**Universidade do Minho**
Escola de Engenharia
Licenciatura em Engenharia Informática

# Unidade Curricular de Comunicação por Computadores

Ano Letivo de 2022/2023

# Grupo 3.02

Carlos Gustavo Silva Pereira (a96867) Cláudio Alexandre Freitas Bessa (a97063) João Miguel Ferreira Loureiro (a97257)

January 5, 2023

# Indice

# 1 Introdução

O presente documento, relata a nossa abordagem ao desenvolvimento de componentes de *software* que integram um sistema DNS, no âmbito da Unidade Curricular de Comunicação por Computadores. Para melhor o fazer, é necessária uma melhor compreensão sobre o que é, para que serve e como funciona um sistema DNS.

Como seres humanos, somos bastante bons a memorizar e usar nomes para identificar o que quer que seja, no entanto, a utilização dos protocolos TCP/IP exige que os computadores sejam identificados a partir de endereços numéricos. Dada a nossa dificuldade em memorizar diversos endereços númericos houve necessidade de criar um sistema lógico prático que auxiliasse o acesso a recursos espalhados pela rede. A solução passa por associar um nome, que nos é mais fácil de memorizar sendo pronunciável, a um respetivo endereço numérico. Para pôr em prática este sistema lógico foi necessário criar aquilo que se veio a chamar de *Domain Name System (DNS)*, um sistema que mapeia os endereços numéricos dos computadores ao seu respetivo nome lógico. Este sistema é composto por diversos componentes, cada um com a sua função, desde o registo dos nomes dos computadores, passando pelo registo das árvores de domínios, até aos componentes cuja função é extrair estas informações desses registos. O **[RFC 1034]**[1] introduz o DNS, explicando de forma mais aprofundada as funções destes três elementos.

A nossa implementação deste sistema foca-se, em três tipos de servidores que integram o sistema DNS e suportam os domínios da rede, dando resposta a *queries* de DNS com base nos seus registos. Na prática, estes servidores serão uma só componente de *software* que implementa as bases de dados, os que tem acesso à mesma, para cada instância do servidor e correm continuamente o serviço de resposta às *queries*.

Existe uma clara influência a nível de implementação através de bibliotecas já existentes em diferentes linguagens de programação, contudo optamos por não utilizar nenhuma dessas e apenas recorrer a bibliotecas que fazem parte dessa biblioteca geral. Esta biblioteca geral em específica denominada em *python* por *dnslib*[2] onde seguimos de perto um exemplo bastante simples e concreto consoante o solicitado.

1. *"Domain Name System"*, https://www.rfc-editor.org/rfc/rfc4180.txt, Accessed: 2022-11-18.
2. *"Simple DNS server (UDP and TCP) in Python using dnslib.py"*, https://gist.github.com/pklaus/b5a7876d4d2cf7271873, Accessed: 2022-11-5.

# 2 Arquitetura do Sistema

## 2.1 Topologia

Para diferentes operações nas várias camadas de redes são necessários diferentes protocolos. Com isto, há todo um conjunto de métodos com base nesses protocolos, quer ao nível de *hardware* quer a nível de *software*. Esses tipos de protocolos podem efetuar ligações físicas ou lógicas que compõe uma topologia, onde é explicado como a rede se torna operacional. Pode-se dizer então que uma topologia é uma representação da organização de uma rede, retratando a forma como os componentes da mesma estão distribuídos e se interligam.

Neste trabalho prático, é nos solicitada a criação de uma topologia que funcionará como ambiente de testes dos componentes de *software* desenvolvidos. No contexto do trabalho, é imperativo que esta apresente: um domínio base (*root domain*, representando por ".") auxiliado por dois servidores, um domínio de topo *.reverse*, dois domínios de topo, cada um com um servidor primário e dois secundários, e um subdomínio para cada domínio de topo, também suportados com um servidor primário e dois secundários. Esta hierarquia de domínios pode ser representada como se observa na 2.1, onde ".ccm", ".bgj" e "reverse" representam os domínios de topo anteriormente mencionados; "barrete" e "lambreta", os respetivos subdomínios. De igual modo foi acrescentado nesta fase final, um servidor de resolução para poder receber as *queries* em primeira mão. Nestes moldes, construiu-se a topologia recorrendo ao *CORE*, visível em 2.2.

