Programación Funcional

Lucas Di Cunzolo

Febrero 2022

1 Introducción

En este trabajo se plantea la implementación parcial del protocolo DNS propuesto por la RFC 1035 [1]

2 Motivación

La motivación radica principalmente en querer aplicar los conceptos aprendidos en la materia en un proyecto real. La elección del protocolo DNS viene de la intriga de conocer un poco más en detalle la implementación del protocolo.

3 Alcance

El alcance de la implementación está acotado a los tipos de consultas más usados entre los que se encuentran *A*, *AAAA*, *MX*.

Además, el servidor DNS no aplica el uso de TTLs ni respuestas recursivas.

4 Modelo de datos

El proyecto se inició modelando un mensaje DNS tal cual se define en la RFC, en el que se pueden distinguir 4 tipos bien definidos, el encabezado del mensaje

```
data DNSHeader = DNSHeader {
  identifier            :: Word16,
  qr                    :: Bool,
  opcode                :: OPCODE,
  authoritativeAnswer   :: Bool,
  truncatedMessage      :: Bool,
  recursionDesired      :: Bool,
  recursionAvailable    :: Bool,
  z                     :: Bool,
  rccode                :: RCODE,
  qdcount               :: Word16,
  ancount               :: Word16,
  nscount               :: Word16,
  arcount               :: Word16
} deriving (Show, Eq)
```

Listing 1: data DNSHeader - *src/FDNS/Types.hs:81*

Dentro del encabezado, se pueden reconocer 2 tipos definidos que modelan el tipo de operación (OPCODE) y el tipo de respuesta (RCODE).

El segundo tipo de dato definido es la pregunta

```haskell
data DNSQuestion = DNSQuestion {
  qname                 :: String,
  qtype                 :: QTYPE,
  qclass                :: QCLASS
} deriving (Show, Eq)
```

Listing 2: data DNSQuestion - *src/FDNS/Types.hs:100*

En este caso, se cuenta con 2 tipos de datos que representan el tipo de registro DNS (QTYPE) y su clase (QCLASS)

En tercer lugar, se modeló el concepto de recurso, que aplica tanto para la sección de respuesta, de respuestas autoritativas y respuestas adicionales.

```haskell
data DNSResource = DNSResource {
  rname                 :: String,
  rtype                 :: QTYPE,
  rclass                :: QCLASS,
  ttl                   :: Word32,
  rdlength              :: Word16,
  rdata                 :: String
} deriving (Show, Eq)
```

Listing 3: data DNSQuestion - *src/FDNS/Types.hs:106*

Por último, todos estos tipos de datos se combinan en el tipo de dato que modela el mensaje DNS como tal

```haskell
data DNSMessage = DNSMessage {
  header                :: DNSHeader,
  question              :: [DNSQuestion],
  answer                :: [DNSResource],
  authority             :: [DNSResource],
  additional            :: [DNSResource]
} deriving (Show, Eq)
```

Listing 4: data DNSQuestion - *src/FDNS/Types.hs:115*

5 Conceptos de la materia

Para el desarrollo de este proyecto, se utilizaron conceptos aprendidos en la materia tales como

5.1 Recursión estructural

Utilizada para el desarrollo de las funciones de desempaquetado, ya que es necesaria una forma de recorrer la trama de bytes, acotando en cada iteración el conjunto de bytes. En este caso, la condición de corte está dada por la información provista por el encabezado.

Luego, se utilizan varias de las funciones que hacen uso del concpeto tales como *map* o *foldl*

5.2 Funciones de alto orden

Aplicadas ampliamente a lo largo del proyecto, me permitió desacoplar mucho más la lógica en varios puntos, para desarrollar funciones genéricas que permitan aplicarse en varios casos. Uno de los casos más claros son las funciones *packList* y *unpackList* usadas para trabajar con todos los recursos representados como listas del mensaje DNS (preguntas, respuestas, respuestas autoritativas y respuestas adicionales)

5.3 Funciones currificadas

Como sabemos, en haskell todas las funciones son currificadas, esto nos permite escribir código compacto y potente. A lo largo de trabajo se intentó desarrollar un código reutilizable.

