Trabalho de Conclusão de Curso

Um estudo do Modelo de Atores aplicado à concorrência de recursos

Monografia apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Bacharel em Ciência da Computação

Fellipe Souto Sampaio
Orientadora: Prof. Dra. Ana Cristina Viera de Melo

Universidade de São Paulo
Dezembro de 2016

Mussum Ipsum, cacilds vidis litro abertis. Si u mundo tá muito paradis? Toma um mé que o mundo vai girarzis! Detraxit consequat et quo num tendi nada.

Resumo

Mussum Ipsum, cacilds vidis litro abertis. Si u mundo tá muito paradis? Toma um mé que o mundo vai girarzis! Detraxit consequat et quo num tendi nada. Mé faiz elementum girarzis, nisi eros vermeio. Nec orci ornare consequat. Praesent lacinia ultrices consectetur. Sed non ipsum felis.

Abstract

Lorem ipsum dolor sit amet, perpetua evertitur tincidunt ei mei, et qui diam vivendo minimum, ea ius iisque virtute moderatius. Doming iuvaret habemus ne vim, eos cu malis nonumy partiendo, populo erroribus nec in. Ex his dicunt legimus corrumpit. Sonet patrioque nec cu, ea mentitum mediocrem iracundia sed.

Vix choro consequat philosophia ea, ius animal suscipiantur cu, mundi necessitatibus mei eu. Ut suas percipit vivendum has, ponderum patrioque adolescens cu eos. Mel expetenda cotidieque eu, quo ad illud consulatu eloquentiam, pri ne timeam quaerendum. Repudiandae voluptatibus necessitatibus est id, at eleifend ocurreret deseruisse mea. Cu per populo primis. Ne est solet placerat reprehendunt, mei at altera iriure accusata.

Begin at the beginning, the King said gravely, “and go on till you come to the end: then stop.”

Lewis Carroll, *Alice in Wonderland*

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

1.1 Motivação

Problemas concorrentes surgem de forma natural em todos os lugares, um simples cruzamento de automóveis ou um sistema de ferrovias podem ser descritos como um complexo problema de concorrência, onde diversos agentes disputam algum tipo de acesso ou recurso. O estudo da concorrência teve sua gênese muito antes de se tornar uma área dentro da ciência da computação.

O seu estudo como conhecemos hoje começou de fato na década de 60 e é atribuído a Dijkstra a publicação do primeiro artigo na área [1]. Durante os anos que se seguiram, a área recebeu inúmeras contribuições de cientistas da computação famosos, como Hoare, Milner, Lamport, Tanenbaum, Andrews entre outros. Com as contribuições vieram teorias, modelos, técnicas e novos problemas para serem estudados.

Percebeu-se, com o passar do tempo, que lidar com um problema concorrente ou escrever um software que funcione desta forma não se tratava de uma tarefa trivial. É da natureza humana pensar de forma sequencial e linear, nosso cérebro não está habituado a ter que trabalhar com vários problemas ao mesmo tempo, ao passo que uma máquina de propósito geral é idealizada para fazer isso. Dada a atual conjuntura do mercado de computadores, de aumentar a quantidade de unidades de processamento em um único processador, saber como 'pensar' de forma concorrente e criar programas com as ferramentas certas, que façam o bom uso dessa capacidade computacional é determinante para resolver problemas concorrentes de forma correta e satisfatória.

1.2 Objetivos

Os objetivos deste trabalho são:

1. Aprender os fundamentos do modelo de atores
2. Entender a concorrência de recursos através do problema do Jantar dos Filósofos
3. Modelar o problema dos Filósofos e implementar três soluções diferentes utilizando a biblioteca Akka e a linguagem de programação Scala
4. Simular o problema e avaliar o desempenho dos algoritmos a partir de diferentes métricas

Capítulo 2

Fundamentos

Neste capítulo apresentaremos os fundamentos da programação concorrente e discutiremos alguns dos principais problemas que aparecem quando escrevemos um programa concorrente. Em seguida será apresentado o modelo de atores, uma forma alternativa de como pensar sobre concorrência e modelar problemas. Por fim, explicaremos sobre a linguagem de programação Scala, a fim de fornecer uma base de entendimento para o código que foi desenvolvido junto com a presente monografia, e sobre o Akka, biblioteca para desenvolvimento concorrente utilizando atores.

2.1 Computação Concorrente

Um programa de computador pode ser divido em duas categorias: sequenciais e concorrentes. O primeiro segue uma ordem muito bem delimitada de passos, recebe uma entrada, realiza computações e por fim nos devolve um resultado. Este comportamento se assemelha muito ao conceito de funções matemáticas, se conhecemos o seu funcionamento interno poderemos fornecer uma estado inicial e deduzir como será seu estado final [2][1]. O segundo por sua vez é um sistema mais complexo, composto por diversas partes independentes. São pequenos programas que executam de forma concorrente entre si, mas que se comunicam ativamente para que exista coesão e unidade na execução. A interpretação funcional utilizada para descrever um programa sequencial não funciona neste caso, a execução do sistema não segue uma ordem bem delimitada e a interferência entre as partes é não-determinística.

Nas duas categorias apresentadas existe um importante elemento para a computação: a comunicação. Diferentes partes de um programa precisam trocar informação, seja em uma execução sequencial ou concorrente. Usualmente as variáveis globais são o mecanismo mais fácil de se utilizar, possibilitam o compartilhamento de estruturas de dados, troca de mensagens e *buffers* de entrada e saída. Em um programa sequencial esse tipo de mecanismo funciona muito bem e não apresenta grandes problemas, no entanto para um programa concorrente a interação entre suas partes[2] utilizando essa técnica cria o o problema da **a exclusão mútua**, como veremos na subseção 2.1.3.

Quando N processos querem acessar um mesmo dado compartilhado é necessário que exista uma coordenação, caso contrário as interações com o recurso podem levar o programa a produzir resultados imprevisíveis. O tempo que cada processo leva para acessar esse recurso varia entre as execuções, criando o que é conhecido como **histórico de execução**.

Este histórico é o reflexo da execução do programa concorrente e do encadeamento das instruções atômicas[3] dos diferentes processos ao longo do tempo. Para elucidar o conceito considere o

[1] Página 2
[2] Daqui em diante indicaremos essas partes independentes como processos ou *threads*
[3] Instruções indivisíveis

seguinte exemplo: existem dois processos T1 e T2 sendo executados, onde cada um possui apenas uma instrução atômica, os possíveis históricos de execução sâo:

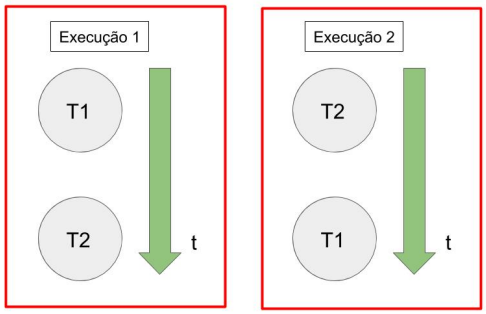

Figura 2.1: *Execução serial e sequencial de dois processos*

Neste caso não existe encadeamento, cada processo começa e termina sua execução, similar a um programa sequencial, no entanto se considerarmos que cada processo possui mais do que uma instrução atômica teremos:

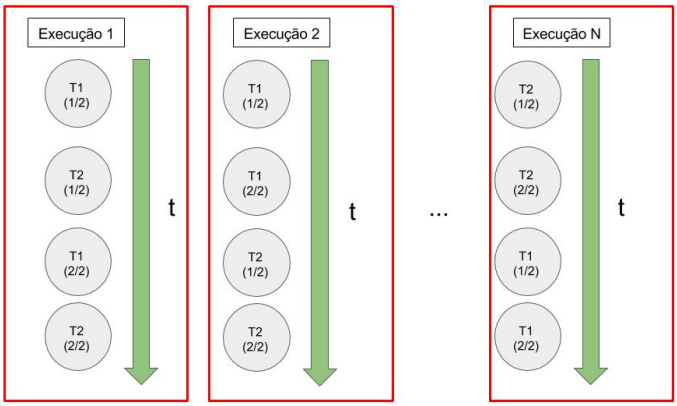

Figura 2.2: *Execução serial e concorrente de dois processos*

Na segunda figura temos encadeamento das instruções, onde cada execução gera um histórico diferente. Diferentes históricos podem gerar o mesmo resultado, todavia queremos ter um controle mais preciso de quais deles serão possíveis de acontecer, descartando os indesejáveis e ficando com um conjunto menor, o qual podemos ter maior previsibilidade dos resultados.

A programação concorrente tenta solucionar este e outros problemas através da implementação de escalonadores de processos justos, protocolos de acesso e execução, algoritmos paralelos e concorrentes, decomposição de problemas, modelos matemáticos entre outras técnicas. As principais vantagens que se pode obter com programas concorrentes são:

- Aumento da velocidade de execução do programa usando diversos núcleos de processamento ao mesmo tempo
- Redução da ociosidade das unidades computacionais, principalmente em programas que envolvem muita entrada ou saída de dados
- Melhor estruturação de programas em unidades independentes e com tarefas delimitadas
- Melhor modelagem para problemas que envolvem comunicação devido a sua natureza concorrente

2.1.1 A Condição de Corrida

A condição de corrida é um comportamento, presente tanto em hardware como em software, onde o resultado de uma computação depende diretamente do encadeamento das instruções que a compõe, de forma que cada histórico de execução pode levar a uma solução diferente. Considere o seguinte exemplo: temos dois processos, P_1 e P_2, por motivo de simplicidade apenas um processador, cada um deles irá executar concorrentemente um trecho do código a seguir:

```
1:  procedure RaceProcess
2:      X ← 0
3:      Y ← 1
4:      Z ← 2
5:
6:      CO
7:          //Trecho de P1
8:          X ← Y + Z                          ▷ Operação não atômica
9:
10:         //Trecho de P1
11:         Y ← 4                              ▷ Operação atômica
12:         Z ← 5                              ▷ Operação atômica
13:     OC
```

Neste código vemos que P_1 tem como instrução a linha 8 e P_2 as linhas 11 e 12. Dizemos que uma operação é atômica quando ela é indivisível e sua execução parece instantânea para todo o sistema [3][4], no nosso exemplo a instrução da linha 7 será decomposta em quatro ações diferentes:

- O valor de Z é carregado no registrador r_1
- O valor de Y é carregado no registrador r_2
- r_1 é somado com r_2 e o resultado é guardado em r_1
- O valor de r_1 é copiado para X

Sendo assim a instrução da linha 8 não é atômica, enquanto 11 e 12 são[5]. Podemos reescrever o código acima envolvendo as instruções atômicas entre <> :

```
1:  procedure RaceAtomicProcess
2:      X ← 0
3:      Y ← 1
4:      Z ← 2
5:
6:      CO
7:          //Trecho de P1
8:          < r1 ← Y >
9:          < r2 ← Z >
10:         < r1 ← r1 + r2 >
11:         < X ← r1 >
12:
13:         //Trecho de P1
14:         < Y ← 4 >
15:         < Z ← 5 >
16:     OC
```

[4] Páginas 42 e 43

[5] Estamos considerando a operação atribuição como atômica

O programa acima pode gerar diversos históricos de execução, onde cada um pode gerar um valor diferente de X como podemos ver na próxima imagem:

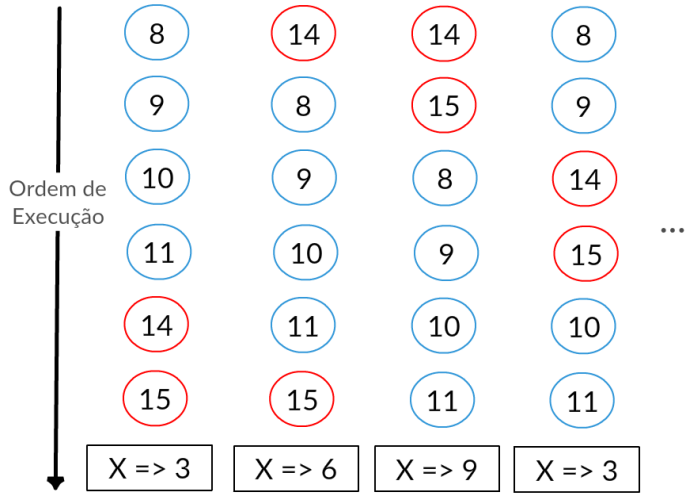

Figura 2.3: *Intercalações de P_1 e P_2*

Conseguimos identificar através da imagem acima uma *race condition* no uso das variáveis Y e Z. Uma possível solução seria escolher um valor esperado para X e reestruturar o código de forma a proteger o acesso e manipulação das variáveis compartilhadas, minimizando assim os possíveis históricos de execução que podem levar a resultados indesejados. Na próxima subseção mostraremos como os mecanismos de exclusão mútua podem ajudar nesse problema.

2.1.2 A Seção Crítica

O problema da seção crítica é um dos mais clássicos na programação concorrente, vem sido extensivamente estudado ao longo dos anos e possui precedentes em diversas áreas da computação, indo de sistemas operacionais a banco de dados . Podemos definir o problema da seguinte forma: Considere que N processos compartilham um recurso e seu acesso é coordenado através de um protocolo de entrada e de saída, o local onde o processo faz uso do recurso é a sua seção crítica. Teremos a seguinte estrutura:

```
1: procedure PROCESS CS[I = 1 TO N]
2:     while TRUE do
3:         Protocolo de entrada
4:         Seção crítica
5:         Protocolo de saída
6:         Seção não crítica
```

O objetivo é encontrar um algoritmo que coordene o acesso a seção crítica. Dijkstra estabelece em [4] que uma solução viável tem que respeitar os seguintes requisitos:

- Em qualquer momento da execução deve haver no máximo um processo na sua seção crítica
- A solução tem que ser simétrica entre os processos, não pode-se prejudicar um processo em detrimento dos outros
- Nada pode ser afirmado sobre a velocidade do processo e quanto tempo ele leva para completar suas ações

E Andrews estipula em [3][6] quatro propriedades desejadas:

- Exclusão Mútua - Apenas um processo deve estar na seção crítica por vez
- Ausência de *Deadlock* e *Livelock* - Se dois ou mais processos estão disputando a entrada na seção crítica ao menos um deles irá conseguir
- Ausência de Espera Desnecessária - Dado que um processo quer entrar em sua seção crítica, e um segundo processo está saindo da seção crítica ou está na seção não crítica, o primeiro não deve ter sua entrada impedida
- Entrada Garantida - Um processo que está tentando entrar na seção crítica irá eventualmente conseguir

Diversas soluções para esse problema foram apresentadas ao longo dos anos, como os algoritmos **Tie-Breaker, Ticket e Bakery** [3][7].

