Universidade da Beira Interior
Departamento de Informática

Nº 143 - 2021

Arrow Distributed Directory Protocol

Implementação, Experimentação e Visualização

Elaborado por:

Guilherme João Bidarra Breia Lopes

Orientadores:

Professor Doutor Simão Melo de Sousa
Professor Doutor Hugo Torres Vieira

19 de maio de 2021

Agradecimentos

...

Conteúdo

Lista de Figuras

Acrónimos

ADDP *Arrow Distributed Directory Protocol*

CSV *Comma-Separated Values*

HTML *HyperText Markup Language*

HTTP *Hypertext Transfer Protocol*

IDE *Integrated Development Environment*

IP *Internet Protocol*

SVG *Scalable Vector Graphics*

URL *Uniform Resource Locator*

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 e visualização dos mesmos.

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

1.2 Objetivos

Os objetivos do projeto foram evoluindo no decorrer do mesmo, pois, inicialmente estava planeada uma implementação apenas concorrente (não distribuída) e uma exploração de especificações formais como o cálculo-pi.

O foco do projeto divergiu para uma implementação de um sistema distribuído, e a visualização deste fazendo uso de uma interface gráfica.

Ou seja, os objetivos definidos durante o desenvolvimento do projeto foram:

- Leitura do *paper* original e proposto e compreensão geral do protocolo.
- Elaboração da especificação 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*).
- Implementação da visualização do histórico do elementos da fila e detentores do acesso ao objeto.

1.3 Resultados Atingidos

Serão descritos os resultados atingidos no desenvolvimento do projeto.

- Foi elaborada uma especificação de todos os elementos pertencentes ao sistema, tendo como maior foco os comportamentos dos nós no sistema. Esta especificação é também descrita neste relatório.
- Desenvolveu-se o programa dos nós, sendo o objetivo de maior importância na elaboração deste projeto, pois um conjunto de instâncias deste programa representam um sistema que segue o protocolo *Arrow*. Para além disso, incluído nesta implementação tem-se:
  - Implementação dos comportamentos dos nós.
  - Comunicação entre nós através do protocolo HTTP.
  - As transformações que os nós podem sofrer.
  - Definição das mensagens transmitidas entre os nós.
- Implementação de uma visualização gráfica das estruturas de dados distribuídas presentes na rede, como um grafo que representa o diretório da rede e uma tabela que representa a fila de chegada do acesso.
- O uso da ferramenta *Docker* para a execução de várias instâncias do programa dos nós de forma automática e distribuída.
- A implementação de uma ferramenta de auxílio para a transformação de uma topologia/definição de uma rede no formato CSV para o formato “docker-compose.yml”, usado como ficheiro de execução da ferramenta *Docker*.

1.4 Organização do Documento

Este documento está organizado da seguinte forma:

1. **Introdução** - Citação do artigo de referência usado para a implementação, os objetivos do projeto e os resultados atingidos no mesmo.

2. **Motivação** - Apresentação das vantagens e a necessidade de utilização de sistema distribuídos, tais como complexidade do desenvolvimento dos mesmos. Descrição do protocolo de referência e citação de trabalhos relacionados.

3. **Especificação** - Descrição ao pormenor de todos os elementos que pertencem ao sistema, incluindo os seus atributos e comportamentos.

4. **Engenharia de Software** - Planeamento e decisões tomadas antes da implementação do projeto.

5. **Implementação** - Descrição de todo o processo de implementação, como linguagens, ferramentas e bibliotecas utilizadas, a implementação de todos os elementos descritos na especificação e implementação da visualização do sistema.

6. **Validação experimental, Simulação e Testes** - Explicação de como é possível construir o sistema, como a interface funciona e de como testes foram feitos.

7. **Conclusão** - Conclusão final do projeto.

Capítulo

2

Motivação

2.1 Introdução

Ao longo dos anos, o crescimento da utilização de serviços informáticos, de dispositivos mais acessíveis à população mundial e uma maior cobertura de serviços de Internet, tem levado ao aumento da necessidade de serviços com uma maior escalabilidade, para garantir o acesso a mais utilizadores, com uma maior disponibilidade, pois erros ou *Downtime* torna-se uma inconveniência para os utilizadores, e distribuídos geograficamente, para garantir o acesso mais rápido ao serviço. Também o aumento do número de núcleos nos processadores ou a conveniência e eficiência de várias máquinas trabalharem em conjunto, tem vindo a aumentar a utilização de programas concorrentes e sistemas distribuídos.

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/programas sequenciais/de um único processo. 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.

Um dos tópicos relacionado com o protocolo estudado é o acesso ao mesmo recurso por vários processos.

Por exemplo, várias máquinas numa rede têm a possibilidade de aceder a um ficheiro, tanto para fazer uma leitura como uma escrita. Duas máquinas (ou processos) pretendem fazer alterações num ficheiro na rede, ambas começam por fazer uma leitura do ficheiro, e após essa leitura, uma das máquinas escreve as alterações, e logo de seguida a outra máquina escreve as alterações. A falha nesta ocorrência foi que a última máquina a escrever sobrescreveu totalmente as a alterações da primeira, visto que ambas leram o mesmo estado do ficheiro mas a uma das máquinas escreveu após a escrita da outra.

Este tipo de ocorrências são denominadas de falha de Condição de Corrida ou *Race Condition*, que podem levar a estados inesperados de qualquer elemento se possa ser alterado (como ficheiros, estruturas de dados, etc), que por último podem levar a grandes catástrofes.

Um exemplo catastrófico foi o "Apagão de 2003" [2] que afetou os Estados Unidos da América e Canada.

Após falharem 3 linhas de eletricidade, foram enviados 3 pedidos ao "mesmo tempo", que provocou uma *Race Condition* na escrita de nova informação num servidor central, deixando assim o servidor num estado corrupto/errado. Essa corrupção levou a que o sistema de alarme entrasse em Ciclo Infinito, fazendo com que não fosse possível receber novos avisos de outras linhas.

No protocolo estudado, estes problemas são evitados através da criação de uma fila, esta distribuída, que ordena/sincroniza o acesso ao objeto por parte dos vários nós do sistema. A formação desta fila é descrita com maior pormenor na secção 2.2.

2.2 Descrição do Protocolo

Nesta secção será descrito o funcionamento e estrutura do *Arrow Distributed Directory Protocol* (*Arrow*) [1].

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 diretório pode ser considerado como um grafo, em que os Nós do Grafo são os vários elementos que pertencem ao sistema (por exemplo, várias instâncias de um programa) e as arestas são as várias ligações no diretório entre os vários Nós.

Consideramos que o objeto é "móvel", no entanto esta é uma abstração do **acesso** ao objeto, e este acesso tem a possibilidade de se movimentar na rede entre os nós.

O objeto pode ser um processo, um ficheiro ou qualquer outra estrutura de dados.

Cada nó tem uma só ligação, deste para outro nó, no entanto podem existir vários nós com ligação a um único nó.

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

Consideremos que a noção que o diretório é uma árvore de extensão mínima, e consideremos que o acesso ao objeto (ou o nó que o detém) está localizado na raiz da árvore e todas as ligações apontam em direção à raiz.

Quando é realizado um pedido, este irá chegar à raiz, pois todas as ligações apontam para esta. No final da circulação do pedido, todas as ligações por onde passou o pedido estão invertidas, para que estas passem a apontar para o nó que fez o pedido.

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.

Exemplo da organização das ligações

Vejamos o seguinte exemplo da rede 2.1, em que o (acesso ao) objeto se encontra atualmente no nó **H**.

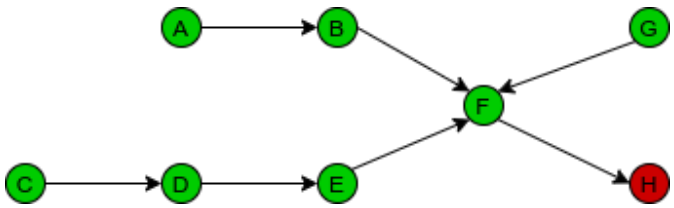

Figura 2.1: Estado inicial do diretório

O nó **A** pretende obter o acesso ao objeto, e para tal envia um pedido através da sua ligação (para o nó **B**). O pedido passará pelos nós **B**, **F** e **H**, por esta ordem.

Vejamos o estado após o pedido chegar ao nó **F**, como demonstrado na imagem 2.2:

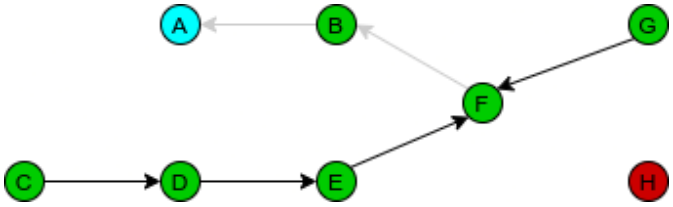

Figura 2.2: Estado do diretório após o pedido chegar ao nó **F**

Note que as ligações por onde o pedido do nó **A** passou inverteram-se.

Quando o nó **A** faz o pedido e este chega ao **H** este passa a ser a raiz da árvore, e os seguintes pedidos serão direcionados até ao **A**.

Vejamos agora outro exemplo 2.3, em que exploramos o caso de existirem mais do que uma mensagem a circular no diretório:

Consideremos que cada passo é uma transmissão de pedidos entre nós, e que os passos que cada pedido executa são sincronizados com os restantes, algo que não é garantido na rede, mas que temos em conta para se demonstrar com maior simplicidade este processo.

Consideremos o mesmo estado anteriormente referido, o estado do diretório após o pedido do nó **A** chegar ao nó **F**, e que ao mesmo tempo o nó **D** pretende obter o acesso, que para tal este enviou um pedido de acesso que atualmente se encontra no nó **E**.

