Universidade da Beira Interior
Departamento de Informática

Nº 143 - 2021: *Distributed Data-structures in Go*

Elaborado por:

Guilherme João Bidarra Breia Lopes

Orientador:

Professor/a Doutor/a [NOME ORIENTADOR(A)]

13 de fevereiro de 2021

Agradecimentos

...

Conteúdo

Lista de Figuras

Lista de Tabelas

Acrónimos

HTTP Hypertext Transfer Protocol

IP Internet Protocol

URL Uniform Resource Locator

IDE *Integrated Development Environment* - Ambiente de Desenvolvimento Integrado.

SVG Scalable Vector Graphics

Capítulo

1

Introdução

1.1 Enquadramento

Este projeto envolve o estudo e implementação de um protocolo para sistemas distribuídos, o *Arrow Distributed Directory Protocol* (ADDP) [1]. A realização deste compreende os tópicos de programação em Linguagem *Go*, Estruturas de Dados, concorrência, sistemas e algoritmos distribuídos.

A elaboração deste projeto decorreu-se na unidade de projeto de final de curso da Licenciatura em Engenharia Informática.

1.2 Objetivos

Inicialmente foram estabelecidos objetivos, no entanto durante a implementação do projeto, o tema distanciou-se do inicialmente proposto. Os objetivos no desenvolvimento do projeto foram:

- Leitura do *paper* oirignal e proposto e compreensão geral do protocolo.
- Elaboração da especificiação dos elementos, ligações entre estes e os seus comportamentos.
- Implementação inicial do programa.
- Execução de várias instâncias do programa com o uso da ferramenta *Docker*.
- Implementação da visualização do grafo representativo do diretório.
- Implementação da visualização da fila (*Queue*).

1.3 Organização do Documento

1.

2.

3.

Capítulo

2

Motivação

2.1 Introdução

Atendendo à crescente utilização de grandes infraestruturas, como em redes sociais, grandes empresas e a demanda de maior capacidade de processamento, como por exemplo, o aumento do número de núcloes nos processadores ou a conveniência de várias máquinas trabalharem em conjunto, a necessidade do uso de sistemas distribuídos tem vindo a aumentar ao longo dos anos.

Das várias vantagens no uso de sistemas distribuídos, as que mais se destacam são as seguintes.

Escalabilidade Horizontal Os sistemas distribuídos permitem que se aumente a capacidade de processamento e armazenamento ao introduzir máquinas no sistema, ao invés de melhorar uma única máquina.

Maior tolerância a erros Quando o processamento está distribuído por vários nós, a falha de um único nó não levará a que o sistema num todo falhe.

Mais eficientes O uso de algoritmos distribuídos permite um maior número de máquinas em execução concorrente, e consequentemente, a divisão do trabalho pelas várias máquinas. Também é possível a distribuição da localização física das máquinas, o que permite conexões de maior velocidade em outros locais no mundo.

No entanto, os sistemas distribuídos, que inerentemente são concorrentes, têm uma complexidade maior no desenvolvimento em comparação com

sistemas e programas sequênciais. A falta de um relógio central, a possibilidade de vários processos necessitarem de aceder ao mesmo recurso ao mesmo tempo, ordem indeterminada de quais quer pedidos, a necessidade de se usar uma rede para comunicar informação entre os nós e dificuldade de controlar os vários sistemas independentes pertencentes ao sistema distribuídos são alguns dos fatores para esta complexidade.

2.2 Descrição do Protocolo

Nesta secção será descrito o funcionamento e estrutura do *Arrow Distributed Directory Protocol* (*Arrow*).

Este sistema consiste numa rede, que permite o envio assíncrono de mensagens (ou a comunicação) entre os nós, e um diretório, que permite localizar um objeto na rede e garantir o acesso exclusivo a este.

O objeto é considerado "móvel", porque há a possibilidade de este se movimentar na rede entre os nós. O objeto pode ser considerado um processo, um ficheiro ou qualquer outra estrutura de dados.

Cada nó tem uma só ligação, deste para outro nó, no entanto pode haver qualquer número de nós com ligação a um único.

Quando um nó pretende aceder ao objeto, este envia um pedido de acesso pela sua ligação.

Caso um nó receba um pedido de acesso,a ligação deste passa a apontar para o nó vizinho que lhe fez chegar o pedido, ou seja, por onde passa o pedido há uma inversão do sentido da ligação.

Se um nó recebeu um pedido de acesso e este detém ou espera pelo acesso ao objeto, o nó que realizou o pedido passa a estar à espera que este nó que recebeu lhe ceda o objeto e a circulação do pedido termina.

Numa vista global sobre a rede, quando um nó realiza um pedido, as ligações dos nós por onde esse pedido passou passam a indicar onde é que futuramente estará o objeto, mas é possível a circulação de vários pedidos na rede, o que

Se um novo nó, que foi afetado pela passagem do pedido, realizar um pedido de acesso, este chegará ao nó que anteriormente fez o pedido.

Esta inversão das ligações permite que, o nó apenas detendo uma ligação, faça chegar o seu pedido ao nó que detém o objeto ou a um nó que deterá o objeto mas que espera por ele.

Caso um nó que espera pelo objeto, quer porque o seu pedido ainda está em circulação na rede ou o nó detentor do objeto ainda não o cedeu, receba um pedido de acesso de outro nó, então este armazena o uma ligação nó que realizou o pedido.

Quando o nó detém o objeto,se recebe um pedido ou tem em espera um outro nó, o objeto é cedido através de uma ligação direta entre dois nós, evitando a passagem do objeto pelo diretório.

A particularidade deste protocolo deve-se à mudança tanto da localização do objeto como a sua origem, pois esta muda-se para o nó que tem o acesso ao objeto, e não há um único nó que detém a localização atual do objeto, mas a disposição de todas as ligações permite a localização do objeto por todos os nós.

O facto da origem do objeto estar em constante alteração evita que apenas um nó detenha o acesso ao objeto, ou seja, o acesso exclusivo, e que e que nenhum nó se torne num foco, ou seja, que nenhum nó receba demasiados pedidos em relação a outros nós, provocado um engarrafamento/*bottleneck* no acesso ao objeto.

2.2.1 Estrutura do Diretório

O diretório constitui um grafo, mais especificamente uma árvore de extensão mínima, em que os vértices são os nós pertencentes ao diretório e as arestas as ligações entre os nós.

2.2.2 Estruturas de Dados

2.3 Trabalhos relacionados

De muitos algoritmos distribuídos e suas implementações, há dois trabalhos muito relacionados com o protocolo tratado neste projeto. Estes são, o *Ivy* [**?**] e o *Arvy* [**?**]. O *Ivy* apresenta um funcionamento muito similar ao *Arrow*, em que a origem do objeto muda com a posição atual deste, isto é,

Na implementação do mesmo protocolo retratado neste projeto existe apenas...

2.4 Conclusões

Capítulo

3

Especificação

3.1 Introdução

3.2 *Node*

Neste diretório existe apenas um *Node* que detém o acesso ao objeto, pois um dos propósitos deste protocolo é garantir o acesso exclusivo a este.

No entanto podem existir outros *Nodes* que pretendem ter acesso ao objeto, e que ficam à espera deste.

Também é possível que *Nodes* detenham ou que esperam pelo acesso ao objeto, recebam um pedido de acesso de outro *Node*, ou seja, os *Nodes* podem ter pedidos em espera.

Ou seja, é possível que um *Node* detenha o acesso ao objeto ou futuramente deterá e ter um outro *Node* à espera que este ceda o acesso.

Estes fatores diferenciam os vários *Nodes* no diretório, e de seguida será feita uma enumeração destes fatores, sendo que estes podem coexistir:

- Se o *Node* detém o acesso ao objeto - ***Owner*/Dono**
- Efetuou um pedido acesso ao objeto. - ***Waiter*/*Node* em espera.**
- Não tem pedidos em espera. - ***Terminal***
- Tem um pedido em espera. - ***With Request*/Com Pedido.**
- Nenhum dos anteriores. - ***Idle*/Inativo.**

A partir da combinação destes fatores temos os seguintes tipos de *Nodes*:

Owner Terminal -
Node que detém o acesso ao objeto e não tem pedidos em espera.

Owner With Request -
Node que detém o acesso ao objeto mas tem um pedido em espera.

Waiter Terminal -
Node que efetuou um pedido de acesso ao objeto mas não tem pedidos em espera.

Waiter With Request -
Node que efetuou um pedido de acesso ao objeto mas tem um pedido em espera.

Idle -
Node que não detém o acesso ao objeto nem efetuou pedido de acesso.

3.3 Ligações

Os *Nodes* têm a possibilidade de comunicar entre si, quer usando o diretório, quando estes pretendem pedir o acesso ao objeto, ou diretamente, quando estes transmitem o acesso ao objeto para outro *Node*.

As ligações (ou comunicações) entre os *Nodes* são feitas a partir de *Channels* (canais). Estes *Channels* funcionam como caixa de entrada ou endereços dos *Nodes*, em que podem ser enviados para outros *Nodes*, e que, caso um *Node* decide transmitir alguma mensagem para o outro *Node*, esta transmissão é feita usando um *Channel* do recetor da mensagem.

Uma outra forma de definir o uso dos *Channels* seria que este é formado por duas partes, a entrada e a saída, em que a entrada é usada pelo *Node* transmissor e a saída está no *Node* recetor.

Os *Channels* são usados quando um *Node*:

- Pretende enviar uma mensagem para outro *Node*, este fá-lo usando o *Channel* pertencente ao recetor.
- Pretende receber mensagens de outros *Nodes* este envia o seu *Channel*.

Ligações no diretório

Como referido no capítulo anterior, o diretório é um grafo (mais especificamente uma árvore de extensão mínima), em que os vértices são os *Nodes* e as arestas são as ligações, e cada ligação é um conjunto de uma entrada (no

transmissor) e uma saída (no recetor), o que indica que este grafo é dirigido, visto que há uma direção pela qual cada mensagem é transmitida por um *Channel* (da entrada para a saída).

Neste tipo de ligação no diretório foram definidas duas partes nas ligações, o "Link", parte de entrada da ligação que está presente no transmissor, que se liga a um "Find" de um *Node*, esta sendo a parte da saída das mensagens, que está presente no recetor.

Então, no grafo que representa o diretório, considera-se que existe uma ligação/um arco de um *Node* **X** para um *Node* **Y** quando o "Link" do *Node* **X** é (ou aponta para) o "Find" do *Node* **Y**.

Então o *Node* **Y** é o *Child Node* do **X**, e o *Node* **X** é o *Parent Node* do **Y**.

Estas ligações do diretório são usadas apenas para a transmissão e circulação dos pedidos de acesso ao objeto, visto que um *Node* não tem informação da localização deste mas tem informação sobre a ligação para o seu *Child Node*, e o conjunto das ligações aponta sempre para a localização atual (o *Owner*) ou uma futura localização (um *Waiter*), ou seja, o *Node* não tem informação sobre a localização atual ou uma futura, mas o conjunto de todas as ligações forma essa informação.

Estes dois tópicos serão retratados em subcapítulos seguintes, mas é necessário mencioná-los:

- Quando um *Node* recebe um pedido de acesso, a ligação entre este e o *Node* que lhe transmitiu é invertida. Esta inversão é desempenhada para que as ligações dos *Nodes* tenham sempre a direção para onde está ou estará o objeto.

