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)]

1 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

1.2 Objetivos

1.3 Organização do Documento

1.

2.

3.

Capítulo

2

Motivação

2.1 Introdução

2.2 Trabalhos relacionados

2.3 Descrição do Protocolo

2.3.1 Estrutura do Diretório

2.3.2 Características do Diretório

2.3.3 Estruturas de Dados

2.4 Conclusões

Capítulo

3

Especificação

3.1 Introdução

3.2 *Node*

Neste capítulo serão descritos os diferentes *Nodes* presentes no diretório, tais como os seus atributos, que estão definidos em detalhe no capítulo seguinte 3.3.

Existem vários fatores que diferenciam os *Nodes*:

- Detém o acesso ao objeto. - *Owner*/Dono do objeto.
- Efetuou um pedido de aquisição do acesso ao objeto. - *Waiter*/*Node* em espera.
- Tem um pedido em espera. - *With Request*/Com Pedido.
- Nenhum dos anteriores. - *Idle*/Inativo.

Algumas regras de funcionamento do diretório incluem:

- As ligações entre *Nodes* são realizadas através do *Link* de um *Node* que aponta para o *Channel* ***Find*** de um outro *Node*.
- Cada pedido transmitido (através do *Link*) inclui o *Channel* ***Find*** do transmissor, para que possa haver uma inversão na ligação, ou seja, para que o recetor possa atualizar o seu *Link* para o *Channel* ***Find*** do transmissor.

- Caso o *Node* detenha ou espera pelo acesso ao objeto e receba um pedido, este atualiza o *WaiterChan* que aponta para o *MyChan* do *Node* que fez o pedido.
- Caso o *Node* ceda o acesso ao objeto, este é transmitido através do *WaiterChan*.

O *Child Node* de um *Node* é o
Também serão descritos os comportamentos que cada tipo pode manifestar e transformações que estes podem sofrer, como a atualização de atríbutos e mudança de tipo:

3.2.1 *Owner Terminal*

Node que detém o objeto e não tem pedido em espera.
Atributos:

- Find
- MyChan
- Obj

Receção de um pedido Access Request

O *Node* recebe um pedido ***Access Request*** no seu *Channel* ***Find***, que foi remetido pelo seu *Parent Node*, ou seja, um pedido transmitido por um *Node* que aponta para o *Node* em questão.

Como o *Node* é o detentor do acesso ao objeto, este transforma-se em ***Owner With Request***, Atualiza o ***Link*** (para ***NewLink***), que aponta para o **Find** do seu *Parent Node*, havendo uma inversão da ligação. - **OwnerWithRequest(find, MyChan, Obj, NewLink, WaiterChan)**.

3.2.2 *Owner With Request*

Node que detém o objeto e tem um pedido em espera.
Atributos:

- Find
- MyChan
- Link

- Obj
- WaiterChan

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***.

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

3.2.3 *Idle*

Node não detém o objeto, nem fez qualquer pedido.

Atributos:

- Find
- MyChan
- Link

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***.

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* (Esta informação provêm informação comunicada do pedido) - **Idle(find, MyChan, NewLink)**.

Realização de um pedido de acesso

O *Node* envia no ***Link*** o ***MyChan*** e o ***Find*** para o *Child Node*.

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

3.2.4 *Waiter Terminal*

Node aguarda pelo acesso ao objeto.

Atributos:

- Find
- MyChan

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***.

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 atualiza o ***WaiterChan*** (para **NewWaiterChan**) (Esta informação provêm informação comunicada do pedido) - **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*** - **OwnerTerminal(find, MyChan, Obj)** .

3.2.5 *Waiter With Request*

Node aguarda pelo acesso ao objeto e tem um pedido em espera.

Atributos:

- Find
- MyChan
- Link
- WaiterChan

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***.

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***, e atualiza o ***Link*** (para ***NewLink***), que aponta para o **Find** do seu *Parent Node* (Esta informação provêm informação comunicada do pedido) - **WaiterWithRequest(find, MyChan, NewLink, NewWaiterChan)**.

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*** - **OwnerWithRequest(find, MyChan, Obj, Link, WaiterChan)**.

3.3 Atributos do *Node*

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

Find

Este atributo representa o *Channel* por onde o *Parent Node* difunde os pedidos de acesso para o *Node*. 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 seu *Child Node*. Contém o *Channel Find* do *Child Node*.

Obj

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

WaiterChan

Este atributo representa o *Channel* do *Node* sucessor da fila. Contém o *MyChan* do *Node* em espera.