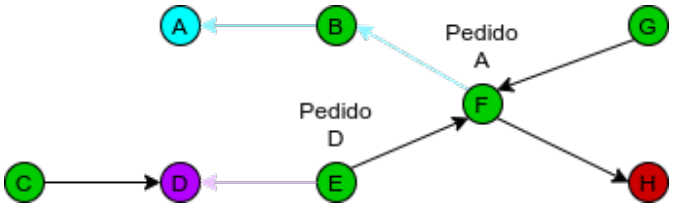

Figura 2.3: Estado do diretório após o pedido do nó **A** chegar ao nó **F**, as etiquetas com "Pedido" indicam onde o pedido de cada nó se encontra no atual estado.

Como o nó **F** aponta/está ligado ao nó **B**, o pedido do nó **D** será então transmitido para este ao invés do **H**. Vejamos o passo seguinte deste estado

na imagem 2.4:

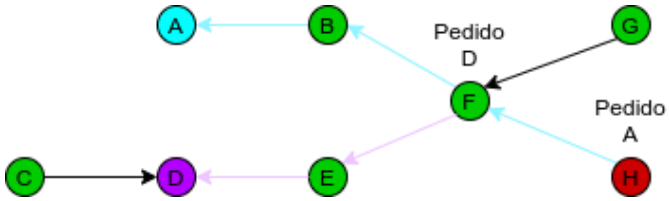

Figura 2.4: Estado do diretório após o pedido do nó **D** chegar ao nó **F**

Nota: Caso os dois pedidos chegassem ao nó **F**, e que nenhum dos dois ainda não foi tratado, os pedidos terão de ser tratados de cada vez. Se o pedido do nó **A** fosse o primeiro a ser processado, este seria transmitido para o **H** e o pedido do **D** continuava a circulação em direção ao **A** e vice-versa.

No passo seguinte 2.5, o pedido do **D** encontra-se no nó **B**:

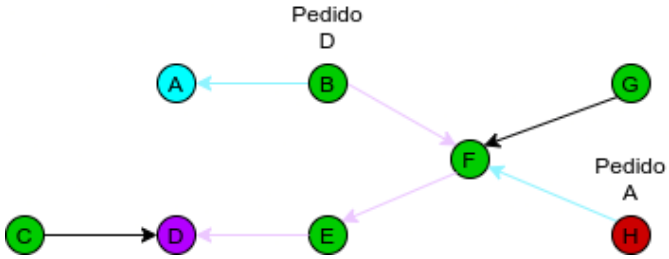

Figura 2.5: Estado do diretório após o pedido do nó **D** chegar ao nó **B**

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, ou seja, na direção do nó que fez o pedido e que espera pelo acesso ao objeto. Qualquer outra "sobreposição" das alterações nas ligações fará com que as ligações apontem em direção dos nós que realizaram os pedidos que passaram por essas ligações, isto é, independentemente da quantidade de pedidos a circular na rede, se seguirmos as direções das ligações em qualquer estado da rede estas apontam sempre para um nó que detém ou espera pelo acesso ao objeto.

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 uma ligação com o 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.

Exemplo da formação de uma fila de espera

Continuando o exemplo anterior. Vejamos um passo seguinte, em que o pedido do nó **D** chega ao nó **A**, o nó **H** ainda não cedeu o acesso ao nó **A**, e o nó **C** decidiu realizar um pedido que atualmente se encontra no nó **D**, que também está à espera do acesso ao objeto. Vejamos a imagem 2.6

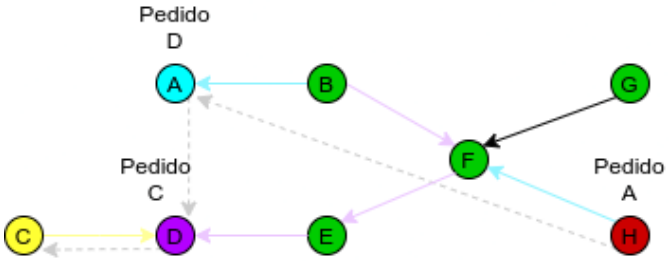

Figura 2.6: Estado do diretório com 3 pedidos em circulação.

Nota: As ligações a tracejado representam as ligações entre os nós fora do diretório, usadas para a transmissão do acesso ao objeto.

Temos assim os seguintes nós à espera:

1. O nó **A** espera pelo nó **H**.
2. O nó **D** espera pelo nó **A**.
3. O nó **C** espera pelo nó **D**.

Ou seja, a ordem de chegada e fila do acesso de pedidos será a **A**,**D**,**C**.

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 "Casa" do objeto estar em constante alteração garante que apenas um nó detenha o acesso ao objeto (acesso exclusivo), e que o único detentor do objeto se torne num foco, isto é, que nenhum nó receba demasiados

pedidos em relação a outros nós, provocado um engarrafamento/*bottleneck* no acesso ao objeto.

Estruturas de Dados

Neste protocolo existem duas estruturas distribuídas, em que cada uma é formada por um tipo de ligação entre os nós.

Grafo os vértices são todos os nós do sistema, e as arestas são as ligações entre os nós utilizadas no envio de pedidos de acesso.

Fila os vértices são todos os nós do sistema que esperam pelo acesso ao objeto e as arestas são as ligações entre os nós mas exterior ao diretório, pois neste circula o acesso ao objeto.

2.3 Trabalhos Relacionados

Existem dois algoritmos cujo o seu funcionamento é semelhante com o ADDP. Estes são o *Ivy* [3] e o *Arvy* [4], este último sendo uma generalização de ambos o *Ivy* e o algoritmo estudado (ADDP). Este três algoritmos (ou classe de algoritmos) garantem o acesso exclusivo ao objeto (ou o acesso a este) devido ao mecanismo de mudança de ligações. O *Ivy* apresenta um funcionamento muito similar ao *Arrow* (ADDP), em que o objeto (ou o acesso a este) varia de localização, mas destaca-se do *Arrow* na formação das ligações, que, ao invés de as ligações por onde o pedido passa se inverterem, os *Nodes* que recebem os pedidos passam a estar ligados ao *Node* que realizou o pedido.

Como referido anteriormente, o *Arvy* é uma generalização de ambos. Neste algoritmo, quando um *Node* recebe um pedido este pode ligar-se a qualquer *Node* por onde atravessou o pedido.

Quanto a implementações, existem três relacionadas com a desenvolvida neste projeto. Estas são, o *Aleph Toolkit*[5], a qual é a única implementação conhecida do ADDP, mas esta é muito diferente da obtida no decorrer deste projeto, a implementação do *Arvy* [6], a qual já foi referida, e uma implementação do *Bully Algorithm* [7], que inspirou o uso de uma Página *Web*, esta que comunica com um Nó de Visualização para a visualização do sistema distribuído, e também inspirou o uso do *Docker* para o *Deploy*/execução do sistema, no entanto este tópico será exposto num capítulo seguinte (Capítulo 4).

2.4 Conclusões

Neste capítulo foi descrito o aumento da necessidade de sistemas e algoritmos distribuídos bem construídos, tal como a complexidade de implementação dos mesmos. No contexto deste projeto, das falhas que ocorrem somente em sistemas concorrentes (e por sua vez, distribuídos), a que se pretende evitar (principalmente) é a Condição de Corrida/*Race Condition*, pois este tipo de falhas provoca resultados/estados inesperados de sistemas e informação, por vezes provocando catástrofes ou grandes problemas. O protocolo em questão evita este tipo de falhas ao garantir que, a qualquer momento, apenas um elemento detém o (acesso ao) objeto. Na descrição desenvolvida sobre este algoritmo há uma maior ênfase na mudança das ligações entre os nós e o que esta mudança provoca no sistema numa vista global, pois este é o mecanismo que evita o acesso concorrente ao objeto, que, após um primeiro pedido chegar ao atual detentor do objeto, o detentor cederá o objeto ao autor do pedido, e os subsequente pedidos de acesso serão redireccionados em direção a um futuro detentor. Foram também referidos dois trabalhos relacionados com o ADDP, os quais diferem principalmente na formação das novas ligações após a recepção de pedidos de acesso.

Capítulo

3

Especificação

3.1 Introdução

Neste capítulo é apresentado a especificação de todos os elementos pertencentes ao sistema. Este sistema é formado por vários **nós** (*Nodes*), estes podendo ter vários comportamentos que alteram o seu estado (atributos). Os **nós** comunicam entre si através de **ligações**, pelas quais são transferidas dois tipos de mensagens, que contém a informação necessária para fazer chegar pedidos ou o acesso ao objeto aos **nós**. Estas mesmas ligações formam as várias estruturas distribuídas existentes no diretório. Ao contrário do que foi descrito no capítulo 2, este capítulo foca-se nas alterações de cada elemento, ao invés da alteração do sistema "num todo".

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.

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.

Por último, podem existir *Nodes* que nem detenham o acesso ao objeto nem estejam à espera deste, e então reencaminham os pedidos. Estes são os únicos *Nodes* que podem pedir o acesso ao objeto.

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 através das ligações o diretório (grafo), quando estes pretendem pedir o acesso ao objeto, ou através das ligações da fila (exteriores ao diretório), 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*, que podem ser enviados para outros *Nodes*, e que, caso um *Node* decide transmitir alguma mensagem para o outro *Node*, esta é feita usando um *Channel* do *Node* de destino.

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, o ponto de saída corresponde assim à caixa de entrada do *Node*, enquanto que o ponto de entrada é detido por quem envia mensagens.

Ligações no diretório

Como referido no capítulo 2, 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 (pedido de acesso) é transmitida por um *Channel* (da entrada para a saída).

Neste tipo de ligação no diretório foram definidas duas partes nas ligações. Denominamos de "Link" a parte de entrada da ligação que está presente no transmissor e denominamos de "Find" a parte da saída das mensagens.

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 *Parent Node* do **X**, e o *Node* **X** é o *Child 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 *Parent 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 fornece 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 na fila

Um *Node* que seja o detentor do acesso ao objeto ou que espera por este pode receber um pedido de acesso. Quando isto acontece, o *Node* que realizou o pedido de acesso aguarda que o *Node* que recebeu o pedido lhe transmita o acesso, passando a haver uma ligação direta entre estes (sem fazendo uso do diretório).