5.4 Pattern matching

Utilizado ampliamente en la definición de las funciones presentadas como utilidades para interactuar con los tipos algebraicos definidos.

5.5 Mónadas

En el desarrollo del proyecto se usaron varias mónadas, de las que se pueden desatacar

- IO, usada en el programa principal, y las principales funciones de top-level, como el servidor UDP, o las funciones para cargar la configuración desde el disco. Permtie agregar todas las interacciones con el mundo exterior.
- Maybe, usada para tratar con funciones que naturalmente serían funciones parciales por la naturaleza de los datos, sobre todo cuando hablamos de funciones propias del protocolo que reciben datos desempaquetados de una trama de bytes.
- Either, usada de forma similar a la mónada Maybe, pero en este caso, me permite agregar un poco más de información. En particular, se utilizó para trabajar con los errores posibles del mensaje DNS, que luego terminan impactando en el código de respuesta del mensaje.

6 Bibliotecas y módulos usados

A continuación se van a describir las principales bibliotecas y módulos que se utilizaron en el desarrollo de este proyecto.

6.1 network [2]

Para implementar el servidor UDP se utilizaron los módulos *Network.Socket* y *Network.Socket.ByteString* pertenecientes al paquete *network*

Esta biblioteca se desarrollo una función de top-level que interactúa con los datos recibidos por el cliente a través de un socket UDP.

6.2 GetOpt [3]

La interfaz definida para este programa es a través de líneas de comandos (CLI), para desarrollarla, se utilizó el módulo *System.Console.GetOpt*, perteneciente al paquete *base* de haskell.

Para esto, se definió un tipo *Option* el cual representa todas las opciones posibles.

```
data Options = Options {
  optConfig       :: FilePath,
  optBindAddress  :: String,
  optPort         :: String,
  optHelp         :: Bool
} deriving Show
```

Listing 5: data Options - *src/FDNS/Commands.hs:14*

A su vez, se definió una función que declara la descripción de las opciones (*OptDescr*). Esta biblioteca permitió armar una interfaz completa, que ya contempla el parseo de los argumentos, simplemente pasandole los argumentos que se recuperan directamente usando la función *getArgs*.

```
data Options = Options {
  optConfig       :: FilePath,
  optBindAddress  :: String,
  optPort         :: String,
  optHelp         :: Bool
} deriving Show
```

Listing 6: Ejemplo de uso

6.3 Yaml [4]

El modo de configuración de los registros DNS del servidor se quiso mantener simple, por lo que se definió un archivo de configuración básico en formato YAML, descrita en la siguiente sección.

6.3.1 Definición

```
# Listado de dominios
domains:
  # Dominio en cuestion
  - name: .example.com
    # Listado de registros del dominio
    records:
        # Tipo del dominio
        # Puede ser A, AAAA, MX o TXT
      - type: A
        # Valor del registro
        # El formato depende del tipo de registro
        value: 1.2.3.4
```

Listing 7: Ejemplo de configuración

Teniendo esto definido, ahora podemos llamar a nuestro servidor de la siguiente forma

```
./fdns --config ./example-config.yaml
```

Listing 8: Ejemplo de llamado usando la CLI y un archivo YAML

6.3.2 Desarrollo

Para poder parsear los archivos yaml, se crearon los tipos de datos que representan cada uno de los objetos del yaml planteados

```
data Record = Record {
  recordType     :: String,
  value          :: String
} deriving (Eq, Show)

data Domain = Domain {
  name           :: String,
  records        :: [Record]
} deriving (Eq, Show)

data Config = Config {
  domains :: [Domain]
} deriving (Eq, Show)
```

