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)]

29 de janeiro de 2021

Agradecimentos

...

Conteúdo

Lista de Figuras

Lista de Tabelas

Acrónimos

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

Capítulo

1

Introdução

1.1 Enquadramento

1.2 Motivação

1.3 Objetivos

1.4 Organização do Documento

1.

2.

3.

Capítulo

2

Estado da Arte

2.1 Introdução

2.2 Trabalhos relacionados

2.3 Destaque da Implementação

2.4 Descrição do Protocolo

2.4.1 Estrutura do Diretório

2.4.2 Características do Diretório

2.4.3 Estruturas de Dados

2.5 Conclusões

Capítulo

3

Tecnologias e Ferramentas Utilizadas

3.1 Introdução

3.2 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*:

3.2.1 *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 3.1: 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 3.2: 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.

3.2.2 *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.

3.3 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*.

3.4 Bibliotecas

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

3.4.1 *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.

3.4.2 *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.

3.5 Outras Tecnologias

3.5.1 *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.

3.6 Outras Ferramentas

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

3.6.1 Github

3.6.2 Lazydocker

3.7 Conclusões

Capítulo

4

Especificação

4.1 Introdução

4.2 *Node*

Neste capítulo serão descritos os tipos de *Nodes* presentes no directório, tais como os seus atributos, que estão definidos em detalhe no capítulo seguinte 4.3.

O tipo de cada *Node* depende dos seguintes fatores:

- Detem 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.

Também serão descritos os comportamentos que cada tipo pode manifestar e tranformações que estes podem sofrer, como a atualização de atríbutos e mudança de tipo:

4.2.1 *Owner Terminal*

Node que detem o objeto e não tem pedido em espera.

Atributos:

- Find 4.3.1

- MyChan 4.3.2
- Obj 4.3.4

4.2.1.1 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 tranforma-se em ***Owner With Request***, e atualiza o ***Link*** (para ***NewLink***), que aponta para o **Find** do seu *Parent Node* (Esta informação provêm informação comunicada do pedido) - **OwnerWithRequest(find, MyChan, Obj, NewLink, WaiterChan)**.

4.2.2 *Owner With Request*

Node que detem o objeto e tem um pedido em espera.

Atributos:

- Find 4.3.1
- MyChan 4.3.2
- Link 4.3.3
- Obj 4.3.4
- WaiterChan 4.3.5

4.2.2.1 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 mantem-se como ***Owner With Request***, e atualiza o ***Link*** (para ***NewLink***), que aponta para o **Find** do seu *Parent Node* (Esta informação provêm informação comunicada do pedido) - **OwnerWithRequest(find, MyChan, Obj, NewLink, WaiterChan)**.

4.2.2.2 Cedência do Objeto

Após a receção de um pedido ***Access Requesst***, 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 detem o objeto, este transforma-se em ***Idle*** - **Idle(find, MyChan, Link)**.

4.2.3 *Idle*

Node não detem o objeto, nem fez qualquer pedido.

Atributos:

- Find 4.3.1
- MyChan 4.3.2
- Link 4.3.2

4.2.3.1 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 mantem-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)**.

4.2.3.2 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)**.

4.2.4 *Waiter Terminal*

Node aguarda pelo acesso ao objeto.

Atributos:

- Find 4.3.1
- MyChan 4.3.2

4.2.4.1 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)**.

4.2.4.2 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)** .

4.2.5 *Waiter With Request*

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

Atributos:

- Find 4.3.1
- MyChan 4.3.2
- Link 4.3.3
- WaiterChan 4.3.5

4.2.5.1 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 mantem-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)**.

4.2.5.2 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)**.

4.3 Atributos do *Node*

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

4.3.1 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*.

4.3.2 MyChan

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

4.3.3 Link

Este atributo representa a ligação do *Node* para o seu *Child Node*. Contém o *Channel Find* do *Child Node*.

4.3.4 Obj

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

4.3.5 WaiterChan

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

4.4 *Channels*

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

4.4.1 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 tipos de *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**.

4.4.2 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.

4.5 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 incluida num módulo *Go* "Nodes" (Ou seja, num diretório com o mesmo nome), e é consituído 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.

5.4.1 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. Esta é a definição da *Struct* dos atributos do *Node*.

```
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   //Liga ao para o child Node
  WaiterChan  string   //Channel do Node que esta na posi ao seguinte da
      fila
  MyAddress   string   //Endere o do Node
  VisAddress  string   //Endere o para onde envia o seu estado atual
     para a atualiza ao da visualiza ao
  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 deter, porém, quando um atributo é "inexistente", este é definido como vazio, ou seja, os atributos "WaiterChan", "VisAddress" e "Link" podem ser *strings* vazias.

5.4.2 Comportamentos

Como referido no capítulo de Especificação 4.3,

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