2.1.3 Mecanismos de Sincronização

No problema da seção crítica diversos processos disputam o acesso a um recurso compartilhado, é fundamental que essa aquisição seja coordenada através de um protocolo de entrada e saída, fazendo uso dos mecanismos de sincronização. Estes mecanismos desempenham papel determinante ao transformar o acesso ao recurso, antes concorrente, em serial.

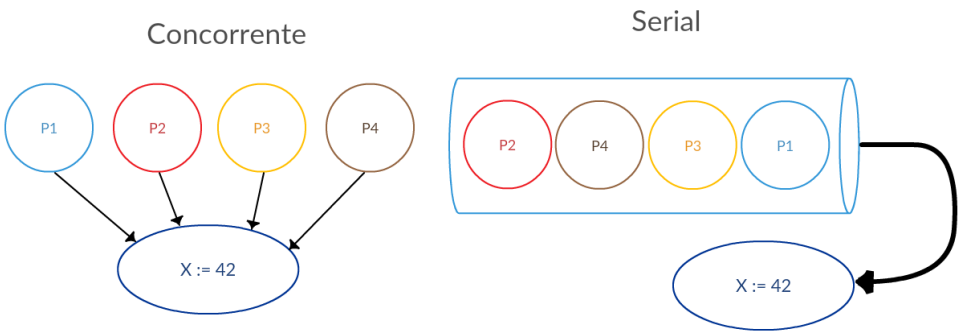

Figura 2.4: *Diferença entre o acesso concorrente e serial a um recurso compartilhado*

Examinaremos a seguir dois mecanismos de sincronização diferentes: **Semáforos** e **Monitores**. Idelizado originalmente por Dijkstra em [5], o semáforo é um tipo especial de variável compartilhada que aceita apenas duas operações P e V, que podem ser definidas da seguinte forma:

1: $sem \quad s \leftarrow 0;$ ▷ s pode assumir valores inteiros não negativos
2: $P(s): \quad < await(s > 0) \quad s = s - 1; >$
3: $V(s): \quad < s = s + 1; >$

A operação V sinaliza o acontecimento de um evento, como o fim da seção crítica, enquanto que a operação P atrasa um processo até que um evento aconteça, como podemos ver na imagem a seguir:

[6] Página 95
[7] Páginas 104, 108 e 111 respectivamente

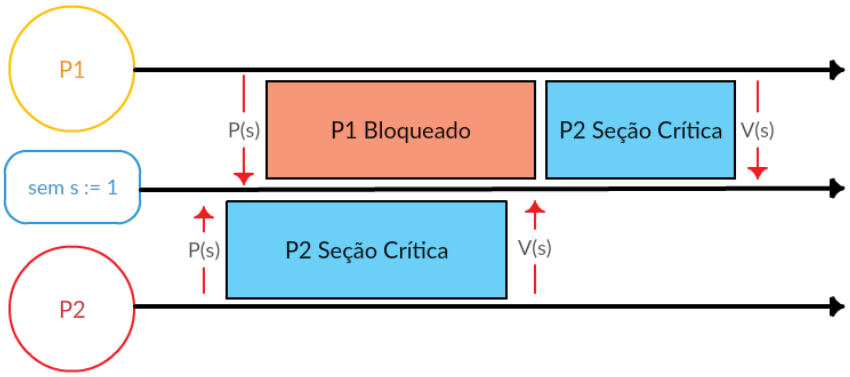

Figura 2.5: *Coordenação de acesso usando semáforos*

Quando o valor de um semáforo alterna apenas entre 0 e 1 o chamamos de **semáforo binário**, neste caso apenas um processo será autorizado a executar a seção crítica, o que verifica as condições da exclusão mútua. A diretiva *await* está relacionada com o atraso do processo até que seja permitido continuar a execução, na implementação da biblioteca POSIX os processos são colocados em uma fila e retirados a medida que as operações V ocorrem [6]. Utilizando semáforos uma solução para o problema da seção crítica seria:

```
1: sem   mutex ← 1;
2:
3: procedure PROCESS CS[I = 1 TO N]
4:     while TRUE do
5:         P(mutex);
6:         Seção crítica
7:         V(mutex);
8:         Seção não crítica
```

Semáforos são mecanismos de baixo nível por conta da sua expressividade e grau de abstração, a utilização das operações P e V devem ser feitas explícitamente para cada variável compartilhada, o que pode se tornar difícil e muito suscetível a erros, caso seja aplicado a estratégia errada a um dado caso.

O conceito teórico dos monitores foi apresentado por Hoare em [7], influenciado pela ideia de *Shared Classes* criado por Brinch-Hansen, disponível em [8][8]. Um monitor é um mecanismo abstrato para controle de recursos, ele dispõem de uma coleção de dados e métodos, como no exemplo a seguir:

[8]O conceito original de *Shared Class* foi apresentado em 1973, um ano antes de Hoare formular sua teoria. No livro citado o autor reapresenta as ideias originais do artigo

```
1: monitor Semaphore {
2:     int   s ← 1;                                  ▷ Dado local do monitor
3:     cond  pos;
4:
5:     procedure Psem(){
6:         while(s == 0)   wait(pos)
7:         s = s − 1
8:     }
9:     procedure Vsem(){
10:        s = s + 1
11:        signal(pos)
12:    }
13: }
```

Os métodos de um monitor serão visiveis a todos os processos em execução, no entanto apenas um processo por vez pode chama-los, de forma que, qualquer chamada subsequente será atrasada até que a anterior termine [7][9] ou que o processo abra mão de sua execução. Este segundo caso pode acontecer caso uma diretiva *wait* seja encontrada no método do monitor, podemos ver esse comportamento na implementação de um semáforo utilizando monitores acima.

O primeiro processo que chamar Psem terá posse do semáforo e decrementará a variável 's'. A chamada seguinte fará com que o segundo processo espere pelo sinal da **variável de condição** 'cond' e libere novas chamadas de Psem. Quando o primeiro processo terminar ele liberará o semáforo e sinalizará com a diretiva *signal* para que o segundo processo continue sua execução. Isso cria dois níveis de serialização, um na chamada do método e outro na variável de condição.

Os monitores são considerados estruturas de mais alto nível, comparado aos semáforos. Eles permitem um elevado grau de abstração sobre a manipulação dos recursos compartilhados, sendo possível encapsular o funcionamento dos protocolos de entrada e saída e apenas expor para os processos uma interface de acesso. Na linguagem de programação Java qualquer objeto pode servir como um monitor, para isso utilizamos a guarda de método *synchronized*, como no exemplo a seguir:

```
1: public class Box {
2:     private Object boxContetns;
3:
4:     public synchronized Object get(){
5:         Object contents = boxContents
6:         boxContents = null
7:         return contents;
8:     }
9:     public synchronized boolean put(Object contents){
10:        if (boxContents != null) return false;
11:        boxContents = contents;
12:        return true;
13:    }
14: }
```

[9] Página 1

2.1.4 Deadlock e Livelock

Na concorrência por recursos o *deadlock* se caracteriza por ser um impasse na execução dos processos, uma vez que um depende do outro. Essa dependência circular faz com que os processos fiquem indeterminadamente no mesmo estado, sem que haja progresso da computação. A representação clássica do *deadlock* é através de um grafo dirigido cíclico, onde os quadrados são os processos e os círculos são os recursos compartilhados.

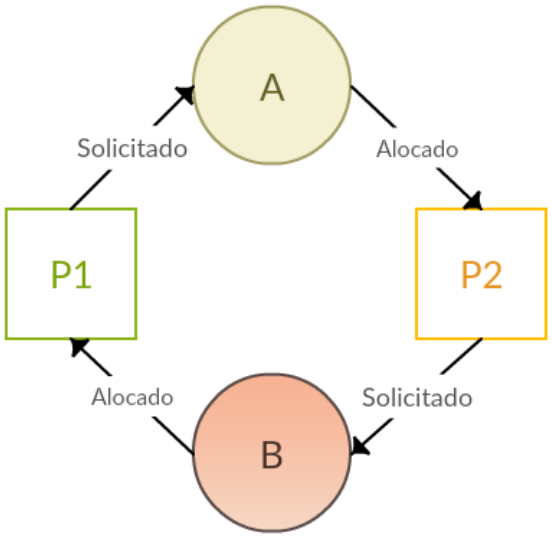

Figura 2.6: *Processos em Deadlock*

Enquanto que no *deadlock* os processos permanecem presos a um estado, no *livelock* existe transições que se repetem indefinidamente, fazendo com que um dos processos não progridam na sua computação, e fique preso em um ciclo de repetição. O *livelock* podem acontecer quando se tenta evitar *deadlocks* testando o estado de uma mecanismo de sincronização associada a um recurso compartilhado antes de utilizá-lo, como por exemplo utilizando a instrução *pthread_ mutex_ trylock*[10] da biblioteca *pthread*.

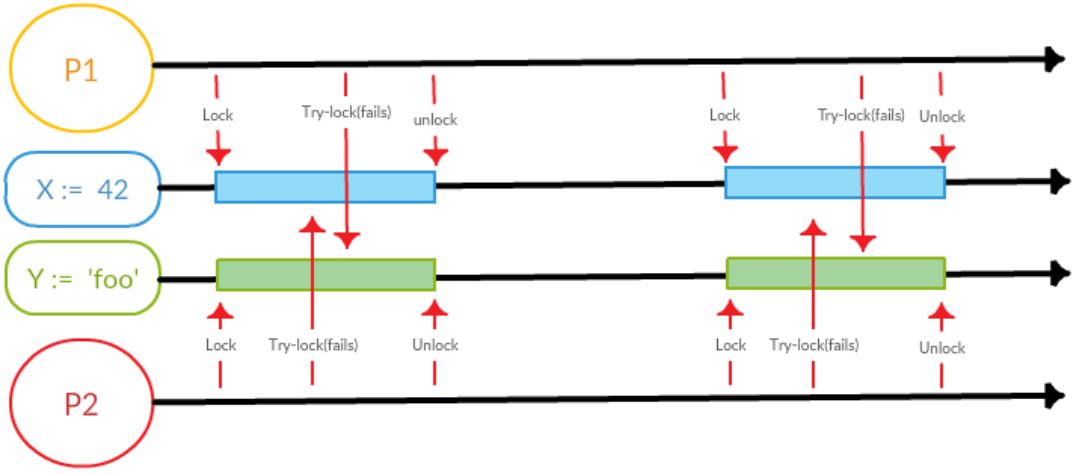

Figura 2.7: *Processos em Livelock*

Na figura acima temos os processos P_1 e P_2 e as variáveis compartilhadas X e Y. Ambos processos precisam das duas variáveis para continuar na suas respectivas computações, P_1 obtem X utilizando

[10] https://linux.die.net/man/3/pthread_mutex_trylock

o comando *Lock*, ao mesmo tempo que P_2 obtem Y. O primeiro processo tenta então bloquear Y utilizando um *Try-Lock*, que falha. Neste ponto se P_1 continuasse esperando por Y teriamos um *deadlock*, mas o comando utilizado retorna uma resposta imediata, fazendo com que P_1 libere a trava de X e tente novamente no futuro. P_2 tem um comportamento análogo e simétrico, o que leva os processos a sempre ficarem na mesma transição de estado indefinidamente.

2.1.5 Mecanismos e Políticas de Escalonamento

Em um ambiente com vários processos ocorre uma competição pelo uso da CPU e é fundamental que exista uma coordenação justa desta utilização. No sistema operacional Unix, por exemplo, existe um programa responsável por arbitrar essa disputa e escolher, através do uso de algoritmos específicos e informações internas do processo, qual deles terá chance de ser executado por um período de tempo.

Este programa é conhecido como **escalonador de processos** e tem a tarefa de distribuir um *time slice* de execução dentro de um determinado processador entre os processos para que possam executar suas instruções. Ao término desse tempo, o escalonador escolhe outro processo que irá substituir o anterior, fazendo com que esse volte para a fila de execução e tenha que esperar por sua vez de ser executado novamente. Esse mecanismo é fundamental para que exista o conceito de multiprogramação; milhares de programas executando em intervalos de tempo sobrepostos e que dão ao usuário final a impressão de execução simultânea.

Os mecanismos de escalonamento utilizam algoritmos específicios para decidir qual processo ganhará controle do processador, em [9][11] são apresentados alguns dos mais utilizados:

- *First in, First Served* - Os processos entram em uma fila de espera pela CPU, e quando conseguem a sua vez podem executar pelo tempo que desejarem
- *Shortest Job First* - Os processos informam o escalonador quanto tempo de CPU que precisam, então são colocados em uma fila de espera ordenada do menor tempo para o maior
- *Round Robin* - Cada processo recebe um tempo fixo de CPU, chamado de *quantum*. Ao terminar esse tempo o processo sofre preempção e volta para a fila de espera
- *Priority Scheduling* - Os processos informam ao escalonador qual seu nível de prioridade de execução, então são colocados em uma fila de esperada ordenada da maior prioridade para a menor
- *Shortest Job next* - Segue a mesma premissa do *Shortest Job First*, a diferença é que o processo pode sofrer preempção caso um novo processo chegue na fila e tenha um tempo menor que o seu
- *Lotery* - O escalonador distribui bilhetes numerados entre os processos e sorteia randomicamente diversos números, aqueles que vencerem terão a oportunidade de usar a CPU

Um mecanismo de escalonamento é útil em um ambiente com programas distintos sendo executados concorrentemente, entretanto esse controle não é tão fino no caso de um programa possuir diversos processos filho. Na própria codificação do problema, o desenvolvedor pode querer dar maior importância para determinados processos que desempenham um papel chave na execução. De acordo com Tanenbaum [9][12] é importante separar o **mecanismo de escalonamento** da **política de escalonamento**.

[11] Páginas 92-96
[12] Página 97

A política de escalonamento se preocupa em determinar qual será o próximo processo a ser executado de forma justa. O conceito de justiça junto com as características citadas na subseção 2.1.2 são importantes para programas concorrentes. Pode-se estabelecer certas métricas para definir se um programa é justo, já que tal característica pode variar muito entre os domínios de aplicação. Como por exemplo, em um programa que possui muitos recursos compartilhados, a quantidade de vezes que tais recursos são acessados e o tempo de espera para isso deveria ser uniforme entre os processos. Esses dados gerados podem indicar a presença de deadlocks, livelocks ou até mesmo se os processos estão sofrendo de **inanição** ao esperar indefinidamente por um recurso que nunca é disponibilizado.