Listing 9: Tipos de datos YAML

Junto a eso, a cada uno de estos nuevos tipos, se le definieron las funciones *parseJSON* de la clase *FromJSON*

```
instance FromJSON Record where
  parseJSON (Y.Object v) =
    Record <$>
    v .:   "type"        <*>
    v .:   "value"
  parseJSON k = fail ("Expected Object for Config value: " ++ show k)

instance FromJSON Domain where
  parseJSON (Y.Object v) =
    Domain <$>
    v .:   "name"        <*>
    v .:   "records"
  parseJSON k = fail ("Expected Object for Config value: " ++ show k)

instance FromJSON Config where
  parseJSON (Y.Object v) =
    Config <$>
    v .:   "domains"
  parseJSON _ = fail "Expected Object for Config value"
```

Listing 10: Funciones parseJSON

Por último, se desarrolló una función que, usando la función *decodeFileEither*, se encarga de leer el archivo de configuración en formato yaml, y retornar, un *Config* usando la mónada IO

```
readConfig :: String -> IO Config
readConfig configFilePath =
  either (error . show) id <$>
  (decodeFileEither configFilePath)
```

Listing 11: Funcion readConfig

6.4 co-log [5]

Para mejorar la visibilidad del funcionamiento del servidor, se plantearon una serie de logs, usando la biblioteca co-log. La ingtegración fue bastante simple usando el logger ya provisto por el módulo *Colog*

```
let logger = logStringStdout
logger <& ("Load config file from \"" ++ configFile ++ "\"")
```

Listing 12: Ejemplo de uso de co-log

6.5 Hspec [6]

Para facilitar el desarrollo de nuevas funcionalidades, se estableció un flujo de TDD, en donde toda nueva funcionalidad que se quiera introducir, primero tiene que ser planteada en formato de test.

En este caso, se aplicó test sobre las funciones principales de empaquetado y desempaquetado, ya que es simple plantear una correspondencia entre un conjunto de bytes en formato ByteString, con el tipo *DNSMessage*.

Para simplificar la configuración de la librería, se utilizó una estructura de directorios estándar, en donde, cada módulo de nuestro código se corresponde con un módulo de spec. El mapeo se puede encontrar en el archivo de inicio de los tests *tests/Spec.hs*

```
{-# OPTIONS_GHC -F -pgmF hspec-discover #-}
```

Listing 13: Archivo tests/Spec.hs

Además, se agregó el apartado *test-suite* en cabal.

```
test-suite spec
  type:                 exitcode-stdio-1.0

  -- .hs or .lhs file containing the Main module.
  main-is:              Spec.hs

  other-modules:        FDNS.Parsers.PackSpec
                      , FDNS.Parsers.UnpackSpec
                      , FDNS.UtilsSpec

  -- Directories containing source files.
  hs-source-dirs:       tests

  -- Other library packages from which modules are imported.
  build-depends:        base >=4.10.1.0 && <4.15
                      , bytestring
                      , fdns
                      , hspec

  -- Base language which the package is written in.
  default-language:     Haskell2010

  build-tool-depends: hspec-discover:hspec-discover == 2.*
```

Listing 14: Apartado test-suite del archivo fdns.cabal

Esto nos permite ejecutar el conjunto de tests de la siguiente forma

```
cabal test all
```

Listing 15: Ejemplo de ayuda

7 Interfaz

La interfaz se planteó de forma que sea simple empezar a probar el servidor DNS, contando con un archivo de configuración como el planteado en el apartado de YAML.

```
$ fdns --help
Usage: fdns -c FILE [-H HOST] [-p PORT]
  -h, -?   --help          Show this help message
  -c FILE  --config=FILE   Config file
  -H HOST  --host=HOST     Address to bind
  -p PORT  --port=PORT     Port to bind
```

Listing 16: Ejemplo de ayuda

8 Desarrollo del proyecto

El proyecto se planteó separando completamente la lógica del protocolo DNS, de la interacción con el cliente (socket UDP) y el usuario (CLI).

