CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
CURSO DE ENGENHARIA DE COMPUTAÇÃO

SISTEMA MULTI-AGENTES DISTRIBUÍDO PARA SIMULAÇÃO DE FORRAGEAMENTO DE CRIATURAS ARTIFICIAIS COM SISTEMA NERVOSO

FELIPE DUARTE DOS REIS

Orientador: Henrique Elias Borges
CEFET-MG

BELO HORIZONTE
JUNHO DE 2017

FELIPE DUARTE DOS REIS

SISTEMA MULTI-AGENTES DISTRIBUÍDO PARA SIMULAÇÃO DE FORRAGEAMENTO DE CRIATURAS ARTIFICIAIS COM SISTEMA NERVOSO

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia da Computação do Centro Federal de Educação Tecnológica de Minas Gerais, como requisito parcial para a obtenção do título de Bacharel em Engenharia da Computação.

Orientador: Henrique Elias Borges
CEFET-MG

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
CURSO DE ENGENHARIA DE COMPUTAÇÃO
BELO HORIZONTE
JUNHO DE 2017

Esta folha deverá ser substituída pela cópia digitalizada da folha de aprovação fornecida pelo Programa de Pós-graduação.

“A poesia está guardada nas palavras — é tudo que eu sei.
Meu fado é de não saber quase tudo.
Sobre o nada eu tenho profundidades.
Não tenho conexões com a realidade.
Poderoso para mim não é aquele que descobre ouro.
Para mim poderoso é aquele que descobre as insignificâncias (do mundo e as nossas).
Por essa pequena sentença me elogiaram de imbecil.
Fiquei emocionado e chorei.
Sou fraco para elogios.”

(Manoel de Barros)

Resumo

Sistemas multi-agente (MAS) são frequentemente empregados na modelagem e simulação de sistemas ecológicos e biológicos. Uma das aplicações é na simulação de vida artificial, onde uma criatura com sistema nervoso deve explorar um mundo desconhecido e aprender como sobreviver a partir das suas interações com o mundo. Este tipo de sistema é de alta complexidade devido ao compromisso com a plausabilidade biológica do modelo, que é assíncrono e não-determinístico. Esse modelo de simulação é incompatível com o modelo clássico de concorrência que utiliza memória compartilhada, no qual o controle de concorrência é síncrono. Este trabalho portanto propõe utilizar o modelo de concorrência assíncrona baseado em atores para implementar uma simulação de forrageamento em larga escala de criaturas artificiais com sistema nervoso assíncrono. A implementação utilizará o *toolkit* Akka desenvolvido em Java para implementar o mundo e a criatura artificial. Cada circuito de estimulação do sistema nervoso será implementado e testado separadamente, e ao final uma simulação de forrageamento será executada para verificar e validar o modelo.

Palavras-chave: Sistemas Multi-Agentes. Arquiteturas cognitivas. Modelo de atores.

Lista de Figuras

Sumário

Capítulo 1

Introdução

Dentre as abordagens propostas na computação para a construção de sistemas inteligentes, tiveram maior destaque os sistemas especialistas, a lógica fuzzy, as redes neurais artificiais, sistemas bio-inspirados, e os sistemas multi agentes (MAS), sendo essas não excludentes. MAS são aplicados na modelagem e simulação de sistemas complexos, *e.g.* sistemas ecológicos, sociais e econômicos, a fim de estudar a emergência de fenômenos na dinâmica do sistema (NIAZI; HUSSAIN, 2011). Essa abordagem tem sido aplicada também nos mais diversos setores da indústria e economia com sucesso (MÜLLER; FISCHER, 2014).

Segundo Niazi e Hussain (2011) MAS são frequentemente empregados em modelagem de sistemas ecológicos e biológicos, em especial nas aplicações de vida artificial e cognição. Estudar a emergência de fenômenos como adaptação, aprendizagem, linguagem e consciência é um desafio nessa área de pesquisa (BEDAU et al., 2000). Duch, Oentaryo e Pasquier (2008) classifica várias arquiteturas cognitivas baseado-se na abordagem usada na construção de cada uma. Essas classes são: simbólicas, emergentes e híbridas. Arquiteturas simbólicas são baseadas em um processador de símbolos de alto nível, e partem de uma perspectiva *top-down*. Arquiteturas emergentes partem de um fundamento conexionista, baseado em redes neurais e abordam a cognição da perspectiva *bottom-up*. A categoria de arquiteturas híbridas combina ambas as abordagens para produzir um modelo que exiba um comportamento plausível com o fenômeno biológico da cognição, em especial, o que se observa nos seres humanos. Nesta última categoria se enquadram algumas arquiteturas cognitivas multi-agentes populares, como por exemplo, LIDA e CLARION. Uma outro trabalho baseado na teoria incorporada da cognição é a arquitetura Artífice, apresentada por Campos et al. (2015). Essas arquiteturas em geral são complexas, compostas de vários componentes que interagem entre si, podendo eles serem concorrentes e não-determinísticos, que é o caso da última citada.