2.2 O Modelo de Atores

O Modelo de atores é um modelo matemático de computação concorrente onde as unidades funcionais do programa são conhecidas como atores. Esse modelo foi idealizado por Carl Hewitt em 1973 [10], e foi fortemente influenciado por outros modelos matemáticos de computação como a *Máquina de Turing*[13], o *Labda-calculus*[14] e *Petri-nets*[15], e por linguagem de programação como *Smalltalk-72*[16], *Simula*[17], *LISP*[18] e *PLANNER*[19] [11].

2.2.1 Regras e Funcionamento

O modelo utiliza a passagem de mensagens como mecanismo de comunicação entre diferentes atores, que por sua vez são unidades fundamentais de computação que englobam três características: **processamento, armazenamento e comunicação**. Ao receber uma mensagem um ator pode executar um pequeno conjunto de ações:

- Tomar decisões locais e alterar variáveis internas
- Mandar mensagens para si mesmo ou para outros atores
- Criar novos atores
- Mudar seu comportamento interno ao processar a próxima mensagem

Ao enviar uma mensagem, o ator deve fornecer o endereço do destinatário, o qual precisa estar salvo em algum tipo de estrutura de dados interna[20]. Ao receber uma mensagem ela será armazenada em uma caixa de entrada de tamanho ilimitado até que seja processada. A ordem de entrega das mensagens é não-determinística, ou seja, duas mensagens enviadas para o mesmo destinatário podem chegar em qualquer instante de tempo, além da ordem das mensagens recebidas ser tanto arbitrária como desconhecida [12][21].

Cada ator tem sua caixa de mensagens própria e uma lista de endereço dos seus possíveis destinatários, entretanto considere o seguinte caso: existem dois atores, A_1 e A_2. Caso A_1 possua o endereço de A_2 isso não implica o contrário, mas A_1 pode enviar uma mensagem para A_2 informando seu endereço. Um ator pai sabe o endereço dos atores filhos, no entanto o contrário não é exigido pelo modelo.

[13] http://mathworld.wolfram.com/TuringMachine.html
[14] https://en.wikipedia.org/wiki/Lambda_calculus
[15] https://en.wikipedia.org/wiki/Petri_net
[16] https://en.wikipedia.org/wiki/Smalltalk
[17] https://en.wikipedia.org/wiki/Simula
[18] https://en.wikipedia.org/wiki/Lisp_(programming_language)
[19] https://en.wikipedia.org/wiki/Planner_(programming_language)
[20] Elemento 5 da figura 1.8
[21] Página 22

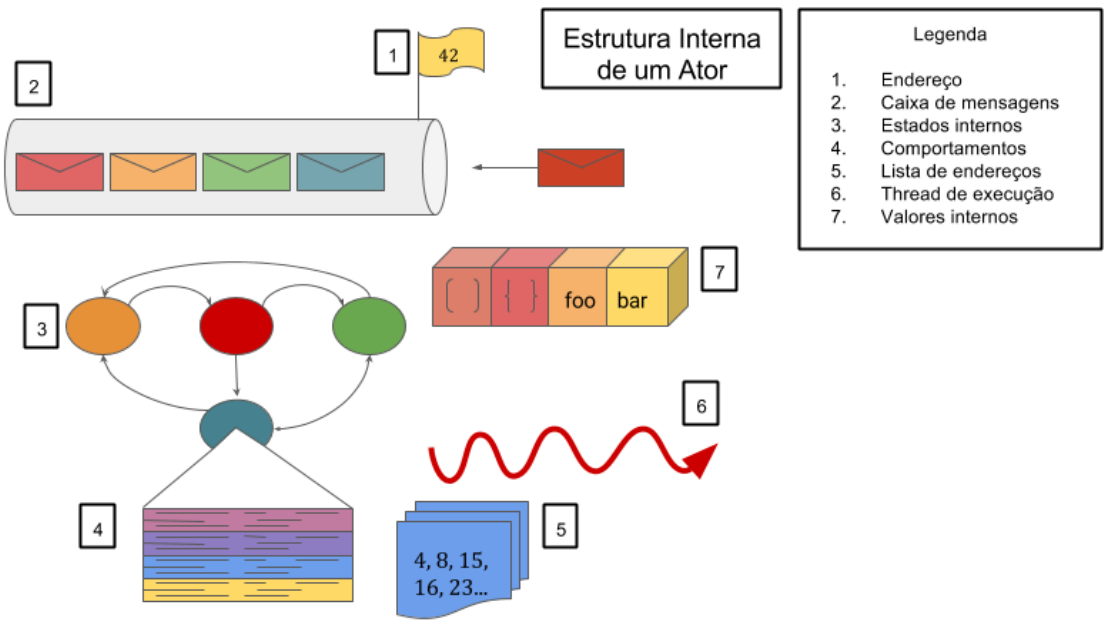

Figura 2.8: *Estrutura interna de um ator*

O funcionamento interno de um ator pode ser pensando como uma *máquina de estados finitos*[22], onde cada estado é um comportamento possuí um conjunto ações[23]. Uma transição de estado é a resposta interna de um ação ao recebimento de uma determinada mensagem e não possuí *side effects*[24] externos.

Cada estado interno do ator pode reagir de diferentes formas, dependendo do tipo de mensagem recebida. Além disso, podem existir estados nos quais um ator não saberá responder a uma determinada mensagem. Um ator pode ter variáveis internas, e essas podem ser compartilhadas com outros atores através de uma cópia, nunca através de uma variável compartilhada, isso evita alguns dos problemas de concorrência de recursos que vimos anteriormente, contudo acarreta uma redundância entre variáveis existentes e uma desincronização entre elas. O problema do compartilhamento de recursos utilizando o modelo de atores pede uma abordagem diferente como veremos na próxima subseção.

2.2.2 Compartilhamento de Recursos no Modelo de Atores

Como definimos na subseção 2.1.2, o problema da exclusão mútua ocorre quando dois processos não podem acessar ao mesmo tempo um recurso compartilhado. Devido a natureza auto-contida de um ator o acesso de variáveis só é possível através da troca de mensagens. Além disso, cada recurso precisa estar encerrado dentro de um ator específico, o qual daremos o nome de *guard actor*. Esse ator irá intermediar o uso de suas variáveis por outros atores, respondendo a solicitações de utilização quando elas estiverem livres ou ocupadas, de acordo com seu estado interno.

Considere o seguinte exemplo: temos um *guard actor* chamado G_1 que tem uma variável compartilhada **X**. G_1 aceita mensagens do tipo *Lock* e do tipo *Unlock* e tais mensagens precisam estar assinadas com o nome do ator solicitante. O comportamento interno de G_1 será o seguinte:

[22] https://pt.wikipedia.org/wiki/M%C3%A1quina_de_estados_finitos
[23] Elemento 4 da figura 2.8
[24] https://en.wikipedia.org/wiki/Side_effect_(computer_science)

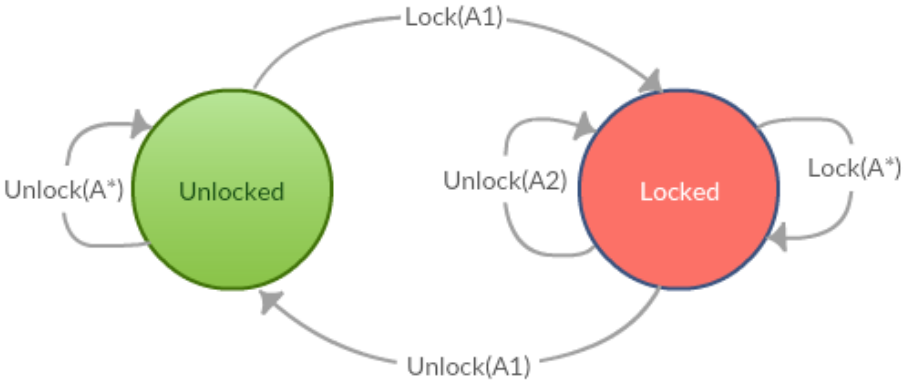

Figura 2.9: *Diagrama de estados de* G_1

O ator A_1 envia uma mensagem do tipo *Lock* para G_1, que se encontra no estado *Unlocked*. Isso garantirá à A_1 acesso exclusivo a **X** por tempo indeterminado. Novas solicitações recebidas por G_1 no estado *Locked* serão respondidas negativamente. Esse tipo de mecanismo garante que apenas um ator detenha o acesso a X por vez. A_1 irá liberar o recurso através da mensagem *Unlock*, e caso seja necessário, esse comando poderia atualizar o valor de X para qualquer novo valor calculado por A_1.

Essa estratégia garante a exclusão mútua no acesso de variáveis. No exemplo apresentado, só existe uma variável e dois possíveis estados, no entanto não é difícil generalizar essa solução para N variáveis. O problema aqui pode ser o crescimento exponencial dos estados e transições possíveis, contudo isso não se torna um dificuldade na prática, implementações do modelo de atores como o Akka (2.4) lidam com isso fazendo uso de construções sofisticadas e *pattern matching* para encontrar um estado específico e aplicar a transição correta.

Em [12][25]. Agha cria o conceito de **atores insensível** para caracterizar atores que esperam uma *become communication*, que seria um tipo de comunicação que impõe qual será o próximo comportamento, alterando assim o seu estado interno. No exemplo anterior G_1, pode ser considerado um ator insensível e as mensagens *Lock*(A_1) e *Unlock*(A_1) são comunicações que impõem um novo comportamento.

No Capítulo 3, o problema do compartilhamento de recursos será reestudado sob a ótica de um famoso problema da área de concorrência, **O Jantar dos Filósofos**. Três soluções distintas serão apresentadas e implementadas utilizando o modelo de atores.

2.3 A Linguagem Scala

A linguagem de programação Scala teve sua gênese no ano de 2001 pelas mãos do professor Martin Odersky e de sua equipe na EPFL[26], tendo sua primeira versão em 2003[13][27]. Scala significa *scalable language*, e tem a pretensão de ser uma linguagem de programação que cresça seguindo as demanadas dos seus usuários.

[25] Página 75
[26] École Polytechnique Fédérale de Lausanne, na Suíça
[27] Página 7

Ela é considerada uma linguagem multiparadigma, é possível aplicar conceitos de orientação a objetos, bem como de programação funcional. A combinação desses dois paradigmas resulta em uma linguagem muito simples, legível e robusta, capaz de expressar novos padrões de programação e abstração de componentes[14][28]. Por fim, podemos citar como características marcantes o seu poderoso sistema de tipos, sua interoperabilidade com linguagem Java e seu processo de compilação, que gera como resultado *bytecode* que será executado dentro da JVM[29].

2.3.1 Variáveis e Valores

Em Scala podemos declarar dois tipos de variáveis, mutáveis e imutáveis. Variáveis mutáveis se comportam de maneira semelhante as suas correspondentes em outras linguagens de programação, como o Java, e são declaradas com a palavra **val**. As variáveis imutáveis, declaradas com a palavra **val**, tem o seu valor atribuído apenas uma vez, não podendo ser alterado posteriormente. Os dois tipos de variáveis tem tipagem estática, ou seja, mesmo que o valor do dado armazenado mude o seu tipo deve se manter o mesmo. Podemos entender todo esse funcionamento com o exemplo a seguir:

```
var foo = 'foo'                                  ▷ Variável mutável
val bar = 'bar'                                  ▷ Variável imutável

foo = bar                                        ▷ Novo valor 'foo'
bar = '42'                                       ▷ Erro de execução

val answer: Int = 42                             ▷ Declarando uma variável com tipo expícito
```

2.3.2 Orientação a Objetos

Embora Scala seja executado na JVM e sua orientação a objeto se pareça muito com a existente em Java, o purismo de sua implementação se assemelha mais a linguagem Smalltalk. Em Scala todos os valores são tratados como objetos[14][30] e as operações como chamadas de métodos. O exemplo a seguir ilustra parte da orientação a objetos em Scala:

```
class BBox(private secretKey: String) {▷ Declaração de uma classe com um construtor implícito
    private var content: Int = 0                 ▷ Declaração de uma variável privada

    def setBox(secretKey: String, value: Int): Boolean = {    ▷ Definição de um método público
        return if(validateKey(secretKey)) {
            self.content = value
            true
        } else { false }
    }
    def getBox(secretKey: String): Boolean = {
        return if(validateKey(secretKey)) {
            self.content
        } else { 0 }
    }
}
```

[28] Página 27
[29] Máquina virtual do Java
[30] Página 95

```
    private def validateKey(secretKey: String): Boolean = { ▷ Definição de um método privado
        return secretKey == self.secretKey
    }
}
val key1: String = 'waka'
val key2: String = 'foo'
val newBox = new BBox(key1)                                  ▷ Criação de um novo objeto

newBox.setBox(key1, 42)                                      ▷ Invocação de método
newBox.getBox(key2)                                          ▷ => 0
```

2.3.3 Case Class e Pattern Matching

Em Scala existe a noção de *Case Class*, que nada mais é do que uma classe com quatro propriedades especiais[15]:

- As variáveis de uma *Case Class* são imutáveis por padrão
- Pode ser decomposta através de *Pattern Matching*
- A comparação de *Case Classes* é feito através de sua estrutura, não de referências
- Sua criação e operação é mais sucinta que uma classe normal

Um *Case Classes* pode ser aplicado da seguinte forma:

```
abstract class Box
case class RedBox(content: Int, key: String) extends Box     ▷ Definição de uma Case Class
case class BlueBox(content: Int, key: String) extends Box

val box1 = RedBox(42, "redKey1")                             ▷ Instanciação sem o uso da palavra new
val box2 = BlueBox(23, "blueKey")
val box3 = RedBox(23, "redKey2")
val box4 = BlueBox(42, "blueKey")

box1.content                                                 ▷ => 42
box2.key                                                     ▷ => blueKey
box3.content = 0                                             ▷ => Erro de execução
```

O verdadeiro poder de um *Case Class* surge ao ser utilizado em conjunto com *Pattern Matching*. Um *Pattern Matching*, que pode ser definido com a palavra **match**, é o casamento de um determinado padrão com uma expressão, e pode ser definido genericamente como[13][31]:

[31] Página 24

```
VALOR match {
      case PADRAO_1 => EXPR_1
      case PADRAO_2 => EXPR_2
      case PADRAO_3 => EXPR_3
...
      case PADRAO_N => EXPR_N
      case _ => EXPR_DEFAULT
}
```

Combinando o *Pattern Matching* com o *Case Class* podemos construir expressões da seguinte forma:

```
def checkBox(box: Box): Unit = {
      box match {
            case RedBox(42, "redKey1") => println("Redbox 42")
            case BlueBox(_, "blueKey") => println("BlueBox ? ")
            case _ => println("Caso não parametrizado")
      }
}
checkBox(box1)                                  ▷ => Redbox 42
checkBox(box2)                                  ▷ => Bluebox ?
checkBox(box3)                                  ▷ => Caso não parametrizado
```

2.4 A Biblioteca Akka

A biblioteca Akka[32] é um conjunto de ferramentas(*toolkit*) para design e criação de aplicações concorrentes, distribuídas, tolerantes a falhas e baseada em eventos, que rodam na JVM. Tem suas bases no modelo de atores (2.2) e no Erlang OTP[33], plataforma para construir aplicações assíncronas e distribuídas baseado em atores utilizando a linguagem de programação Erlang[13][34].