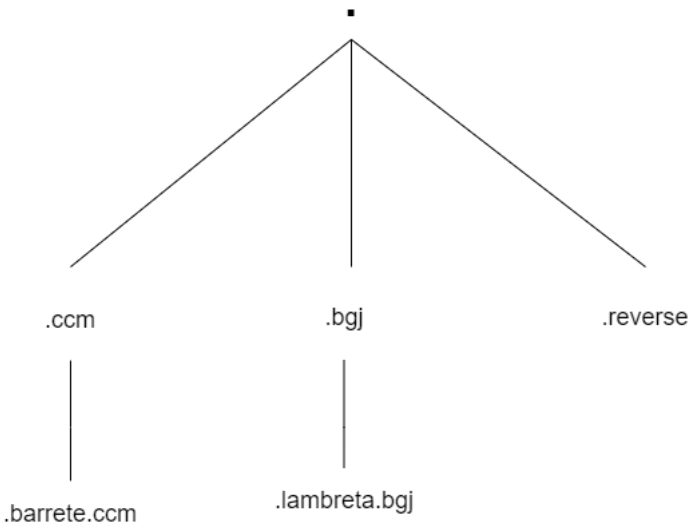


Figure 2.1: Hierarquia de domínios da topologia

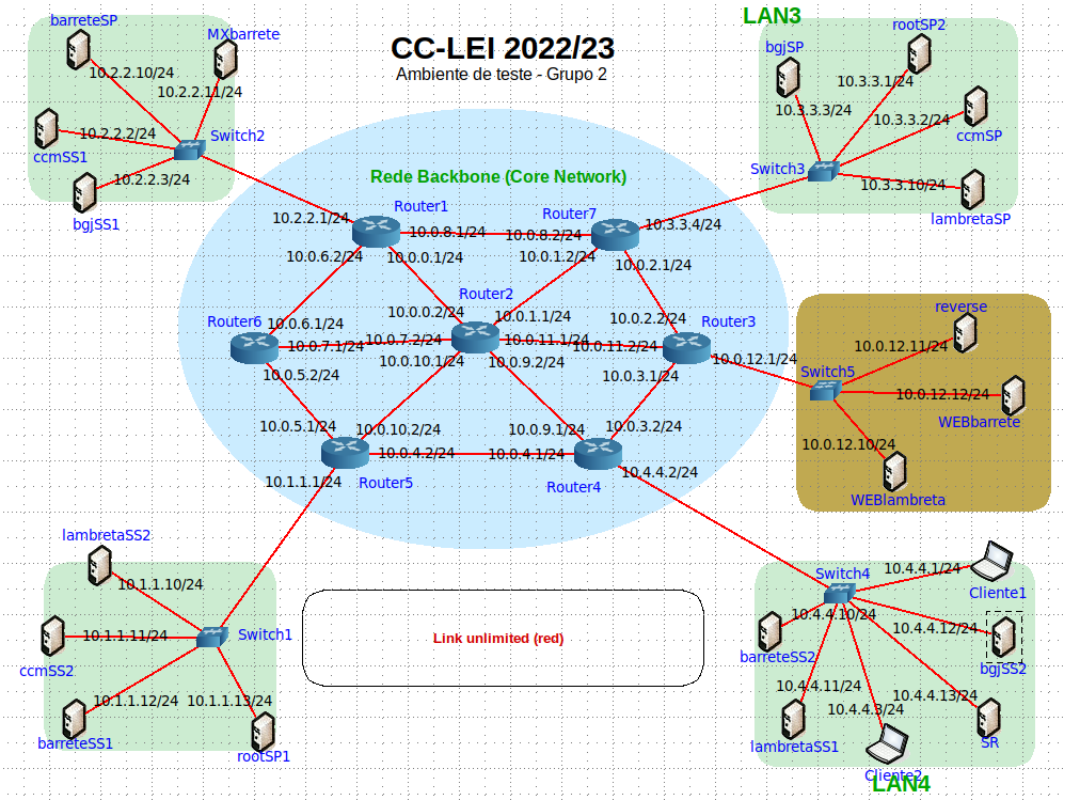


Figure 2.2: Ambiente de Teste - CORE

## 2.2 Requisitos funcionais

Tendo em conta o enunciado do trabalho, determinamos alguns dos requisitos para cada um dos componentes deste projeto.

### 2.2.1 Servidor Principal

- Reponde e efetua *queries* DNS
- Tem acesso direto à base de dados dum domínio DNS
- Atualiza informações sobre um domínio na base de dados
- Recebe como *input* de um fichero de configuração, ficheiro de base de dados e ficheiro indicador dos servidores de topo que este irá gerir
- Devolve um ficheiro de *log* como *output*
- Permite ser inicializado em modo *debug* imprimindo as *logs* na consola em tempo real

**Nota:** Nesta primeira fase, o Servidor Principal tem apenas como *input* o caminho para o ficheiro de configuração e outro para o ficheiro de base de dados, ainda não implementado os servidores de topo.

```
# Configuration file for primary server for barrete.ccm
barrete.ccm DB /home/core/CC32/db/barrete.txt

barrete.ccm SS 10.1.1.12:5353
barrete.ccm SS 10.4.4.10:5353

barrete.ccm LG logs/barrete.log
all LG logs/all.log
```

Figure 2.3: Exemplo de um ficheiro de configuração de um Servidor Primário

```
# DNS database file for domain barrete.ccm
# It also includes a pointer to the primary server

@ DEFAULT barrete.ccm.
TTL DEFAULT 86400

@ SOASP ns1.barrete.ccm. TTL
@ SOAADMIN admin.barrete.ccm. TTL
@ SOASERIAL 0114692022 TTL
@ SOAREFRESH 14400 TTL
@ SOARETRY 3600 TTL
@ SOAEXPIRE 604800 TTL

@ NS ns1.barrete.ccm. TTL
@ NS ns2.barrete.ccm. TTL
@ NS ns3.barrete.ccm. TTL

@ MX mx1.barrete.ccm. TTL 10

ns1 A 10.2.2.10 TTL
ns2 A 10.1.1.12 TTL
ns3 A 10.4.4.10 TTL
mx1 A 10.2.2.11 TTL

sp CNAME ns1 TTL
ss1 CNAME ns2 TTL
ss2 CNAME ns3 TTL
mail1 CNAME mx1 TTL
```