3.4 *Channels*

Nesta especificação foram apenas definidos dois *Channels*. A comunicação entre os *Nodes* é feita por canais, pelos quais são comunicados canais. Neste capítulo serão descritos os *Channels*:

Access Request

Este tipo de *Channel* é usado para fazer chegar o pedido ao último elemento da fila de espera (de acesso ao objeto). 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 dar acesso ao objeto a quem fez o pedido, por outras palavras, é usado pelo atual ***Owner*** para transmitir o acesso ao *Waiter* que estava na posição da fila de espera.

3.5 Conclusões

Capítulo

4

Tecnologias e Ferramentas Utilizadas

4.1 Introdução

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*:

4.2.1 *Go*

Ao contrário de outras linguagens, a implementação de **sistemas concorrentes** em *Go* é **simples**, sendo uma das razões para a utilização desta tecnologia. É simples "ao ponto" de adicionando a palavra "*go*" antes de qualquer procedimento, esse procedimento irá correr em uma nova *Goroutine*, ou seja, de forma concorrente em relação a todas as outras *Goroutines* já em execução. Uma "Goroutine" é um *lightweight* ***thread*** gerido pelo *runtime* do *Go*.

Outras razões são:

- Simplicidade - tem poucas funcionalidades que por si são simples.
- Rapidez - o código é compilado em apenas um ficheiro executável.

- Transparência - há poucas formas de resolver o mesmo problema, ou seja,

Por exemplo, comparando (parcialmente) *Go* e *Java*, um programa concorrente que mostra 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 execução de procedimentos concorrentes, a linguagem oferece bibliotecas de apoio a problemas 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 um assunto de grande relevância em problemas de sistemas distribuídos, pois o próprio sistema funciona num paradigma concurrente, visto que os vários elementos 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 consistente (ou o diretório), o *Node* que recebeu os pedidos terá de os tratar de forma sincronizada, isto é, tem de abordar um pedido de cada vez.

Estas são razões que induzem a liguagem *Go* a ser uma ótima escolha para a resolução de problemas num contexto distribuído, pois esta oferece múltiplas ferramentas concorrência de origem.

4.2.2 *JavaScript*

Para a implementação da visualização foi usado **JavaScript**. Esta linguagem permite que a informação de páginas *Web* seja alterada após a sua renderização/carregamento. É nos útil no desenho de grafos e alteração de tabelas que dispõe a informação sobre a visualização da rede.

4.3 Ferramentas de Edição de Código

Neste capítulo serão descritas ferramentas que foram usadas para escrita de código de forma mais eficiente:

- *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*.

4.4 Bibliotecas

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

4.4.1 *gorilla/mux*

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

4.4.2 *D3.JS*

A biblioteca *D3.JS*, em que *D3* significa "***D**ata-**D**riven **D**ocuments*", em português, "documentos baseados 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.

4.5 Outras Tecnologias

4.5.1 *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, e mantendo as aplicações isoladas do sistema hospedeiro.

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.

4.6 Outras Ferramentas

Nesta secção irão ser mencionadas ferramentas de menor destaque na elaboração do projeto. Estas serviram de auxílio

4.6.1 Github

4.6.2 Lazydocker

4.7 Conclusões

Capítulo

5

Implementação

5.1 Introdução

5.2 Escolhas de Implementação

5.3 Detalhes de Implementação

5.4 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 as várias (funcionalidades ?) estão distribuídas por estes ficheiros.

Estes são:

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

5.5 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 5.1: 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. Também é usado na construção da *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 *OwnerTerminal* ou *OwnerWithRequest*, este tem o acesso ao objeto.

5.6 Inicialização do objeto "Self Node"

5.7 Tipos de Nodes

5.8 Comportamentos

Como referido no capítulo de Especificação 3.3, 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 .

5.8.1 Receção de um pedido Access Request

Todos os tipos de *Nodes* têm a possibilidade de receber um pedido de *AccessRequest*, 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 5.2: *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 5.3: 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.

```
newAccessRequest.Link = node.Find
node.SendThroughLink(newAccessRequest)
node.OwnerWithRequest(accessRequest.Link, node.WaiterChan)
```

Excerto de Código 5.4: Comportamento do *Node* tipo *Owner Terminal* 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.

```
newAccessRequest.Link = node.Find
node.SendThroughLink(newAccessRequest)
node.OwnerWithRequest(accessRequest.Link, node.WaiterChan)
```

Excerto de Código 5.5: Comportamento do *Node* tipo *Owner Terminal* 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 "My-