- Os pedidos de acesso contém informação necessária para a inversão das ligações e para que seja possível o envio do objeto do atual *Owner* para o *Node* que realizou o pedido.

Ligação direta

É possível que um *Node* tenha outro *Node* à espera que lhe envie o acesso ao objeto. No entanto para que este tenha um pedido em espera é necessário que este seja o detentor do objeto ou que futuramente tenha o acesso (*Waiter*).

Quando o *Node* tem outro *Node* em espera, há uma ligação direta entre estes. Esta ligação é utilizada no envio do acesso ao objeto, pois o *Node* quando receber o acesso, este poderá ceder ao *Node* que espera através desta ligação.

Neste tipo de ligação são definidas duas partes, o "WaiterChan", parte de entrada da ligação que está presente no transmissor, que se liga ao "MyChan" de um *Node*, sendo esta a parte da saída do acesso ao objeto, que está presente no recetor.

Com a mudança das ligações no diretório e o facto de que um *Node* pode ter um (e apenas um) outro *Node* à espera, isto resulta na circulação de um pedido até este chegar a um *Node* que detém ou futuramente deterá (mas que espera) o acesso ao objeto e que não tenha qualquer outro pedido.

Neste sistema é possível existirem *Nodes* à espera de outros *Nodes* que também estão à espera, e cada um tem no máximo um outro *Node* à espera. Este comportamento dos *Nodes* origina a criação de uma fila (*Queue*), pois o *Node* que espera pelo atual detentor do objeto irá ser o primeiro da fila e posteriormente enviará o acesso ao objeto ao *Node* que espera por este, e assim seguidamente.

A cabeça desta fila será o atual detentor do objeto, que está diretamente ligado a um outro *Node*, e o último elemento da fila será o *Node* na fila que não tem qualquer outro *Node* à sua espera.

3.4 Atributos do *Node*

Neste secção serão descritos os atributos que podem constituir um *Node*.

Find

Este atributo representa o *Channel* do *Node* onde este recebe pedidos de acesso ao objeto. Está presente em todos os *Nodes*.

MyChan

Este atributo representa o *Channel* do *Node* para o qual é transmitido o objeto. Está presente em todos os *Nodes*.

Link

Este atributo representa a ligação do *Node* para o "Find" de um outro *Node*, ou seja, uma ligação para o seu *Child Node*.

WaiterChan

Este atributo representa o *Channel* para o "MyChan" de um outro *Node*, ou seja, o seu sucessor na fila.

Obj

Este atributo representa o acesso ao objeto por parte do *Node*. Em qualquer estado da rede, apenas um *Node* dispões deste atributo.

Atributos de cada tipo de *Node*

Neste subcapítulo serão definidos os atributos que cada tipo de *Node* apresenta.

Owner Terminal

- Find
- MyChan
- Obj

Owner With Request

- Find
- MyChan
- Link
- Obj
- WaiterChan

Idle

- Find
- MyChan
- Link

Waiter Terminal

- Find
- MyChan

Waiter With Request

- Find
- MyChan
- Link
- WaiterChan

3.5 Tipos de mensagens

Nesta especificação foram apenas definidos dois *Channels*. A comunicação entre os *Nodes* é feita por canais, pelos quais são comunicados canais (as mensagens).

Access Request

Este tipo de *Channel* é usado para fazer chegar o pedido a um *Node* que detém ou futuramente irá deter o acesso ao objeto. O *Node* de destino depende da estrutura atual do diretório, no entanto o *Node* que realizou o pedido não terá informação sobre o destino, mas o recetor de destino terá a informação necessária para que este tenha a possibilidade de ceder o acesso ao objeto ao *Node* que realizou o pedido.

Para tal, neste *Channel* são comunicados dois *Channels*:

- O *Channel* ***MyChan*** do *Node* que fez o pedido.
- O *Channel* que identifica quem fez chegar o pedido, ou seja, o ***Find*** do **Parent Node**.

Give Access

No entanto, há um tipo de *Channel* que é usado para ceder o acesso ao objeto a quem fez o pedido, por outras palavras, é usado pelo atual ***Owner*** para transmitir o acesso ao *Waiter* que estava à sua espera, ou seja, ao primeiro da *Queue*.

3.6 Comportamentos dos *Nodes*

Neste capítulo serão

Owner Terminal

Receção de um pedido Access Request

O *Node* recebe um pedido ***Access Request*** no seu *Channel* ***Find***, que foi remetido pelo seu *Parent Node*.

Como o *Node* é o detentor do acesso ao objeto, este transforma-se em ***Owner With Request***, isto é, é o detentor do acesso objeto mas existe um *Node* que espera pelo acesso. Atualiza o ***Link*** (para ***NewLink***), que aponta para o **Find** do seu *Parent Node*, havendo uma inversão da ligação entre o *Node* e o *Parent Node*, e passa a deter o ***WaiterChan*** que aponta para o *MyChan* do *Node* que realizou o pedido, isto para que seja possível o envio do acesso ao objeto.

O *Node* sofre a transformação **OwnerWithRequest(Find, MyChan, Obj, NewLink, WaiterChan)**.

Owner With Request

Receção de um pedido Access Request

O *Node* recebe um pedido ***Access Request*** no seu *Channel* ***Find***, que foi remetido pelo seu *Parent Node*.

Este envia pelo ***Link*** o ***WaiterChan*** do pedido ***Access Request*** e o seu *Channel* ***Find***. É enviado o "Find" para que o seu *Child Node* tenha a possibilidade de inverter a sua ligação, ou seja, para que possa haver um arco do *Child Node* para o *Node*, e o "WaiterChan" para que se faça chegar o *MyChan* do *Node* que fez o pedido a um *Node Waiter* ou *Owner*.

Como o *Node* já tem em espera um pedido de acesso, este mantém-se como ***Owner With Request***, mas atualiza o ***Link*** (para ***NewLink***), que aponta para o **Find** do seu *Parent Node*, havendo uma inversão da ligação.

O *Node* sofre a transformação **OwnerWithRequest(Find, MyChan, Obj, NewLink, WaiterChan)**.

Cedência do Objeto

Após a receção de um pedido ***Access Request***, o *Node* pode ceder o acesso ao objeto ao *Node* que fez o pedido. Para tal, este envia pelo *Channel* ***WaiterChan*** (O *MyChan* do *Node* que fez o pedido) um *Channel* ***Give Access***.

Como o node não detém o objeto e satisfez o pedido, este transforma-se em ***Idle*** - **Idle(Find, MyChan, Link)**, em que apenas mantém o *Find*, o *MyChan* e o *Link*.

Idle

Receção de um pedido Access Request

O *Node* recebe um pedido ***Access Request*** no seu *Channel* ***Find***, que foi remetido pelo seu *Parent Node*.

Este envia pelo ***Link*** o ***WaiterChan*** do pedido ***Access Request*** e o seu *Channel* ***Find***.a É enviado o "Find" para que o seu *Child Node* tenha a possibilidade de inverter a sua ligação, ou seja, para que possa haver um arco do *Child Node* para o *Node*, e o "WaiterChan" para que se faça chegar o *MyChan* do *Node* que fez o pedido a um *Node Waiter* ou *Owner*.

Como o *Node* não tem o acesso ao objeto, este mantém-se como ***Idle***, e atualiza o ***Link*** (para ***NewLink***), que aponta para o **Find** do seu *Parent Node*. **Idle(find, MyChan, Link)**.

Realização de um pedido de acesso

O *Node* envia no ***Link*** o ***MyChan*** e o ***Find*** para o *Child Node*. O *MyChan* para que seja possível chegar o acesso ao objeto a este *Node*, e o *Link* para que o seu *Child Node* possa inverter a ligação.

Como fez um pedido, este transforma-se em ***Waiter Terminal***, e deixa de apresentar o ***Link*** - **WaiterTerminal(Find, MyChan)**.

Waiter Terminal

Receção de um pedido Access Request

O *Node* recebe um pedido ***Access Request*** no seu *Channel* ***Find***, que foi remetido pelo seu *Parent Node*.

Este envia pelo ***Link*** o ***WaiterChan*** do pedido ***Access Request*** e o seu *Channel* ***Find***.

É enviado o "Find" para que o seu *Child Node* tenha a possibilidade de inverter a sua ligação, ou seja, para que possa haver um arco do *Child Node* para o *Node*, e o "WaiterChan" para que se faça chegar o *MyChan* do *Node* que fez o pedido a um *Node Waiter* ou *Owner*.

Como o *Node* não tem o acesso ao objeto mas aguarda pelo acesso ao objeto, este transforma-se em ***Waiter With Request***, atualiza o ***Link*** (para ***NewLink***), que aponta para o **Find** do seu *Parent Node*, e passa a deter o ***WaiterChan***, que aponta para o *MyChan* do *Node* que realizou o pedido, isto para que seja possível o envio do acesso ao objeto. Como o *Node* passa a ter um pedido em espera, este sofre a transformação **WaiterWithRequest(Find, MyChan, NewLink, NewWaiterChan)**.

Receção acesso ao objeto

O *Node* recebe acesso ao objeto (**Obj**) no seu *Channel* ***MyChan***. Como o *Node* não tem pedidos, este transforma-se em ***Owner Terminal***. Como o *Node* é detentor do objeto, deixam de existir ligações a partir do *Node*, sofrendo a transformação **OwnerTerminal(Find, MyChan, Obj)** .

Waiter With Request

Receção de um pedido Access Request

O *Node* recebe um pedido ***Access Request*** no seu *Channel* ***Find***, que foi remetido pelo seu *Parent Node*.

Este envia pelo ***Link*** o ***WaiterChan*** do pedido ***Access Request*** e o seu *Channel* ***Find***.

É enviado o "Find" para que o seu *Child Node* tenha a possibilidade de inverter a sua ligação, ou seja, para que possa haver um arco do *Child Node* para o *Node*, e o "WaiterChan" para que se faça chegar o *MyChan* do *Node* que fez o pedido a um *Node Waiter* ou *Owner*.

Como o *Node* não tem o acesso ao objeto, aguarda pelo acesso ao objeto e tem um pedido em espera, este mantém-se como ***Waiter With Request***, porque ainda não satisfez o pedido que tem em espera, e atualiza o ***Link*** (para ***NewLink***), que aponta para o **Find** do seu *Parent Node* Sofre a transformação **WaiterWithRequest(Find, MyChan, NewLink, WaiterChan)**.

Receção acesso ao objeto

O *Node* recebe acesso ao objeto (**Obj**) no seu *Channel* ***MyChan***. Como o *Node* tem pedidos, este transforma-se em ***Owner With Request***, sendo a transformação **OwnerWithRequest(Find, MyChan, Obj, Link, WaiterChan)**.

3.7 Conclusões

Neste capítulo foram estudados todos os elementos que pertencem ao sistema, tais como os seus comportamentos e utilizades. A separação das várias propriedades destes elementos definidas na descrição serviu de auxílio para a implementação deste, com principal destaque o comportamento de cada tipo de nó e a mudança de estado após a despoletação deste.

Capítulo

4

Implementação

4.1 Introdução

Neste capítulo serão descritos todos os elementos envolvidos na implementação do protocolo, como as tecnologias utilizadas, em que é feita uma descirção da razão de escolha de cada uma destas a implementação do *Node* do sistema o foco deste capítulo é atribuído ao processo de desenvolvimento dos *Nodes* do sistemas, ou o programa que define um *Node*, pois o conjunto destes descreve o protocolo em questão, e por último a implementação da visualização, mais especificamente o tratamento de informação que estará disponível neste componente.