Para esto, se cuenta con una función *main* como punto de entrada, en donde se parsean los argumentos usando *GetOpt*, y dependiendo los argumentos enviados, se muestra una ayuda, o se inicia el servidor UDP.

El servidor UDP toma todas sus configuraciones necesarias de los argumentos al programa, e inicia un socket a la espera de datos.

Una vez recuperados los datos, en formato ByteString, se llama a la función de desempaquetado.

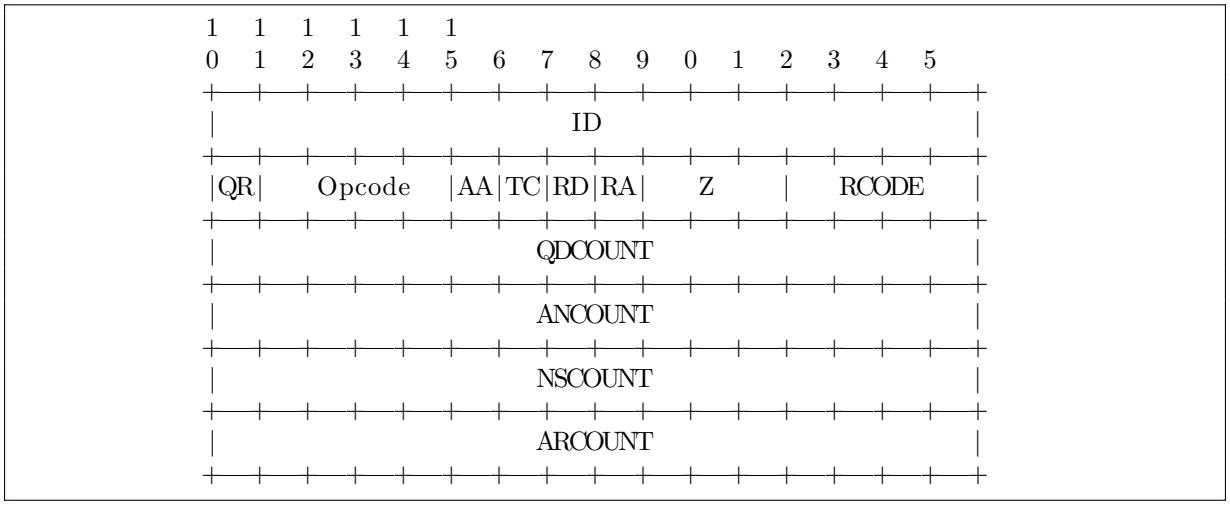
9 Desarrollo del protocolo

Las funciones referentes al protocolo se pueden diferenciar en 2 grandes gruopos, las de desempaquetado, que toman una trama de bytes, y retornan uno de los tipos definidos del modelo de datos. En su contraparte, tenemos las funciones de empaquetado, que reciben tipos del modelo de datos, y retornan su representación en bytes.

9.1 Desempaquetado

El primer paso es identificar y desempaquetar los primeros 12 bytes, que representan el encabezado del mensaje. En el mismo vamos a tener toda la información necesaria para poder trabajar con el resto del mensaje.

