respectivamente, por onde o servidor de atualizações aceita conexões.

Além disso, é necessário configurar qual será a porta por onde o *slave* atenderá às requições de computação. Essa informação é especificada pelo `keyword serverPort`.

Abaixo um exemplo do conteúdo de um arquivo `ConfigFile` que configura um *slave*:

```
serverPort:8001
updaterIP:127.0.0.1
updaterPort:7999
```

6.5 Configuração dos servidores

Quanto ao servidor principal e ao servidor de atualizações, para estes não são necessárias novas implementações. As implementações obtidas com os Códigos 8 e 9 são suficientes para serem utilizadas, necessitando somente da configuração de cada servidor.

Nesta seção é tratado da configuração do servidor principal e do servidor de atualizações.

Ambos servidores necessitam de arquivos `ConfigFile` diferentes e ambos utilizam, como foi dito na Seção 2.2, o mesmo arquivo `BookLocations`.

Uma vez que as principais informações trabalhadas por esses servidores encontram-se no arquivo `BookLocations`, a especificação no arquivo `ConfigFile` se torna mínima. O único `keyword` necessário de se definir para cada um desses servidores em seus respectivos arquivos `ConfigFile` é o `serverPort`, que é o `keyword` utilizado para especificar por qual porta cada servidor atenderá às requisições. Desse modo, abaixo encontra-se um exemplo simples que mostra o conteúdo de um arquivo `ConfigFile` que pode ser utilizado por um dos dois servidores (não pelos dois, é claro, pois eles rodam no mesmo computador e portanto necessitam de portas diferentes):

```
serverPort:8000
```

7 Conclusão e trabalhos futuros

Uma vez que o principal objetivo buscado durante a realização deste trabalho foi um sistema distribuído assíncrono transparente, ou seja, que implementa o maior número possível das transparências descritas na Seção 1, nesta seção é esclarecido sobre quais foram as transparêcias conseguidas com a realização da implementação com a linguagem de programação Haskell. Além disso, nesta seção são apresentados alguns pontos a serem trabalhados futuramente sobre a implementação do sistema distribuído apresentado neste trabalho.

Das sete transparências descritas na Seção 1 (acesso, localização, migração, relocação, replicação, concorrência e falha), somente cinco delas foram obtidas com a realização deste trabalho. As transparências de migração e concorrência não foram obtidas pelos motivos mencionados mais adiante.

A transparência de migração não pôde ser obtida, pois uma vez delegada uma tarefa a um recurso (*slave*), obtém-se um `Handle` de comunicação para se obter o resultado; caso o *slave* alterar sua localização a conexão irá se perder e o resultado da computação não poderá ser obtido.

Além disso, os resultados obtidos após a computação pelos *slaves* não são salvos em disco, implicando assim a impossibilidade de se manter o resultado da computação com a movimentação do *slave*.

Quando uma tarefa é delegada a um *slave* e este altera sua localização após receber a tarefa, a computação realizada por ele será desperdiçada. No momento de se obter o resultado a tarefa poderá ser novamente delegada ao mesmo *slave*, com sua localização nova, e a computação será realizada novamente. Mesmo assim não se obtém a transparência de migração, pois a computação realizada antes da movimentação é perdida.

A transparência de concorrência não pôde ser obtida, pois em Haskell não se tem o conceito de variáveis (muito menos variáveis globais), ou seja, em Haskell os dados são persistentes [6] e portanto não se tem a ideia de processos independentes alterarem o conteúdo de uma área de memória.

Como foi dito na Seção 3.1.1, as transparências de acesso e localização foram obtidas com a implementação das funções `compute` e `getResult` do módulo `Client`, que são funções responsáveis por comunicar com o servidor e *slaves* requisitando e obtendo resultados de computações.

Na Seção 5.2.1 foi mostrado que a transparência de relocação foi obtida, pois a movimentação dos *slaves* é possível, uma vez que a execução do *slave* em uma nova localização fará com que o servidor seja informado de sua localização e a partir daí novas tarefas poderão ser delegadas a ele.

Qualquer *slave* replicado será informado ao servidor. Na Seção 4.3.1 é mostrado como a transparência de replicação foi obtida, fazendo com que várias instâncias de recursos possam ser utilizadas, aumentando assim a confiança do sistema como um todo.

A transparência à falha foi obtida com base no módulo `MaybeExceptions`, descrito na Seção 2.1. Qualquer problema que venha a ocorrer com algum *slave* ou na comunicação

com os *slaves*, a exceção é capturada e novas tentativas de computação são realizadas (ver Seção 3.1.2). Mesmo em casos extremos, onde há problemas com todos os *slaves* que computam a função desejada, o programa cliente não está comprometido, pois ao obter o resultado, este é obtido na mônada `Maybe`, fazendo com que o usuário possa verificar se foi possível obter o resultado.