Akka implementa as ideias fundamentais do modelo de atores, com algumas diferenças sutis em relação ao modelo original. O código apresentado a seguir foi baseado em [16][35] e ajuda a clarificar essas sutilezas.

```
1   import akka.actor._
2
3   class AtorQueComprimenta extends Actor {
4     def receive = {
5       case name:String => println(s"Ola $name, sou um ator que comprimenta")
6       case _       => println("Me desculpe, mas nao entendi sua mensagem")
7     }
8   }
9
10  object Main extends App {
11    val system = ActorSystem("ComprimentadorSystem")
12    val greeter = system.actorOf(Props[AtorQueComprimenta], name="greeter")
13    greeter ! "Mundo" // Ola Mundo, sou um ator que comprimenta
14    greeter ! 42 // Me desculpe, mas nao entendi sua mensagem
```

[32] http://akka.io
[33] https://github.com/erlang/otp
[34] Página 55
[35] Página 2

```
15
16        system.shutdown()
17      }
```

As funcionalidades relacionadas com a manipulação de atores estão definidas dentro do pacote *akka.actor*, importado na linha 1. Na linha 3 um novo ator é definido, para isso basta extender a classe *Actor* e sobrescrever o método *receive*. Esse método é o comportamento inicial de um ator e será chamado sempre que uma mensagem nova for recebida. Ele ira aplicar um *Pattern Matching* na mensagem recebida, a fim de identificá-la e usar o processamento correto, caso ela não seja identificada o *Pattern Matching* cairá no caso não parametrizado. Definir um processamento para este caso torna o comportamento mais completo e resiliente, o ator poderá responder ao seu emissor, caso possuia seu endereço, que desconhece como processar esse tipo de mensagem.

O *ActorSystem*, criado na linha 11, é a entidade responsável por criar, executar e finalizar o sistema de atores. Na linha 12 usamos ele para instanciar um novo ator AtorQueComprimenta. Nas linhas 13 e 14 duas mensagens são enviadas para o ator que acaba de ser criado. Mensagens enviadas com o operador **!** são assíncronas, ou seja, a mensagem da linha 13 não precisa ser entregue para que a da linha 14 seja enviada. Além disso, as mensagens serão processadas, uma por vez, assim que o ator estiver livre.

Os atores no Akka são objetos que implementam as três características fundamentais do modelo de atores (2.2.1). Cada ator possui um comportamento interno corrente, e esse comportamento é responsável por processar as mensagens recebidas através do *Pattern Matching*, como explicamos anteriormente. Um ator pode mudar seu comportamento através do método *become(novoComportamento)*, nesse novo comportamento o processamento das mensagens pode se manter ou ser alterado. Esse comportamento se assemelha a uma máquina de estados, como pode ser visto na figura 2.8.

Os dados armazenados por um ator podem ser variáveis de instância, de classe ou de escopo. De acordo com o modelo não é permitido que exista compartilhamento de dados entre atores, entretanto essa restrição pode ser facilmente quebrada. Pelo fato do Akka ser executado dentro da JVM não existe um mecanismo que force estritamente a separação de dados entre os atores, apenas o bom senso e as boas práticas do desenvolvedor. É possível enviar a referência de uma estrutura de dados mutável de um ator para outro, o que configuraria um compartilhamento de dados e estaria quebrando a regra de encapsulamento do modelo [16][36].

```
1     import akka.actor._
2
3     abstract class Communication
4     case class VoiceMessage(message: String, telephoneNumber: String)
          extends Communication
5     case class TextMessage(message: String, telephoneNumber: String) extends
          Communication
6     case class Email(sender: ActorRef, message: String, emailAddress:
          String, anexo: String) extends Communication
7
8     class MessengerActor extends Actor {
9       def receive = {
10        case VoiceMessage(msg, number) =>
11          println(s"Telefone: $number. Msg de Voz: $msg")
12
13        case TextMessage(msg, number) =>
14          println(s"Telefone: $number. Msg de texto: $msg")
15
```

[36] Página 3

```
16          case Email(sender, msg, dest, anexo) =>
17            println(s"De: $sender, Para: $dest. Msg: $msg, Anexo: $anexo")
18
19          case _ =>
20            println("Me desculpe, mas nao entendi sua mensagem")
21        }
22      }
23
24      object Main extends App {
25        val system = ActorSystem("ComprimentadorSystem")
26        val messenger = system.actorOf(Props[MessengerActor], name="messenger")
27        messenger ! VoiceMessage("Alo! quem fala?", "+55-011-99999-8888")
28        messenger ! TextMessage("Ola! essa e euma mensagem",
              "+55-011-99999-8888")
29        messenger ! Email("Segue em anexo o documento", "johndoe@gmail.com",
              "wakafoobar")
30        messenger ! 42 // Me desculpe, mas nao entendi sua mensagem
31
32        system.shutdown()
33      }
```

A comunicação entre os atores acontece através da troca de mensagens. Elas podem ser definidas como *Case Classes*, o que permite uma interação mais flexível com o *Pattern Matching* realizado pelo método que recebe a mensagem. Essa ideia está ilustrada no código acima, onde três tipos de mensagens foram definidas: *VoiceMessage*, *TextMessage* e *Email*. As mensagens enviadas por um ator, de forma síncrona ou assíncrona, seguem duas regras [17]:

- *At-most-once delivery* - Para cada mensagem enviada, ela será entregue zero ou uma vez, em outras palavras a mensagem enviada pode se perder
- *Message ordering per sender-receiver pai* - Mensagens enviadas diretamente, de um ator para outro, sempre serão recebidas em ordem

O funcionamento dos atores no Akka é orientada a eventos, um conjunto de threads alternam entre si para executar diversos atores. Caso um evento ocorra, como o recebimento de uma nova mensagem, o ator é acordado e executado. A configuração do sistema fica a cargo do *Message-Dispatcher*[18], ele é responsável por definir diversos tipos de funcionamentos, como por exemplo: quantos processos estarão disponíveis para executar os atores, qual o grau de paralelismo máximo e mínimo e quantas mensagens deverão ser processadas por um ator até que a sua thread mude para outro.

Capítulo 3

O Jantar dos Filósofos

Neste capítulo apresentaremos o problema do jantar dos filósofos, explicaremos sua origem e suas características como um problema de programação concorrente. Por fim introduziremos três algoritmos diferentes que resolvem o problema.

3.1 O Problema

O problema do jantar dos filósofos, ou também conhecido como problema dos filósofos famintos, é um clássico problema de programação concorrente utilizado para estudar e desenvolver técnicas de sincronização e controle de concorrência a recursos compartilhados.

Foi inicialmente idealizado por Dijkstra em 1965 como um exercício para seus estudantes, onde diversos computadores competiam pelo acesso a fitas periféricas [19]. Posteriormente Hoare formulou o problema como conhecemos hoje em seu famoso livro Communication Sequential Process [20][1].

Cinco filósofos estão dispostos em uma mesa circular, na frente de cada um existe um prato espaguete. Para comer, um filósofo precisa da ajuda de dois garfos ao mesmo tempo, ele tem a liberdade de escolher se irá pegar primeiro o garfo da esquerda e depois o da direita ou vice-versa. Quando um filóso não está comendo ele está pensando.

Existe apenas 1 garfo entre cada prato vizinho, e este só pode ser segurado por 1 filósofo por vez, ou seja, um filósofo só poderá comer se os seus dois garfos estiverem livres implicando que filósofos adjacentes não podem comer ao mesmo tempo.

Após comer, um filósofo libera seus garfos para que os outros possam usar, a limpeza do garfo é um mero detalhe que não tem relevância na modelagem do problema. Um filósofo pode pegar um garfo da mesa e aguardar pelo outro, mas só poderá começar a comer quando tiver os dois garfos. É admitido que os filósofos sempre tenham fome e que o prato de espaguete é reposto infinitamente. Não é permitido que os filósofos se comuniquem, isso poderia quebrar os ciclos de pensamento e alimentação que cada um realiza.

[1] Página 55 a 61

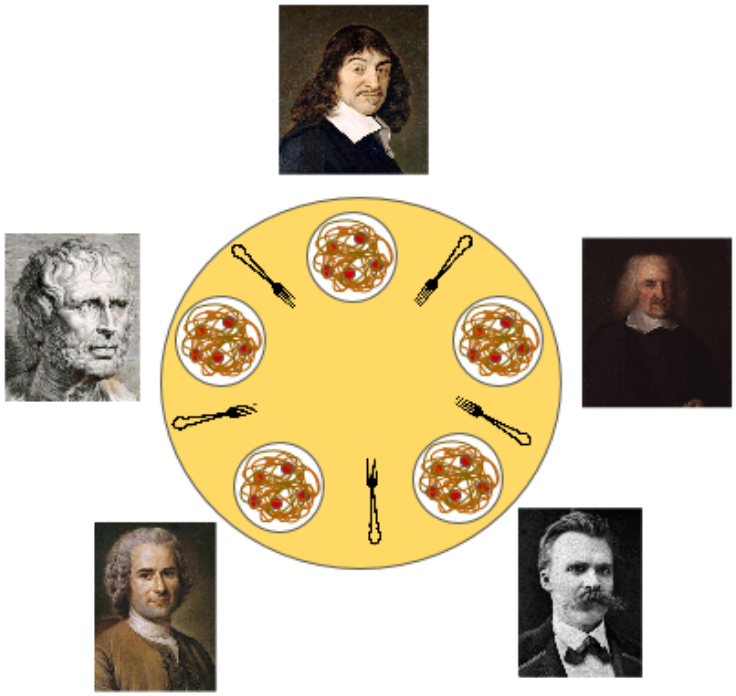

Figura 3.1: *Uma ilustração do problema*

O desafio é definir um comportamento para os filósofos, através de um algoritmo concorrente, de forma que todos consigam ter acesso aos recursos compartilhados e que possam indefinidamente alternar entre os estados 'pensando' e 'comendo'. Este problema serve como abstração para situações reais onde processos distribuídos disputam o acesso simultâneo a um conjunto de recursos compartilhados e impasses como *deadlock*, *livelock*, *starvation* e *race condition* são suscetíveis a acontecer.

3.2 Algoritmos Propostos

Para entender melhor a natureza do problema podemos propor um simples algoritmo e analisar com ele se comporta. Cada filósofo executará as seguintes instruções:

1. Pense até que o garfo esquerdo esteja disponível, então o pegue
2. Pense até que o garfo direito esteja disponível, então o pegue
3. Quando estiver segurando os dois garfos coma por uma quantidade fixa de tempo
4. Quando terminar largue o garfo direito
5. Largue o garfo esquerdo
6. Volte para o passo 1

O algoritmo apresentado seria útil se fosse livre de *deadlocks*. Considere que os 5 filósofos executam o passo 1, todos pegam ao mesmo tempo o garfo da esquerda e ficam indefinidamente pensando, até que o garfo da direita seja liberado, algo que no algoritmo acima nunca irá acontecer. Utilizando a representação de um *deadlock* introduzida na subseção 2.1.4 podemos visualizar o que aconteceu:

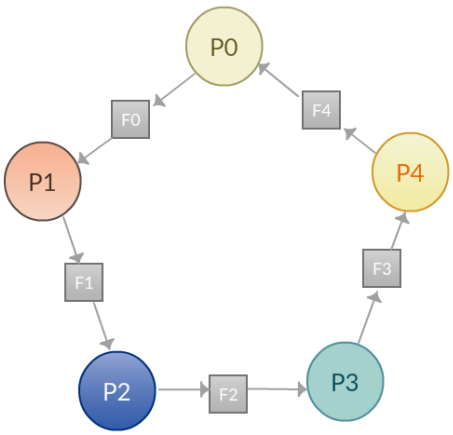

Figura 3.2: *Filósofos em deadlock*

Os filósofos estão representados por círculos e os garfos por quadrados, uma aresta indo de um garfo para um filósofo significa que ele detêm a posse do recurso, no sentido inverso significa que ele espera pelo recurso. O algoritmo apresentado não se protege contra *deadlock*, e caso tentássemos alterar o algoritmo estabelecendo um tempo máximo de espera pelo segundo garfo antes de recomeçar o algoritmo poderia acontecer um *livelock*. Veremos a seguir três algoritmos que solucionam o problema original e todos os impasses.

3.2.1 Hierarquia de Recursos

O primeiro algoritmo análisado é conhecido como **Hierarquia de Recursos** e foi baseado em duas soluções apresentadas por Dijkstra em [4] e [21]. O algoritmo começa numerando os garfos de 0 até 4 e os filósofos de P_0 até P_4, dispostos na mesa em sentido anti-horário, semelhante a figura anterior. Cada filósofo terá o seguinte comportamento:

1. Pense por uma quantidade fixa de tempo então sinta fome
2. Ao sentir fome tente pegar o garfo de menor índice que estiver ao seu alcance, caso consiga prossiga para o próximo passo, caso contrário volte para o passo 1
3. Tente pegar o garfo restante, caso consiga prossiga para o próximo passo, caso contrário solte o garfo que tem em mãos e volte para o passo 1
4. Comece a comer por uma quantidade fixa de tempo
5. Largue o garfo de menor índice
6. Largue o garfo restante
7. Volte para o passo 1

Numerar os garfos e estabelecer uma ordem que deve ser seguida ao requisitar o recurso faz com que apenas o último filósofo, que tem os garfos F_0 e F_4, tenha um comportamento diferente dos demais, ele irá requisitar primeiro o recurso da direita, enquanto que os outros escolherão o da esquerda. Utilizando um diagrama semelhante à fígura 2.2 teremos a seguinte configuração:

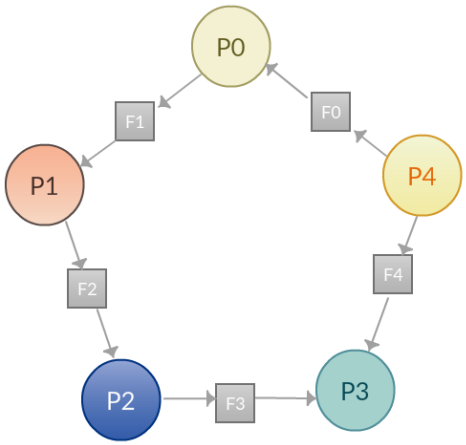

Figura 3.3: *Estado inicial do algoritmo*

A partir do estado inicial nós podemos deduzir que o primeiro filósofo a comer será P_3, seguido por P_2, P_1 e P_0. Garantimos a ausência de deadlock ao fazer com que P_4 tenha um comportamento diferente do demais, no pior dos casos, em que todos os filósofos querem comer ao mesmo tempo, P_4 quebrará o ciclo e apenas um deles irá conseguir, enquanto que os outros terão que voltar para passo inicial. A ausência *livelock* será garantida através dos passos 2 e 3 do algoritmo, um filósofo que não consegue comer abre mão de seus recursos e volta a pensar, abordagem diferente do primeiro algoritmo apresentado, onde tinhamos uma espera infindável e desnecessária.