El encabezado tiene el siguiente formato



Listing 17: Ejemplo de ayuda

Esto se traduce de la siguiente forma

- Los primeros 2 bytes se usan para representar el ID
- El tercer byte se usa para representar 4 campos
  - El primer bit representa si es una query (0) o una respeusta (1)
  - Los siguientes 4 bits, representan el código de operación
  - El sexto bit se utiliza solamente en las respeustas, y represnta si es una respeusta autoritativa.
  - El septimo bit se utiliza para marcar que el mensaje va a estar truncado por el tamaño máximo del canal.
- El cuarto byte se usa para representar otros 4 campos
  - El primer bit marca si se quiere realizar una pregunta recursvia.
  - El segundo bit, es modificado por la respuesta, y marca si el servidor sopoerta respuestas recursivas.
  - Los siguientes 3 bits, estan reservados para futuras implementaciones
  - Los siguientes 4 bits representan el tipo de respuesta
- El quinto y sexto byte representan la cantidad de preguntas que contiene el mensaje.
- El septimo y octavo byte representan la cantidad de respuestas que contiene el mensaje.
- El noveno y décimo byte representan la cantidad de respuestas autoritativas que contiene el mensaje.
- El onceavo y doceavo byte representan la cantidad de respuestas adicionales que contiene el mensaje.

Esto se traduce en una función destinada a desempaquetar el encabezado, que recibe los bytes en el formato *ByteString*, y retorna un *DNSHeader*

En este caso, separamos cada uno de los bytes usando una función definida en el módulo *FDNS.Parsers.Internal.Utils*, la cual añade una capa de control usando la mónada *Maybe*.

Una vez separados los bytes, se empieza a construir el encabezado recuperando los valores con *fromMaybe*, tomando por defecto el valor nulo de los campos. Esto permite armar siempre un encabezado bien formado para retornar, aunque la trama de bytes sea incorrecta, marcando el error en el campo *rcode* del ebncabezado.

El siguiente paso, y haciendo uso de los datos recuperados anteriormente, es desempaquetar tantas preguntas como el valor de *qdcount* del encabezado. Para esto, se definió una función de alto orden que hace uso de recursión estructural para recorrer la trama de bytes e ir desempaquetando aplicando la función que recibe como primer parámetro

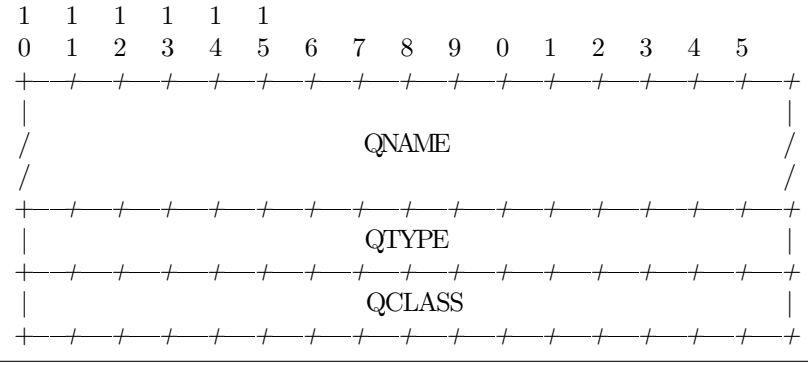
```
unpackList :: (BS.ByteString -> (Either DNSError a, BS.ByteString))
              -> Word16
              -> BS.ByteString
              -> [Either DNSError a]
unpackList unpack 0 bytes = []
unpackList unpack n bytes = let (resource, bytes') = unpack bytes in
                            resource : unpackList unpack (n-1) bytes'

unpackQuestions :: Word16 -> BS.ByteString -> [Either DNSError DNSQuestion]
unpackQuestions = unpackList unpackQuestion
```

Listing 18: Funcion unpackList y unpackQuestions

En este caso, vamos a recorrer la trama de bytes en cada llamado de la recursión, usando la función *unpackQuestion*, sobre una subtrama de bytes diferente, y con la condición de corte dada por que la cantidad de preguntas a desempaquetar llegue a 0.