Neste tipo de ligação no diretório foram definidas duas partes nas ligações. Denominamos de "WaiterChan" a parte de entrada da ligação que está presente no transmissor e denominamos de "MyChan" 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 diretamente apenas um) outro *Node* à espera, isto resulta na circulação de um pedido até este chegar a um *Node* que detém ou futuramente deterá (que está à 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 *Node* ao qual o detentor do objeto está ligado (com este tipo e ligações), o último elemento da fila será o *Node* na fila que não tem qualquer outro *Node* à sua espera e esta fila está vazia quando não há nenhum *Node* à espera no atual detentor do objeto.

3.4 Atributos do *Node*

Nesta 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*.

	Owner Terminal	Owner With Request	Idle	Waiter Terminal	Waiter With
Find	X	X	X	X	X
MyChan	X	X	X	X	X
Link		X	X		
Obj	X	X			
WaiterChan		X			X

Link

Este atributo representa a ligação do *Node* para o "Find" de um outro *Node*, ou seja, uma ligação para o seu *Parent 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 subsecção serão definidos os atributos que cada tipo de *Node* apresenta. Os atributos de cada *Node* estão apresentados na tabela 3.4

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 **Child 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 descritos os comportamentos que os *Nodes* podem ter, tais como as razões que despoletam estes comportamentos.

Owner Terminal

Receção de um pedido Access Request

O *Node* recebe um pedido ***Access Request*** no seu *Channel* ***Find***, que foi remetido pelo seu *Child 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 *Child Node*, havendo uma inversão da ligação entre o *Node* e o *Child 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 *Child Node*.

Este envia pelo ***Link*** o ***WaiterChan*** do pedido ***Access Request*** e o seu *Channel* ***Find***. É enviado o "Find" para que o seu *Parent Node* tenha a possibilidade de inverter a sua ligação, 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 *Child 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 *Child Node*.

Este envia pelo ***Link*** o ***WaiterChan*** do pedido ***Access Request*** e o seu *Channel* ***Find***. É enviado o "Find" para que o seu *Parent Node* tenha a possibilidade de inverter a sua ligação, 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 *Child Node*. **Idle(find, MyChan, Link)**.

Realização de um pedido de acesso

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

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

É enviado o "Find" para que o seu *Parent Node* tenha a possibilidade de inverter a sua ligação, 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 *Child 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 *Child Node*.

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

É enviado o "Find" para que o seu *Parent Node* tenha a possibilidade de inverter a sua ligação, 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 *Child 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 descritos todas as alterações em cada elemento após vários acontecimentos, tais como quando um *Node* recebe um pedido ou quando um *Node* recebe ou cede o acesso ao objeto. A separação das várias propriedades destes elementos definidas na descrição serviu de base para a implementação deste, com principal destaque ao comportamento de cada tipo de nó e a mudança de estado após inicialização deste.

Capítulo

4

Engenharia de Software

4.1 Introdução

Neste capítulo irá ser descrito o planeamento da implementação do projeto, e decisões tomadas antes de qualquer desenvolvimento, sendo o facto de se planear implementar um sistema distribuído ao invés de apenas concorrente o fator que mais influenciou essas decisões. Estas decisões incluem a Linguagem usada na Implementação, o método usado para testar o funcionamento da implementação, o método de comunicação entre os Nós visto que se optou por um implementação distribuída e o modelo do sistema.

4.2 Sistema Distribuído

Como referido no capítulo 1, a proposta inicial consistia em simular o protocolo num único programa, mas o desenvolvimento de um sistema distribuído provaria-se uma implementação de maior interesse do algoritmo e "fiel", pois o ADDP foi desenvolvido para esse mesmo contexto.

Também foi ponderada a utilização de uma linguagem que fornecesse comunicação por canais, visto que seria a noção mais próxima da comunicação entre os *Nodes* tal como foi definida no *paper* de referência, na qual se escolheu a Linguagem *Go* (ver), pois seria necessária a comunicação entre vários *Threads*, estes simulando *Nodes*. No entanto, devido à alteração do contexto da implementação de Concorrente para Distribuído, não foi possível fazer uso deste tipo de canais, e optou-se por usar outra tecnologia que garantisse a comunicação remota entre os vários *Nodes*.

4.3 Linguagem Go

A linguagem *Go* facilita o desenvolvimento de sistemas concorrentes, pois, nativamente, oferece várias ferramentas para o desenvolvimento deste tipo de sistema.

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*** gerida 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 na especificação de procedimentos concorrentes, a linguagem oferece bibliotecas nativas 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 recebe 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 (ver 4.5) que, por omissão, todos os pedidos HTTP que o servidor recebe são tratados em *goroutines* diferentes, 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".

4.4 Visualização do Sistema

Foi decidida a implementação de uma visualização do sistema. Esta visualização permitiria a testagem da programa durante a implementação do sistema,

tornaria mais fácil a compreensão do funcionamento do sistema e possivelmente provar o seu bom funcionamento.

Surgiram várias hipótese de implementação, como a utilização de bibliotecas de bibliotecas gráficas para desenvolver a interface de visualização, no entanto o suporte destas bibliotecas na Linguagem *Go* não era o desejado, ficando então decidido o uso de uma página *Web*.

Mesmo que a atualização da interface numa página *Web* não fosse tão eficiente como a de uma biblioteca gráfica, esta provou-se útil pelo facto de ser possível aceder a esta interface fora do sistema, isto é, sem ser necessário que uma das máquinas do sistema distribuído possuísse um monitor, e também garantindo o acesso remoto a esta mesma visualização.

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.

4.5 Comunicação entre os *Nodes*

Como referido em na secção 4.2, o facto de se ter planeado implementar um sistema distribuído em vez de um programa concorrente, tornou-se impossível o uso de comunicações por canais do *Go*, e para a comunicação seria necessário um método de comunicação por rede.

Inicialmente procurou-se reproduzir o conceito de comunicação por canais da linguagem *Go*, mas que esta fosse por rede, ou seja, algo que possivelmente usa-se uma sintaxe ou conceitos semelhantes, sendo que as únicas soluções encontradas ou já não estavam em desenvolvimento, ou seja, possivelmente existiriam *Bugs*, ou a biblioteca nativa *netchan*, que se encontrava descontinuada(*deprecated*). Também seria possível obter o mesmo efeito através de *RPC*, para o qual seria necessário bibliotecas externas, ou *Sockets*.

No entanto usou-se comunicação por HTTP, visto que este protocolo iria também ser usado na visualização, pela qual são enviadas mensagens com o formato JSON, uma vez que, tanto a linguagem *Go* como a linguagem *JavaScript* fornecem mecanismos de serialização de mensangens deste formato nativamente.

4.6 Uso de *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*, 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* pode ser considerado *Node* do sistema ou uma máquina que cada um tem uma instância do programa a correr, o seu próprio endereço IP e que podem comunicar entre si.

A implementação deste programa permite a execução de um serviço distribuído sem o uso desta ferramenta, pois esta foi usado apenas para o *Deploy* mais rápido e simples para testar durante o desenvolvimento do projeto.

4.7 Modelo do Sistema

Há vários Nodes, que são instâncias do programa *Node*, estes que usam ligações numa rede.

As ligações feitas entre entre os *Nodes* do sistema usam o protocolo HTTP, pelas quais são enviadas mensagens no formato JSON.

Para ser possível a visualização do estado da rede, cada *Node* envia regularmente a informação do seu próprio estado para um *Node* especial, este que também pertence à mesma rede.

Quando se pretende visualizar a informação que se encontra no *Node* de visualização, faz-se uso de uma página *Web*, a qual faz pedidos constantes dessa mesma informação e a dispõe numa animação que vai evoluindo com a alteração do estado da rede. Usando esta página é também possível forçar um ou vários *Nodes* a realizar um pedido de acesso, em que a página envia um pedido (para cada *Node* a forçar) para o *Node* de visualização. Este *Node* depois enviará os pedidos aos *Nodes*.

Vejamos os casos de uso descritos no diagrama de casos de uso 4.1, no qual é possível observar que o utilizador tem a possibilidade de interagir com o sistema, ao ter a poder forçar um *Node* em específico ou todos ao mesmo tempo a realizar um pedido de acesso, também pode consultar várias informações do sistema, como as diferentes estruturas de dados distribuídas e históricos de eventos e também tem a possibilidade de interagir com a interface, quando decide parar as atualizações da interface ou ativar/desativar o modo de filas, no qual passa a mostrar as ligações das filas ao invés das do diretório.

Os possíveis casos de uso da interface estão descritos no diagrama 4.1.

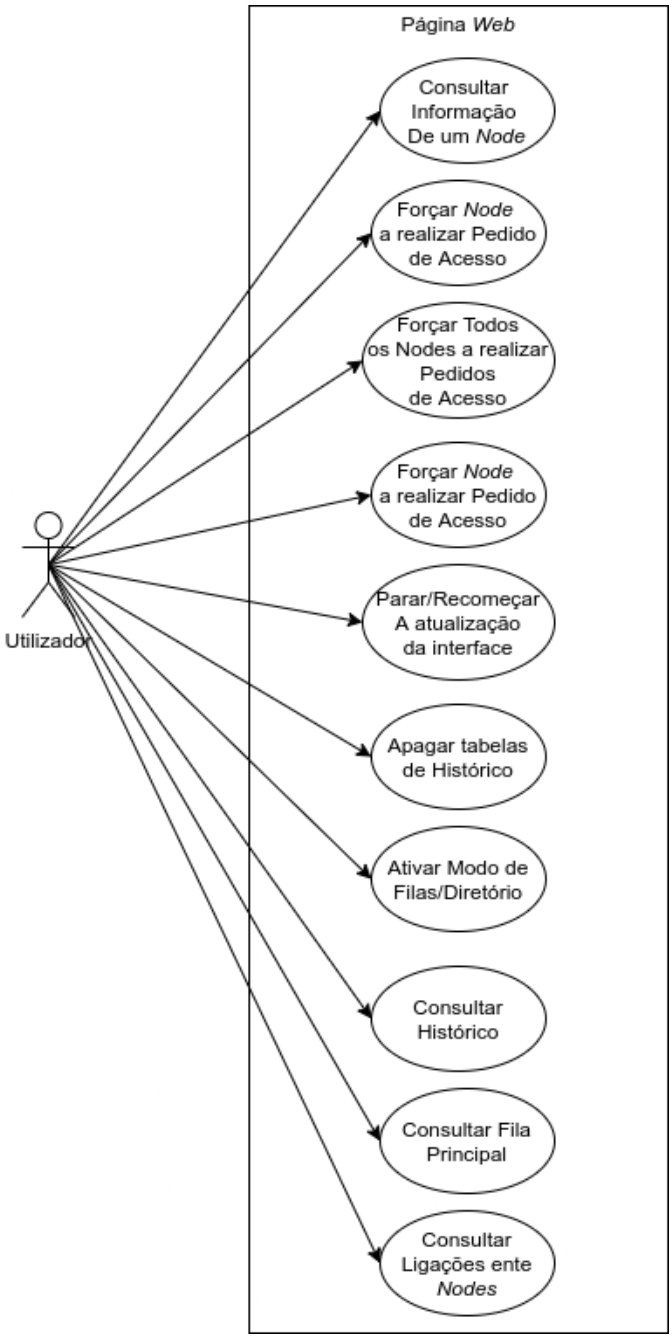

Figura 4.1: Diagrama de Casos de Uso da Interface de Visualização.