3.2.2 Waiter

O segundo algoritmo é baseado na solução proposta por Hoare em [20][2]. Nesta solução os filósofos poderiam estar sentados ou não em seus respectivos lugares, além disso existia um agente, chamado de *Footman*, que era encarregado de controlar que poderia sentar ou se levantar da mesa. No algoritmo ***Waiter*** também existe uma unidade central, um garçom, que terá o seguinte comportamento:

1. Espere até receber uma nova mensagem de um filósofo P_i
2. Ao receber uma nova mensagem verifique o tipo dela
3. Se for do tipo 'requisição':
   (a) Verifique se os garfos de P_i estão livres
   (b) Em caso positivo autorize P_i a comer, casos contrário impeça que P_i coma
   (c) Volte para o passo 1
4. Se for do tipo 'liberação':
   (a) Libere os garfos de P_i
   (b) Volte para o passo 1

Por sua vez, os filósofos se comportarão da seguinte forma:

1. Pense por uma quantidade fixa de tempo então sinta fome
2. Ao sentir fome mande uma mensagem do tipo 'requisição' para o garçom e aguarde sua resposta por um período de tempo máximo, caso esse tempo se esgote volte para o passo 1

[2]Páginas 57-61

3. Se for autorizado pelo garçom vá para o próximo passo, caso seja impedido volte para o passo 1
4. Comece a comer por uma quantidade fixa de tempo
5. Mande uma mensagem do tipo 'liberação' para o garçom
6. Volte para o passo 1

Ao inserir um garçom que arbitrará o acesso aos recursos compartilhados garantimos que não existira impasses, cada mensagem será processada seguindo a ordem de recebimento e a resposta será dada prontamente ao filósofo emissor, fazendo com que ele consiga comer ou tenha que voltar a pensar. O *trade off* dessa solução está na diminuição da concorrência causada pela inserção de uma unidade central, a velocidade com que ele responderá cada mensagem é determinante para que os filósofos não entrem em inanição.

3.2.3 Algoritmo de Chandy-Misra

O último algoritmo, que recebe como nome os sobrenomes dos autores, foi proposto em [22] e apresenta o problema do jantar dos filósofos como um caso especial do *drinking philosopher problem*, que consiste em um número arbitrário de filósofos dispostos em uma mesa circular, e entre cada um deles existe uma garrafa de bebida. Os filósofos fazem diferentes sessões de degustação, onde pode escolher qualquer conjunto de garrafas: a da esquerda, a da direita ou ambas.

O algoritmo funciona a partir de um conjunto de regras definidas, variáveis booleanas e um grafo de precedência H. As variáveis são as seguintes:

$fork_u(f_i)$: O filósofo u detêm o garfo f_i

$reqf_u(f_i)$: O filósofo u detêm um *request token* para o garfo f_i

$dirty_u(f_i)$: O garfo f_i está em posse do filósofo u e está sujo

$thinking_u/hungry_u/eating_u$: O filósofo u está pensando/faminto/comendo

O grafo de precedência H se define da seguinte forma: um vértice orientado entre dois filósofos vizinhos **u** e **v** vai de **u** para **v** se e somente se (1) **u** detêm a posse do garfo compartilhado entre ele e **v**, ou (2) **v** detêm o garfo, e o garfo está sujo, ou (3) o garfo está indo de **v** para **u**.

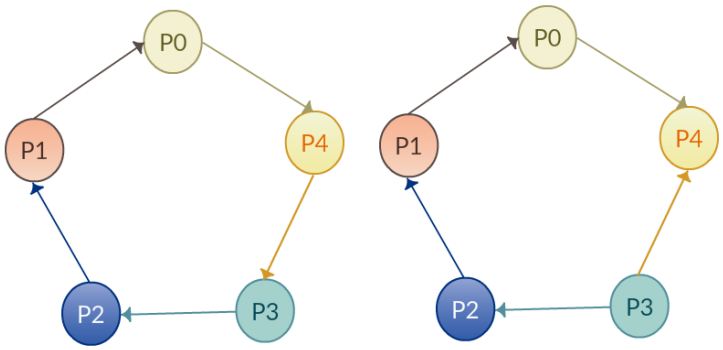

Figura 3.4: *Dois grafos de precedência*

Os filósofos seguem o conjunto de regras abaixo para enviar ou receber garfos. As regras estão escritas no padrão g ⇒ A, onde g é uma condição e A é uma sequência de ações a serem tomadas [22][3].

[3]Página 638

- Filósofo u requisita o garfo f_i:

  $hungry_u$, $reqf_u(f_i)$, $\neg fork_u(f_i) \Rightarrow$
  Mande o *request token* para o filósofo que detêm f_i;
  $reqf_u(f_i) := false$

- Filósofo u libera o garfo f_i:

  $\neg eating_u$, $reqf_u(f_i)$, $dirty_u(f_i) \Rightarrow$
  Mande o garfo para o filósofo que enviou o *request token* de f_i;
  $dirty_u(f_i) := false$;
  $fork_u(f_i) := false$

- Filósofo u recebe um *request token* para o garfo f_i:

  Ao receber um *request token* para o garfo $f_i \Rightarrow$
  Mande o garfo para o filósofo que enviou o *request token* de f_i;
  $reqf_u(f_i) := true$

- Filósofo u recebe o garfo f_i:

  Ao receber o garfo $f_i \Rightarrow$
  $fork_u(f_i) := true$;
  $\neg dirty_u(f_i)$

- Filósofo u está usando o garfo f_i:

  $eating_u$, $fork_u(f_i) \Rightarrow$
  $dirty_u(f_i) := true$

As regras apresentadas definem como será a transição de estado dos filósofos entre os estados 'pensando', 'faminto' e 'comendo'. Um requisito fundamental para o funcionamento dessas regras é a comunicação entre os filósofos, ação não permitido na formulação original do problema. Esse relaxamento do problema não faz com que ele perca generalidade, sua natureza concorrente, os impasses e a necessidade de arbitrar o acesso a recursos compartilhados se mantem, contudo os filósofos terão que saber como criar e atender solicitações.

Junto com as regras e as variáveis apresentadas anteriormente o algoritmo estabelece que o estado inicial do problema tem que verificar três condições:

1. Todos os garfos estão inicialmente sujos
2. Para cada garfo f_i que é dividido entre dois filósofos um deles segurará o *request token* e o outro o garfo
3. O grafo de precedência H é acíclico

O algoritmo garante que H será sempre acíclico e que todos os filósofos conseguirão comer em algum momento [22][4], através disso concluímos que o a solução é livre de impasses e que não existirá casos de inanição.

[4]Página 639

Capítulo 4

Modelagem e Implementação

Neste capítulo o problema do jantar dos filósofos será modelado utilizando o modelo de atores, juntamente com as suas três soluções que foram apresentadas na seção 3.2. Em seguida será explicado como o programa foi estruturado e implementado, quais construções da linguagem Scala e da biblioteca Akka foram utilizadas em cada algoritmo e também como os dados produzidos pelo programa foram coletados, analisados e se transformaram nos gráficos que serão apresentadas na seção 4.4.

Falaremos também sobre as características da simulação, em que tipo de ambiente foi executado, quais softwares foram utilizados e quais métricas foram escolhidas para analisar os algoritmos. Por fim, os resultados das simulações serão apresentados na forma de gráficos de barras.

4.1 Modelando os Algoritmos Através do Modelo de Atores

Para modelar o problema primeiro precisamos identificar quem são as unidades funcionais, que tipo de processamento eles irão realizar, quais seus valores internos e com quais unidades funcionais irão se comunicar. Começamos com os filósofos, cada um deles tem que realizar uma sequência de passos bem definido que estariam associados a um conjunto estados. Pensando de forma geral teriamos ao menos os estados **Pensando**, **Faminto** e **Comendo**, eles estariam presente em todos os algoritmos já que descrevem a ideia original do problema.

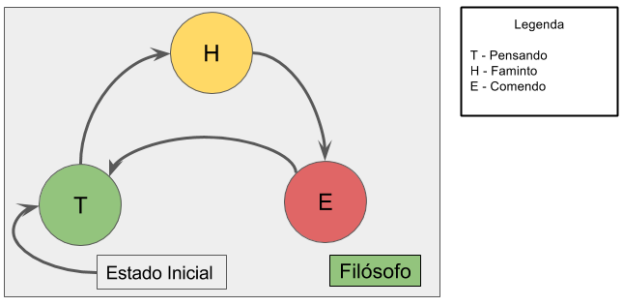

Figura 4.1: *Estados básicos de um filósofo*

Cada filósofo disputa o acesso a dois recursos compartilhados, os garfos. Como foi discutido na subseção 2.2.2 atores se comunicam apenas por troca de mensagens e o compartilhamento de dados não é permitido. Para contornar esse problema iremos considerar que cada garfo será um ator, e eles serão responsáveis por prover algum mecanismo de proteção enquanto estivem em uso pelos filósofos. Queremos um comportamento semelhante a um *guard actor*, com os estados **Disponível** e **Ocupado**.

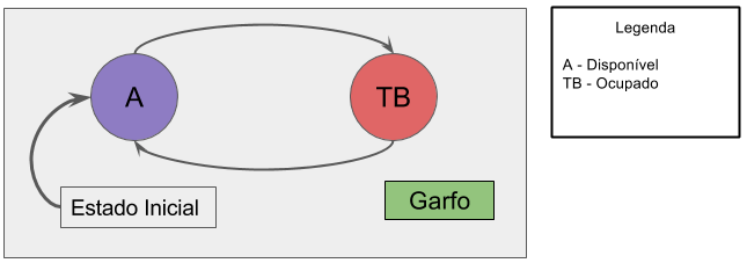

Figura 4.2: *Garfo como guard actor*

Dessa forma os filósofos entrarão em contato com os garfos e solicitarão o seu uso, caso a solicitação seja atendida o estado do garfo mudará para ocupado. Consideramos que o filósofo que provocou essa mudança está em posse do recurso compartilhado, sendo ele o único capaz de liberar o uso do garfo.

Atores precisam se comunicar, para isso o endereço do destinatário deve ser conhecido. No caso dos filósofos, eles precisam saber o endereço de seus recursos compartilhados, os quais serão imutáveis durante a execução. O garfo, por outro lado, precisa sabe quem é o filósofo que o está usando, informação que se altera ao longo da execução. Essa informação precisaria estar relacionada com o estado do garfo, sempre que houver uma transição de estado para 'Ocupado' o endereço do filósofo precisa ser recriado, isso ajuda a manter a ideia de imutabilidade dos atores.

A comunicação entre um filósofo e um garfo acontece em dois momentos, quando o recurso precisa ser adquirido e quando ele precisa ser liberado. Ao tentar adquirir o recurso o filósofo irá mandar uma mensagem para o garfo, e irá esperar sua resposta, que pode ser assíncrona ou síncrona. O garfo pode estar disponível ou em uso, independente do estado ele precisa saber responder qualquer tipo de solicitação feita pelos filósofos. Essa comunicação ativa entre as partes faz com que o filósofo não fique em *deadlock* esperando infinitamente pelo recurso, caso ele consiga o garfo existirá uma progressão nos estados até que ele consiga comer, caso contrário ele precisará voltar para o estado 'Pensando' e tentar novamente após algum tempo.

O comportamento apresentado acima é geral para os três algoritmos, cada solução irá adicionar novos processamentos, valores internos e comunicações. Apresentaremos a seguir como cada um dos algoritmos foi modelado utilizando o modelo de atores e as ideias que serviram como base.

4.1.1 Hierarquia de Recursos

A primeira particularidade do algoritmo Hierarquia de Recursos é a numeração dos garfos, iremos considerar que eles vão de f_0 até f_4. Os garfos serão dispostos em sentido anti-horário, colocando primeiro f_0 entre P_0 e P_4, como ilustrado na figura 3.3. Como o recurso de menor índice está sempre a esquerda, exceto no caso de P_4 que está a direita, definimos que P_0 até P_3 serão filósofos canhotos, sempre pegarão o garfo da esquerda, neste caso o de menor índice, e em seguida o da direita. Essa ordem também será seguida quando os garfos forem liberados. O único filósofo destro será P_4 e seguirá a sequência inversa dos passos dos demais filósofos. A orientação do filósofo será definida como um valor chamado *handedness*, o qual será utilizado para decidir qual garfo deve ser pego primeiro.

Como os filósofos precisam pegar primeiro o garfo de menor índice e em seguida o de maior índice, a menos que tenham conseguido o primeiro, será necessário dois estados adicionais além dos apresentados na figura 4.1. Os estados dos filósofos, dos garfos bem como as mensagens trocadas estão expressas na figura 4.3.