Esta función *unpackQuestion* se aplica para desempaquetar las preguntas, que tienen el siguiente formato



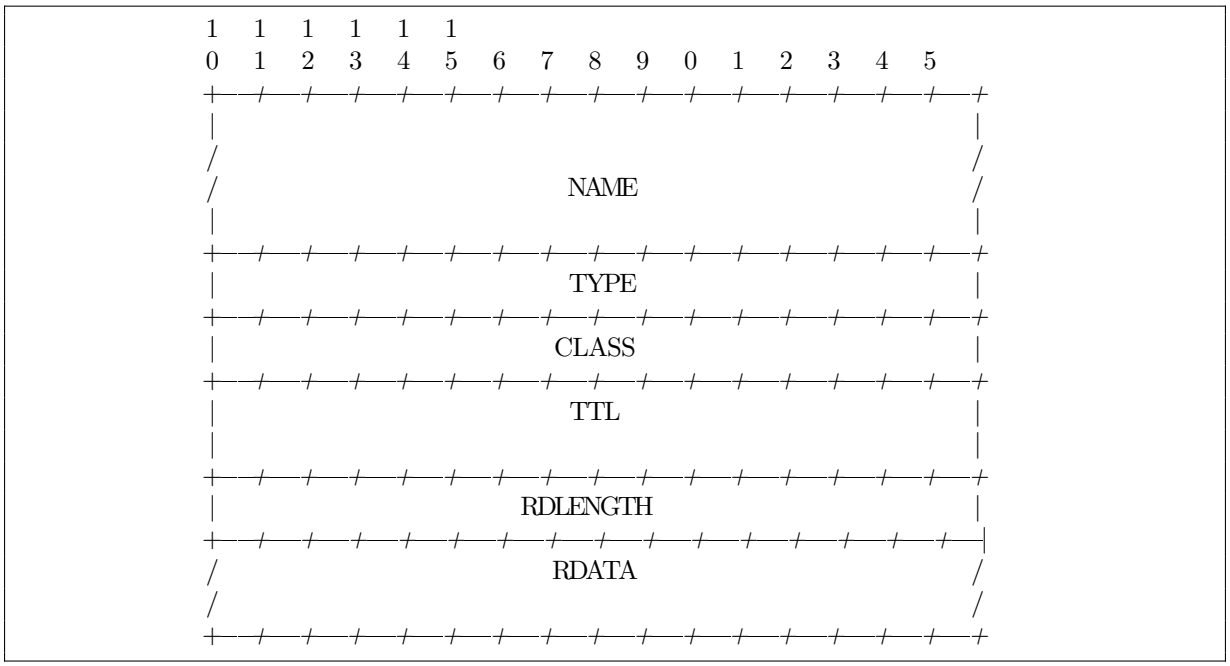
Listing 19: Formato preguntas DNS

Por último, y de forma similar al proceso anterior, se tienen que desempaquetar las respuestas, respuestas autoritativas y respuestas adicionales que, recordando la definición del protocolo, comparten su definición.

```
unpackResources :: Word16 -> BS.ByteString -> [Either DNSError DNSResource]
unpackResources = unpackList unpackResource
```

Listing 20: Función unpackResources

En este caso, usamos la función *unpackResource*, aplicada con una trama de bytes que tiene que repretar el siguiente formato



Listing 21: Formato preguntas DNS

9.2 Trabajando con el tipo DNSMessage

Una vez analizado el mensaje DNS, se tiene que recuperar, de existir, el registro que satisfaga la pregunta. Esto lo vamos a hacer usando la función *lookup*

```
lookup :: Config -> String -> String -> [Record]
lookup config name' recordType' =
  case find (\d -> name d == name') (domains config) of
    (Just domain) -> filter
                       (\r -> recordType r == recordType')
                       (records domain)
    Nothing       -> []
```

Listing 22: Ejemplo de ayuda

Con esto vamos a tener una lista de *Record*, que es fácilmente transformable a una lista de *DNSResource* usando un map y una función auxiliar que dado un *Record* retorna un *DNSResource*