4.2 Linguagens de Programação

Tal como o título deste projeto sugere, ***Go*** (frequentemente referido como *Golang*) é a linguagem usada na implementação do protocolo. Para além de *Go*, também foi utilizada ***JavaScript*** como linguagem de implementação da visualização.

De seguida serão descritas as razões de utilização da linguagem Go e brevemente detalhes sobre o uso de *JavaScript*:

Go

A implementação de **sistemas concorrentes** em *Go* é **simples**, sendo a razão príncipal para a utilização desta tecnologia. Adicionando a palavra "*go*" antes de qualquer procedimento, esse procedimento irá correr em uma nova *Goroutine*, de forma concorrente em relação a todas as outras *Goroutines* já em

execução. Uma "Goroutine" é um *lightweight* ***thread*** gerido pelo *runtime* do *Go*.

Por exemplo, comparando (parcialmente) dois programas concorrentes, um em *Go* e outro em *Java*, que mostram os números inteiros de 0 a 10:

```
func main() {
  var wg sync.WaitGroup
  wg.Add(2)

  go count(&wg, "Goroutine-1")
  go count(&wg, "Goroutine-2")


  wg.Wait()
}

func count(wg *sync.WaitGroup, goroutineName string) {
  defer wg.Done()
  for i := 0; i < 10; i++ {
    fmt.Printf("Thread %s, %d\n", goroutineName, i)
    time.Sleep(time.Second * 40)
  }
}
```

Excerto de Código 4.1: Exemplo em *Go*, usando a *keyword* "go" para começar uma *Goroutine*.

```
class RunnableDemo implements Runnable {
   private Thread t;
   private String threadName;

   RunnableDemo( String name) {
      threadName = name;
   }
   public void run() {
      try {
         for(int i = 10; i < 10; i++) {
            System.out.println("Thread: " + threadName + ", " + i);
            Thread.sleep(40);
         }
      } catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
      }
   }
   public void start () {
      if (t == null) {
         t = new Thread (this, threadName);
         t.start ();
      }
```

```
    }
}

public class TestThread {

    public static void main(String args[]) {
        RunnableDemo R1 = new RunnableDemo( "Thread-1");
        R1.start();

        RunnableDemo R2 = new RunnableDemo( "Thread-2");
        R2.start();
    }
}
```

Excerto de Código 4.2: Exemplo em *Java*, usando a *interface* "Runnable" e uma classe"RunnableDemo" para começar *threads*.

Além da simplicidade de especificação de procedimentos concorrentes, a linguagem oferece bibliotecas de apoio a programas concorrentes, como a biblioteca "sync" que disponibiliza primitivas de sincronização simples (como por exemplo *WaitGroups* e *Mutex Locks*), e canais que permitem a comunicação e partilha de dados entre *Goroutines*.

A concorrência é inerente aos sistemas distribuídos, visto que os vários elementos do sistema executam de forma independente e em simultâneo.

Um exemplo relacionado com o tema deste projeto seria o caso em que um *Node* recebe vários pedidos de outros *Nodes*. De forma a manter o sistema (ou o diretório) consistente , o *Node* que recebeu os pedidos terá de os tratar de forma sincronizada, isto é, tem de endereçar um pedido de cada vez.

Na implementação fez-se uso de servidores e pedidos HTTP que, por omissão, todos os pedidos HTTP que o servidor recebe são tratados em *goroutines* diferente, de forma concorrente, o que permite a receção de vários pedidos e para a sincronização das *goroutines* usou-se um "Mutex" proveniente da biblioteca "sync".

JavaScript

Para a implementação da visualização foi usado **JavaScript**. Esta linguagem permite que a informação de páginas *Web* seja alterada após o seu carregamento. É usado no desenho de grafos e alteração de tabelas que dispõem a informação da visualização da rede.

4.3 Ferramentas e Bibliotecas utilizadas

Ferramentas

As ferramentas usadas durante a implentação deste projeto foram:

GoLand IDE especializado para *Go*. Inclui *Plugins* de *Debugging*, sugestão de código, etc.

VIM Editor de texto/conjunto de atalhos de teclado. Permite escrever texto de forma eficiente e apenas usando o teclado. Pode ser usado como *Plugin* no IDE *GoLand*.

GitHub Esta plataforma foi útil no controlo de versões do projeto e na partilha do código e especificação do protocolo.

Bibliotecas

Nesta secção irão ser referidas bibliotecas utilizadas na implementação e as suas funcionalidades.

gorilla/mux

Multiplexador de pedidos HTTP. Esta biblioteca da linguagem *Go* foi utilizada para simplificar a declaração de caminhos do servidor HTTP. É usada nos *Nodes* e no servidor HTTP da visualização.

D3.JS

A biblioteca *D3.JS*, em que *D3* significa "***D**ata-**D**riven **D**ocuments*", em português, "documentos dirigidos em dados", é usada para a representar gráficamente dados. No contexto deste projeto, esta é utilizada para a visualização da rede que estamos a testar, sendo que esta é representada como um grafo. Esta biblioteca permite a atualização periódica da representação do estado da rede de forma simples. Faz uso do elemento *SVG* do ***HTML!***, que é um *Standard*, o que permite a funcionalidade desta em grande maioria dos *Browsers* modernos, e é "leve", a qual nos possibilita uma grande taxa de atualização do grafo com dados mais recentes.

Outras Tecnologias

Docker

Docker é uma plataforma aberta/ferramenta construída de forma a tornar mais acessível a criação e execução de programas usando *containers*.

Estes *containers* podem ser comparados com *Virtual Machines*, ambos tendo o mesmo propósito, mas os *containers Docker* são mais leves, mais rápidos e portáveis.

No entanto, esta tecnologia foi utilizada para simular uma rede distribuída, em que cada *container* simula um *Node* ou uma máquina que cada um tem uma instância do programa a correr, o seu próprio endereço IP e que podem comunicar entre si.

4.4 Classe *Node*

A Classe *Node* desempenha a função de armazenar o estado atual do próprio *Node*, define os métodos/procedimentos que este pode executar, e uma enumeração dos 5 diferentes tipos de *Nodes*.

Nesta Implementação, esta classe está incluída num módulo *Go* "Nodes" (Ou seja, num diretório com o mesmo nome), e é constituído por 5 ficheiros, sendo que o código está distribuído por entre os ficheiros da seguinte forma:

Node.go - Contém *Struct* que define os **atributos**.

NodeBehaviours.go - Define os possíveis **comportamentos**.

NodeCommunications.go - Conjunto de **métodos de comunicação** de informação para outros *Nodes*.

NodeTranformations.go - **Transformações**/Mudanças de tipo que o *Node* pode sofrer. A diferenciação destes comportamentos deve-se ao facto de estes mudarem o estado atual do *Node*.

NodeType.go - Enumeração dos **tipos** que o *Node* pode ser.

4.5 Atributos da Classe

Como referido no capítulo de Especificação, um *Node* tem, no máximo 5 atributos, no entanto, na sua implementação este incluí no total 8, tendo a mais os atributos "MyAddress", "Type" e "VisAddress". Esta é a definição da *Struct* dos atributos do *Node*.

Este código está presente no ficheiro Node.go.

```
type Node struct {
  Type        NodeType //Tipo do Node, ver Tipos de Nodes
  MyChan      string   //Channel onde recebe acesso ao objeto
  Find        string   //Channel onde recebe pedidos
  Link        string   //Ligacao para o child Node
  WaiterChan  string   //Channel do Node que esta na posicao seguinte da
      fila
  MyAddress   string   //Endereco do Node
  VisAddress  string   //Endereco para onde envia o seu estado atual para
      a atualizacao da visualizacao
  Obj         bool     //Se tem objeto ou nao
}
```

Excerto de Código 4.3: Definição da estrutura *Node*

Na *Struct* estão definidos todos os atributos que o *Node* pode ter, porém, quando um atributo é inexistente, este é definido como vazio, ou seja, os atributos "WaiterChan", "VisAddress" e "Link" podem ser *strings* vazias.

O atributo **"MyChan"** é o *Channel* do *Node* onde este vai receber o acesso ao objeto. Assim, é um dos componentes da mensagem (*Struct*) que irá ser enviada para o *Child Node* quando este decide pedir o acesso ao objeto.

O atributo **"Find"** é o *Channel* do *Node* onde este vai receber pedidos de acesso de outros *Nodes*. Também é usado na construção dos dois tipos de pedidos, no pedido de *AccessRequest* quando o *Node* decide fazer um pedido e quando este reencaminha um pedido para o *Child Node*, ao construir um novo pedido de *AccessRequest*, mantendo o "WaiterChan" mas substituindo o "Link" do pedido pelo seu "Find".

O atributo **"Link"** é o *Channel "Find"* do *Child Node*. Pode existir ou não, caso não exista este é representado como uma *string* vazia. Este é usado para o reencaminhamento e difusão de quaisquer pedidos "AccessRequest", quer estes sejam construídos pelo *Node* ou pedidos que chegaram ao seu "Find".

O atributo **"WaiterChan"** é o *Channel "MyChan"* do *Node* que espera pelo acesso ao objeto. Pode existir ou não, caso não exista este é representado como uma *string* vazia. Este é usado na atribuição do objeto ao *Node* que está à espera do acesso ao objeto, por parte do *Node*.

O atributo **"MyAddress"** é o endereço IP do próprio Node. Este é usado para identificação do *Node* na rede e para a construção dos *Channels "Find"* e *"MyChan"*, pois este é usado na inicialização do servidor HTTP do *Node*.

O atributo **"VisAddress"** é o URL usado para a atualização do estado do *Node* na visualização.

O atributo **"Obj"** apenas indica se o *Node* tem acesso ou não ao objeto. Este atributo é redundante, pois a mesma informação pode ser adquirida a partir do tipo do node ("Type"), em que, caso o *Node* seja do tipo *OwnerTer-*

minal ou *OwnerWithRequest*, este tem o acesso ao objeto.

4.6 Inicialização do objeto "Self Node"

Nesta secção será descrito o processo de inicialização do objeto "Self Node", isto é, a definição dos atributos do *Node* que o programa irá definir.

Para tal, os atributos do *Node* provêm de argumentos da linha de comandos, isto para facilitar a inicialização de *Nodes* sem interface gráfica ou com uso de "scripts"/"Dockerfiles" (Ficheiros de construção de imagens *Docker*).

Na linguagem "Go" existe uma função especial, "init", que é sempre executada (e a primeira) no carregamento de um módulo *Go*, que foi usada para garantir que a inicialização do *Node* esteja completa antes da execução de qualquer outra função.

Nesta função, é instanciado o objeto "selfNode" da classe "Node", e é armazenado o ponteiro para o mesmo, para que possa haver alteração do mesmo por parte dos seus métodos.