Figure 2.4: Exemplo de um ficheiro de configuração de uma Base de Dados

### 2.2.2 Servidor Secundário

- Reponde e efetua *queries* DNS
- Tem acesso a uma cópia da base de dados dum domínio DNS
- Atualiza a sua cópia da base de dados periodicamente
- Recebe como *input* de um fichero de configuração e ficheiro de servidores de topo
- Devolve um ficheiro de *log* como *output*

**Nota:** Nesta primeira fase, o Servidor Secundário tem somente como input o caminho para o ficheiro de configuração, ainda não implementado os servidores de topo.

```
# Configuration file for secondary server for barrete.palha
barrete.palha SP 10.2.2.10:5353

barrete.palha LG logs/barrete.log
all LG logs/all.log

root ST /home/core/CC32/db/rootservers.db
```

Figure 2.5: Exemplo de um ficheiro de configuração de um Servidor Secundário

### 2.2.3 Servidor Resolução

- Realiza queries DNS sobre qualquer domínio como intermediário
- Implementado a nível da API da nossa aplicação
- Escolhe o modo em que faz a *query* (iterativo ou recursivo)
- Recebe como *input* um ficheiro de configuração e um ficheiro com a lista de servidores de topo
- Devolve como *output* um ficheiro de *log*

### 2.2.4 Cliente

- Realiza *queries* DNS
- Escolhe o tipo de serviço que pretende
- Escolhe o modo em que faz a *query* (iterativo ou recursivo)
- *Input* e *output* através da linha de comandos
- Define o *timeout*, tempo de espera pela resposta à *query* que realiza

Uma vez que o Servidor de Resolução já se encontra implementado, o cliente pode realizar pedidos em modo iterativo, isto é, todas as iterações são realizadas pelo cliente uma vez, esperando a resposta até que o servidor de resolução obtenha a solicitada. Se a resposta de um servidor a uma *query* for apenas um endereço para um outro domínio que tenha a informação, o próprio servidor de resolução agora, ao contrário da primeira fase, encarrega-se de reenviar para esse outro endereço.

## 2.3 Componentes

Existindo quatro grandes componentes que sustentam toda esta arquitetura, servidor principal, servidor secundário, servidor resolução e cliente, decidimos apresentar, a nível de *software*, três módulos, **sp**, **ss**, **sr** e **cliente**. Desta forma conseguimos reutilizar o código em diferentes ficheiros sendo apenas necessário aplicar restrições dependendo do tipo de servidor configurado à inicialização, explicado em maior detalhe mais à frente.

Consideramos a existência de um módulo *servidor* que podia exercer o papel de Servidor Primário e Servidor Secundário, porém, vistos os papéis distintos que estes assumem, mantivemos a ideia inicial de implementar um módulo específico para cada. Com isto em mente, e visto que de forma iterativa não seria útil implementar diferentes classes tanto para os servidores de topo, como para os servidores de domínio de topo, o servidor de resolução efetua o reencaminhamento tanto para um como para o outro, não necessitando de correr a outras classes sem ser aos servidores primários e/ou possíveis servidores secundários através de transferência de zona. Desta forma, além dos mencionados anteriormente, recorremos a um módulo *parser*, um módulo *logs*, um módulo *db* e um módulo *dns-controller*.

### 2.3.1 *Interpreter*

A existência do *interpreter* (*parser*) é de facto uma das componentes chave para o funcionamento deste sistema. Tem a função de verificar a integridade dos ficheiros de *input*, sendo eles o ficheiro de configuração do(s) servidor(es), base de dados e um documento com a lista de servidores de topo a esse domínio. De igual forma, a classe é responsável por popular a base de dados no servidor caso eles sejam diferentes dos inseridos como *default*.

### 2.3.2 *Logs*

Tem como objetivo a escrita em ficheiros de *logs* e a possível amostra desse mesmo conteúdo no terminal caso o servidor opere em modo *debug*.

**Nota:** de momento as *logs* dos servidores são todas feitas apenas para o ficheiro *all.log*.

### 2.3.3 *Base de Dados*

O módulo *Database*, além de suportar as estruturas de dados essenciais ao funcionamento dos servidores, também escreve nos ficheiros de base de dados dos servidores onde é instanciado.

### 2.3.4 *Controlador DNS*

Esta classe tem como objetivo a execução e gestão das *queries* solicitadas pelo cliente ao servidor. Uma espécie de *parser* específico para as mensagens DNS, onde numa próxima fase será responsável pela codificação e descodificação em binário.

### 2.3.5 *Cache*

Por fim a *cache* tem o objetivo de guardar a informação de um X número de *queries* aumentando a eficiência e consequentemente reduzindo o tempo de espera do cliente pela resposta à mensagem DNS enviada. Uma re-implementação de um dicionário em *python* mas com mais funcionalidades e ou capacidades.

# 3 Modelo de Informação

## 3.1 Especificações

No presente capítulo, existe uma especificação a nível semântico e sintático mais aprofundado dos módulos principais para o funcionamento das interações entre servidores e clientes. Devido a isso prosseguiremos na explicação mais específica do módulo *Interpreter* que possui todas as funções/métodos e processos auxiliares que verificam e registam as propriedas necessárias.