Um modelo fiel de um processo cognitivo é naturalmente complexo e a implementação de uma arquitetura baseada em tal modelo certamente terá um alto custo computacional.

Prasad, Lesser e Lander (1996) mostraram que ao aumentar a complexidade de um sistema multi-agente acarreta em impactos na sua performance, escalabilidade e estabilidade. O autor também argumenta que a escalabilidade não é um atributo do modelo, e sim, da implementação, e está relacionada principalmente com as tecnologias adotadas. Portanto, para estudar sistemas complexos, principalmente os que modelam fenômenos biológicos, executando simulações em larga escala, é necessário escolher um ferramental tecnológico adequado.

Quando a área de estudo envolve sistemas dinâmicos, ecológicos ou qualquer outro processo de origem biológica, como é o caso de processos cognitivos de criaturas artificiais - contexto no qual este trabalho se insere, há dois aspecto a se considerar: tais processos são inerentemente não-determinísticos e assíncronos. Consequente, um modelo matemático e/ou computacional que vise manter algum grau de plausibilidade biológica para com o fenômeno modelado, deverá consequentemente manter pelo menos uma destas duas características, quais sejam o não-determinismo e a assincronia. Todavia, no que concerne ao aspecto computacional, a larga maioria dos modelos apresentados na literatura recorre ao uso de *threads* e controle de concorrência síncronos.

O mecanismo de concorrência por *threads* surgiu na década de 70 com o objetivo de produzir sistemas mais confiáveis (HANSEN, 2013). Uma *thread* é um trecho de código de um programa que pode executar simultaneamente à outros. Quando duas *threads* tentam acessar a mesma posição de memória ao mesmo tempo isso causa uma condição de corrida, produzindo um estado inválido da memória. Para controlar os acessos concorrentes usa-se um mecanismo chamado semáforo binário ou *mutex*, que bloqueia o acesso de uma *thread* enquanto outra está fazendo o acesso. Esse tipo de solução bloqueante não é escalável uma vez que a medida que o número de *threads* cresce, a concorrência pelos recursos compartilhados aumenta, fazendo que o sistema execute praticamente de forma sequencial, o que diminui o desempenho computacional.

Atualmente outros modelos de concorrência tem surgido, principalmente com a necessidade de se produzir sistemas mais escaláveis. Um destes é o modelo de atores (TASHAROFI; DINGES; JOHNSON, 2013; HEWITT, 2010; HALLER, 2012), proposto inicialmente por Hewitt, Bishop e Steiger (1973) como um formalismo para construção de softwares inteligentes. Um ator é uma primitiva universal da computação digital, não compartilha estado com outros atores e pode se comunicar com estes via troca de mensagens assíncrona. Um ator, ao receber uma mensagem, pode: alterar seu estado interno, criar um número finito de atores, ou enviar um número finito de mensagens a outro ator. A assincronia intrínseca do modelo de atores permite produzir sistemas mais escaláveis, aproveitando ao máximo vários núcleos de um mesmo computador, bem como vários computadores em um *cluster*. Essa característica do modelo é bem compatível com os sistemas multi-agentes

que modelam processos cognitivos, onde os eventos também, nos trabalhos anteriormente apresentados, podem ocorrer de forma concorrente.

Apesar de tanto o modelo de atores quando o de *threads* terem surgido praticamente na mesma época, o primeiro começou a ganhar popularidade somente agora com a explosão das aplicações de *cloud computting*. Uma das implementações mais robustas atualmente é o Akka, inicialmente feito em linguagem Scala e, posteriormente na linguagem Java.