Os argumentos da linha de comandos são analisados/*Parsed* e transformados/*Casted* de texto para o tipo correspondente ao do atributo (por exemplo, o tipo é *Casted* de texto para um inteiro), sendo que os únicos atributos não opcionais são o endereço ("MyAddress") do *Node* e o seu tipo.

```
selfNode = new(Nodes.Node) //Instanciacao do ''selfNode'' e
    armazenamento do ponteiro para o mesmo

// \emph{Parsing} dos argumentos da linha de comandos
// as funcoes do modulo ''flag'' devolvem ponteiros para a
    instanciacao de cada argumento \emph{Parsed}
myAddress := flag.String("address", "", "Node's Address (Required)")
myType := flag.Int("type", -1, "Node's Type (0-4)(Required)") //Por
    definicao de todos os tipos
visAddress := flag.String("visualization", "", "Visualization address.
    ")
link := flag.String("link", "", "Link")
requests := flag.Bool("requests", true, "If this Node, when Idle,
    preforms Object Requests")

// Chamado apos todas as ''flags'' serem definidas, para que seja
    processado o \emph{Parsing}
flag.Parse()
```

Excerto de Código 4.4: *Parsing* dos argumentos da linha de comandos.

No entanto, atributos como "Link", "MyChan", "Find", e "Type" que dependem do endereço dos *Nodes*, que apenas são formados se o *Parsing* for bem sucedido.

Dependendo do tipo do *Node*, irão ser executas instruções para garantir o bom funcionamento do diretório/simulação, como por exemplo, caso o *Node* seja do tipo "OWNER", é definido que este tem o objeto implicitamente, ou caso seja do tipo "IDLE" irá ser inicializada uma *goroutine* que executará o método "AutoRequest" do *Node*, para que este tenha a possibilidade de realizar pedidos automáticamente/aleatoriamente.

```
selfNode.MyAddress = *myAddress

// instanciacao dos \emph{Channels ``Find''} e \emph{``MyChan''}
// concatenacao de ``http://'' que indica ao \emph{Node} que
    transmitira o pedido que este deve ser um pedido \acs{HTTP}
// concatenacao dos caminhos ``/find'' e ``/myChan'' para indicar aos
    \emph{Nodes} para que \emph{Channel} sera enviado

selfNode.MyChan = fmt.Sprintf("http://%s/myChan", *myAddress)
selfNode.Find = fmt.Sprintf("http://%s/find", *myAddress)

selfNode.Type = Nodes.NodeType(*myType)



selfNode.VisAddress = fmt.Sprintf("http://%s", *visAddress)

selfNode.Link = *link
if *link != "" {
  selfNode.Link = fmt.Sprintf("http://%s/find", *link)
}

// Caso seja do tipo ``OWNER'', o \emph{Node} tem o acesso ao objeto
if selfNode.Type == Nodes.OWNER_TERMINAL || selfNode.Type == Nodes.
    OWNER_WITH_REQUEST {
  selfNode.Obj = true


  //caso seja ``IDLE'', sera executado o metodo ``AutoRequest'' numa
      nova \emph{Goroutine}
} else if selfNode.Type == Nodes.IDLE && *requests{
  go selfNode.AutoRequest()
}
```

Excerto de Código 4.5: Instanciação dos atributos do *Node*.

4.7 Tipos de Nodes

Os 5 tipos de *Nodes* foram definidos em uma enumeração de constantes inteiras.

```
type NodeType int


const (
  OWNER_WITH_REQUEST NodeType = iota //0
  OWNER_TERMINAL                     //1
  IDLE                               //2
  WAITER_WITH_REQUEST                //3
  WAITER_TERMINAL              //4
)
```

Excerto de Código 4.6: Definição da enumeração dos tipos de *Node*

4.8 Comportamentos

Como referido no capítulo de Especificação 3.4, o *Node* pode ter vários comportamentos, que dependem do seu tipo e de fatores que os desencadeiam, como por exemplo receber um pedido de acesso ou o acesso ao objeto e o próprio *Node* (no caso desta implementação) tomar decisões, como ceder o acesso ao objeto ou pedir o mesmo.

Este código está presente no ficheiro NodeBehaviours.go .

Receção de um pedido Access Request

Todos os tipos de *Nodes* têm a possibilidade de receber um pedido de *Access-Request*, no entanto, o comportamento (e transformação) desencadeado por este evento é diferente entre tipos.

Quando o *Node* recebe um pedido *Access Request* (*Handler* "findRoute" no ficheiro "controller.go") o e método "HandleFind" da classe *Node* é executado. O objeto de entrada provêm da descodificação dos dados transmitidos pelo *Parent Node*.

Na implementação deste método, foi usada a estrutura condicional "Switch" para escolher que comportamento será tomado dependendo do tipo atual do "selfNode".

O acesso à secção crítica deste método é sincronizada com o uso do *Mutex* "Mutex", isto é, o acesso ao estado atual do "selfNode" só é adquirido por uma *goroutine* de cada vez.

O pedido de *Access Request* recebido, "accessRequest" é copiado para um novo objeto "newAccessRequest", quer irá ser utilizado na construção de um novo "AccessRequest", caso o *Node* redirecione o pedido para o seu *Child Node*.

O último procedimento a ser executado neste método é o método "UpdateVisualization" da classe "Node", em uma nova *goroutine*. Este método é usado para atualizar o estado atual do *Node* na visualização.

```
Mutex.Lock() // fecho do Mutex
defer Mutex.Unlock() // a primitiva defer indica que o codigo de
    abertura do Mutex sera corrido caso a execucao deste metodo termine

newAccessRequest := accessRequest // copia do pedido

// Decisao do comportamento que depende do tipo "Type" atual do Node
switch node.Type {
  case OWNER_TERMINAL:
  case OWNER_WITH_REQUEST:
  case IDLE:
  case WAITER_TERMINAL:
  case WAITER_WITH_REQUEST:
  }
go node.UpdateVisualization()
```

Excerto de Código 4.7: *Switch* de decisão do comportamento.

Será feita uma descrição do comportamento que cada tipo de *Node* pode ter, isto é, a implementação de cada caso do "Switch".

Caso OWNER_TERMINAL

Caso o tipo seja "OWNER_TERMINAL", o *Node*, o método "OwnerWithRequest" da classe *Node* irá ser executado, que transforma o *Node* em *Owner With Request*.

Os parâmetros de entrada desse método serão o "Link" e "WaiterChan" (do atributo "GiveAccess" do pedido), que correspondem ao "Find" do *Parent Node* e ao "MyChan" do *Node* que fez o pedido, respetivamente. Isto é, a ligação entre o *Node* e o transmissor do pedido inverte-se, e o *Node* passa a ter um *Node* em espera.

Após a transformação, será executado o método "releaseObj" do *Node*, que irá despoletar o comportamento de Cedência do Objeto, em uma nova *goroutine*.

```
node.OwnerWithRequest(accessRequest.Link, accessRequest.GiveAccess.
    WaiterChan) // transfomacao em Owner With Request
go node.releaseObj() //comportamento de Cedencia do Objeto
```

Excerto de Código 4.8: Comportamento do *Node* tipo *Owner Terminal* caso receba um pedido *Access Request* no *Channel "Find"*

Caso OWNER_WITH_REQUEST

Caso o tipo seja "OWNER_WITH_REQUEST", o Atributo "Link" do objeto "newAccessRequest" será substituído pelo "Find" do *Node*, para o seu *Child Node* inverter a ligação.

Após feita esta alteração do objeto, o *Node* envia o objeto "newAccessRequest" pelo seu "Link" para o seu *Child Node*.

Por último, o método "OwnerWithRequest" da classe *Node* irá ser executado, em que o *Node* mantém o tipo mas é feita uma alteração do seu "Link" para o "Find" que provêm do pedido "Access Request" que recebeu, isto é, para o "Find" do seu *Parent Node*, invertendo a ligação, mas mantendo o *WaiterChan* porque o *Node* que fez o pedido ainda não tem o acesso ao objeto.

```
//alteracao do ''Link'' do novo pedido para o ''Find'' do \emph{Node}
newAccessRequest.Link = node.Find
// Transmissao do ''newAccessRequest'' pelo ''Link''
node.SendThroughLink(newAccessRequest)
// Transformacao \emph{OwnerWithRequest}, que mantem o ''WaiterChan'',
    mas atualiza o ''Link''
node.OwnerWithRequest(accessRequest.Link, node.WaiterChan)
```

Excerto de Código 4.9: Comportamento do *Node* tipo *Owner With Request* caso receba um pedido *Access Request* no *Channel "Find"*

Caso IDLE

Caso o tipo seja "IDLE", o Atributo "Link" do objeto "newAccessRequest" será substituído pelo "Find" do *Node*, para o seu *Child Node* inverter a ligação.

Após feita esta alteração do objeto, o *Node* envia o objeto "newAccessRequest" pelo seu "Link" para o seu *Child Node*.

Por último, método "Idle" da classe *Node* irá ser executado, que mantém o tipo do *Node*, mas é feita a alteração do seu "Link" para o "Find" que provêm do pedido "Access Request" que recebeu, ou seja, para o "Find" do seu *Parent Node*, que inverte a ligação.

```
//alteracao do ''Link'' do novo pedido para o ''Find'' do \emph{Node}
newAccessRequest.Link = node.Find
// Transmissao do ''newAccessRequest'' pelo ''Link''
node.SendThroughLink(newAccessRequest)
// Transformacao \emph{Idle}, que atualiza o ''Link''
node.Idle(accessRequest.Link)
```

Excerto de Código 4.10: Comportamento do *Node* tipo *Idle* caso receba um pedido *Access Request* no *Channel "Find"*

Caso WAITER_TERMINAL

Caso o tipo seja "WAITER_TERMINAL", o *Node* o método "WaiterWithRequest" do *Node* será executado, que transforma o *Node* em *WAITER_WITH _REQUEST*.

Os parâmetros de entrada são o "Link" e "WaiterChan" (do atributo "GiveAccess" do pedido), que correspondem ao "Find" do *Parent Node* e ao "MyChan" do *Node* que fez o pedido, respetivamente. O "Link" é o "Find" do *Parent Node*, o que causa a inversão da ligação.

```
// Transformacao \emph{OwnerWithRequest}, que mantem o ''WaiterChan'',
    mas atualiza o ''Link''.
node.WaiterWithRequest(accessRequest.Link, accessRequest.GiveAccess.
    WaiterChan)
```

Excerto de Código 4.11: Comportamento do *Node* tipo *Waiter Terminal* caso receba um pedido *Access Request* no *Channel "Find"*

Caso WAITER_WITH _REQUEST

Caso o tipo seja "WAITER_WITH _REQUEST", o Atributo "Link" do objeto "newAccessRequest" será substituído pelo "Find" do *Node*, para que o *Child Node* possa inverter a sua ligação.

Após feita esta alteração do objeto, o *Node* envia o objeto "newAccessRequest" pelo seu "Link" para o seu *Child Node*.

Por último, este sofre uma transformação. O método "WaiterWithRequest" da classe *Node* irá ser executado, em que é feita uma alteração do seu "Link" para o "Find" que provêm do pedido "Access Request" que recebeu, ou seja, para o "Find" do seu *Parent Node*, mas mantêm o *WaiterChan* porque o *Node* que fez o pedido ainda não tem o acesso ao objeto. O "Link" é o "Find" do *Parent Node*, o que causa a inversão da ligação.

```
newAccessRequest.Link = node.Find // Alteracao do atributo Find do
    objeto newAccessRequest para o Find do Node
node.SendThroughLink(newAccessRequest) // Envio do objeto
    newAccessRequest pelo link
node.WaiterWithRequest(accessRequest.Link, node.WaiterChan) //
    transformacao do Node. Mantem-se o WaiterChan mas altera-se o Link
```

Excerto de Código 4.12: Comportamento do *Node* tipo *Waiter With Request* caso receba um pedido *Access Request* no *Channel "Find"*

Cedência do Objeto

No caso do *Node* ser do tipo "WAITER_TERMINAL" e receber um pedido "Access Request", este sofre uma transformação, muda de tipo para "WAITER_WITH_REQUEST" e passou a deter o atributo "WaiterChan" ("MyChan" do *Node* que fez o pedido).