Chan" do *Node* que fez o pedido, respetivamente. O "Link" é o "Find" do *Parent Node*, o que causa a inversão da ligação.

```
node.WaiterWithRequest(accessRequest.Link, accessRequest.GiveAccess.
    WaiterChan)
```

Excerto de Código 5.6: Comportamento do *Node* tipo *Owner 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 5.7: Comportamento do *Node* tipo *Owner Terminal* caso receba um pedido *Access Request* no *Channel "Find"*

5.8.2 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 5.8: *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 5.9: 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 5.10: Acesso à secção crítica, transmição do Objeto, transformação em *Node* "IDLE", *goroutine* de decisão de pedido, e atualização na visualização

5.8.3 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 *goroutine*.

```
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 5.11: 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 5.12: 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.

5.8.4 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 execu-

tado. 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 5.13: Método “AutoRequest”

5.9 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:

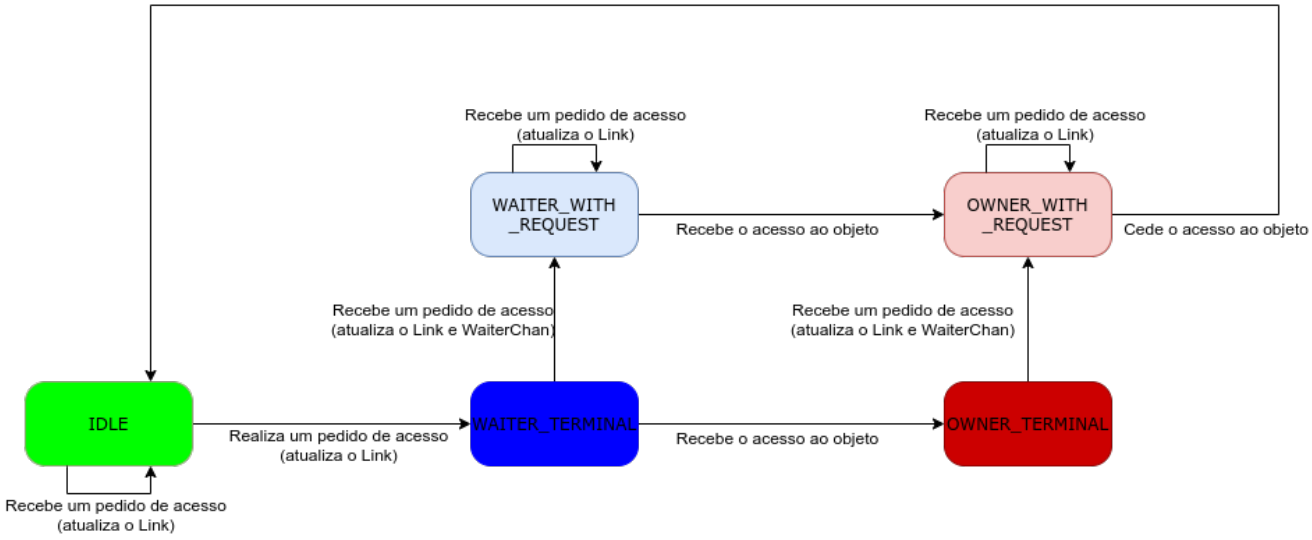

Figura 5.1: Diagrama de Estados do *Node*

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

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
}
```

Excerto de Código 5.14: Método/transformação "Idle"

WaiterTerminal

Se o *Node* é do tipo "IDLE" e realizar um pedido de acesso, este transforma-se em "WAITER_TERMINAL". 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 5.15: Método/transformação "Idle"

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", mas transforma-se em "OWNER_TERMINAL".

```
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 5.16: Método/transformação “Idle”

5.10 Comunicação entre Nodes

5.11 Classes de *Channels*

5.12 Implementação da Visualização

5.13 Conclusões

Capítulo

6

Visualização

6.1 Introdução

6.2 Atualização dos Dados

I

6.3 Representação da Rede

6.3.1 *Nodes*

6.3.2 *Links*

6.3.3 Fila

6.3.4 Histórico de Pedidos

6.3.5 Histórico de *Owners*

6.4 Conclusões

Capítulo

7

Reflexão Crítica

7.1 Introdução

7.2 Escolhas de Implementação

7.3 Detalhes de Implementação

7.4 Conclusões

Capítulo

8

Conclusões e Trabalho Futuro

8.1 Conclusões Principais

8.2 Trabalho Futuro

Bibliografia