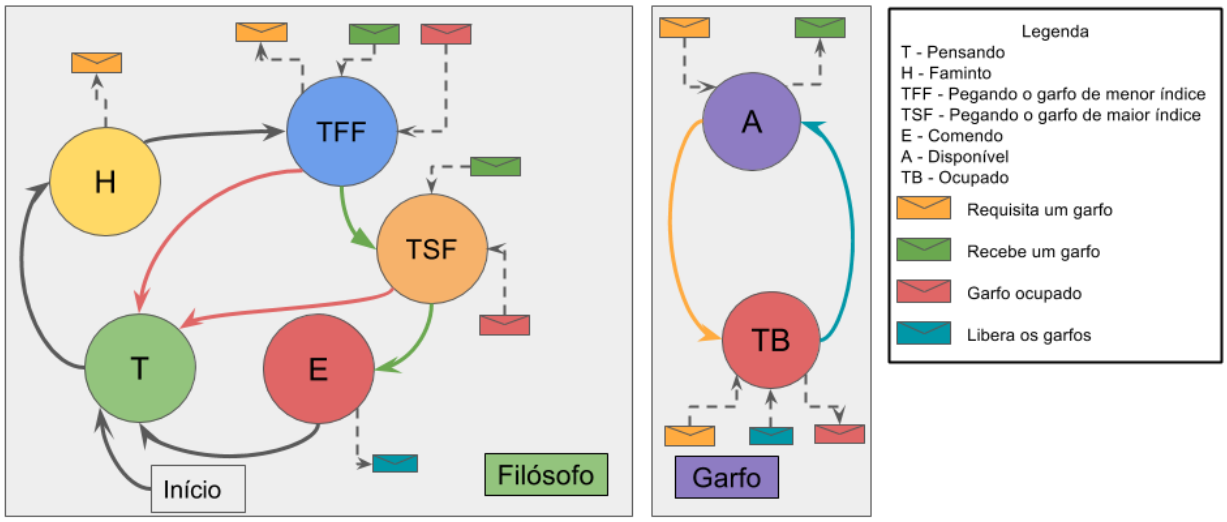

Figura 4.3: *Estados e comunicações do algoritmo Hierarquia de Recursos*

Nos filósofos as transições em preto acontecem através do agendamento de uma mensagem para si mesmo, programada para um futuro próximo. As transições coloridas são o resultado de comportamentos específicos ao receber uma determinada mensagem. Na caixa de legenda temos a explicação de cada estado e mensagem trocada no sistema.

4.1.2 Waiter

Neste algoritmo temos a presença do garçom, um ator que irá receber mensagens do tipo **Requisição** e **Liberação**. A comunicação entre o garçom e os filósofos podem ser feita de forma assíncrona, o filósofo mandará uma requisição para o garçom e ficará esperando no estado seguinte, quando o garçom processar sua mensagem ele poderá responder de forma positiva ou negativa.

Por sua vez, o garçom precisa se comunicar com os garfos, perguntado se eles estão livre ou não. Essa comunicação precisa ser feito de forma síncrona, caso os dois garfos de um filósofo P_i estejam liberados o garçom irá responder de forma positiva para o filósofo solicitante. Como uma mensagem é processada por vez pelo garçom a comunicação com os garfos não poderia ser assíncrona, caso fosse o garçom poderia responder afirmamente para dois filósofos vizinhos ao mesmo tempo, criando uma condição de corrida pelo recurso comum a eles dois.

Embora o filósofo saiba o endereço de seus recursos compartilhados esse tipo de comunicação não acontece diretamente, quem tem essa função é o garçom, ele recebe nas mensagens dos filósofos o endereço dos garfos que terá que interagir. A figura 4.4 descreve a interação dos atores no algoritmo Waiter.

4.1.3 Chandy-Misra

Os estados internos do algoritmo Chandy-Misra são iguais aos descritos na figura 4.1. Diferente dos demais algoritmos aqui os garfos podem estar ou não em posse dos filósofos, e é apenas no primeiro caso que eles poderão ser usar para comer. Outra peculiaridade do algoritmo é que os filósofos precisam saber o endereço dos seus vizinhos, bem como de dois pares de duas variáveis booleanas, o primeiro par indica a posse dos garfos e o segundo a posse dos *tokens*. Essas variáveis serão alteradas ao longo do tempo a fim de refletir as regras do algoritmo.

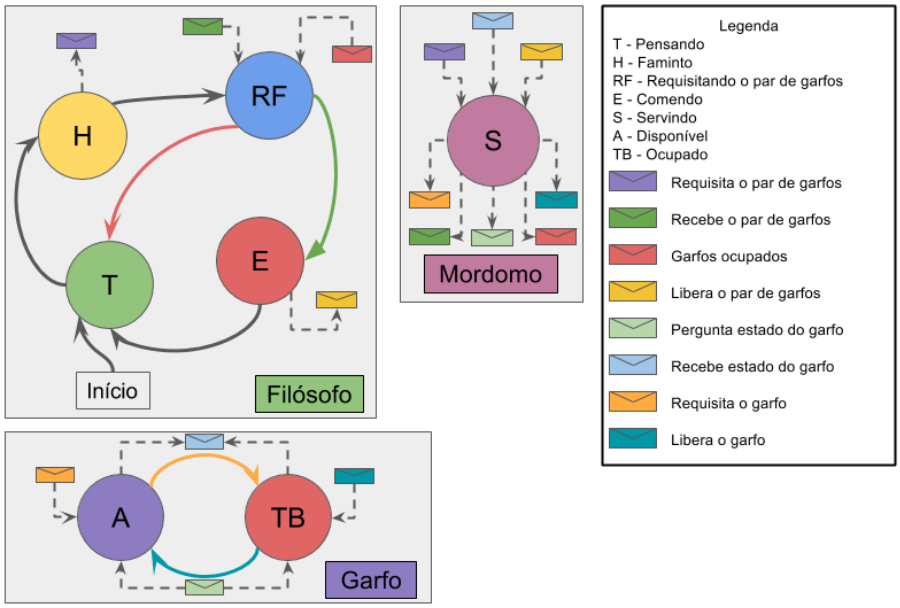

Figura 4.4: *Estados e comunicações do algoritmo Waiter*

Como o algoritmo estabelece comunicação entre os filósofos será preciso que todos os estados tenham os comportamentos de receber uma requisição ou uma resposta. Em cada um dos estados esses comportamentos serão diferentes, como podemos ver a seguir.

Pensando

- Caso o filósofo receba um requisição e o garfo esteja em seu poder ele irá verificar se o garfo está sujo, se estiver o filósofo limpará o garfo e o enviará para o solicitante. Caso contrário ele irá salvar o *request token* recebido e continuará sua execução
- Se a resposta for positiva o filósofo irá alterar as suas variáveis boolenas, no caso negativo a resposta será desconsiderada

Faminto

- Caso o filósofo receba um requisição ele irá salvar o *request token* recebido e continuará na sua execução
- Se a resposta for positiva o filósofo irá alterar as suas variáveis boolenas, no caso negativo a resposta será desconsiderada
- O filósofo tentará pegar os garfos, caso os dois estejam em seu poder ele enviará mensagens para os garfos, para sinalizar que eles estão ocupados. Se um dos garfos, ou os dois, não estiverem disponíveis o filósofo enviará um requisição para seu vizinho que está detêm a posse do garfo. Importante mencionar que a requisição será enviada apenas se o emissor estiver com o *request token* em mãos, isso evita que esse passo seja repetido inumeras vezes

Comendo

- Caso o filósofo receba um requisição ele irá salvar o *request token* recebido e continuará na sua execução
- Qualquer tipo de resposta pode ser descartada já que o filósofo detêm os recursos compartilhados

- Ao terminar de comer, o filósofo irá verificar se tem algum *request token* em aberto. Se houver ele limpará o garfo e enviará diretamente para o filósofo que enviou a requisição

O algoritmo estabelece um estado inicial para o programa antes de começar a simulação, uma dessas condições é que o grafo de precedência H seja aclícico. A configuração escolhida foi a apresentada na figura 3.4, o segundo grafo da imagem. Essa configuração é feita manipulando as variáveis booleanas de cada filósofo antes de começar a simulação. Por fim, a figura 4.5 descreve a interação dos atores no algoritmo Chandy-Misra.

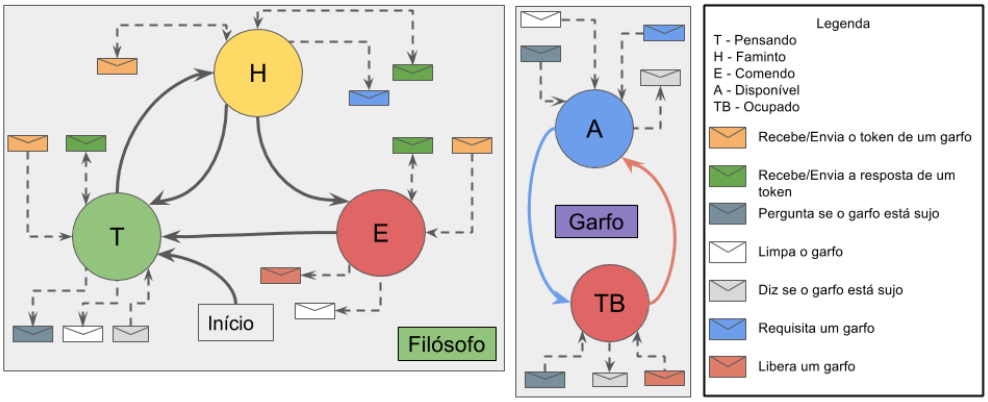

Figura 4.5: *Estados e comunicações do algoritmo Chandy-Misra*

4.2 Estrutura e Implementação

Os softwares escritos para este trabalho podem ser divididos em: o simulador do jantar dos filósofos, *scripts* para automatização da execução, *scripts* para parseamento dos dados de saída e *scripts* para apresentação dos dados. Inicialmente focaremos nossa atenção no simulador.

O simulador foi escrito na linguagem de programação Scala e tem a biblioteca Akka com principal dependência. O processo de *build* do código-fonte e execução foi feito através da ferramenta **SBT**[1]. A figura 4.6 descreve as diferentes classes existentes no programa e a interação de cada uma.

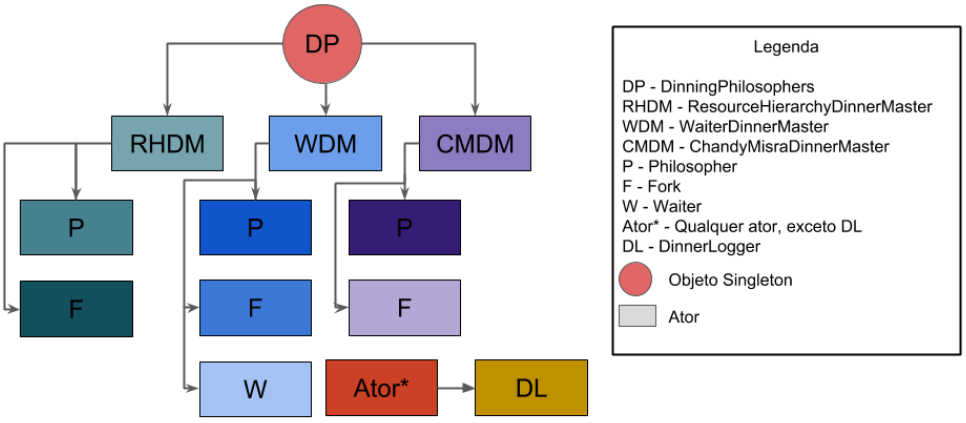

Figura 4.6: *Classes, objetos e Atores da simulação*

A classe *DinningPhilosophers* é um *singleton* e é responsável por receber via linha de comando, ou menu de opção, três parâmetros para execução da simulação:

[1] http://www.scala-sbt.org/

1. Tempo de simulação: Quanto segundos a simulação ficará em execução
2. Algoritmo: Qual algoritmo será usado na simulação[2]
3. Opção randômica: Essa *flag* determina se o tempo que cada filósofo leva para pensar e comer, podendo fixo em 5 segundos ou um valor aleatório entre 1 e 10 segundos. Usando a opção **Y** o tempo será fixo

Um exemplo de chamada através da linha de comando seria:

```
$ sbt "run-main br.usp.ime.fllsouto.dinningActors.DinningPhilosophers 60 C
    Y"
```

E através do menu de opções:

```
$ sbt "run-main br.usp.ime.fllsouto.dinningActors.DinningPhilosophers"
[info] Set current project to dinningActors (in build
    file:/home/fsouto/Study/ime-usp/tcc/dining_actors_https/)
[info] Running br.usp.ime.fllsouto.dinningActors.DinningPhilosophers
Insert the simulation time in integer seconds: 30
Choose the algorithm for the simulation
[R]esource Hierarchy
[W]aiter
[C]handyMisra

[Q]uit the program

Option: W
Run with fake random thinking and eating time (Y/N): N
Total of duration : 30 seconds
Fake Random Option: false
Running simulation using Waiter algorithm

Time left: 29 s
Time left: 27 s
Time left: 25 s
Time left: 23 s
Time left: 21 s
...
```

Após receber os parâmetros de entrada o *singleton* irá identificar qual o algoritmo escolhido e baseado nessa escolha irá criar um ator de uma das três classes disponíveis: *ResourceHierarchyDinnerMaster*, *WaiterDinnerMaster* e *ChandyMisraDinnerMaster*. Os parâmetros da simulação serão passados para o ator através da mensagem *StartDinner*. O comportamento das três classes é análogo e pode ser descrito da seguinte forma:

- Receba os parâmetros da simulação
- Crie o ator *DinnerLogger*
- Crie um número fixo de filósofos, dado a lista de nomes definida no método *run*
- Para cada filósofo criado gere dois números aleatórios, um para o tempo de pensar e outro para o tempo de comer. Caso a opção fornecida seja de um *fakeRandom* trunque todos os valores em 5 segundos
- Adicione as variáveis particulares de cada algoritmo aos filósofos

[2] [R]esource Hierarchy,[W]aiter,[C]handyMisra

- Crie os garfos e associe aos filósofos
- Crie as estruturas adicionais de cada algoritmo
- Faça os setups iniciais de cada algoritmo
- Comece a simulação mandando a mensagem *Think* para cada um dos filósofos
- Agende o envio da mensagem *TerminateDinner* para si mesmo, com uma espera igual ao tempo de simulação escolhido

O ator *DinnerLogger* desempenha um papel primordial na simulação do jantar. A cada nova mensagem recebida pelos atores do sistema, exceto os *DinnerMasters*, os atores enviarão uma mensagem para o *DinnerLogger* informando em qual estado estão, qual mensagem estão processando, qual seu nome e um *timestamp* de quando o evento aconteceu. Esses dados serão recebidos, organizados e escritos em um arquivo no formato *.JSON*[3]. Esse processo se repetirá em todos os estados dos filósofos, garfos e garçom. Ao fim da simulação será gerado um arquivo com centenas de milhares de mensagens que descrevem a simulação, como o exemplo a seguir:

```
[
  ...
  {
    "actor": "Fork",
    "state": "takenBy",
    "message": "Release ",
    "fork": "Fork_3",
    "philo": "Rousseau--2",
    "timestamp": 1479591122271
  },
  {
    "actor": "Philosopher",
    "state": "hungry",
    "message": "FellHungry",
    "philo": "Nietzsche--3",
    "timestamp": 1479591123228
  },
  {
    "actor": "Fork",
    "state": "available",
    "message": "Take",
    "fork": "Fork_3",
    "philo": "Nietzsche--3",
    "timestamp": 1479591123230
  },
  {
    "actor": "Philosopher",
    "state": "requestForks",
    "message": "TakenForks",
    "philo": "Nietzsche--3",
    "timestamp": 1479591123230
  },
  {
    "actor": "Philosopher",
    "state": "thinking",
    "message": "Think",
    "philo": "Rousseau--2",
    "timestamp": 1479591123288
  },
```