A classe *Parser* presente em *interpreter.py*, guarda as linhas do ficheiro dado como *input* para ler como matriz, i.e. *array* de *array*. Seguidamente irá percorrer cada linha da matriz verificando se os seus componentes são válidos, isto na função *check-line*. Este mesmo método invoca outros métodos auxiliares que ocorrem para diferentes casos, quer dependendo da variabilidade do comprimento de determinada linha, quer dependendo do ficheiro e obviamente dependente de determinado parâmetro que ocorra nessa linha. Apesar de ser um ficheiro aulixiar, é bastante denso e contém bastantes funções auxiliares, maioritariamente para verificar parâmetros, e seguir mais facilmente um raciocínio.

## 3.2 Controlo de erros

Nas comunicações TCP ao contrário das UDP já existe um controlo de erros. Ao contrário do UDP, onde após uma *request* se segue apenas uma *response* do servidor, no TCP é efetuada uma *synchronization* inicial, onde se verifica a possibilidade de efetuar a conexão, e uma *synchronization acknowledgment* a confirmar sucesso, indicando que o servidor está à escuta. Após este processo, onde de forma objetiva, não se perde pacotes na mensagem, agora sim se envia a mensagens pretendidas, titulado de*final acknowledgment*. Obviamente, é um controlo de erros longe da perfeição uma vez que a existência de métodos que lidam com este tipo de acontecimentos na camada aplicacional tornam-se cruciais para um melhor desempenho e consequentemente sucesso entre as conexões efetuadas.[1]

1. *"TCP vs. UDP: Understanding 10 Key Differences"*, https://www.spiceworks.com/tech/networking/articles/tcp-vs-udp/, Accessed: 2022-11-16.

### 3.2.1 Ficheiro de Configuração

A maior parte do controlo de erros efetuado pelo sistema é sobre o ficheiro de configuração, uma vez que, sem ele, não há qualquer tipo de ação disponível. No módulo *Parser* existe um método titulado de *check-file* que recorre de forma iterativa ao método *check-line* que divide a linha pelo espaços existentes na mesma. Verificando se cada segmento representa devidamente o parâmetro definido pelo corpo docente. Caso contenha qualquer tipo de erro o sistema irá informar através de *exception*, i.e. através de uma mensagem de erro no terminal, ao utilizador que existiu algum tipo de erro na leitura do ficheiro e descarta-o.

### 3.2.2 Ficheiro de *Logs*/Base de Dados

Ambos partilham a funcionalidade de registo, sendo alvos de escrita pelo sistema ao longo da execução. Difere apenas a necessidade de controlo de erros na leitura de base de dados ao arranque, semelhante ao existente no *parser*, para deteção de anomalias e, futuramente, o controlo de acessos concorrentes aos ficheiros de logs caso estes sirvam para reportar toda a atividade de um servidor, e.g. um servidor que funciona como primário para múltiplos domínios e quer registar a atividade num único ficheiro.

# 4 Modelo de Comunicação

No protocolo de comunicação de uma mensagem DNS há várias partes, quer *hardcoded* quer *softcoded*, que são repetitivas entre servidores onde há algum código repetitivo pois há premissas que tem que ser alcançadas, construídas e respeitadas.

## 4.1 Servidores

Os servidores, relativamente ao primário e secundário efetuam uma comunicação entre si por transferência de zona através de TCP.Durante as diferentes atividades efetua chamadas ao módulo de *logs* onde registará as atividades que participou durante o seu tempo de vida. De igual modo, o servidor de resolução, mais recente implementado, possui a capacidade de efetuar ligações através de UDP entre servidores de topo, domínio e principais efetuando, igualmente, chamadas aos *logs*.

Primeiramente os servidores tem como variável de classe um dicionário, similar a um *Map* em outras linguagens como *Java*, *Javascript*, C ou C++ mas com uma definição forte sobre tipos e não sobre objetos, onde guardará todos os possíveis atributos a modificar e de interesse em relação à utilização de um servidor. Variáveis de instância essas sendo: domínio, endereço, porta,*timeout*, modo de *debug* em que o valor *default* é *shy*, caminho para o ficheiro de configuração, objeto configuração, lista de caminho para os ficheiros de *logs*, objeto *Logs* que é um dicionário de *Logger*s', caminho para o ficheiro de base de dados, objeto de base de dados, lista de servidores de topo, tamanho do header em que valor *default* é $1Kb$ e o *encoder* da mensagem DNS em valor *default* é $utf-8$.

Todos esses valores são atualizados segundo a função *get-args* do módulo *Parser* onde percorre o ficheiro de configuração e atualiza os valores do dicionário de propriedades mencionado acima. Caso não seja inserido na linha de comandos como *input* assume os valores definidos como *default*. Por sua vez, na linha de comandos será obrigatoriamente inserido o caminho para o ficheiro de configuração e opcionalmente, porta, tempo antes de *timeout* e modo de *debug*. Esses parâmetros são passados ao *get-args* de forma a atualizar as suas propriedades *a posteriori*.