Esta secção de código é executada concorrentemente (numa *goroutine*) com qualquer outras *goroutines* que estejam a ser executadas, isto para permitir que o *Node* receba outros pedidos, que os transmita, e que outros comportamentos ou transformações possam ocorrer enquanto este espera para ceder o acesso, isto é, enquanto que o *Node* é "OWNER_WITH_REQUEST".

Referente a um caso real, o *Node* transmitiria o objeto pelo *Channel "WaiterChan"*, quando, por exemplo, o acesso a este não fosse mais necessário. No entanto, por questões de se pretender criar uma simulação deste protocolo, o *Node* decide ceder o acesso ao objeto após um tempo aleatório (entre 1 a 2 segundos) depois de receber o pedido para o seu acesso.

É executada a primitiva "defer" com o método "Unlock" do objeto "Mutex" para, quando esta função terminar, o *Mutex* ser desbloqueado, para que outras *goroutines* possam aceder ao estado atual do objeto.

```
defer Mutex.Unlock() // o ''Mutex'' e desbloqueado quando a execucao
    deste metodo terminar
randomSleep := utils.RandomRange(1, 2) // Gera um numero aleatorio,
    neste caso, 1 ou 2
time.Sleep(time.Second * time.Duration(randomSleep)) // A \emph{
    goroutine} espera durante o tempo aleatorio gerado (em segundos)
```

Excerto de Código 4.13: *Node* espera 1 ou 2 segundos antes de ceder o objeto.

De seguida é criado um objeto da classe/tipo "GiveAccess" que será transmitido para o *Node* em espera através do "WaiterChan".

```
accessObject := Channels.GiveAccess{WaiterChan: node.WaiterChan}
```

Excerto de Código 4.14: Criação do objeto "accessObject", da classe "GiveAccess"

Este método ("releaseObj"), contém uma secção crítica, pois acede ao objeto "selfNode". O acesso à secção crítica deste método é sincronizada com o uso do *Mutex* "Mutex", ou seja, o acesso ao estado atual do "selfNode" só é adquirido por uma *goroutine* de cada vez, para prevenir a alteração do estado enquanto que o *Node* transmite o acesso ao objeto, e para que a transformação (método "Idle") ocorra de seguida ao *Node* deixar de ter o acesso ao objeto.

Quando tiver a possibilidade de aceder à secção crítica, este irá transmitir o acesso do objeto ao *Node* em espera. De seguida, como já não possui o

acesso ao objeto, este sofre um transformação, mudando-se para um *Node* do tipo "IDLE", mantendo o "Link".

Como o *Node* transformou-se em "IDLE", é inicializada uma *goroutine* que executará o método "AutoRequest". Este método desempenha a função de decidir se o *Node* faz um pedido de acesso. No entanto, este é descrito num capítulo diferente.

De seguida, o procedimento a ser executado neste método é o método "UpdateVisualization" da classe "Node", em uma nova *goroutine*. Este método é usado para atualizar o estado atual do *Node* na visualização.

```
Mutex.Lock() //Pedido de acesso a seccao critica
node.SendObjectAccess(accessObject) //Transmissao do objeto, atraves
    do envio do objeto ''accessObject''
node.Idle(node.Link) //transformacao em ''IDLE'', mas mantendo o ''
    Link''
go node.AutoRequest() // goroutine que decidira se o \emph{Node} faz
    um pedido de acesso
go node.UpdateVisualization() // atualiza o estado do Node na
    visualizacao
```

Excerto de Código 4.15: Acesso à secção crítica, transmição do Objeto, transformação em *Node* "IDLE", *goroutine* de decisão de pedido, e atualização na visualização

Realização de um pedido de acesso

No caso do *Node* ser do tipo "IDLE", este tem a possibilidade de pedir o acesso ao objeto. Qualquer outro tipo de *Node* não pode fazer pedidos de acesso.

Neste método há acesso ao estado atual do *Node*, logo faz-se uso de um "Mutex" para o acesso ser sincronizado.

Como, a partir da visualização e da "Shell" do *Node* é possível forçar o *Node* a realizar um pedido, quando a *goroutine* acede à secção crítica do método (acesso ao estado atual do *Node*) é verificado se o tipo do *Node* é "IDLE".

Caso seja do tipo "IDLE", irá ser instanciado um objeto do tipo "Channels.AccessRequest", em que o atributo "Link" contém o *Channel "Find"* do próprio *Node*, ou seja, o URL do método onde o *Node* recebe os pedidos "AccessRequest", e o atributo "GiveAccess"(do tipo "Channels.GiveAccess"), que contém o *Channel "MyChan"* do próprio *Node*, o URL do método onde o *Node* recebe o acesso ao objeto. Depois da instanciação, é executado o método "SendThroughLink" do *Node*, que envia o objeto pelo "Link" do *Node* para o seu *Child Node*.

O "Link" deste objeto servirá para o *Child Node* do *Node* inverter a ligação, isto é, para que o *Child Node* possa fazer a ligação de volta para o *Node*.

O atributo "GiveAccess" será usado pelo próximo *Node Terminal* (quer este seja *Owner* ou *Waiter*), para quando esse próximo *Node* obtiver acesso ao objeto, este redirecioná-lo para o *Node* que realizou o pedido.

Como o *Node* realizou um pedido de acesso e espera pelo acesso ao objeto, este transforma-se em "WAITER_TERMINAl", deixando de ter "Link".

Por último, o *Node* atualiza o seu estado na visualização em uma nova *go-routine*.

```
func (node *Node) Request() {
  Mutex.Lock() // Sincronizacao do acesso a seccao critica
  defer Mutex.Unlock() // O metodo ''Unlock'' do objeto ''Mutex'' sera
      executado caso o metodo ''Request'' termine

  //Existe para evitar:
  //que ou o utilizador faca um request e o node ja mudou de tipo
  //que se faca um request a partir do metodo do Node de pedidos remotos
  if node.Type != IDLE {
    fmt.Printf("Can't request an object if not Idle.")
    return
  }
  fmt.Printf("Requesting.")

  //Instanciacao do pedido de acesso
  accessRequest := Channels.AccessRequest{
    GiveAccess: Channels.GiveAccess{
      WaiterChan: node.MyChan,
    },
    Link: node.Find,
  }

  //Envio do pedido de acesso
  node.SendThroughLink(accessRequest)


  //transfomacao em WaiterTerminal, visto que este espera pelo acesso
  node.WaiterTerminal()

  //atualizacao do estado atual do Node na visualizacao
  go node.UpdateVisualization()
}
```

Excerto de Código 4.16: Método "Request"

É possível o utilizador provocar a realização do pedido pelo *Node*. Pode ser feito a partir de uma "Shell" que é iniciada com o programa, ou através da visualização, ao clicar duas vezes no *Node* do grafo correspondente ao *Node* em questão.

No entanto, por questões de simulação do diretório, caso o estado inicial

do *Node* ou este mude de tipo para "IDLE", é iniciada uma *goroutine* que executa o método "AutoRequest" da classe *Node*.

Neste método existe um ciclo "infinito", cuja a única condição de saída é o *Node* realizar o pedido.

A cada ciclo é gerado um valor aleatório, que será, em segundos, o tempo que esta *goroutine* irá esperar.

Após a espera, um outro valor aleatório é gerado (0 ou 1), que indicará se o *Node* irá fazer um pedido (ou seja, se será executado o método "Request" da classe "Node").

Caso faça o pedido (seja o valor 1), o ciclo irá terminar. Caso contrário, o *Node* terá de esperar um tempo aleatório até ao próximo ciclo.

```
func (node *Node) AutoRequest() {
  var randomSleep int

  // Ciclo infinito
  for {

    randomSleep = utils.RandomRange(5, 15) // E gerado um numero inteiro
         aleatorio entre 5 e 15

    fmt.Printf("\nTrying to Request the Object in %d seconds.",
        randomSleep)


    // A \emph{goroutine} espera durante o valor de ``randomSleep'' (em
        segundos)
    time.Sleep(time.Second * time.Duration(randomSleep))

    //E gerado um numero inteiro, 0 ou 1.
    //Caso o valor seja 1, o \emph{Node} ira realizar um pedido de
        acesso e o ciclo termina
    if requests := utils.RandomRange(0, 1); requests > 0 {
      node.Request()
      break
    } else {
      //Caso seja 0, o \emph{Node} ira gerar um numero inteiro aleatorio
           entre 5 e 20
      cooldown := utils.RandomRange(5, 20)
      fmt.Printf("Didn't request. Retrying in %d seconds.", cooldown)
      // A \emph{goroutine} espera durante o valor de ``cooldown'' (em
          segundos)
      time.Sleep(time.Second * time.Duration(cooldown))
    }
  }

}
```

Excerto de Código 4.17: Método “AutoRequest”

Este método permite que *Node* realize pedidos em tempo aleatório com uma chance de 50%, ou seja, há a possibilidade de o *Node* não realizar um pedido.

Receção acesso ao objeto

Caso o *Node* seja do tipo *WAITER_TERMINAL* ou *WAITER_WITH_REQUEST* este pode receber o acesso ao objeto.

Quando o *Node* recebe um pedido *GiveAccess* (*Handler* “myChanRoute” no ficheiro “controller.go”) o método “ReceiveObj” da classe *Node* é executado. O objeto de entrada do método provêm da descodificação dos dados transmitidos pelo *Owner* que cedeu o acesso ao objeto, isto é, uma desserialização dos dados provenientes de um método HTTP “POST” num objeto do tipo “GiveAccess”.

Como neste método, é feito um acesso ao estado atual do *Node* e há transformações, o acesso a esta secção crítica é sincronizada com o uso de um *Mutex*. Para tal, é executado o método “Lock” do objeto “Mutex”. No fim da execução deste método é necessário desbloquear o “Mutex”, para tal a primitiva “defer” é usada, que executará o método “Unlock” do objeto “Mutex”.

Como este programa se trata de uma simulação, este objeto não tem qualquer uso, no entanto, num caso de uso, este objeto seria um *Channel* de comunicação com um ficheiro/base de dados, ou qualquer outro objeto em que o seu acesso seria sincronizado.

Quando o *Node* é “WAITER_TERMINAL”, ao receber o acesso ao objeto, este transforma-se em “OWNER_TERMINAL”, pois não tem nenhum outro *Node* em espera.

Quando o *Node* é “WAITER_WITH_Request”, ao receber o acesso ao objeto, este transforma-se em “OWNER_WITH_REQUES”, pois ainda tem outro *Node* em espera. Com o *Node* tem outro *Node* à espera do acesso, é inicializada uma *goroutine* que irá executar o método “releaseObj” da classe *Node*.