Este sistema é feito das várias ligações entre os vários componentes do sistema, isto é, ligações entre *Nodes*, ligações entre os *Nodes* e um *Node* de

visualização e entre o *Node* de visualização e uma página *Web*. No diagrama 4.2 é possível observar que cada nó se pode ligar no máximo a um único nó, mas que vários nós podem ligar-se ao mesmo e que o envio de pedidos ou do acesso ao objeto pela rede formada por estas ligações pode ser feito nos dois sentidos. Também é possível observar que cada nó comunica com um nó de visualização, este que recebe atualizações de estado dos nós, e também tem a possibilidade de forçar um ou todos os nós caso o utilizador o pretenda fazer. É possível observar que o nó de visualização serve de nó intermédio na atualização da interface a partir das atualizações que recebe dos nós.

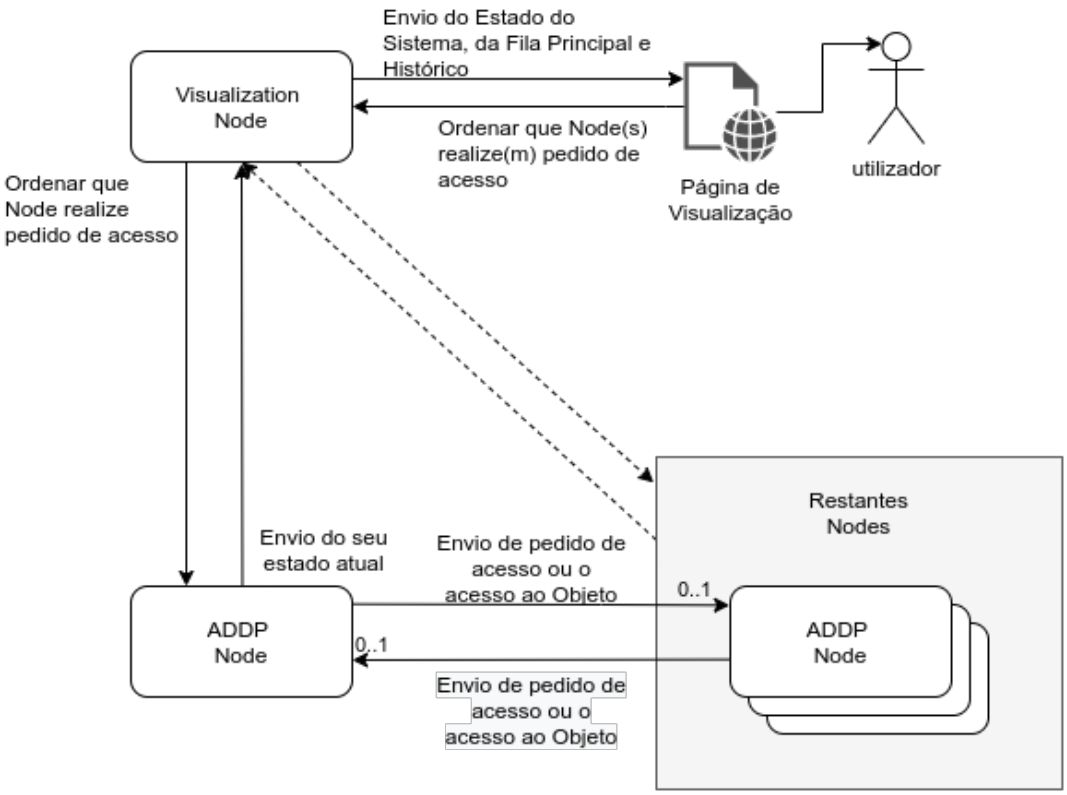

Figura 4.2: Arquitetura do Sistema.

O próprio *Node* para além de possuir os atributos descritos no capítulo 3.4, este também possui um Tipo e o endereço do *Node* de visualização para que seja possível a atualização do seu estado na interface. O *Node* pode sofrer transformações, receber pedidos de acesso ou o acesso ao objeto, realizar um pedido de acesso, que, para que não seja necessária a intervenção do utilizador, este último comportamento pode ocorrer de forma automática. Estes comportamentos foram descritos no capítulo 3.6, e para tal foi desen-

volvida uma classe "*Node*" tendo como atributos os atributos do *Node* e os comportamentos como os seus métodos. O diagrama de classe do *Node* está apresentado na ilustração 4.3, no qual é possível observar que a classe "Node" implementa os métodos de transformação, de comunicação através das ligações, "SendThroughLink" e "SendObjectAccess", métodos de realização de pedidos, o "Request", e como referido anteriormente, o nó tem a possibilidade de realizar um pedido automaticamente, fazendo uso do método "AutoRequest", e por último implementa dois métodos que revela informação sobre o nó, estes sendo o "OutputState", que escreve na consola no qual o programa se encontra em execução e e "UpdateVisualization" que é utilizado quando o nó pretende atualizar o seu estado atual no nó da visualização. Também está definido um "Enum", que representa os vários nós como descrito na secção 3.2 e duas classes que representam os dois tipos de *Channels* que podem ser comunicados entre os nós.

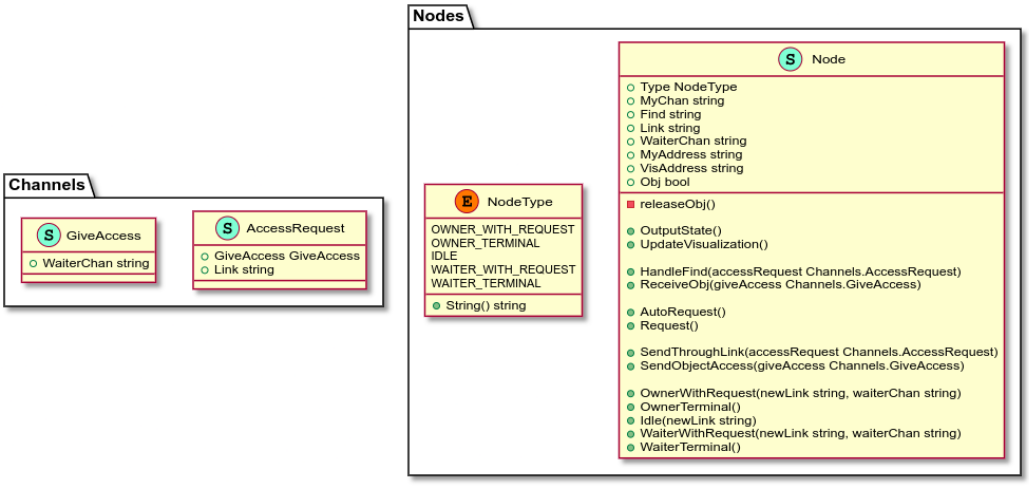

Figura 4.3: Diagrama da Classe "*Node*".

4.8 Conclusão

Neste capítulo foi exposta a o porquê da da mudança do projeto de um programa concorrente para um sistema distribuído e as alterações que essa decisão implicou no trajeto do projeto, os motivos que levaram à escolha da Linguagem *Go*, o planeamento de uma visualização do protocolo e do sistema em execução para uma prova parcial e de como seria implementada essa mesma interface, o uso da ferramenta/tecnologia *Docker* para o *Deploy* do sistema, a

escolha do método de comunicação entre os *Nodes* e o modelo do funcionamento do sistema.

Capítulo

5

Implementação

5.1 Introdução

Neste capítulo serão descritos todos os elementos envolvidos na implementação do protocolo, como as tecnologias utilizadas, a razão pela sua escolha. É feita a descrição dos pontos de maior interesse da implementação do *Node* do sistema, que foi desenvolvida após a escrita da Especificação (3). Esta parte da implementação foi dividida em vários pontos de interesse, sendo estes: a classe *Node* e os seus atributos, os comportamentos dos *Nodes*, as suas transformações entre os cinco possíveis tipos e a comunicação entre os *Nodes*. Também é descrita a implementação de um *Node* especial, usado para a visualização, para o qual são enviadas atualizações dos *Nodes* que existem no sistema, que são usadas para a renderização da visualização e reconstrução das estruturas de dados distribuídas, estas que também são demonstradas na visualização.

5.2 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.

Make Ferramenta de execução automática de comandos.

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.

5.3 Construção do Sistema

Na implementação deste projeto foi necessário testar os *Nodes* num contexto distribuído, no entanto, para que não seja necessário a execução de várias instâncias do programa em várias máquinas diferentes, fez-se uso de *containers Docker* para se obter o mesmo efeito.

Além disso, o uso desta ferramenta provou-se útil na execução de múltiplas instâncias do programa, não apenas como "Máquinas Virtuais", de forma rápida e simples.

Na execução deste sistema para testes fez-se uso de três sistemas de construção, sendo estes "docker", "docker-compose" e "Make", cada um dependendo do anterior, por esta ordem.