De forma similar, os servidores invocam a classe *Parser* para verificar a veracidade dos ficheiros de configuração e ficheiro de base de dados, dando como argumentos o caminho do ficheiro e um *booleano* que identifica qual é o ficheiro que está a ser inserido. Dessa forma podemos operar segunda diferentes classes como a classe *Database*, responsável pela

base de dados e a classe *Logs* mencionada múltiplas vezes ao longo deste documento.

Ainda em ambos os servidores, principal e secundário, é efetuado os métodos para a realização de ligações/conexões TCP, objetivo principal destes módulos e portanto cruciais para todo o funcionamente deste sistema. Seguimos então um procedimento de conexões TCP dentro do fornecido pelo professor Pedro. Onde através do *multi-threading* o servidor A espera receber informação pela transferência de zona do servidor B e/ou vice-versa.[1]

### 4.1.1 Servidor Primário

O servidor principal de cada domínio possui tanto a ligação por UDP com o servidor de resolução que solicita a resposta à *query* DNS, como também possui a criação de ligação por TCP para a troca de mensagens na transferência de zona. Tem também acesso exclusivo direto e a qualquer instante à base de dados.[2]

### 4.1.2 Servidor Secundário

Uma das poucas diferenças de um servidor secundário para um servidor primário é o facto da base de dados não ser atualizada em tempo real mas sim apenas passado um determinado tempo, definido no ficheiro de base de dados. A outra diferença é que este não se liga diretamente ao cliente, apenas devolve a resposta a determinada *query* que possa provir, do servidor principal desse domínio, no qual não encontra resposta e questiona dentro da transferência de zona aos seus subdomínios. Essas conexões internas, mais uma vez, são apenas efetuadas dentro do protocolo TCP retornando a resposta em mensagem DNS ao servidor primário que por sua vez encaminhará a mesma para o cliente, i.e. solicitador de *query* DNS.

### 4.1.3 Servidor Resolução

Já o servidor de resolução é bastante diferente dos outros servidores. Faz apenas um tipo de conexão que é em UDP. Esse tipo de conexão é utilizada tanto para a ligação com o cliente, como para o envio e receção das diversas procuras nos diferentes servidores de topo e/ou domínio. Recebendo a *query* do cliente irá procurar se encontra a resposta para tal na sua *cache*, caso não consiga irá procurar a *query* no servidor de topo. Este não tendo na *cache* reenviará para o *DNS resolver* que encaminhará para o servidor de domínio. De igual forma, não tendo a resposta para tal em uma das *entries* da *cache* responde ao SR com os A-*records* dos servidores que possui. Por fim o servidor resolução procurará no

1. *"DNS: O que é, como funciona e como configurar o DNS de um domínio"*, https://rockcontent.com/br/blog/dns/, Accessed: 2022-11-16.
2. *"Mariii-23 Github Repository"*, https://github.com/Mariii-23/CC_20-21, Accessed: 2022-10-31.

servidor principal do mesmo domínio a resposta, cujo poderá efetuar transferência de zona ou não para encontrar a resposta ao pedido inicial. Caso este processo todo seja em vão, passará para o próximo servidor de topo, onde por fim enviará a respota ao cliente tal como já era esperado.

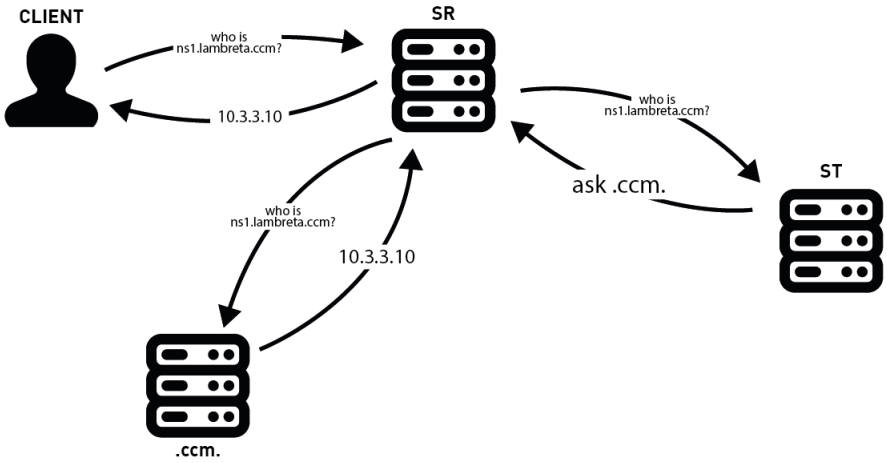


Figure 4.1: Fluxo query DNS

### 4.1.4 Transferência de Zona

A transferência de zona é um dos requisitos mais relevantes num sistema DNS. Esta é fundamental nas relações entre um servidor primário (SP) e os seus respetivos servidores secundários (SS) uma vez que os permite estar em sintonia.

De forma concreta, os servidores secundários pedem uma cópia da base de dados relativa a um domínio ao servidor primário logo no seu arranque e vão-se mantendo atualizados, questionando periodicamente o servidor primário se possui uma nova versão da sua base de dados, *i.e* se houve mudanças ou acrescentos na base de dados do domínio.