Ao final, é inicializada uma *goroutine* que executará o método “UpdateVisualization”, para que o estado do *Node* seja atualizado na visualização.

```
func (node *Node) ReceiveObj(giveAccess Channels.GiveAccess) {
  Mutex.Lock() // Sincronizacao do acesso a seccao critica
  defer Mutex.Unlock() // O metodo ''Unlock'' do objeto ''Mutex'' sera
      executado caso o metodo ''Request'' termine

  fmt.Printf("Received Access:")
  fmt.Println(giveAccess)
```

```
  switch node.Type {
  case WAITER_TERMINAL:
    node.OwnerTerminal()
    break
  case WAITER_WITH_REQUEST:
    node.OwnerWithRequest(node.Link, node.WaiterChan)
    go node.releaseObj()
    break
  }

  go node.UpdateVisualization()

}
```

Excerto de Código 4.18: Método "ReceiveObj"

4.9 Transformações do *Node*

Neste capítulo serão descritas as transformações que o *Node* pode sofrer, tais como as suas implementações.

O *Node* sofre transformações quando este executa qualquer comportamento, no entanto, uma transformação não significa uma mudança de tipo, mas uma mudança de estado.

Estas são as possíveis transformações que o *Node* pode sofrer:

O nome dos métodos das transformações provêm do nome to tipo para o qual se vai mudar ou manter.

Idle

Se o *Node* é do tipo "IDLE" e este recebe um pedido de acesso, após transmitir o pedido para o seu *Child Node*, sofre a transformação "Idle" (mantém o tipo) mas atualiza o *Link* para o *Find* do seu *Parent Node*. O novo "Link" ("NewLink") é o parâmetro de entrada deste método.

Se o *Node* é do tipo "OWNER_WITH_REQUEST" mas cedeu o acesso ao objeto, visto que este já não o possui o *Node* transforma-se em "IDLE", mas mantém o *LINK*, isto é, o seu "Link" atual é usado como parâmetro de entrada deste método.

```
func (node *Node) Idle(newLink string) {
  node.Type = IDLE     //Alteracao do tipo para ''IDLE''
  node.Link = newLink //Atualizacao do ''Link''
  node.Obj = false     //Caso seja ''OWNER\_WITH\_REQUEST'', deixa de ter
       acesso ao obj
  node.WaiterChan = ""//Caso seja ''OWNER\_WITH\_REQUEST'', deixa de ter
       o ''Node'' em espera
```

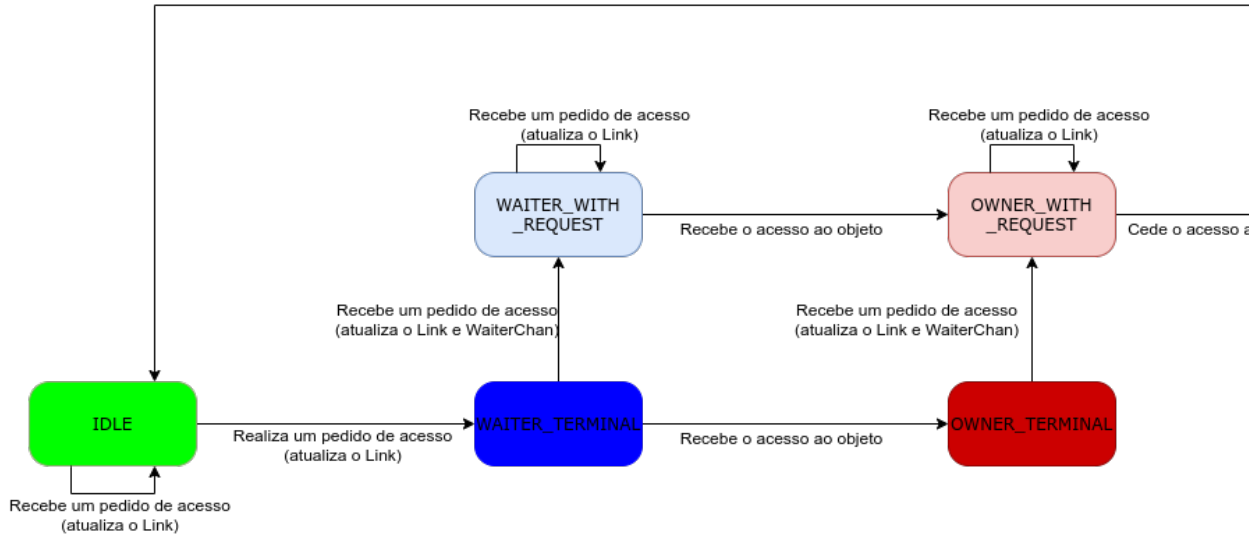

Figura 4.1: Diagrama de Estados do *Node*

Nota: O estado inicial do *Node* não está presente porque este pode começar em qualquer estado/tipo.

```
}
```

Excerto de Código 4.19: Método/transformação "Idle"

WaiterTerminal

Se o *Node* é do tipo "IDLE" e realizar um pedido de acesso, este sofre a transformação "WaiterTerminal". Como foi o *Node* que realizou o pedido de acesso, este não aponta para nenhum outro *Node* (o "Link" passa a vazio/nulo).

```
func (node *Node) WaiterTerminal() {
  node.Type = WAITER_TERMINAL //Alteracao do tipo para ''WAITER\
      _TERMINAL''
  node.Link = ""            // Como foi o \emph{Node} quem realizou o
      pedido, este nao aponta para nenhum outro \emph{Node}
  node.WaiterChan = ""      //redundante
}
```

Excerto de Código 4.20: Método/transformação "WaiterTerminal"

OwnerTerminal

Se o *Node* é do tipo "WAITER_TERMINAL" e receber o acesso ao objeto, como não recebeu qualquer pedido, este mantém-se sem "Link" ou "WaiterChan",

então sofre a esta transformação, mudando o tipo para "OWNER_TERMINAL", pois tem acesso ao objeto mas não tem qualquer pedido.

```
func (node *Node) OwnerTerminal() {
  node.Type = OWNER_TERMINAL //Alteracao do tipo para ''OWNER\{_}
      TERMINAL''
  node.Link = ""
  node.Obj = true          //Como se transformou em ''OWNER\{_}TERMINAL''
      significa que passou a deter o acesso ao objeto
  node.WaiterChan = ""        //Redundante, mas um \emph{Node} deste tipo
       nao tem \emph{Node} a espera do acesso ao objeto
}
```

Excerto de Código 4.21: Método/transformação "OwnerTerminal"

OwnerWithRequest

Se o *Node* é do tipo "OWNER_TERMINAL" e receber um pedido de acesso, como este recebeu um pedido de acesso, o "Link" é atualizado e como este é detentor do acesso ao objeto, o *Node* passa a ter um pedido em espera, logo o "WaiterChan" é atualizado para o "MyChan" do *Node* em espera. Então sofre a transformação "OwnerWithRequest", mudando o tipo para "OWNER_WITH_REQUEST", sendo o "NewLink" e o "WaiterChan" os valores de entrada do método.

Caso o *Node* seja do tipo "OWNER_WITH_REQUEST", ao receber um pedido de acesso, o "Link" é atualizado, pois, mesmo sendo o *Node* com acesso ao objeto, este objeto será cedido a outro *Node* e não ao *Node* que realizou o pedido.

```
  func (node *Node) OwnerWithRequest(newLink string, waiterChan string)
      {
    node.Type = OWNER_WITH_REQUEST
    node.Link = newLink
    node.Obj = true
    node.WaiterChan = waiterChan
  }
```

Excerto de Código 4.22: Método/transformação "OwnerWithRequest"

WaiterWithRequest

Se o *Node* é do tipo "WAITER_TERMINAL" e receber um pedido de acesso, como este recebeu um pedido de acesso, o "Link" é atualizado e como este será detentor do acesso ao objeto, o *Node* passa a ter um pedido em espera, logo o "WaiterChan" é atualizado para o "MyChan" do *Node* em espera. Então sofre a transformação "WaiterWithRequest", mudando o tipo para "WAI-

TER_WITH_REQUEST", sendo o "NewLink" e o "WaiterChan" os valores de entrada do método.

Caso o *Node* seja do tipo "WAITER_WITH_REQUEST", ao receber um pedido de acesso, o "Link" é atualizado, pois, mesmo que este *Node* terá o acesso ao objeto, este objeto será cedido a outro *Node* (do primeiro pedido que transformou o *Node* em' 'WAITER_WITH_REQUEST") e não ao *Node* que realizou este pedido.

```
func (node *Node) WaiterWithRequest(newLink string, waiterChan string)
    {
  node.Type = WAITER_WITH_REQUEST
  node.Link = newLink
  node.WaiterChan = waiterChan
}
```

Excerto de Código 4.23: Método/transformação "WaiterWithRequest"

4.10 Comunicação entre Nodes

Neste capítulo serão tratadas as conexões que os *Nodes* fazem entre si, Como referido anteriormente, é feito uso do protocolo HTTP nas ligação/pedidos entre *Nodes*, pois este é um "industry standard" (ou seja, o protocolo mais usado neste tipo de sistemas). Para tal, cada *Node* inicia um servidor HTTP que espera pela entrada de pedidos de HTTP, isto é, espera que outros *Nodes* transmitam pedidos do tipo "AccessRequest" e "GiveAccess". A implementação do servidor HTTP está presente no módulo "controller" (no ficheiro "controller.go").

No entanto, também foram implementados métodos para permitir ao *Node* a realização de pedidos, ou seja, de poder enviar informação para outros *Nodes*. A implementação destes pedidos HTTP está presente no módulo "Node" (no ficheiro "NodeCommunications.go").

Servidor HTTP

Após a inicialização do programa do *Node*, é inicializado o servidor HTTP, executando a função "StartServer", em que é criado um objeto do tipo "Router" (roteador), no qual são registados os vários caminhos e *Handlers*(manipuladores) destes caminhos. Isto é, são registados os métodos deste servidor e as funções que serão executadas caso sejam feitos pedidos em cada um desses métodos.

Os métodos e *Handlers* registados são:

/find findRoute
Saida do *Channel "Find"* do *Node*

/myChan myChanRoute
Saida do *Channel "MyChan"* do *Node*

/remoteRequest remoteRequest
Metodo usado na visualizacao para forcar o *Node* a realizar um pedido accessRequest

Por fim, é executado a função "ListenAndServe" do módulo "http", que escuta por pedidos de *HTTP* de entrada. Os parâmetros de entrada deste método são o endereço do *Node* ("MyAddress") e o "Router" instanciado.

```
  r := mux.NewRouter() //Instanciacao do objeto r da classe ''Router''

  // Registo dos caminhos e \emph{Handlers}
  r.HandleFunc("/find", findRoute).Methods("POST")
  r.HandleFunc("/myChan", myChanRoute).Methods("POST")
  r.HandleFunc("/remoteRequest", remoteRequest).Methods("GET")


  // Inicializacao do servidor
  if err := http.ListenAndServe(selfNode.MyAddress, r); err != nil {
    log.Fatal(err)
  /* Caso ocorra um erro (err != nil), a execucao e terminada,
    o programa termina a execucao e e mostrado o erro.
  */
}
```

Excerto de Código 4.24: Instanciacão e inicialização do servidor HTTP

De seguida, serão descritos os caminhos e as funções executadas:

/find

Este método equivale à saída do channel "Find" (referido na Especificação). Quando um *Node* transmite um pedido de "AccessRequest" para outro *Node*, esta transmissão é feita através de um pedido HTTP "POST", em que no corpo do pedido está incluído a informação do pedido "AccessRequest", no formato "JSON".

Este método aceita pedidos HTTP "POST".

Neste *Handler*, é feita uma desserialização do corpo do pedido de "JSON" para um objeto do tipo "AccessRequest", para este ser usado como parâmetro de entrada no método "HandleFind" da classe "Node".