Tendo em vista que existem trabalhos na área de sistemas multi-agentes que necessitam de escalabilidade, mas que utilizam modelos de concorrência que dificultam o crescimento desses sistemas, que existem modelos alternativos de concorrência que favorecem a escalabilidade, e cujas características se adaptam bem à simulação de sistemas cognitivos; o que se propõe neste trabalho é utilizar o modelo de atores para construir um simulador de vida artificial, distribuído e assíncrono, dotado de um sistema nervoso artificial, que seja escalável. Este simulador será baseado na arquitetura Artífice, citada anteriormente. Posto este objetivo geral, os objetivos específicos do trabalho são:

1. Compreender o modelo de atores aplicado a construção de sistemas multi-agentes
2. Estudar e compreender a arquitetura Artífice
3. Apresentar uma nova versão da arquitetura, baseando-se no modelo de atores
4. Apresentar um modelo de simulação em ambiente distribuído
5. Implementar o modelo proposto utilizando o *toolkit* Akka
6. Executar simulações da arquitetura proposta, extraindo dados da execução, para compará-lo com a literatura do projeto Artífice

Feita esta introdução, o trabalho está organizado da seguinte maneira: o Capítulo 2 introduz o modelo de concorrência por estado compartilhado utilizando *threads* e como surgem os principais problemas de sincronização, apresenta o modelo de atores, uma de suas implementações, o *toolkit* Akka e como ele resolve esses problemas. O Capítulo 3 apresenta uma breve descrição de algumas arquiteturas cognitivas, em destaque a arquitetura artífice, e como ela se diferencia dos trabalhos na literatura. O Capítulo 4 apresenta o a metodologia, bem como o desenvolvimento do trabalho, e as principais escolhas de projeto. Por fim o Capítulo 6 faz as considerações finais.

Capítulo 2

Fundamentação Teórica

Neste capítulo serão apresentados conceitos pertinentes para a compreensão do presente trabalho. A Seção 2.1 apresenta os fundamentos de controle de concorrência usando o modelo de *threads*. A Seção 2.2 apresenta o modelo de atores e uma de suas implementações, provendo exemplos que explicam como são eliminados os problemas existentes no clássico de concorrência. Por fim na **??** está apresentado um pequeno histórico da arquitetura Artífice e como se construíram seus principais componentes funcionais.

2.1 Concorrência usando threads

Programação concorrente surge na década de 70 com a necessidade de sistemas operacionais executarem vários processos ao mesmo tempo, e seu desenvolvimento ao longo dos anos teve o objetivo de produzir sistemas mais confiáveis (HANSEN, 2013). Com ele muitos problemas de pesquisa surgiram, foram abordados e solucionados. Atualmente a maioria das linguagens de programação tem mecanismos consolidados que partiram desse desenvolvimento.

Assim como em um sistema operacional em que muitos processos podem executar concorrentemente, dentro de um mesmo processo várias linhas de execução, doravante *threads*, podem executar ao mesmo tempo. O uso de programação concorrente é fundamental atualmente para fazer uso das arquiteturas *multi-core* modernas. A diferença fundamental entre *threads* é que as primeiras compartilham o mesmo espaço de endereçamento, *i.e.* elas tem acesso às mesmas variáveis e podem executar operações de leitura e escrita nelas simultaneamente. O acesso concorrente a posições na memória compartilhadas sem o devido controle pode causar problemas a execução de programas *multi-thread*, uma vez que o escalonamento das *threads* é não-determinístico. Os principais problemas que tem de ser controlados são *deadlocks*, *starvation*, e condições de corrida.

Uma **condição de corrida** acontece quando duas threads entram em uma **seção crítica**

e executam ao mesmo tempo uma operação em memória produzindo um estado inválido. Uma **seção crítica** é um trecho de códdigo em que a execução de mais de uma thread não deve ser permitida. O Programa 2.1 mostra um exemplo de código em C que pode exibir um problema de condição de corrida.