Com o uso de estes sistemas de construção apenas é necessário executar o comando "make run" para executar um sistema com um *Node* de visualização e um número arbitrário (mas definido num ficheiro "docker-compose.yml", ver capítulo seguinte) de *containers* que executam o programa do *Node*.

Como referido no capítulo 4, o *Node* de visualização faz uso de uma página *Web* para demonstrar o estado (conhecido) do sistema. O endereço desta página depende do ficheiro "Dockerfile" (ver subsecção 5.3).

docker

Este sistema é usado na criação de *docker containers* a partir da leitura de 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.

Para a execução de *Containers* que executam o programa do *Node* foi usado o seguinte "Dockerfile".

```
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 5.1: "Dockerfile" do *Node*

Para a execução de *Containers* que executam o programa do *Node* foi usado o seguinte "Dockerfile".

```
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 . .

# compilacao do programa do \emph{Node} de visualizacao
RUN go build -o vis

# execucao da instancia do \emph{Node} de visualizacao
CMD ./vis
```

Excerto de Código 5.2: "Dockerfile" do *Node*

docker-compose

Foram criadas várias tipologias 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, sendo que este ficheiro faz uso de ficheiros "Dockerfile".

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 atributos 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 atraves da visualizacao
  network_mode: host
```

Excerto de Código 5.3: Ficheiro docker-compose.yml

Make

Na execução do sistema foi usado o sistema "Make", que executa comandos do Sistema Operativo. Para tal é necessário um ficheiro "Makefile" onde são

descritos os vários comandos que serão executados ao executarmos o comando "make".

Por exemplo, caso se pretenda executar todo o sistema, apenas é necessário executar o comando "make" com o argumento (ou regra) "start", ou seja, "make start", invés de se executarem 7 comandos.

O "Makefile" usado neste projeto é o seguinte:

```
# constroi a imagem do \emph{Node} de visualizacao
build_viz:
  docker build -t vis ./visualization

# comeca o \emph{container} que executa o \emph{Node} de visualizacao
start_viz:
  -docker stop vis
  -docker rm vis
  docker run -it --net=host --env   address=:8000 --publish 8000:8000 --
      detach --name vis vis:latest

# comeca os \emph{containers} que executam os varios \emph{Nodes}
start_containers:
  -docker-compose stop
  docker-compose up --build --force-recreate -d

# Para a execucao de todos os \emph{containers} usados pelo sistema
stop_all:
  -docker stop vis
  -docker-compose stop


# Executa todas as regras necessarias para a execucao do sistema
run:
  $(MAKE) stop_all
  $(MAKE) start_viz
  $(MAKE) start_containers
```

Excerto de Código 5.4: Ficheiro "Makefile"

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 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.

5.5 Atributos da Classe *Node*

Como referido no capítulo 3, 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 5.5: 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 *OwnerTerminal* ou *OwnerWithRequest*, este tem o acesso ao objeto.

5.6 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 de texto para o tipo correspondente ao do atributo (por exemplo, o tipo é transformado 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 5.6: *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, 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 5.7: Instanciação dos atributos do *Node*.

5.7 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 5.8: Definição da enumeração dos tipos de *Node*

5.8 Comportamentos

Como referido no capítulo 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 .

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 5.9: *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 5.10: 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 5.11: 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 5.12: 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 5.13: 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 5.14: 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. Como se prentende simular o funcionamento 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 5.15: *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 5.16: 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.

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 5.17: 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 *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 5.18: 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.

Como pretendemos demonstrar o protocolo em funcionamento sem necessidade de interação humana, 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*, para que os *Nodes* decidam fazer pedidos automaticamente.

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 5.19: Método "AutoRequest"

Este método permite que *Node* realize pedidos em tempo aleatório com uma probabilidade de 50%, ou seja, é possível o *Node* não realizar um pedido durante algum tempo.

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*.

No 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 5.20: Método "ReceiveObj"

5.9 Transformações do *Node*

Nesta secção 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.

As possíveis transformações são demonstradas na imagem 5.1.

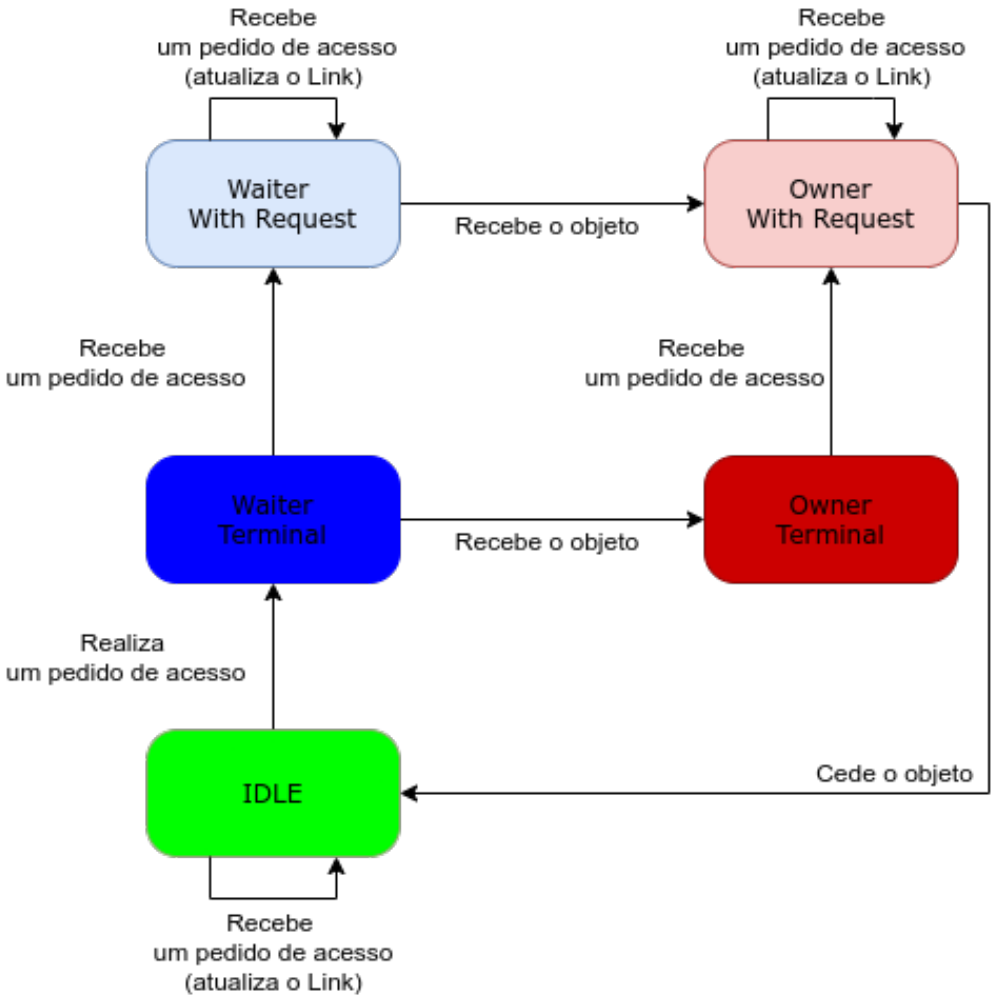

Figura 5.1: Diagrama de Estados do *Node*

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 5.21: 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 5.22: 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 5.23: 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 5.24: 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 “WAITER_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 5.25: Método/transformação "WaiterWithRequest"

5.10 Comunicação entre Nodes

Nesta secção 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 5.26: 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 5.27: *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 5.28: *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 5.29: *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 5.30: 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 5.31: 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 5.32: 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 5.33: Método "UpdateVisualization"

5.11 Classes de *Channels*

Nesta secção 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 5.34: *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 5.35: *Struct* "GiveAccess"

5.12 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, os detalhes desta não são centrais ao conceito do projeto, então não serão apresentados.

Por isso serão apenas tratados os conteúdo de maior interesse desta parte da implementação, como os métodos/caminhos do servidor HTTP do *Node* de visualização e como cada um processa a informação necessária, como, por exemplo, o *Node* de visualização tem acesso à informação atual do diretório e a formação da "Queue".

Servidor HTTP

Tal como descrito na secção de inicialização do servidor HTTP dos *Nodes* (5.10), são registados múltiplos 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 desafios/dificuldades que foram endereçadas, sendo estes:

- A atualização por parte dos *Nodes* não garante a ordem de chegada da informação, pois a rede (não o diretório) usada na comunicação de pedidos HTTP não garante a ordem de, por exemplo, o tempo de chegada de um pedido ao *Node* de visualização pode variar entre *Nodes*, e se há muita informação a circular na rede pode haver um engarrafamento na chegada de atualizações.

- O método de atualiação dos *Nodes* é executada numa *goroutine*, o que não garante que a execução deste método ocorra o mais rápido/cedo possível.
- Com todos os contragimentos referidos anteriormente é necessário sincronizar a atualização da informação sobre as várias filas possivelmente existentes no sistema, provocando uma alteração da ordem do processamento das várias atualizações.

Mesmo com as limitações referidas, é possível apresentar uma aproximação do estado da rede, usando vários mecanismos, que serão retratados nas subsecções seguintes.

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", da biblioteca "sync", 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. Este "Map" permite a leitura e escrita concorrente de dados por várias *goroutines* (estas que são inicializadas a cada pedido neste método), o que leva a uma atualiação da visualização de forma mais fluída e rápida.

Caso o estado contido na atualização seja de um *Node* do tipo "Idle" (2), esta atualização é ignorada.

Faz-se uso de um *Buffered Channel* "ChangeChannel", uma primitiva da linguagem "*go*", que permite o envio das atualizações das várias *goroutines* que processam a atualização do "Map" para uma *goroutine*. A razão de utilização de um *Channel* deve-se à propriedade de sincronização ordendado que estes dispõe. Como referido anteriormente, não é garantido que as atualizações chegam de forma ordenada, mas este mecanismo permite, pelo menos, a ordenação das atualizações por parte das *goroutines*.