No final da execução deste *Handler*, é feita uma resposta para o *Node* que transmitiu o pedido, com o conteúdo "Successful" e é fechado o corpo do pedido, usando o método "Close" do objeto "Body" do pedido "r", através do uso

da primitiva “defer”, que garante a execução deste método após a execução do “Handler terminar”.

```
func findRoute(w http.ResponseWriter, r *http.Request) {
  defer r.Body.Close() //defer garante que este procedimento e executado
       quando a execucao deste ''Handler'' termina

  //definicao que o tipo de conteudo do corpo tem o formato ''JSON''
  w.Header().Set("Content-Type", "application/json")

  // desserializacao do corpo para um objeto do tipo ''AccessRequest''
  var accessRequest Channels.AccessRequest
  _ = json.NewDecoder(r.Body).Decode(&accessRequest)

  fmt.Printf("\nGot a find request")
  fmt.Printf("\n%s", utils.StructToString(accessRequest))

  // execucao do metodo HandleFind do node
  selfNode.HandleFind(accessRequest)

  json.NewEncoder(w).Encode("Successful") // Resposta ao ''Node'' que
      realizou o pedido \acs{HTTP}

}
```

Excerto de Código 4.25: *Handler* “findRoute” do método “/find”

/myChan

Este método equivale à saída do channel “MyChan” (referido na Especificação). Quando um *Node* transmite um pedido de “GiveAccess” para outro *Node*, esta transmissão é feita através de um pedido HTTP “POST”, em que no corpo do pedido está incluído a informação do pedido “GiveAccess”, no formato “JSON”.

Este método aceita pedidos HTTP “POST”.

Neste *Handler*, é feita uma desserialização do corpo do pedido de “JSON” para um objeto do tipo “GiveAccess”, para este ser usado como parâmetro de entrada no método “ReceiveObj” da classe “Node”.

No final da execução deste *Handler*, é feita uma resposta para o *Node* que transmitiu o pedido, com o conteúdo “Successful” e é fechado o corpo do pedido, usando o método “Close” do objeto “Body” do pedido “r”, através do uso da primitiva “defer”, que garante a execução deste método após a execução do “Handler terminar”.

```
func myChanRoute(w http.ResponseWriter, r *http.Request) {
  defer r.Body.Close() //defer garante que este procedimento e executado
       quando a execucao deste ''Handler'' termina
```

```
  //definicao que o tipo de conteudo do corpo tem o formato ``JSON''
  w.Header().Set("Content-Type", "application/json")

  // desserializacao do corpo para um objeto do tipo ``GiveAccess''
  var giveAccess Channels.GiveAccess
  _ = json.NewDecoder(r.Body).Decode(&giveAccess)

  fmt.Printf("\nGot Access To The Object!")
  fmt.Printf("\n%s", utils.StructToString(giveAccess))

  // execucao do metodo ReceiveObj do node
  selfNode.ReceiveObj(giveAccess)

  json.NewEncoder(w).Encode("Successful") // Resposta ao ``Node'' que
      realizou o pedido \acs{HTTP}

}
```

Excerto de Código 4.26: *Handler* "myChanRoute" do método "/myChan"

/remoteRequest

Este método não tem qualquer uso dentro diretório. É apenas utilizado para forçar o *Node* a realizar um pedido a partir da visualização (ou qualquer outra forma de realização de pedidos HTTP).

O método "Request" da classe *Node* é executado, este método é usado para a realização de um pedido de acesso. Por último, é feita uma resposta ao "Client" que realizou o pedido HTTP e a conexão é terminada com o o fecho do corpo do pedido, usando o método "Close" do objeto "Body" do pedido "r".

```
  func remoteRequest(w http.ResponseWriter, r *http.Request)  {
    defer r.Body.Close() //defer garante que este procedimento e
        executado quando a execucao deste ``Handler'' termina
    selfNode.Request() //metodo de realizacao de pedidos AccessRequest
    json.NewEncoder(w).Encode("Successful") // Resposta ao ``Client''
        que realizou o pedido
  }
```

Excerto de Código 4.27: *Handler* "remoteRequest" do método "/remoteRequest"

Pedidos HTTP do *Node*

O *Node* tem a possibilidade de realizar pedidos *HTTP*, todos eles do tipo "POST".

Para tal foram definidos 3 caminhos diferentes, em que todos executam a mesma função "sendDataTo".

Esta função "sendDataTo" é polimórfica, pois o segundo parâmetro de entrada é do tipo "interface", isto é, o segundo parâmetro de entrada desta função pode ser de qualquer tipo (mas que seja possível a sua serialização para o formato "JSON").

O primeiro parâmetro é o URL para o qual irá ser enviado o pedido "POST".

Na existência de falhas no envio do pedido, são feitas no máximo 4 tentativas de envio, com um tempo de 4 segundos de espera entre cada tentativa.

```
// o parametro ''data'' pode ser de qualquer tipo/classe
func sendDataTo(toURL string, data interface{}) {

  //serializacao da informacao ''data'' para uma string
  message, err := json.Marshal(data)
  if err != nil {
    log.Fatal(err)
  }

  // Contador de tentativas efetuadas
  retries := 0

  for retries < maxRetries {

    resp, err := http.Post(toURL, "application/json", bytes.NewBuffer(
        message))

    //Caso existam falhas no envio
    if err != nil {
      fmt.Fprintln(os.Stderr, err)

      // incrementacao do valor de tentativas efetuadas
      retries++
      // a \emph{goroutine} espera durante 4 segundos
      time.Sleep(time.Second * time.Duration(4))

      continue
    }

    _, err = ioutil.ReadAll(resp.Body)
    if err != nil {
      resp.Body.Close()
      log.Fatal(err)
    }

    //Fecho do corpo do pedido e da conexao
    resp.Body.Close()
    break
```

```
    }

}
```

Excerto de Código 4.28: Método "sendDataTo" para envio de dados para outros *Nodes*

Os três tipos de pedidos (HTTP) possíveis que o *Node* pode realizar são:

SendThroughLink

Envia um pedido do tipo "AccessRequest" para o seu *Child Node*. Tem como parâmetro de entrada um objeto da classe "AccessRequest". Executa a função "sendDataTo" como valores de entrada o "Link" do *Node*, isto é, o "Find" do seu *Child Node* e o objeto da classe "AccessRequest".

```
func (node *Node) SendThroughLink(accessRequest Channels.AccessRequest)
    {
  go sendDataTo(node.Link, accessRequest)
}
```

Excerto de Código 4.29: Método "SendThroughLink"

SendObjectAccess

Envia um pedido do tipo "GiveAccess" para o *Node* que espera pelo acesso ao objeto. Tem como parâmetro de entrada um objeto da classe "GiveAccess". Executa a função "sendDataTo" como valores de entrada o "WaiterChan" do *Node*, isto é, o "MyChan" do *Node* que espera pelo acesso ao objeto e o objeto da classe "GiveAccess".

```
func (node *Node) SendObjectAccess(giveAccess Channels.GiveAccess) {
  go sendDataTo(node.WaiterChan, giveAccess)
}
```

Excerto de Código 4.30: Método "SendObjectAccess"

UpdateVisualization

Atualiza o estado do *Node* na visualização. Executa a função "sendDataTo" como valores de entrada o "VisAddress" do *Node*, isto é, o URL do método da visualização de atualização de dados e o objeto da classe "Node".

```
func (node *Node) UpdateVisualization(){
  go sendDataTo(node.VisAddress, node)
}
```

Excerto de Código 4.31: Método "UpdateVisualization"

4.11 Classes de *Channels*

Neste capítulo serão descritos os "Channels"/Pedidos utilizados pelos *Nodes*, sendo estes "AccessRequest" e "GiveAccess".

As "Struct" destes estão definidas no módulo "Channels", no ficheiro **channels.go**.

GiveAccess

Esta classe encapsula o *Channel "MyChan"* do *Node* realizou o pedido de acesso. Num caso real, este também incluiria um *Channel* que daria acesso ao objeto, ou o próprio objeto, no entanto como nesta implementação se pretende criar uma simulação deste protocolo, esse atributo não é necessário.

```
type GiveAccess struct {
  WaiterChan string `json:"waiterChan"`
}
```

Excerto de Código 4.32: *Struct* "GiveAccess"

AccessRequest

Esta classe encapsula um pedido de acesso ao objeto. Os atributos são o "Link", que contém o "Find" do *Node* que transmitiu o pedido e o "GiveAccess" instanciado pelo *Node* que fez o pedido.

Ou seja, o atributo "GiveAccess" mantém-se entre transmissões dos *Nodes* mas o atribuição "Link" é alterado em cada *Node* para o seu *Channel "Find"*.

```
type AccessRequest struct {
  GiveAccess GiveAccess `json:"giveAccess"` // #2
  Link       string     `json:"link"`       // #1
}
```

Excerto de Código 4.33: *Struct* "GiveAccess"

4.12 Implementação da Visualização

Na implementação deste protocolo a visualização serviu de auxílio na depuração do diretório e testar o seu funcionamento. Cada *Node*, quando sofre qualquer alteração de estado, este transmite o seu estado atual para um *Node* especial da visualização.

Este *Node* de visualização mantém o estado mais recente de cada *Node*. Esta informação armazenada é depois usada na visualização do diretório.

Para a parte gráfica desta secção fez-se uso de uma página *Web*, que é atualizada usando *JavaScript* e pedidos *HTTP* feitos ao *Node* da visualização, que devolve a informação do diretório.

A componente da visualização, mesmo que útil, a sua implementação é de pouca relevância ao contexto do projeto, ou seja, a parte do projeto implementada em *JavaScript*. Por isso serão apenas tratados os conteúdo de maior interesse, como os métodos/caminhos do servidor HTTP do *Node* de visualização e como cada um trata/cálcula a informação necessária, como, por exemplo, o *Node* de visualização tem acesso à informação atual do diretório e a formação da "Queue".

Servidor HTTP

Como na secção de inicialização do servidor HTTP dos *Nodes*, são registados multiplos caminhos e handlers neste servidor.

Os caminhos, handlers, tipos e as suas funcionalidades são:

/ root
: Devolve a página *Web* e código *JavaScript* necessários para a renderização gráfica do diretório.

/data data
: Devolve os estado de cada *Node* e *Links* armazenados no *Node* da visualização.

/queue queue
: Devolve a *Queue* atual no diretório.

/updateState updateState
: Utilizado pelos *Nodes* para atualizarem o seu estado atual.

/requestAll requestAll
: Utilizado para forçar todos os *Idle Nodes* a realizar pedidos de acesso.

No entanto, esta implementação tem defeitos.

- A sincronização do acesso ao "Map" causa um efeito de bottleneck e não garante que o acesso seja ordenado.
- A atualização por parte dos *Nodes* também não garante a ordem de chegada da informação, pois a rede (não o diretório) usada na comunicação de pedidos HTTP não garante a ordem, por exemplo o, o tempo de chegada de um pedido ao *Node* de visualização pode variar entre *Nodes*.

- Por último, o método de atualiação dos *Nodes* é executada numa *go-routine*, o que não garante que a execução deste método ocorra o mais rápido possível.

Atualização da visualização

Para que o *Node* da visualização armazene a informação mais recente do diretório, como referido anteriormente, cada *Node* transmite o seu estado atual para este *Node*.

Esta informação é armazenada numa estrutura de dados "Map", em que as chaves deste são os endereços ("MyAddress") de cada *Node* e o valor é o estado do *Node* a qual o endereço corresponde à chave.