[3] https://pt.wikipedia.org/wiki/JSON

```
    {
      "actor": "Philosopher",
      "state": "hungry",
      "message": "FellHungry",
      "philo": "Rousseau--2",
      "timestamp": 1479591140469
    },
    ...
  ]
```

Quando o tempo da simulação chegar ao fim o *DinnerMaster* receberá a mensagem *TerminateDinner*. No processamento dela outras mensagens do tipo *PoisonPill* serão enviadas, uma para cada ator em execução no sistema. Essa mensagem será recebida como outra qualquer e ao ser processada causará o termino da execução do ator. Essa mensagem funciona de forma assíncrona, uma abordagem diferente é usado com o *DinnerLogger*, com ele é utilizado a mensagem síncrona *gracefullStop*, o *DinnerMaster* irá aguardar uma resposta de volta por um tempo máximo de 5 segundos, após receber uma resposta ou ultrapassar o tempo ele mandará uma mensagem *PoisonPill* para si mesmo e terminará a simulação. Podemos ver essa finalização através do *log* de execução a seguir:

```
...
Time left: 21 s
Time left: 19 s
Time left: 17 s
Time left: 15 s
Time left: 13 s
Time left: 11 s
Time left: 9 s
Time left: 7 s
Time left: 5 s
Time left: 3 s
Time left: 1 s
Finishing Philo Descartes--0
Finishing Fork Fork_0
Finishing Fork Fork_3
Finishing Philo Nietzsche--3
Finishing Philo Seneca--1
Finishing Waiter Alfred
Finishing Fork Fork_2
Finishing Fork Fork_1
Finishing Philo Hobbes--4
Finishing Fork Fork_4
Finishing Philo Rousseau--2
Finishing logger actor, output file saved in :
   WaiterDinnerMaster-T30-oN-1479591111120.json
WaiterDinnerMaster Finishing!
```

De forma a automatizar todo processo de execução, parseamento e análise dos dados alguns scripts foram escritos na linguagem Ruby[4], o funcionamento de todo processo está descrito na figura 4.7.

[4] https://www.ruby-lang.org

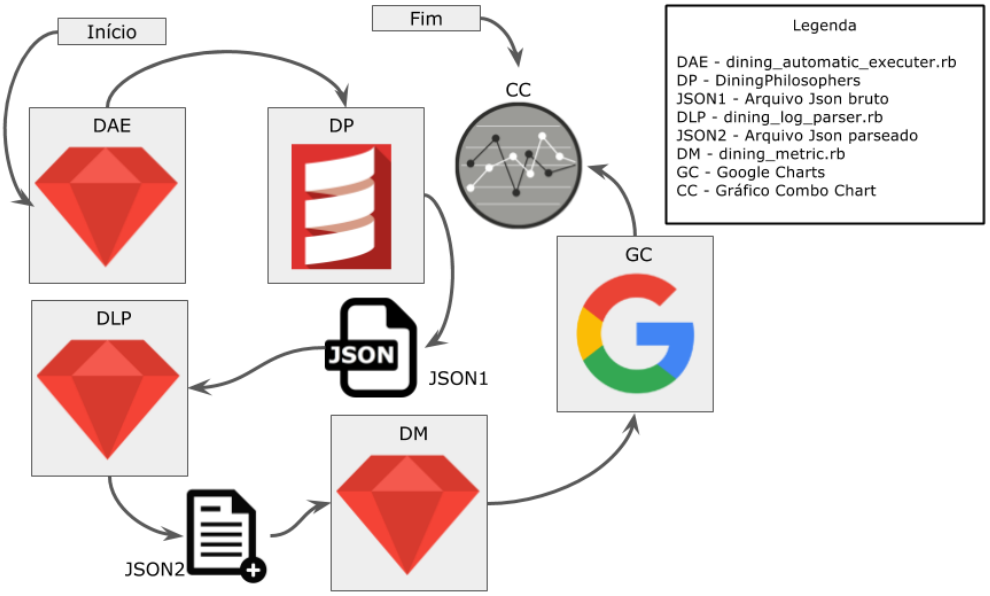

Figura 4.7: *Estrutura da simulação e interação com os scripts*

4.3 Critérios de Análise e Métricas

Na seção 3.1 mencionamos de forma breve algumas características que uma solução teria que verificar para ser considerada satisfatória. Precisamos de uma definição mais objetiva para nos ajudar a criar bons estimadores para os algoritmos implementados. Definimos que para uma solução ser considerada válida deve verificar as seguintes propriedades:

- Fornecer um comportamento para os filósofos
- Possibilitar que todos os filósofos consigam comer
- Evitar que os filósofos morram de fome
- Coordenar o acesso aos recursos compartilhados
- Ser justa, de forma que todos os filósofos comam em média a mesma quantidade e esperem em média o mesmo tempo para acessar os recursos

Baseado nessa propriedades criaremos três métricas para avaliar o desempenho dos algoritmos. Esses dados serão extraídos do arquivo *.JSON* gerado durante a simulação, seguindo a sequência de passos descritos na figura 4.7. As métricas escolhidas são:

- Quantidade de vezes que os filósofos se alimentaram
- Quantidade de vezes que os filósofos foram bloqueados de comer
- Tempo médio de espera para comer

Para a primeira métrica contaremos quantas vezes cada filósofo atingiu o estado 'Comendo', enquanto que na segunda contaremos quantas vezes cada filósofo teve que voltar para o estado 'Pensando'. A terceira métrica será calculada através da média simples de quanto tempo levou para os filósofos sairam do estado 'Pensando' e chegarem no estado 'Comendo'.

4.4 Simulação e Resultados

Por motivos de praticidade e limitação de equipamento a simulação foi realizada em apenas um computador, o notebook pessoal do autor dessa monografia. As configurações da máquina e os softwares utilizados são os que seguem:

Especificação do Hardware

- Lenovo Thinkpad T420
- Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz com 4 núcleos
- 8gb RAM
- SSD 250gb Kingston UV400
- Ubuntu 16.04 LTS Mate Version
- Kernel Linux 4.4.0-47-generic (x86_64)

Especificação dos Softwares

- Scala version 2.11.8
- Java version 1.8.0_101
- Akka version 2.4.8
- Sbt version 0.13.12

Foram realizadas 9 simulações diferentes, 3 para cada algoritmo, com duração de 1800 segundos, 3600 segundos e 7200 segundos[5]. Em todas elas o parâmetro *fakeRandom* estava configurado como **Y**, fazendo com que o tempo de pensamento e alimentação estivessem truncados em 5 segundos. A seguir apresentamos os resultados das simulações para cada uma das métricas.

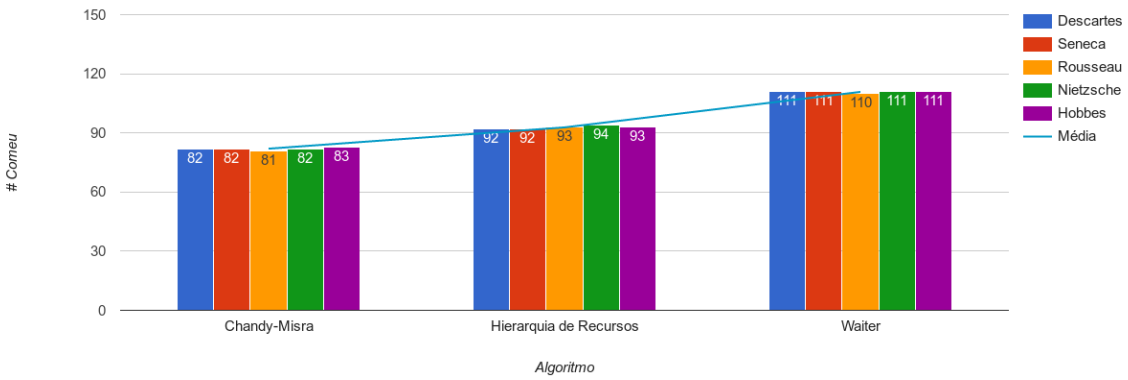

Figura 4.8: *Quantidade de vezes que os filósofos se alimentaram (30m)*

[5] 30 minutos, 1 hora e 2 horas respectivamente

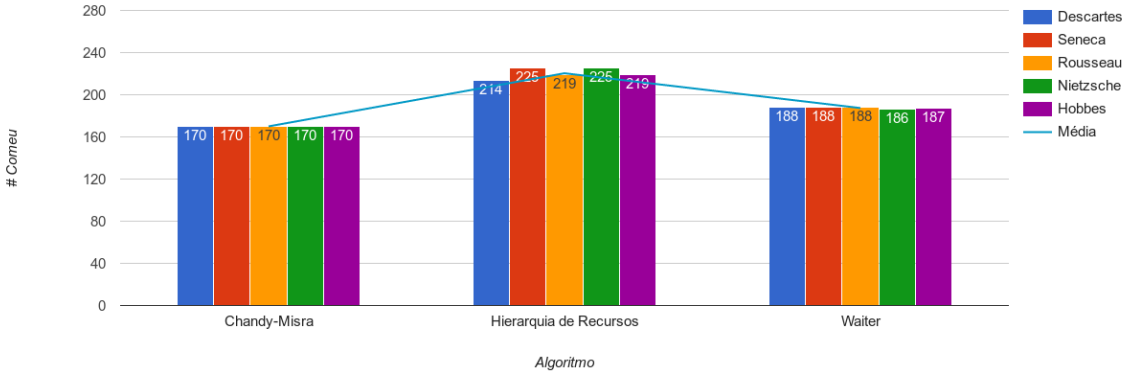

Figura 4.9: *Quantidade de vezes que os filósofos se alimentaram (1hr)*

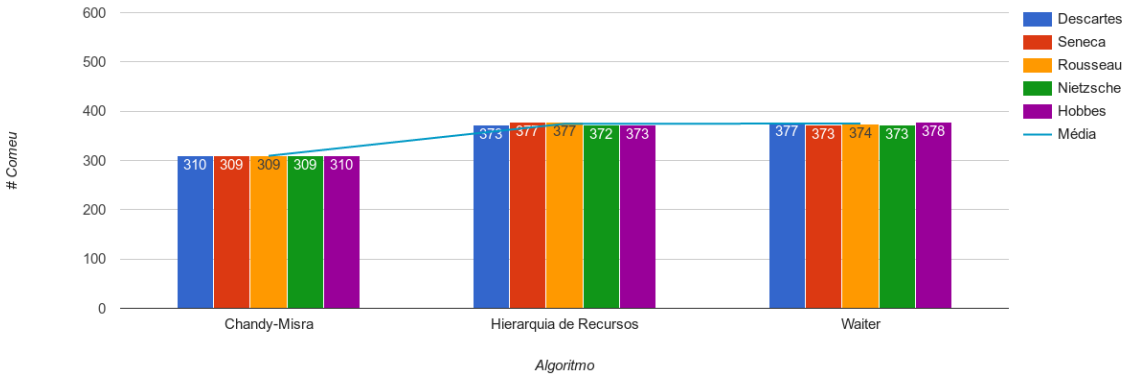

Figura 4.10: *Quantidade de vezes que os filósofos se alimentaram (2hrs)*

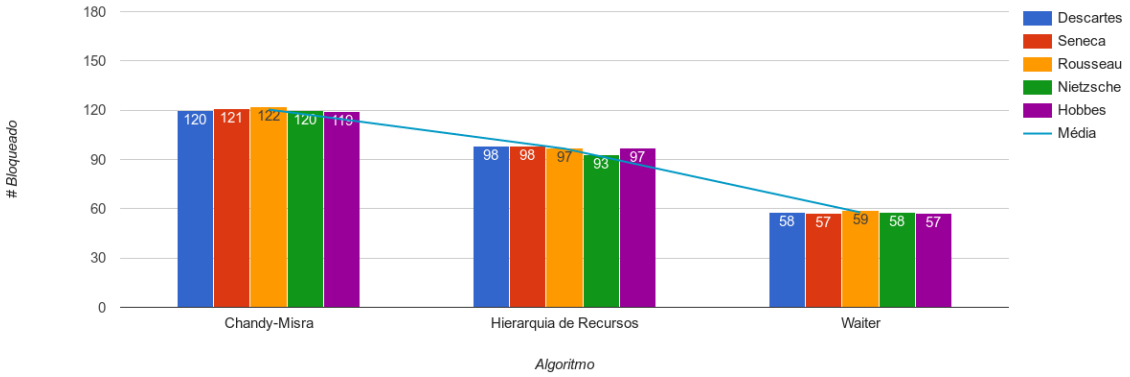

Figura 4.11: *Quantidade de vezes que os filósofos foram bloqueados de comer (30m)*

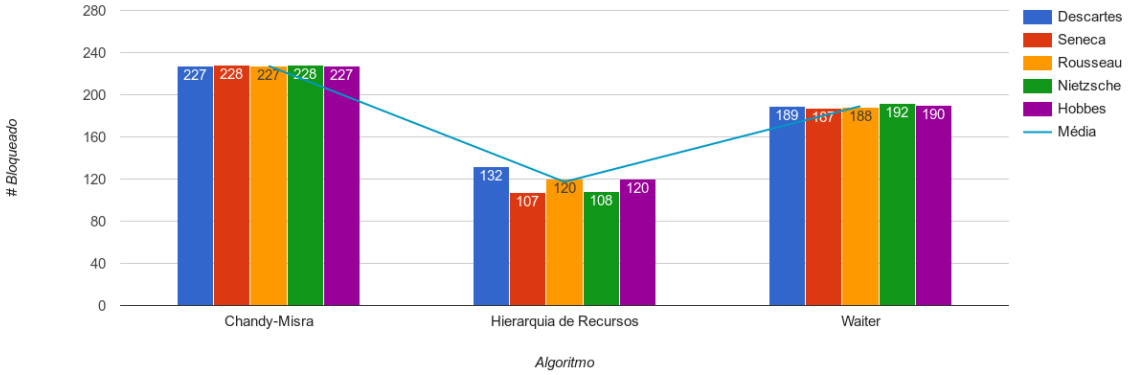

Figura 4.12: *Quantidade de vezes que os filósofos foram bloqueados de comer (1hr)*