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

defer r.Body.Close()

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


var update elements.Node
_ = json.NewDecoder(r.Body).Decode(&update)   //desserializacao da
    atualizacao numa struct do tipo Node
```

```
  // Remocao das prefixos e Sufixo http:// e metodo
  update.Link = re.ReplaceAllString(update.Link, '')
  update.WaiterChan = re.ReplaceAllString(update.WaiterChan, '')


  //Armazenamento no Map
  Nodes.Store(update.MyAddress, update)

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

  //Caso o estado da atualizacao que chegou seja de um Node nao IDLE (2)
  if update.Type != 2 {


    ChangeChannel <- update
  }
}
```

Excerto de Código 5.36: *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 uma ligação, seja esta uma ligação no diretório ou na fila, é instanciado um objeto do tipo "elements.Connection" tendo como atributos o endereço desse *Node* ("Source") e o "Link" ou o "WaiterChan" ("Target"), que ao final é adicionado à lista dos "tempLinks" ou "tempQueueCons", dependo do tipo de ligação (do diretório ou da fila).

```
  response := new(elements.VisResponse)

  var tempNodes []elements.Node
  var tempLinks []elements.Connection
  var tempQueueCons []elements.Connection

  Nodes.Range(func(key, value interface{}) bool {
    v := value.(elements.Node)
    tempNodes = append(tempNodes, v)
```

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

    return true
  },
  )

  response.Nodes = tempNodes
  response.Links = tempLinks
  response.QueueCons = tempQueueCons
```

Excerto de Código 5.37: Iteração pelo "Map" "Nodes", instanciação do objeto e adicionado à lista

Esta informação é serializada para o formato "JSON", que é enviada como resposta deste pedido HTTP. Este método é utilizado pela página *Web* para obter os dados a mostrar.

Queue

A implementação desta secção tem limitações, 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 as suas desvantagens. Serão apresentadas as duas possíveis soluções:

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*.

A atualização dos *Nodes* permite a reconstrução das *Queues* existentes no sistema, apenas usando o estado da atualização.

Esta é a informação que é possível retirar a partir de cada tipo de *Node* das atualizações, como por exemplo, a formação de novas *Queues*, a concatenação de duas *Queues* usando a informação do "WaiterChan" da atualização e a remoção de *Nodes* da *Queue* "Principal", quando o *Node* se transforma em *Owner*.

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 de acordo com o grafo que demonstra o diretório. Além disso é necessário a implementação de mecanismos para garantir que a falta de uma ordem da chegada dos dados não reconstrua as várias *Queues* de forma errada, e estes mecanismos podem não assegurar que uma grande discrepância na chegada de uma atualização não terá consequências no seu funcionamento.

No entanto, fez-se uso desta última solução, pois é a que demonstra a informação mas precisa.

Reconstrução das *Queues* através das atualizações dos *Nodes*

Todas as atualizações dos *Nodes*, à exceção das atualizações cujo o tipo do *Node* é "IDLE", são transmitidas para uma *goroutine* que executa uma função "NodeChange" de forma contínua, através de *channels* da linguagem *"Go"*. A inicialização da *goroutine* e execução da função é iniciada no iniciar do *Node* de visualização.

```
func nodeChange() {
  for { //ciclo infinito
    select {
    case newChange := <-ChangeChannel: //Leitura dos valores de um \emph
        {channel}
      QueuesMutex.Lock()   //Bloqueio e Desbloqueio de um \emph{Mutex}
          para a escrita
      updateQueues(newChange)
      QueuesMutex.Unlock()

        .
        .
```

```
            .
            .
        }
    }
}
```

Excerto de Código 5.38: Função "NodeChange" executada por uma "goroutine" e recebe as atualizações através de um *Channel*

Na implementação da secção seguinte foi usado um "Map" (não concurrente) que guarda informação dos *Nodes* que já estão numa fila, "NodesInQueues", e uma lista bidimensional que armazena as várias *Queues* com os seus elementos, "Queues". Será feita uma descrição da informação que é possível retirar de cada tipo de *Node*.

Waiter Terminal

Indica que há uma nova *Queue* no diretório.

Caso já exista um *Node* **X**, o qual o "WaiterChan" aponta para este, significa que esta atualização chegou depois da atualização do *Node* que recebeu o pedido, logo podemos concatenar a nova *Queue* com a *Queue* que tem o *Node* **X** como último elemento.

Se o *Node* já está em alguma *Queue* ignoramos esta atualização, pois isto significa que esta já é desatualizada.
Se o último elemento de alguma *Queue* tem um "waiterChan" que aponta para o *Node* que fez a atualização, significa que a atualização do *Node* que recebeu o pedido realizado pelo **X** chegou primeiro, e então juntamos o **X** à *Queue* do *Node* que aponta para este.

Se nenhum dos casos anteriores for verdadeiro, é adicionada uma nova *Queue* à lista de *Queues* com o **A** como único elemento.

```
    if _, isIn := NodesInQueues[update.MyAddress]; isIn {
      return
    }
    NodesInQueues[update.MyAddress] = update.Type
    for i, queue := range Queues {
      if len(queue) == 0 {
        continue
      }
      if queue[len(queue)-1].WaiterChan == update.MyAddress {
        Queues[i] = append(Queues[i], update)
        return
      }
```

```
    }
    Queues = append(Queues, []elements.Node{update})

    break
```

Excerto de Código 5.39: Alterações nas filas caso o *Node* seja do tipo "Waiter Terminal"

Waiter With Request

Indica que houve uma concatenação entre a lista a qual o *Node* da atualização, **A**, e a *Queue* do *Node* para o qual o "WaiterChan" do **A** pertence. Designamos o *Node* que fez a atualização que se está a processar de **A**, e o *Node* para o qual o "WaiterChan" do *Node* **A** aponta de **B**.

É possível garantir que esta ligação é verdadeira, no entanto é possível que tanto o *Node* **A** como o **B** ainda não esteja em qualquer uma das *Queues*, pois a sua atualização ainda não chegou.

O *Node* **A**, caso se esteja em alguma fila, este é o último elemento da fila, o *Node* **B**, caso se esteja em alguma fila, este é o primeiro elemento da fila.

Caso os dois já estão em filas, é feita uma concatenação das filas onde estes estão.

```
firstQueue := -1 //fila onde se apresenta o \emph{Node} \textbf{A}
secondQueue := -1 //fila onde se apresenta o \emph{Node} \textbf{A}


//Procura das filas onde cada \emph{Node} esta
for i, queue := range Queues {
  if queue[len(queue)-1].MyAddress == update.MyAddress {
    firstQueue = i
  } else if queue[0].MyAddress == update.WaiterChan {
    secondQueue = i
  }
}


//Se nao foi possivel encontrar uma das filas, ignoramos, pois as filas
    ja se encontram juntas
if secondQueue == -1 || firstQueue == -1 {
  return
}
```

```
//Concatenacao das duas \emph{Queues}
Queues[firstQueue] = append(Queues[firstQueue], Queues[secondQueue]...)

//Remocao da \emph{Queue} onde o \textbf{B} se apresenta
removeFromQueues(secondQueue)
```

Excerto de Código 5.40: Alterações nas filas caso o *Node* seja do tipo "Waiter With Request" e ambos os *Nodes* **A** e **B** se apresentam em filas

Caso o *Node* **A** esteja numa fila mas o **B** não, adicionamos o **B** ao final da fila onde o **A** se encontra, e caso o *Node* **B** esteja numa fila mas o **A** não, adicionamos o **A** ao início da fila onde o **B** se encontra.

```
//Caso o \textbf{A} se encontra mas o \textbf{B} nao.
if IsNodeAIn && !IsNodeBIn {
  for i, queue := range Queues {
    if queue[len(queue)-1].MyAddress == update.MyAddress {
      Queues[i] = append(Queues[i], nextNode)
      return
    }
  }
}

//Caso o \textbf{B} se encontra mas o \textbf{A} nao.
if !IsNodeAIn && IsNodeBIn {
  for i, queue := range Queues {
    if queue[0].MyAddress == update.WaiterChan {
      Queues[i] = append([]elements.Node{update}, Queues[i]...)
      return
    }
  }
}
```

Excerto de Código 5.41: Alterações nas filas caso o *Node* seja do tipo "Waiter With Request" e pelo menos um se apresenta numa fila.

Caso nenhum deles se encontra em filas, e como referido anteriormente, é garantido que há uma ligação entre os dois *Nodes*, logo temos a informação que há uma *Queue* em que os seus dois elementos são o **A** e o **B**, por esta ordem.

No entanto, ainda é possível saber se esta *Queue* já está ligada a outra, pois se existir uma *Queue* cujo o "WaiterChan" do último *Node* aponta para o **A** ou o "WaiterChan" do *Node* **B** aponta para o primeiro elemento de um fila, ou ambos.

```
//Formacao da \emph{Queue} com os elementos \textbf{A} (update) e o \
    textbf{B} (nextNode)
currentQueue := []elements.Node{update, nextNode}
```

```
currentQueueLocation := -1 //Localizacao desta \emph{Queue}
// -1 significa que ainda nao foi concatendada com nenhuma outra \emph{
    Queue}

for i, queue := range Queues {


  //Se o ultimo \emph{Node} de uma fila aponta para o \textbf{A}
  if queue[len(queue)-1].WaiterChan == update.MyAddress {

    // Caso a ''currentQueueu'' ainda nao foi concatenada
    if currentQueueLocation == -1 {

      //E concatenda a \emph{queue}
      Queues[i] = append(Queues[i], currentQueue...)

      //Armazenamos nova posicao da ''currentQueue''
      currentQueueLocation = i

    // Caso a ''currentQueue'' ja foi concatenada
    } else {

      //Concatenamos a \emph{Queue} com a \emph{Queue} onde a ''
          currentQueue'' se encontra
      Queues[i] = append(Queues[i], Queues[currentQueueLocation]...)

      //Removemos a \emph{Queue} onde a ''currentQueue'' se encontra
      removeFromQueues(currentQueueLocation)
    }

  //Se o ''WaiterChan'' do \emph{Node} \textbf{B} (nextNode) apontar
      para o primeiro de uma \emph{Queue}
  } else if nextNode.WaiterChan == queue[0].MyAddress {

    // Caso a ''currentQueueu'' ainda nao foi concatenada
    if currentQueueLocation == -1 {

      // Concatenamos a \emph{Queue} encontrada a ''currentQueue'', que
          passa a estar na mesma posicao (substitui)
      Queues[i] = append(currentQueue, Queues[i]...)

      //Armazenamos nova posicao da ''currentQueue''
      currentQueueLocation = i

    // Caso a ''currentQueueu'' ja foi concatenada
    } else {
      //Concatenamos a \emph{Queue} encontrada a \emph{Queue} onde a ''
          currentQueue'' se encontra
```

```
        Queues[currentQueueLocation] = append(Queues[currentQueueLocation
            ], Queues[i]...)

        //Removemos a \emph{Queue} encontrada
        removeFromQueues(i)
      }
    }

}
//Se nao mudou de sitio entao acrescentamos a queue a lista
if currentQueueLocation == -1 {
  Queues = append(Queues, currentQueue)
}
```

Excerto de Código 5.42: Alterações nas filas caso o *Node* seja do tipo "Waiter With Request" nem o **A** nem o **B** estão em filas.