Para saber se houve qualquer alteração, o servidor secundário envia uma *query* do tipo "ZT" ao servidor primário pedindo a versão atual da sua base de dados. O SP irá responder com a versão atual da sua base de dados e, caso o SS verifique que a sua cópia se encontra desatualizada - ou, no caso de arranque, inexistente - inicia o protocolo de transferência de zona. Este protocolo começa com o envio de uma mensagem do SS para o SP indicando o domínio para o qual pretende fazer a cópia da base de dados. O SP verifica se o remetente desta mensagem possui autorização para receber uma cópia verificando nos seus registos os servidores autoritários.

Caso este tenha autorização, envia-lhe uma mensagem com o número de entradas do seu ficheiro base de dados, esperando ter confirmação, sendo esta confirmação uma mensagem indicando o mesmo valor que enviou, por parte do seu servidor que fez o pedido. Posta

esta confirmação, passa a enviar as entradas do seu ficheiro linha a linha, esperando, para cada uma, uma confirmação de receção. Do outro lado, o SS vai recebendo as entradas, verificando se estas chegam por ordem, visto que vêm indexadas, e enviando mensagens de confirmação. A informação é então inserida na base de dados do servidor secundário de, após a recessão do número de linhas acordado anteriormente, não houver qualquer problema.

Para esta troca de mensagens são utilizados protocolos UDP, no envio e receção da *query* inicial, e TCP, na restante troca de informação mencionada. Numa primeira fase, para nos certificarmos que não há qualquer problema nesta troca, exigimos uma resposta de confirmação ("OK!") ao SS para cada entrada que recebe por parte do SP. Além disso, as entradas da base de dados são enviadas com índices para certificar que não há trocas de linhas indesejadas. De notar que estes dois detalhes se tornam relevantes à sombra do facto de que estamos a utilizar protocolos TCP/IP, o qual já nos garante fazer chegar a informação pretendida de forma ordenada.

## 4.2 Cliente

De igual modo aos servidores o cliente também tem uma variável de classe do tipo dicionário que guarda todas as propriedades necessárias para o funcionamento do envio e receção da mensagem DNS. Após de igual modo recorrer ao *Parser* e efetuar o *alias* corre um pedido de DNS através de UDP para o IP inserido[3] como *input* na linha de comandos. De igual forma o método de conexão UDP também é bastante semelhante ao fornecido pelo professor Pedro.

Para execução da *query*, é necessário construir a mensagem que será enviada. Para essa construção, é utilizado um método da classe **Query** (proveniente do módulo *dns_controller*) que cria uma mensagem DNS vazia, semelhante à apresentada no enunciado. Esta mesma estrutura irá ser utilizada, futuramente, para construir a resposta ao cliente.

## 4.3 Logs

Para a escritura de *logs* nos respetivos ficheiros recorremos maioritariamente a uma biblioteca já existente em python titulado de *logging*. Seguindo e utilizando os padrões e operações desenvolvidas na mesma, efetuando *override* e implementação de outras recorrendo a um projeto *open-source* criado no ano passado na plataforma *GitHub* por um utilizador intitulado de Delgan que efetua atualizações mensais ao projeto..[4] Optamos

3. *"Send DNS request with socket in Python"*, https://stackoverflow.com/questions/58805814/send-dns-request-with-socket-in-python, Accessed: 2022-11-12.

4. *"How to use logging library in python"*, https://github.com/Delgan/loguru, Accessed: 2022-10-30.

pela utilização destas bibliotecas de forma a poupar bastante implementação uma vez que é código já implementado que utiliza controlo de concorrência sendo assim menos um parâmetro para nos preocupar.

Esta classe é invocada recebendo como argumentos uma lista de caminhos para os ficheiros de logs, i.e caso não existam são criados no caminho recebido, um modo de *debug* i.e *shy* se for apenas para escrever no ficheiro e *debug* se for para escrever no ficheiro e no terminal de igual modo. Por fim é dado também como argumento um *booleano* se for cliente ou não. Após a verificação dos argumentos com o respetivo controlo de erros previamente mencionado, efetuamos a criação dos diferentes *Loggers* para determinados ficheiros i.e. cada ficheiro presente na diretoria de *logs* possuí uma só instância *logger* que efetua a escrita nesse ficheiro. Essa mesma instância efetua a escrita num determinado ficheiro, podendo ser invocado em qualquer módulo que recorra à sua solicitação.

```
24:11:22.21:28:21:342: ST 10.2.2.10 5353 255 debug
24:11:22.21:28:21:342: EV @ conf-file-read /home/core/CC32/config/config-barreteSP.txt
24:11:22.21:28:21:343: EV @ initialized
24:11:22.21:28:21:343: EV @ udp-listening-at 10.2.2.10:5353
24:11:22.21:28:21:343: EV @ tcp-listening-on 10.2.2.10:5353
24:11:22.21:29:06:636: QR 10.4.4.1:39889 20757,Q,0,0,0,0;barrete.ccm.,MX;



24:11:22.21:29:06:636: RP 10.4.4.1:39889 20757,A,0,1,3,4;barrete.ccm.,MX;
barrete.ccm. MX mx1.barrete.ccm. 86400 10;
barrete.ccm. NS ns1.barrete.ccm. 86400,barrete.ccm. NS ns2.barrete.ccm. 86400,barrete.ccm. NS ns3.barrete.ccm. 86400;
mx1.barrete.ccm. A 10.2.2.11 86400,ns1.barrete.ccm. A 10.2.2.10 86400,ns2.barrete.ccm. A 10.1.1.12 86400,ns3.barrete.ccm. A 10.4.4.10 86400;
```

Figure 4.2: Exemplo de um ficheiro de logs

## 4.4 Base de Dados

De forma a armazenar toda a informação relevante contida num ficheiro de base de dados, foi criada uma classe **Database**. Esta contem variáveis onde são armazenadas as informações conforme o seu tipo, e de forma conveniente para a realização de futuras queries.