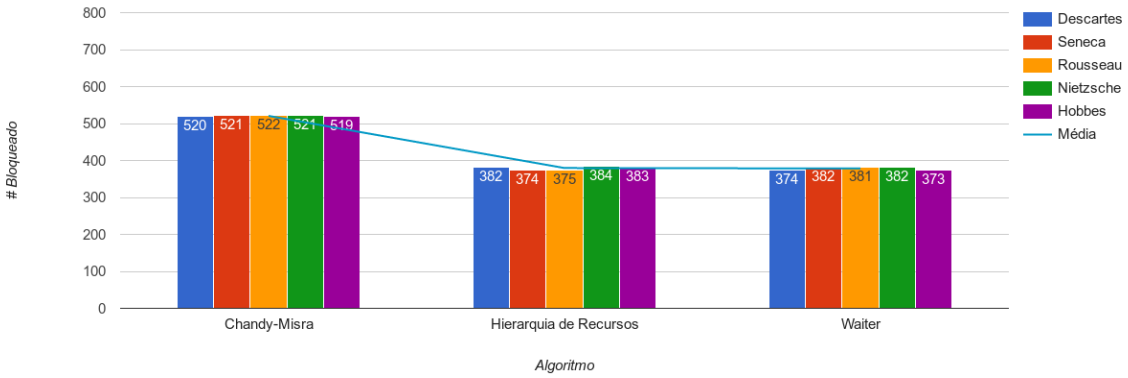

Figura 4.13: *Quantidade de vezes que os filósofos foram bloqueados de comer (2hrs)*

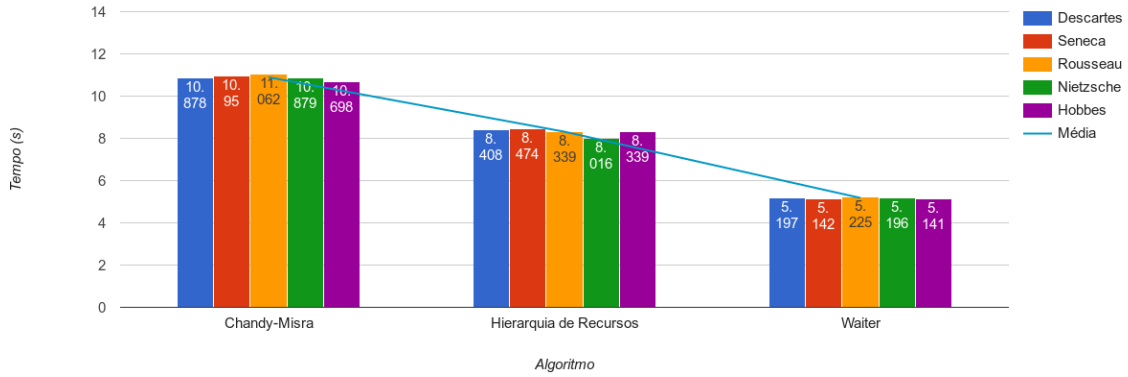

Figura 4.14: *Tempo médio de espera para comer (30m)*

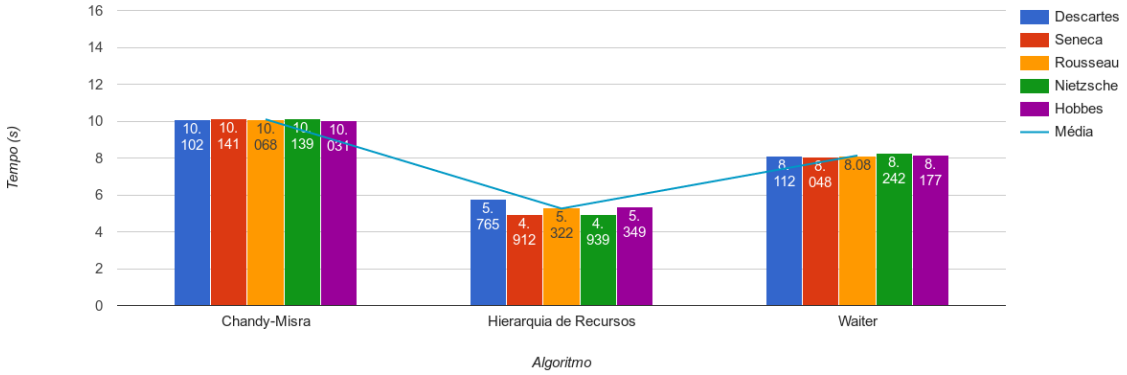

Figura 4.15: *Tempo médio de espera para comer (1hr)*

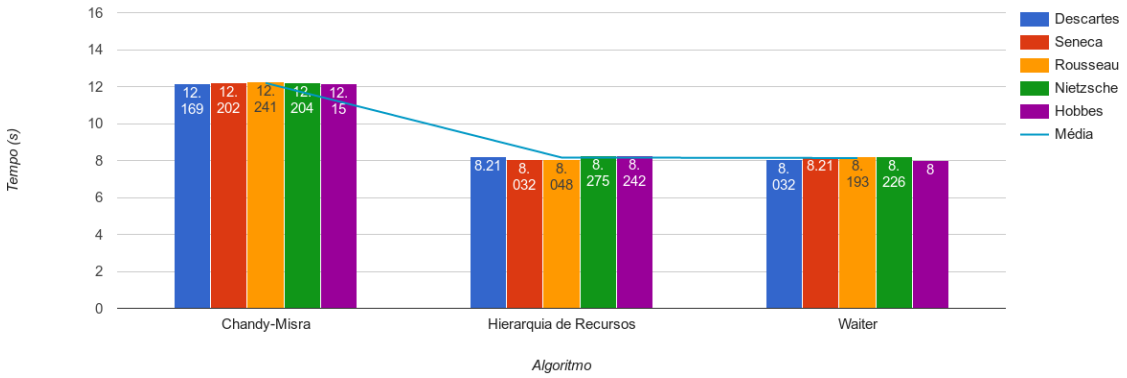

Figura 4.16: *Tempo médio de espera para comer (2hrs)*

Capítulo 5

Conclusões

Esta monografia é o produto final de um ano de estudo sobre computação concorrência, seus modelos, seus problemas e as ferramentas certas para resolve-los. Esse estudo possibilitou um grande aprendizado sobre a área em questão, indo além daquele obtido em sala de aula. O aprendizado se estende também à pesquisa científica, habilidades como buscar fontes, separar os artigos mais relevantes, ler e sintetizar conhecimento foram constantemente exercitadas ao longo deste um ano.

No Capítulo 1 apresentamos a motivação para o estudo da programação concorrente, bem como os objetivos que gostaríamos de atingir com esse estudo. No Capítulo 2 fizemos uma revisão sobre a programação concorrente, discutindo os seus principais desafios. Explicamos também sobre os fundamentos do modelo de atores, quais são suas regras e como ele pode ser usado para compartilhar recursos. Após isso introduzimos brevemente a linguagem de programação Scala, explicando suas variáveis e valores, orientação a objetos e construções funcionais mais avançadas. Terminamos o capítulo falando sobre a biblioteca de concorrência Akka, como ela funciona e implementa o modelo de atores.

No Capítulo 3 apresentamos o problema do jantar dos filósofos, discutimos suas regras e analisamos suas características, com foco principalmente na concorrência de recursos. Propusemos três algoritmos para resolver o problema dos filósofos, explicando detalhadamente o funcionamento de cada um deles.

No Capítulo 4 apresentamos uma modelagem para o problema dos filósofos através do modelo de atores. Cada um dos algoritmos recebeu uma subseção específica, onde discutimos suas particularidades e propusemos máquinas de estado que descrevem os atores envolvidos na implementação de cada uma das soluções. Ainda neste capítulo, explicamos como o código foi estruturado e implementado, detalhando a estrutura de classes e as interações entre elas. Também está detalhado como funciona o processo de simulação e como os dados gerados por ela são manipulados e processados por diferentes *scripts* a fim de obter diferentes gráficos. A motivação por trás de cada métrica é explicada na seção seguinte, juntamente com os critérios de avaliação dos algoritmos.

Na seção 4.4 apresentamos os resultados e iremos analisar eles através das propriedades descritas na seção 4.3.

O comportamento dos filósofos

Cada algoritmo descreve um comportamento para eles, como podemos ver no na seção 3.2. Os filósofos são atores e seus comportamentos são descritos a partir de seus estados internos e comunicações, e foram apresentados nas figuras 4.3, 4.4 e 4.5. Os gráficos apresentados na seção 4.4 evidenciam a diferença do comportamento através dos resultados dos algoritmos.

O estado 'Comendo'

Podemos ver através dos gráficos 4.8, 4.9 e 4.10 que todos os filósofos conseguiram comer. Isso evidencia a ausência de impasses, como *deadlocks* e *livelocks*, nos três algoritmos.

Filósofos em inanição

Podemos ver através dos gráficos 4.14, 4.15 e 4.16 que o tempo de espera é praticamente uniforme entre os filósofos. Isso indica que nenhum filósofo sofreu de inanição.

Coordenação de acesso aos recursos

Como nenhum filósofo sofreu de inanição e o tempo de espera é praticamente uniforme nos três algoritmos podemos concluir que a concorrência pelo recurso compartilhada foi arbitrada de forma satisfatória e simétrica. Cada uma das modelagens ira coordenar o acesso aos recursos de uma forma. Na Hierarquia de Recursos o recurso está encapsulado em um ator e este terá a responsabilidade de coordenar o seu uso pelos filósofos. No Waiter temos uma unidade centralizadora de decisões, o garçom, que ira intermediar o acesso e liberação dos garfos. Por fim, no Chandy-Misra a responsabilidade é compartilhada entre filósofos vizinhos, caso o garfo esteja em posse de um filósofo e não esteja uso ele deve abrir mão do recurso para que seu vizinho possa utilizá-lo.

Resultados justos

Os três algoritmos demonstraram resultados justos nas três métricas aplicadas. Uma pequena variação impediu que os filósofos tenham desempenho igual, entretanto essa variação se manteve consistente ao longo das simulações, o que é tolerável, uma vez que nenhum filósofo foi realmente prejudicado ou beneficiado.

5.1 Trabalhos Futuros

A melhora contínua é uma das regras de ouro no desenvolvimento de software, e a mesma ideia pode ser aplicada ao desenvolvimento de um trabalho acadêmico na área de computação. Muitos aspectos da pesquisa desenvolvida tiveram que ter o escopo reduzido, seja pela falta de tempo ou pela falta de maturidade teórica e prática. Listamos a seguir algumas melhorias que poderiam ser exploradas em trabalhos futuros:

- Na simulação do problema sempre foi considerado cinco filósofos na mesa, esse número poderia ser generalizado para qualquer número inteiro N, que seja positivo e maior que 2
- Os tempos utilizados na simulação poderiam ser da ordem de milisegundos, isso aumentaria muito a quantidade de mensagens trocadas no sistema e por conseguência o total de dados gerados
- Os dados envidas para o *DinnerLogger* poderiam ser armazenadas em um buffer ao longo da execução e enviados apenas uma única vez ao termino da simulação
- Uma análise estatística de mais alto nível poderia ser feita sobre os dados, a fim de identificar padrões de comportamento nos algoritmos
- A implementação poderia ser mais resiliente caso algum erro ocorresse durante a simulação

Bibliografia

[1] Gil Neiger (maintainer). 2002 podc influential paper award, 2003. http://www.podc.org/influential/2002-influential-paper/ [Acessado em: 20-11-2016].

[2] Robin Milner. **Communication and Concurrency**. Prentice Hall; 1 edition, 1989.

[3] Gregory Andrews. **Foundations of Multithreaded, Parallel, and Distributed Programming**. Pearson; 1 edition, 1999.

[4] Edsger W. Dijkstra. Hierarchical ordening of sequential processes. **EWD310**, pages 7–7.

[5] Edsger W. Dijkstra. Over seinpalen. **EWD74**.

[6] Linux man-pages project (Michael Kerrisk). Linux programmer's manual - sem_overview(7), 1998. http://man7.org/linux/man-pages/man7/sem_overview.7.html [Acessado em: 10-10-2016].

[7] C.A.R Hoare. Monitors: An operating system structuring concept. **Communications of the ACM**, pages 1–9, 1974.

[8] Per Brinch Hansen. **Operating System Principles**. Prentice Hall; 2 edition, 2001.

[9] Andrew S. Tanenbaum; Herbert Bos. **Moden Operational Systems**. Prentice Hall; Edição: 4th ed., 2014.

[10] Carl Hewitt; Peter Bishop; Richard Steiger. A universal modular actor formalism for artificial intelligence. **IJCAI'73 Proceedings of the 3rd international joint conference on Artificial intelligence**, pages 235–245, 1973.

[11] Carl Hewitt. Actor model of computation: Scalable robust information systems. **ArXiv e-prints**, pages 31–36, 2010.

[12] Gul A. Agha. **Actors: a Model of Concurrent Computation in Distributed Systems**. MIT Artificial Intelligence Laboratory, 1985.

[13] Thiago Henrique Coraini. Uma infraestrutura para aplicações distribuídas baseadas em atores scala, 2011.

[14] Martin Odersky; Lex Spoon; Bill Venners. **Programming in Scala**. Artima Press; Edição: PrePrint ed 2., 2007.

[15] Scala Documentation project (Copyright © 2011-2015 EPFL). Scala case class, 2015. http://docs.scala-lang.org/tutorials/tour/case-classes.html [Acessado em: 14-11-2016].

[16] Martin Thurau. Akka framework. 2012. https://media.itm.uni-luebeck.de/teaching/ws2012/sem-sse/martin-thurau-akka.io.pdf. University of Lübeck.

[17] Akka Documentation (Copyright © 2015 Typesafe Inc.). Message delivery reliability, 2016. http://doc.akka.io/docs/akka/2.4.2/general/message-delivery-reliability.html [Acessado em: 15-11-2016].

[18] Akka Documentation (Copyright © 2015 Typesafe Inc.). Dispatchers, 2016. http://doc.akka.io/docs/akka/current/scala/dispatchers.html [Acessado em: 15-11-2016].

[19] Edsger W. Dijkstra. Twenty-eight years. **EWD74**.

[20] C.A.R Hoare. **Communication Sequential Process**. Prentice Hall; 1 edition, 1985.

[21] Edsger W. Dijkstra. Two starvation-free solutions of a general exclusion problem. **EWD625**.

[22] K.M. Chandy; J. Misra. The drinking pilosopher problem. **ACM Transactions on Programming Languages and Systems, Vol. 6, No. 4**, pages 632–646, 1984.