Por último, para agregarlo, se tiene un conjunto de funciones que reciben un *DNSMessage* y retornan un nuevo *DNSMessage* modificado con ese agregado

```
-- Agregar una lista de preguntas al mensaje
(<<?) :: DNSMessage -> [DNSQuestion] -> DNSMessage
-- Agregar una lista de respuestas al mensaje
(<<!) :: DNSMessage -> [DNSResource] -> DNSMessage
-- Agregar una lista de respuestas autoritativas al mensaje
(<<@) :: DNSMessage -> [DNSResource] -> DNSMessage
-- Agregar una lista de preguntas adicionales al mensaje
(<<+) :: DNSMessage -> [DNSResource] -> DNSMessage
```

Listing 23: Ejemplo de ayuda

Para simplificar la construcción del nuevo mensaje DNS, se cuenta con la función *dnsResolver*, la cual ya se encarga de recuperar los registros para las preguntas dadas, y retorna un nuevo mensaje DNS con las respuetas ya agregadas.

```
dnsResolver :: Config -> DNSMessage -> DNSMessage
dnsResolver config message = message <<! resources
  where questions = question message
        lookupConfig = FDNS.Config.lookup config
        resources = concat $ lookupConfig <$> questions
```

Listing 24: Ejemplo de ayuda

9.3 Empaquetado

Para poder retornar la respuesta al cliente, es necesario empaquetar el mensaje en una trama de bytes. Para ello, tenemos funciones similares a las vistas en el desempaquetado.

Contamos con una función *packHeader* que recibe un *DNSQuestion*, y retorna un ByteString.

Luego contamos con una función genérica para empaquetar listas, la cual es usada para empaquetar tanto las preguntas (*packQuestions*), como para empaquetar el resto de los recursos (*packResoureces*)

```
packList :: (a -> BS.ByteString) -> [a] -> BS.ByteString
packList pack = foldl
                (\bytes resource -> BS.append bytes (pack resource))
                BS.empty

packQuestions :: [DNSQuestion] -> BS.ByteString
packQuestions = packList packQuestion

packResoureces :: [DNSResource] -> BS.ByteString
packResoureces = packList packResourece
```

Listing 25: Funcion unpackList y unpackQuestions

10 Conclusiones

El planteamiento del modelo de datos fue muy simple a nivel conceptual, y el lenguaje ayudó en gran medida a bajar rapidamente varias ideas para el desarrollo. Realmente me ubiera gustado contar con más tiempo para poder llegar a plantear un menejo de errores más completo en cuanto al protocolo DNS, como así para plantear una mirada más completa del mismo, aunque creo que sirvió como una primera introducción para empezar a entender como pueden plantearse proyectos con una complejidad media, teniendo una separación fuerte de la lógica funcional, de la interacción con el mundo exterior.

10.1 Para mejorar

Como se nombró anteriormente, un manejo de los errores más completo. También creo que la lógica de desempaquetado de un mensaje se puede mejorar para llevarlo a un plano más simple, sobre todo en el manejo de los corrmientos entre secciones del mensaje.

Otro detalle a tener en cuenta sería introducir alguna estructura de datos que permita trabajar de forma simple, e implementar el concepto de compresión de los mensajes, actualmente, los mensajes generados por este servidor DNS van a ser relativamente más pesados ya que en cada aparición de un dominio, se transforma directamente a su representación en bytes. Se había hablado de introducir los Zipper [7].

Referencias

[1] P. Mockapetris. Domain names - implementation and specification, 1987. URL https://datatracker.ietf.org/doc/html/rfc1035.

[2] libraries@haskell.org. network, 2001. URL https://hackage.haskell.org/package/network.

[3] libraries@haskell.org. System.console.getopt, 2002. URL https://hackage.haskell.org/package/base-4.16.0.0/docs/System-Console-GetOpt.html.

[4] Kirill Simonov Michael Snoyman, Anton Ageev. yaml, 2009. URL https://hackage.haskell.org/package/yaml.

[5] snoyberg. co-log, 2018. URL https://hackage.haskell.org/package/co-log.

[6] Simon Hengel. hspec, 2011. URL https://hackage.haskell.org/package/hspec.

[7] Zipper. URL https://wiki.haskell.org/Zipper.