Owner Terminal

Indica que uma *Queue* será removida da lista de *Queues* e indica qual o atual *Owner* do sistema.
Caso a *Queue* a ser removida tem mais elementos para além do *Node* que fez a atualização, isto indica que até à chegada desta atualização houve *Queues* a concatenar com a *Queue one* o *Owner Terminal* se encontra, e que esta atualização é desatualizada. Então nesse caso só é removido o *Node* da atualização da *Queue*.
Caso o *Node* que fez a atualização não se encontra em nenhuma *Queue*, isto significa que uma atualização da transformação do *Node* para *Owner With Request* chegou antes desta atualização, e que esta informação deverá ser ignorada.

```
// Se o \emph{Node} esta em alguma \emph{Queue}
foundNodeQueue := -1


//Indicacao que este \emph{Node} e o atual Owner
currentOwner = update


//Procura da \emph{Queue} cujo o primeiro element e o \emph{Node} que
    fez a atualizacao
for i, queue := range Queues {
  if queue[0].MyAddress == update.MyAddress {
    foundNodeQueue = i
  }
}
```

```
//Se ja nao se apresenta em alguma \emph{Queue} entao ignora-se a
    atualizacao
if foundNodeQueue == -1 {
  return
}

// Se so tiver um unico elemento (o \emph{Node} que fez a atualizacao)
if len(Queues[foundNodeQueue]) == 1 {

  // Eliminacao da \emph{Queue} encontrada
  removeFromQueues(foundNodeQueue)
//Se tem mais elementos
} else {
  //Removemos o primeiro elemento (o \emph{Node} que fez a atualizacao)
  Queues[foundNodeQueue] = Queues[foundNodeQueue][1:]
}

//Eliminacao do \emph{Node} da lista de \emph{Nodes} que estao e \emph{
    Queues}
delete(NodesInQueues, update.MyAddress)
```

Excerto de Código 5.43: Alterações nas filas caso o *Node* seja do tipo "Owner Terminal"

Owner With Request

Indica que chegou um pedido ao atual *Owner* ou que o *Node* para qual o "WaiterChan" do anterior *Owner* apontava (ou o primeiro da *Queue* "Principal"). Para além disso indica qual é a *Queue* "Principal".

Caso o *Node* é o primeiro de uma *Queue*, indica que o tipo anterior deste *Node* (conhecido) era "Waiter With Request", e o *Node* é removido da *Queue*. Caso o "WaiterChan" do *Node* da atualização aponte para o primeiro elemento de uma *Queue*, isto indica que essa *Queue* é a *Queue* "Principal".

```
delete(NodesInQueues, update.MyAddress)
currentOwner = update
for i, queue := range Queues {


  // O \emph{Node} e o primeiro elemento de uma \emph{Queue}
  if queue[0].MyAddress == update.MyAddress {
    if len(queue) > 1 {


      // Remocao do primeiro elemento da \emph{Queue}
      Queues[i] = Queues[i][1:]
```

```
      // Indicacao que esta \emph{Queue} e a ``Principal''
      temp := queue[1:]
      Queues[i] = Queues[0]
      Queues[0] = temp
      return
    } else {
      removeFromQueues(i)
    }

  //caso o \emph{Node} aponte para o primeiro elemento de uma \emph{
      Queue}
  } else if queue[0].MyAddress == update.WaiterChan {
    // Indicacao que esta \emph{Queue} e a ``Principal''
    temp := queue
    Queues[i] = Queues[0]
    Queues[0] = temp
  }
}
```

Excerto de Código 5.44: Alterações nas filas caso o *Node* seja do tipo "Owner With Request"

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 5.45: *Handler* "requestAll" do método "/requestAll"

Mesmo que o representado na visualização possa não ser o mais 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.

5.13 Conclusões

Neste capítulo foi apresentado todo o processo que envolveu a implementação deste projeto, desde as tecnologias, à implementação do *Node* e do *Node* de visualização. A implementação deste projeto segue uma interpretação da Especificação (3) na Linguagem Go, em que é possível executar um sistema distribuído. Existe um maior foco na implementação dos comportamentos dos *Nodes* visto que estes alteram o estado global do sistema. Foram também apresentados os desafios nesta implementação, principalmente no desenvolvimento da Visualização, provocados pela falta de um Relógio Global e o uso de uma Rede Assíncrona, na qual não é possível garantir a sincronia/sequência dos eventos dos *Nodes*.

Capítulo

6

Validação experimental, Simulação e Testes

6.1 Introdução

Neste capítulo será apresentada a interface fornecida na implementação tal como a sua utilidade tanto durante o desenvolvimento do projeto como para uma possível prova parcial do bom funcionamento deste. Irá ser feita uma descrição de como é disposta a informação da rede/sistema, como são representados os nós, as estruturas de dados distribuídas e do histórico de acontecimentos na rede. Este histórico será o elemento principal na prova parcial da implementação, pois para o bom funcionamento do sistema é necessário que dois tipos de eventos tenham a mesma ordem, estes sendo, as entradas dos *Node* na fila principal (que está diretamente ligada ao atual detentor do objeto) têm de seguir a mesma ordem que a chegada do acesso do objeto aos *Nodes* que o pediram o acesso a este.

6.2 Interface de Visualização

Para demonstrar o funcionamento e o estado do sistema, foi desenvolvida uma interface gráfica. Nesta secção serão descritos os componentes apresentados na interface de visualização.

Esta interface contém a informação sobre as duas estruturas de dados presentes no sistema, tabelas que contêm o histórico de vários eventos no sistema, e botões para interagir com o sistema e a visualização.

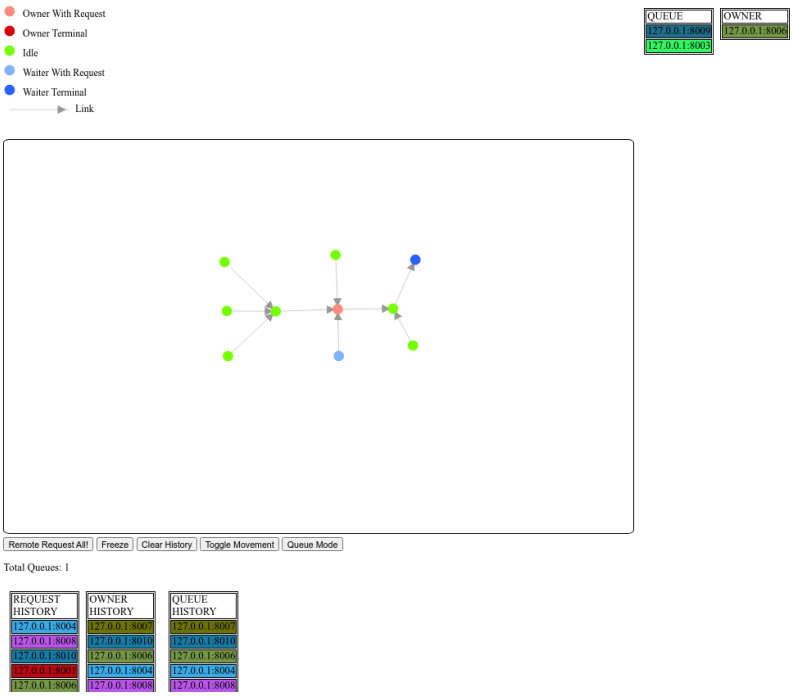

Figura 6.1: Visão geral da interface.

Representação das Estruturas de Dados

Na parte central da interface está apresentado um grafo, no qual os círculos representam os *Nodes* e as setas representam os *Links*. Também está presente uma etiqueta que indica qual o tipo do *Node* corresponde à cor de um *Node*. As ligações (*Links*) entre os *Nodes* e as cores dos *Nodes* são atualizadas dependendo do estado conhecido do sistema.

Ou seja, há uma animação que vai evoluindo conforme o estado da rede conhecido pelo *Node* de visualização, na qual é possível acompanhar que o comportamento de cada *Node* altera a estado do rede.

É possível saber mais informação sobre cada *Node*, deixando o ponteiro do rato por cima de um *Node* e é possível forçar um *Node* a realizar um pedido ao clicar neste.

Existe também o modo de filas/*Queue*, que após pressionar o botão *Queue Mode* (ver subsecção 6.2) as setas entre nós passam a demonstrar as ligações de filas/*Queues* entre os *Nodes*, ao invés das ligações do diretório.

Também estão apresentadas duas tabelas ao lado direito, com os títulos *Queue* e *Owner*, que representam a *Queue* "Principal" e o atual *Owner* conhecido.

Históricos

Na parte inferior da interface estão apresentadas 3 tabelas que contém informação sobre os seguintes históricos:

Request History Histórico dos pedidos realizados pelos *Nodes*.

Owner History Histórico dos *Nodes* que tiveram o acesso ao objeto.

Queue History Histórico dos *Nodes* da fila principal, isto é, a ordem da futura chegada do objeto aos *Nodes*.