Com a implementação deste sistema distribuído, também foi possível obter um sistema assíncrono. Como mostrado na Seção 3.2.1, isso foi facilmente obtido por conta das características da própria linguagem de programação.

Este trabalho apresenta um série de limitações e poderá ser melhorado em trabalhos futuros nos seguintes pontos:

- Encontrar uma maneira de carregar a função a ser computada pelo *slave* em tempo de execução: Pela maneira que foi implementado, é estritamente necessário que cada *slave* com um conjunto diferente de funções aceitas seja um programa diferente e que necessariamente um novo *slave* seja novo programa compilado. Uma maneira de trabalhar em cima disso é encontrando, em Haskell, uma maneira de se fazer o carregamento de uma biblioteca em tempo de execução e obter a possibilidade de utilizar suas funções.

- Uma vez conseguido o item acima, implementar um mecanismo mais dinâmico de informar ao servidor as funções aceitas pelos *slaves*: Com a implementação atual, o servidor somente é notificado de quais funções são computadas por quais *slaves* quando estes iniciam suas execuções, mas esse funcionamento não é satisfatório no caso de se conseguir a implementação do item anterior.

- Implementar a possibilidade de aceitar mais um parâmetro de argumento pelas funções computadas de maneira distribuída: Uma vez que isso for obtido, a utilização do operador `$/` não será mais possível (operadores são sempre binários), a menos que a maneira de se especificar os parâmetros seja por meio de lista polimórfica (talvez implementada com uso de GADTs [5]).

- Procurar uma maneira de não trabalhar com o arquivo `BookLocations`: Toda requisição realizada ao servidor principal faz com que este abra o arquivo `BookLocations` para utilizar das informações lá contidas. Dependendo da aplicação isso pode ser um problema por conta dos repetidos acessos ao disco. Uma maneira de trabalhar em cima disso é encontrando uma maneira de persistir as informações de localização de tempo em tempo, fazendo com que o acesso ao disco seja menos frequente. Isso é um problema em Haskell por conta da não existência de variáveis globais.

- Adicionar mais tentativas de se conectar ao servidor principal: A função `compute` do módulo `Client` tenta somente um vez realizar a comunicação com o servidor principal para obter as localizações dos *slaves*. Caso o servidor não esteja no ar, o resultado já não poderá ser obtido. É possível que em algumas aplicações seja necessário mais de uma tentativa para se conectar a esse servidor.

- Implementar política de decisão entre *slaves* informados: O servidor principal informa todos os *slaves* capazes de computar a função requerida. Pode ser implementada um política de decisão entre os *slaves* informados de modo a decidir entre os

slaves com maior probabilidade de realizarem a tarefa com sucesso e de realizarem mais rapidamente. Para isso deve ser implementado maneiras de armazer dados estatísticos obtidos ao longo da execução do sistema distribuído.

- Implementar salvamento de dados: Implementar o salvamento dos resultados obtidos com a computação das funções, de modo a não ser necessário realizar uma computação caso ela já tenha sido realizada.

- Utilizar o tipo `Either` ao invés do tipo `Maybe`: Com a utilização do tipo `Either` a transparência à falhas poderá ser mantida e no caso de impossibilidade de realizar a computação, o usuário poderá receber a informação do motivo que impossibilitou a computação ser realizada.

- Testes mais elaborados: Para verificação da possibilidade de uso deste sistema distribuído em certas aplicações, faz-se necessário a realização de testes mais elaborados que levem em conta a capacidade do sistema de se manter, por exemplo quando há excesso de requisições.

Referências

[1] ASCII Table. http://www.asciitable.com/, 2011.

[2] Distributed System code. https://github.com/pedrormjunior/A-Distributed-System-Implemented-in-Haskell, 2011.

[3] Jean Dollimore, Tim Kindberg, and George Coulouris. *Distributed Systems: Concepts and Design*. Addison Wesley, 4th edition, 2005.

[4] Fibonacci number. http://en.wikipedia.org/wiki/Fibonacci_number, 2011.

[5] GADTs for dummies. http://www.haskell.org/haskellwiki/GADTs_for_dummies, 2011.

[6] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1st edition, 1999.

[7] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, 1st edition, 2008.

[8] Solving Every Sudoku Puzzle. http://norvig.com/sudoku.html, 2011.