Á medida que o ficheiro de Base de Dados é lido, esta classe, inserida no servidor, faz um parse próprio e insere os dados devidamente.

## 4.5 Mensagens DNS

O cliente envia uma mensagem DNS a um servidor da sua escolha, quando quer efetuar uma query DNS sobre um determinado domínio ou servidor.[5] A mensagem a ser enviada

5. *"DNS: Types of DNS Records, DNS Servers and DNS Query Types"*, https://ns1.com/resources/dns-types-records-servers-and-queries, Accessed: 2022-11-12.

deve seguir o seguinte formato:

{IP do servidor}, {domínio/servidor pretendido}, {tipo da query},{modo de fazer a query}

Cada campo representa, respetivamente:

- **IP do servidor a que pedir para efetuar a query**;
- **nome do domínio/servidor** sobre qual pretende efetuar a query;
- **tipo da query** que se pretende efetuar. Pode ser qualquer um dos **tipos suportados na sintaxe dos ficheiros de base de dados dos SP**
- deve ser **I**, que indica a intenção de executar de forma iterativa (não tendo sido implementada a forma recursiva).

Dois exemplos de pedidos válidos seriam: "**10.2.2.10 barrete.ccm. MX I**" e "**10.3.3.10 lambreta.bgj. A I**". Esta mensagem é então enviada para o servidor em questão através dos sockets e este quando a recebe, atribui um identificador aleatório e utiliza a classe **Query** e os seus métodos para efetuar uma pesquisa na sua **Database** e construir o pedido e a resposta á query.

A resposta é então enviada de volta para o endereço IP do cliente, onde lhe é apresentada a resposta no seguinte formato:

{identificador da mensagem}, {flags}, {código de resposta}, {número de valores de resposta encontrados}, {número de valores autoritativos encontrados}, {número de valores extra encontrados};
{nome do domínio/servidor pedido}, {tipo de query pedido}; {valores de resposta};
{valores autoritativos}; {valores extra}

Cada campo representa, respetivamente:

- número de pedido atribuído pelo o servidor;
- pode ser **Q** (ser for query, se não tiver é resposta a esta), ou **A** (indica se resposta é autoritativa);
- **código de erro** que pode ser **0** (se não tiver erro nenhum), **1** (se o domínio/servidor existir, mas não existir resposta ao tipo pedido), **2**(se o domínio/servidor não existir) ou **3** (usada para fornecer os IPs de domínios relevantes na procura iterativa de uma resposta);
- **número de valores de resposta encontrados;**
- **número de valores autoritativos encontrados;**
- **número de valores extra encontrados;**

- **nome do domínio/servidor** sobre qual a query foi efetuada;
- **tipo de query** que foi efetuada;
- **todos os valores cujo nome e tipo correspondem com o pedido**, podendo este ser **todos os tipos suportados na sintaxe dos ficheiros de base de dados dos SP**;
- **todos os valores cujo nome correspondem com o pedido e cujo tipo é NS**, sendo vazio de o tipo pedido na query for NS;
- **todos os valores cujo nome correspondem com o pedido e cujo tipo é A**, tendo também obtido **correspondência no seu parâmetro com todos os valores obtidos nos dois campos anteriores.** É vazio se o tipo pedido na query for A:

Para demonstrar as respostas, efetuamos os pedidos anteriormente mencionados como exemplo, obtendo:

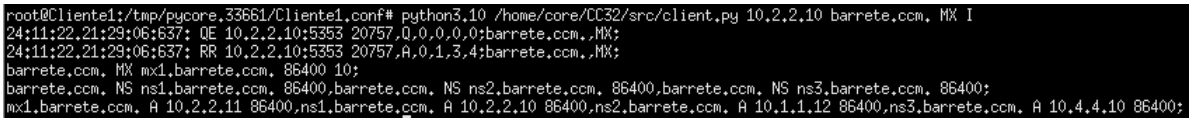


Figure 4.3: Resposta à query "10.2.2.10 barrete.ccm. MX I"

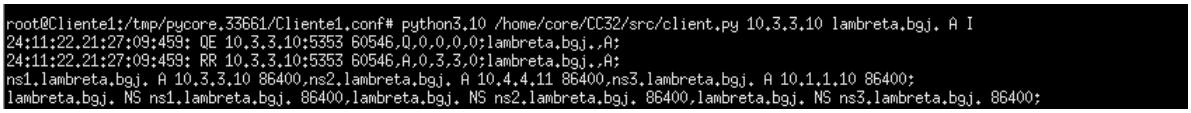


Figure 4.4: Resposta à query "10.3.3.10 lambreta.bgj. A I"

### 4.5.1 Cache

A cache é utilizada pelo o SR de forma a armazenar a informação relacionada com os pedidos de queries antes realizadas, nomeadamente com os *response values* destes, de forma a responder mais rapidamente a pedidos cuja informação está lá contida.