Esta informação não representa o estado da rede, mas é utilizada em testes. No entanto, o seu uso será descrito na secção 6.3.

Botões

Na interface estão disponíveis 5 botões diferentes que permitem a iteração com o sistema e a visualização. A funcionalidade de cada botão é a seguinte:

Remote Request All - Força todos os *Nodes* conhecidos a realizarem pedidos. Usado para testes.

Freeze - Para qualquer atualização da interface de visualização.

Clear History - Apaga as tabelas de histórico.

Toggle Movement - Bloqueia o movimento de todos os círculos que representam os *Nodes* no grafo.

Queue Mode - Ativa o modo de demonstração das filas.

6.3 Testes

No desenvolvimento foi necessário testar a implementação e também provar o seu bom funcionamento, no entanto não foi possível fazer-se uso de métodos formais de prova deste sistema, nem de comparar esta implementação com outra existente, visto que, a implementação feita no decorrer deste projeto é muito diferente da única conhecida [5]. Ainda assim é possível testar o bom funcionamento do sistema fazendo o uso dos elementos presentes na interface gráfica disponibilizada.

Nem todos os elementos da visualização estão atualizados em qualquer momento, por exemplo, ambos o grafo e as tabelas da fila “Principal” podem estar em desacordo com o estado real do diretório, pois ambos dependem da

última informação conhecida pelo *Node* de visualização, que pode ter um *Delay* causado tanto pela atualização (ou *Refresh Rate*) da interface, a rede usada para comunicação, a sincronização da chegada de informação, etc., mas ao longo do decorrer do sistema, estes serão corrigidos.

É possível, na demonstração gráfica do diretório (o grafo) ser mostrado um estado impossível, como, por exemplo, mostrar vários *Owners*, sendo que na realidade só existe um único, ou uma ligação em falta, porém, a implementação estaria errada caso um *Node* se ligar a um *Node* que previamente não era seu vizinho, pois, como definido no protocolo estudado, os vizinhos de um *Node* são sempre as mesmos, apenas as ligações entre estes é que são alteradas (invertidas).

No entanto, o uso mais indicado do grafo e das tabelas da fila e atual *Owner* é de visualização e demonstração de como funciona o sistema/protocolo, e não como prova, devido aos problemas de atualização presentes.

Para uma evidência mais forte, fez-se uso das 3 tabelas dos históricos. Como referido anteriormente, a tabela *Queue History* mostra o histórico dos *Nodes* que entraram na fila "Principal", e a passagem do acesso ao objeto tem de seguir a ordem dessa fila, e a ordem por onde circula o objeto é mostrado na tabela *Owner History*, logo, caso as tabelas *Queue History* e *Owner History* demonstrem os *Nodes* com ordens diferentes, o sistema está errado, visto que o objeto não "seguiu" a ordem esperada.

A tabela *Request History* demonstra que a ordem pela qual os *Nodes* realizam pedidos ou que a ordem que *Node* de visualização recebe a atualização de que esses *Nodes* realizaram pedidos pode ser diferente da ordem na fila "Principal". Esta pode ser apresentada com uma ordem dos *Nodes* diferente à apresentada nas outras tabelas, visto que, caso um *Node* realize um pedido, este pedido pode demorar mais tempo a chegar a um *Node Terminal* do que, por exemplo, um *Node* que realize mais tarde um pedido e que esteja diretamente ligado ao *Node Terminal*.

É possível observar este fenómeno quando há *Nodes* muito distantes de um *Waiter Terminal* ou quando há um grande número de pedidos a circular na rede, que pode ser demonstrado usando o botão "Request All" disponível na interface.

6.4 Conclusões

Neste capítulo foi esclarecida a representação dos vários elementos do sistema, em que se dispõe um grafo para representar as várias ligações entre os nós, quer estas sejam ligações do diretório ou das filas, uma tabela que contém os endereços dos *Nodes* que estam presentes na fila "Principal". Foram

apresentados os usos do botão, estes que servem para a interação com a interface ou até mesmo com o sistema. A secção de maior interesse para este projeto é a secção de Testes, pois, recurreu-se ao uso do histórico dos eventos para se provar parcialmente o bom funcionamento da implementação conseguida no projeto, em que, para que o sistema não apresente falhas é necessário que a ordem de "chegada" dos *Nodes* à fila "Principal" (esta é a fila que tem o atual *Owner* como cabeça de fila) seja a mesma que os *Nodes* recebem o acesso ao objeto.

Capítulo

7

Conclusões e Trabalho Futuro

7.1 Conclusões Principais

Este projeto teve como objetivo principal o estudo e implementação do ADDP, um protocolo que permite a partilha de um objeto móvel através de um diretório, sem que todos os elementos detenham a localização deste e que garante o acesso mutuamente exclusivo a este. Num sistema distribuído definido por este protocolo, os vários nós que o formam têm a possibilidade de requisitar o acesso (exclusivo) a um objeto, fazendo uso das ligações a outros nós, as quais são invertidas após serem usadas na transmissão dos pedidos. Este mecanismo de inversão garante que a direção de qualquer ligação indique a atual ou futura localização do objeto (ou o acesso a este), evitando a necessidade de os nós difundirem a sua localização.

Esta característica garante uma maior escalabilidade do sistema, pois não é necessário que os nós transmitam a informação a todos os outros nós no sistema sobre a localização do objeto, e garantindo o acesso exclusivo a este ao enviá-lo ao nó ao qual o primeiro pedido a ser processado pertence, fazendo com que os seguintes pedidos sejam redirecionados ao futuro detentor. As várias ligações entre os nós e a informação pelas quais é transferida definem duas estruturas de dados distribuídas, sendo estas um grafo, mais especificamente uma árvore de extensão mínima formada pelas ligações de diretório onde são transmitidos os pedidos de acesso ao objeto, e uma fila formada pelas ligações exteriores ao diretório pelo qual circula o acesso ao objeto.

Após a leitura e estudo do artigo que introduz o ADDP [1], foi realizada uma especificação de todos os elementos que pertencem ao sistema, nomeadamente os vários nós, os seus atributos, as ligações entre estes, os tipos de mensagem que por estas circulam e os comportamentos do nó, que serviu de

base para a implementação.

Foi realizada uma implementação na Linguagem *Go* do protocolo, na qual foi possível explorar conceitos de concorrência, comunicação por canais, sincronização e exclusão mútua, na qual foi possível obter um programa que é executado num sistema distribuído. Para a visualização deste sistema, foi desenvolvido o programa de um nó especial com o propósito de reconstruir o estado do sistema. Através de atualizações que cada nó envia periodicamente a este, este nó é capaz de reconstruir o grafo que representa o diretório, as filas existentes no sistema e, sem uso de *timestamps*, reconstruir a ordem dos eventos ocorridos, usando regras do protocolo para obter a sequência. Estas atualizações são depois solicitas por uma página *Web* desenvolvida para a visualização da informação que se encontra no nó de visualização, as quais são usadas para o desenho de um grafo que representa o diretório e tabelas que dispõem a ordem das tabelas e dos elementos das filas. Para além de dispor o estado do sistema, esta página é também interativa, na qual é possível forçar *Nodes* do sistema a realizar pedidos de acesso, ou controlar a animação.

Por fim, ordem de reconstrução dos eventos foi necessária para provar (parcialmente) o funcionamento da implementação feita. Neste sistema existem dois tipos de acontecimento no sistema que têm de seguir a mesma ordem, estes sendo, a entrada de um nó na fila "Principal", fila de espera que está ligada ao nó detentor do objeto, e a chegada do objeto a um nó e foi necessário reconstruir esses mesmos acontecimentos usando apenas as atualizações, estas que não seguiam a ordem de acontecimento nem faziam uso de *timestamps* para as sincronizar. Então foi necessário usar outros fatores no qual há a certeza da ordem, como a mudança do tipo de um nó, ou o facto de se conhecer que um nó está à espera de outro para obter o acesso ao objeto para ser possível a reconstrução da ordem pela qual os nós entram na fila e a ordem que o (acesso ao) objeto segue.

7.2 Trabalho Futuro

A parte gráfica desenvolvida neste trabalho poderia ser utilizada na visualização de outros algoritmos como, por exemplo, os trabalhos referidos na secção 2.3 , o "Ivy" e "Arvy", pois esta foi implementada de mode a ser independente do algoritmo "Arrow", e assim também seria interessante implementar esses mesmos algoritmos e fazer comparações entre estes.

O programa do *Node* poderia ser alterado de modo a ser utilizado como uma biblioteca externa, e aplicado em Sistemas de Ficheiros Distribuídos ou Bases de Dados Distribuídas, em que as várias Bases de Dados partilhariam docu-

mentos/informações fazendo seguindo o protocolo estudado, permitindo o acesso exclusivo a um documento/informação e evitando que algumas das Bases de Dados se tornassem em "Hotspots".

Também seria interessante a adaptar o trabalho atual para, ao invés de ser partilhado apenas um objeto, fosse possível que existissem vários diretórios, e que para tal seria necessário que as ligações entre os nós fossem formadas através de um algoritmo, pois seria trabalhoso especificar manualmente as ligações para cada diretório.

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.

[2] SecurityFocus Kevin Poulsen. Tracking the blackout bug, 2004. [Online] `https://www.securityfocus.com/news/8412`. Último acesso a 15 de maio de 2021.

[3] Kai Li. Ivy: A shared virtual memory system for parallel computing. *ICPP (2)*, 88:94, 1988.

[4] Pankaj Khanchandani and Roger Wattenhofer. The arvy distributed directory protocol. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, page 225–235, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Maurice Herlihy. Aleph Toolkit, 1999. [Online] `https://www.securityfocus.com/news/8412`. Último acesso a 15 de maio de 2021.

[6] Silvan Mosberger. arvy, 2019. [Online] `https://github.com/Infinisil/arvy`. Último acesso a 18 de maio de 2021.

[7] Timothee Tosi. bully-algorithm, 2019. [Online] `https://github.com/TimTosi/bully-algorithm`. Último acesso a 15 de maio de 2021.