O *handler* que implementa esta atualização de dados é

```
func updateState(w http.ResponseWriter, r *http.Request) {
defer r.Body.Close()

w.Header().Set("Content-Type", "application/json")

// desserializacao do estado de JSON para o tipo ``elements.Node''
var update elements.Node
_ = json.NewDecoder(r.Body).Decode(&update)

Mutex.Lock()
AllUpdates = append(AllUpdates, update)

//uso de \emph{RegEx} para remover ``http://'' e ``/find'' do ``Link''
     da atualizacao
update.Link = re.ReplaceAllString(update.Link, ``)

// Alteracao do valor da chave correspondente ao endereco do \emph{
    Node} que fez atualizacao com o seu novo estado
Nodes[update.MyAddress] = update

if update.Type == 4 {
  requestHistory = append(requestHistory, update.MyAddress)
}

Mutex.Unlock()
json.NewEncoder(w).Encode("Successful")

}
```

Excerto de Código 4.34: *Handler* "updateState" do método "/updateState"

Renderização da visualização

Durante o correr da visualização, a atualização do grafo que representa o diretório é feita através de pedidos constantes efetuados ao caminho **/data** do servidor HTTP.

Este devolve duas listas, uma contendo o estado de todos os *Nodes* armazenados no *Node* de visualização, outra contendo todos os arcos do grafo (dirigido), ou seja, pares de endereços, "Source" e "Target" (classe "elements.Link").

É feita uma iteração pelo "Map", em que caso um *Node* tenha o atributo "Link", é instanciado um objeto da classe "elements.Link" tendo como atributos o endereço desse *Node* ("Source") e o "Link" ("Target"), que ao final é adicionado à lista dos "Links".

```
for _, v := range Nodes {
  tempNodes = append(tempNodes, v)

  if v.Link != "" {
    tempLinks = append(tempLinks, elements.Link{
      Source: v.MyAddress,
      Target: v.Link,
    })
  }
}
```

Excerto de Código 4.35: Iteração pelo "Map" "Nodes", instanciação do objeto e adicionado à lista

Queue

A implementação desta secção tem falhas, tais como as referidas na introdução desta secção. pois a qualquer momento não é possível saber com exatidão quais os verdadeiros estados dos *Nodes*, logo não é possível saber o estado atual da *Queue*.

Há duas soluções para o cálculo da fila, sendo que essas soluções têm os seus problemas, provocados pelos defeitos da atualização dos estados do *Nodes* referidos na introdução deste capítulo.

É possível saber quais os *Nodes* que estão e entraram na *Queue*. A *Queue* começa quando o *Owner Terminal* se transforma em *Owner With Request*, pois o seu "WaiterChan" aponta para o primeiro elemento *Queue*. O último elemento da *Queue* é o *Waiter Terminal*, e quando um novo *Node* entra na *Queue*, o *Waiter Terminal* transforma-se em *Waiter With Request*, ou seja, o "WaiterChan" do *Node* que se transformou aponta para o novo elemento. Mas é necessário que esta "Queue" comece no *Owner*, pois é possível a existência de múltiplas filas no diretório.

Por exemplo, um *Node* **X** realizou um pedido de acesso. Logo de seguida, o seu *Parent Node* **Y** realiza também o pedido de acesso, mas o pedido do *Node* **X** ainda não chegou ao último *Node* da "Queue" que começa no *Owner*, ou seja, é formada uma nova "Queue" que não está ligada à "Queue" que começa no *Owner*.

As duas soluções (cada uma com os seus defeitos) são:

Através do estado atual conhecido.

Caso tenhamos conhecimento do atual *Owner* e for feita uma iteração a partir do *WaiterChan* de cada *Node*, isto é, seguir uma lista ligada, em que o "WaiterChan" representa o "Next" (ponteiro para o próximo elemento), o *Owner Terminal* como o "Head" (cabeça da lista) até chegar a um *Node* sem "WaiterChan", é possível conhecer a "Queue".

No entanto, o próprio estado atual conhecido pode não ser o real, e é possível haver mais que um *Owner* no estado armazenado, caso a atualização de um *Node* que se transformou de *Waiter* para *Owner* chegou antes da atualização do *Node* que cedeu o objeto.

Através das atualizações dos *Nodes*.

Caso sigamos o traço das atualizações dos *Nodes* que se transformaram de *Waiter Terminal* para *Waiter With Request*, que indica que um novo elemento entrou na "Queue", e dos *Nodes* que se transformaram de *Owner Terminal* em *Owner With Request*, que indica que uma nova "Queue" começou e qual o primeiro elemento, é possível seguirmos a formação das "Queues".

Porém, o traço conhecido pelo *Node* da visualização poderá estar momentaneamente desatualizado, e mesmo sendo mais preciso que a solução anterior, a "Queue" demonstrada não estará condizente com o grafo demonstrado.

No entanto, fez-se uso da primeira solução, pois a "Queue" demonstrada estará de acordo com o grafo.

Para tal, é feito uma procura do *Owner* atual no "Map" que contém os estados conhecidos dos *Nodes*.

```
for _, node := range Nodes {
  if node.Type < 2 {
    currentOwner = node.MyAddress
  }
}
```

Excerto de Código 4.36: Iteração pelo "Map" "Nodes", para procurar o atual *Owner*

Depois de obtermos o atual *Owner*, iteramos pela "Queue", em que o "WaiterChan" de cada *Node* aponta para/indica qual o próximo elemento da *Queue*, e a iteração termina quando o *Node* atual da iteração tem o "WaiterChan" vazio, ou seja, já não há mais elementos na "Queue".

```
  for currentNode.WaiterChan != "" {
    nextNode = Nodes[re.ReplaceAllString(currentNode.WaiterChan, '')]
    response.QueueNodes = append(response.QueueNodes, nextNode.MyAddress
        )
    currentNode = nextNode
  }
```

Excerto de Código 4.37: Iteração pela "Queue", usando o "WaiterChan" dos *Nodes*

Request All

O método é usado para testar o comportamento dos *Nodes*. Por exemplo, é nos útil identificar se o estado do diretório se mantém estável quando há um grande número de pedidos a percorrer o diretório, se um *Node* é capaz de sincronizar pedidos concorrentes e se o diretório se mantém estável apôs estas várias ocorrências.

```
func requestAll(w http.ResponseWriter, r *http.Request) {
  defer r.Body.Close()

  /*
  \emph{WaitGroup} garante que esta funcao so termina caso todas as \
      emph{goroutines} terminem de executar.

  */
  var wg sync.WaitGroup

  w.Header().Set("Content-Type", "text/plain")

  // iteracao sobre todos os estados conhecidos

  for _, element := range Nodes {

    //wg.Add(1) indica que ha mais uma goroutine pelo qual o wg.Wait()
        precisa de esperar

    wg.Add(1)


    // funcao de pedidos remotos
    go remoteRequest(element.MyAddress, &wg)
  }
```

```
  wg.Wait()
  w.Write([]byte("Successful"))
}

func remoteRequest(address string, wg *sync.WaitGroup) {

  // defer garante que este metodo do objeto ``wg'' (classe \emph{
      WaitGroup}) e executado
  //Isto indica (ao \emph{WaitGroup}) que esta \emph{goroutine} ja
      acabou a execucao
  defer (*wg).Done()

  _, err := http.Get(fmt.Sprintf("http://%s/remoteRequest", address))
  if err != nil {
    fmt.Println(err)
  }
}
```

Excerto de Código 4.38: *Handler* "requestAll" do método "/requestAll"

Mesmo que a visualização pode não ser a atual, o estado da rede terá de se manter, isto é, as seguintes atualizações que se façam na parte gráfica terão que demonstrar que as ligações entre *Nodes* se mantiveram (mesmo que invertidas), que existe apenas um único *Owner* e que qualquer outro estado não possível não é demonstrado.

4.13 Uso de *containers Docker*

Na implementação deste projeto foi necessário testar os *Nodes* num contexto distribuído. Para tal, em vez de correr várias instâncias do programa em várias máquinas diferentes (mas que podessem comunciar entre si), fez-se uso de *containers Docker* para criar o mesmo efeito.

Além disso, o uso desta ferramenta provou-se útil como ferramenta de execução de multiplas instâncias do programa, não apenas como "Máquinas Virtuais".

Para a criação e execução de *containers* foram necessários dois tipos de ficheiros, **Dockerfile docker-compose.yml**.

Dockerfile

Na criação de um *container* é necessário um ficheiro "Dockerfile", que descreve o processo de inicialização do *container*, como que Sistema Operativo/imagem irá usar, a cópia de ficheiros para o *container*, que comandos deve executar e por último que programa deverá executar.

```
FROM golang # que sistema ou imagem ira usar, neste caso e usado a
    imagem ''golang''
WORKDIR /src  # em que diretorio, no container, os seguintes comandos
    irao ser executados
COPY . .
RUN go build -o node # compilacao do programa

# execucao da instancia, as variaveis de ambiente sao marcadas com $, no
     entanto serao descritas a sua origem de seguida
CMD ./node --address=$address --type=$type --link=$link --requests=
    $requests --visualization=$VIS_ADDRESS
```

Excerto de Código 4.39: Iteração pelo "Map" "Nodes", instanciação do objeto e adicionado à lista

docker-compose.yml

Foram criadas várias topologias de redes (por exemplo rede em Anel, Estrela, etc) para demonstrar e testar o diretório. Estes exemplos estão no formato "YML", mais precisamente, no formato de um ficheiro "docker-compose.yml" para que este possa ser lido pelo programa "docker-compose", um *script* que permite a execução de múltiplos *containers* com apenas um ficheiro e um comando.

No entanto estes ficheiros apenas declaram os atributos de cada *container*, como o nome, o endereço IP, os *Ports* que esta necessita para o funcionamento, e no contexto deste projeto, os artibutos iniciais de cada *Node*, como o "Type", o "Link" e o enderço do *Node* de visualização, visto que cada *container* executará uma instância do programa, isto é, cada *container* é um *Node*. Os atributos do *Node* são definidos usando as variáveis de sistema.

```
  # nome do container
  node_0:
    tty: true
    stdin_open: true

    # indicacao da localizacao do ficheiro Dockerfile
    build:
      context: ./src
      dockerfile: Dockerfile

    # definicao dos atributos do node como variaveis de ambiente
    environment:
      address: 127.0.0.1:8001
      type: 2
      link: 127.0.0.1:8005
      VIS_ADDRESS: 127.0.0.1:8000/updateState
```

```
    requests: "true"

  # Ports necessarios para o funcionamento do Container
  ports:
    - "8001:8001"
  # Indicacao que este container sera executado na mesma rede que o
      Host, isto para que seja
  # possivel a realizacao de pedidos remotos atraes da visualizacao
  network_mode: host
```

Excerto de Código 4.40: Iteração pelo “Map” “Nodes”, instanciação do objeto e adicionado à lista

4.14 Conclusões

Capítulo

5

Reflexão Crítica

5.1 Introdução

5.2 Escolhas de Implementação

5.3 Detalhes de Implementação

5.4 Conclusões

Capítulo

6

Conclusões e Trabalho Futuro

6.1 Conclusões Principais

6.2 Trabalho Futuro

Bibliografia

[1] Michael J Demmer and Maurice P Herlihy. The arrow distributed directory protocol. In *International Symposium on Distributed Computing*, pages 119–133. Springer, 1998.