Implementamos a cache através da classe **Cache** que contem um array de tamanho fixo (20 entradas, neste caso) de objetos da classe **Entry**, que representam, como o nome indica, as várias entradas da cache. Contém também todos os métodos relacionados com a gestão de dados em *caching* (procura, atualização e inserção de entradas).

Cada **Entry** contém o nome, tipo, valor, ttl, origem (SP ou OTHERS, sendo OTHERS por omissão), tempo em que foi inserido e estado (FREE ou VALID). Estes campos são todos derivados de uma resposta a uma query.

**Nota:** Devido a facilitar a gestão e inserção na cache de um servidor, todos as entradas são inseridas com origem OTHERS, o que permite á cache não ficar sobrecarregada com entradas SP (que não são normalmente libertadas na pesquisa.)

Ao receber um pedido de Query, o servidor faz uma pesquisa pela a sua cache a verificar se contém entradas com informação para responder ao pedido, ao mesmo tempo libertando entradas cujo tempo na cache ultrapassou o seu TTL. Se tiver, responde ao cliente com essa mesma informação, caso contrário, continua, de forma iterativa, a contrução de uma resposta ao cliente.

Terminada a pesquisa, e se foi obtida uma resposta com *Response Values*, o servidor utiliza os métodos da classe Cache para desconstruir a resposta, criar uma entrada a partir desta e inserir-la na sua cache para posterior utilização.

# 5 Divisão de Trabalhos

| | Gustavo | João | Cláudio |
|---|---|---|---|
| **1ª FASE** | | | |
| Arquitetura de Sistema | | | X |
| Modelo de Informação | | X | X |
| Modelo de Comunicação | X | X | X |
| Planeamento do Ambiente de Teste | X | X | |
| Transferência de Zona | X | | |
| Tratamento de Queries | | X | |
| Logs | | | X |
| Testes | X | | |
| **2ª FASE** | | | |
| Atualização da Arquitetura do Sistema | | | X |
| Atualização do Modelo de Informação | X | X | X |
| Atualização do Modelo de Comunicação | X | X | X |
| Planeamento do Ambiente de Teste | X | | |
| Implementação de SP, SS, ST, SDT e SR | X | X | X |
| Implementação da Cache | | X | |
| Testes | X | X | X |

Como se pode observar, o desenvolvimento do trabalho foi igualmente dividida pelo grupo, sendo que, naturalmente, cada elemento se focou mais numa área em particular.

# 6 Conclusão e Trabalho Futuro

Numa primeira fase do trabalho conseguimos aprofundar e aplicar os nossos conhecimentos sobre o DNS e dos protocolos TCP e UDP, sendo estas as bases deste trabalho prático. Definimos também o planeamento a seguir para o trabalho prático inteiro, e conseguimos implementar com sucesso as funcionalidades requisitadas para esta fase.

Já nesta segunda fase ou fase final, conseguimos aplicar os nossos objetivos de implementar a topologia por completo recorrendo à implementação de um *DNS resolver*, i.e. um servidor de resolução, capaz de verificar a resposta direta em *cache* a uma mensagem proveniente do cliente e/ou redirecioná-las para os devidos servidores que, possivelmente, terão a resposta para a mesma.

Contudo, não foram implementadas por completa algumas funcionalidades, ou por ser opcionais ou por *bugs* que não conseguimos corrigir dentro do prazo solicitado para a entrega final. Entre eles temos a existência do rDNS que está implementado apenas como parâmetro PTR da base de dados, as prioridades e tempos de vida (TTL) apenas estão implementados nas *queries* e na *cache*. Por fim não implementamos totalmente o processo recursivo para obter a resposta à mensagem DNS, não estando funcional.

# Bibliography

*"DNS: O que é, como funciona e como configurar o DNS de um domínio"*. https://rockcontent.com/br/blog/dns/. Accessed: 2022-11-16.

*"DNS: Types of DNS Records, DNS Servers and DNS Query Types"*. https://ns1.com/resources/dns-types-records-servers-and-queries. Accessed: 2022-11-12.

*"Domain Name System"*. https://www.rfc-editor.org/rfc/rfc4180.txt. Accessed: 2022-11-18.

*"How to use logging library in python"*. https://github.com/Delgan/loguru. Accessed: 2022-10-30.

*"Mariii-23 Github Repository"*. https://github.com/Mariii-23/CC_20-21. Accessed: 2022-10-31.

*"Send DNS request with socket in Python"*. https://stackoverflow.com/questions/58805814/send-dns-request-with-socket-in-python. Accessed: 2022-11-12.

*"Simple DNS server (UDP and TCP) in Python using dnslib.py"*. https://gist.github.com/pklaus/b5a7876d4d2cf7271873. Accessed: 2022-11-5.

*"TCP vs. UDP: Understanding 10 Key Differences"*. https://www.spiceworks.com/tech/networking/articles/tcp-vs-udp/. Accessed: 2022-11-16.