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  int cont;
5
6  void* inc(void* n) {
7      cont = cont + 1;
8      return NULL;
9  }
10
11 int main() {
12     pthread_t threads[3];
13     cont = 0;
14     for (int i = 0; i < 3; ++i)
15         pthread_create(&threads[i], NULL, &inc, NULL);
16     for (int  i = 0; i < 3; ++i)
17         pthread_join(threads[i], NULL);
18
19     printf("%d\n", cont);
20     return 0;
21 }
```

Programa 2.1 – Exemplo de programa *multithread* em C onde ocorre uma condição de corrida

O programa cria três *threads* na linha quinze que executam a função que está definida na linha 6. Essa função simplesmente incrementa o contador global **cont**. O programa espera todas as *threads* terminarem na linha 17 e exibe o resultado na linha 19. A depender da ordem que os comandos são executados, o resultado final do contador pode não ser igual ao número de *threads* que executaram. A lista abaixo exibe uma ordem das *threads* executadas para Programa 2.1 que produz o valor final 2 enquanto o esperado seria 3:

1. A *thread* 1 lê o valor 0 da variável cont da memória principal
2. A *thread* 2 lê o valor 0 da variável cont da memória principal
3. Ambas as *threads* 1 e 2 executam o incremento produzindo o valor 1
4. A *thread* 1 escreve 1 na variável cont
5. A *thread* 2 escreve 1 na variável cont

6. A *thread* 3 executa produzindo o valor 2 na variável cont

Esse problema de ordenação dos comandos pode ser resolvido com uma primitiva de sincronização chamada semáforo binário, ou *mutex*. Essa estrutura controla o acesso à seção crítica, permitindo apenas uma *thread* acessá-la por vez. Se uma segunda *thread* tenta fazer o acesso enquanto primeira está executando a seção crítica, aquela é posta em estado de espera até que o *mutex* seja liberado.

Quando uma *thread* entra na seção crítica ela deve garantidamente deixar a seção crítica em algum momento, caso contrário as outras *threads* que tentarem acessar a região podem ficar em ***deadlock***, *i.e.*, esperando por um recurso que jamais será liberado. Esse problema pode acontecer quando duas *threads* precisam bloquear dois recursos A e B ao mesmo tempo para poderem executar, mas uma *thread* bloqueou o recurso A e outra bloqueou o recurso B. Este problema pode ser resolvido checando antes, ao conseguir bloquear o primeiro recurso, se o segundo está liberado. Se não estiver, desbloqueia-se o primeiro e entra-se em estado de espera até que ambos sejam liberados.

Starvation é um problema parecido com *deadlocks* no sentido de que uma *thread* fica esperando por tempo indeterminado por um recurso por nunca ser escalonada. Esse problema é mais raro mas pode acontecer quando o escalonamento é prioritário e uma das *threads* sempre perde prioridade para as demais.

Como visto, o modelo de programação concorrente usando estado compartilhado pode ser problemático se o controle de concorrência não for feito adequadamente utilizando as primitivas de sincronização. Ainda assim, depurar esse tipo de problema pode ser trabalhoso pois o escalonador das *threads* é geralmente não-determinístico e seria necessário reproduzir o mesmo estado da memória e a mesma ordem de execução dos comandos para produzir o erro esperado.

O modelo que utiliza compartilhamento de memória é inadequando quando se pensa em sistemas escaláveis, tanto horizontal quanto verticalmente. Por **escalabilidade vertical** se entende a capacidade de um sistema de utilizar eficientemente todos os recursos de uma única máquina. Em outras palavras, quanto melhor a configuração da máquina, melhor o desempenho do sistema. Por **escalabilidade horizontal** se entende a capacidade de um sistema de aproveitar ao máximo a quantidade de máquinas disponíveis na rede.

A escalabilidade vertical do sistemas em memória compartilhada fica comprometida porque, ainda que uma máquina tenha muitos processadores, as *threads* competem pelos mesmos recursos e bloqueiam a execução uma da outra, impedindo o estado do programa de avançar. O modelo de *threads* também não oferece nenhum mecanismo nativo para que uma *thread* se comunique com uma outra que esteja em outra máquina; para que isso

aconteça é necessário implementar a comunicação usando *sockets* ou invocação remota de método (RMI), o que dificulta o projeto de um sistema distribuído e escalável baseado em *threads*.

Eliminando a necessidade de compartilhamento de estado é possível chegar a um novo modelo de concorrência mais escalável que funciona somente por troca de mensagens, que é objeto da próxima seção.

2.2 Modelo de atores

O modelo de atores é uma alternativa ao modelo clássico de de programação concorrente (AGHA, 1985). Foi proposto por Hewitt, Bishop e Steiger (1973) como um formalismo puramente matemático para inteligência artificial, baseado em uma única entidade, a saber, atores. **Atores** são uma primitiva universal da computação digital que se comunicam por troca de mensagens.

Um ator é uma entidade computacional isolada e deve ser capaz de criar outros atores, mudar seu estado interno e enviar mensagens a outros atores (**??**). No que diz respeito às mensagens, a única restrição sobre elas é que tenham um tipo. Ele é isolado no sentido de que não compartilha estado com outros atores e toda comunicação é feita, a principio, por troca de mensagens assíncrona. Essa característica intrínseca do modelo de atores implica na não existência de condições de corrida, por isso não existem semáforos nem primitivas de sincronização no modelo.

Os modelos clássicos de computação, como o de Turing e o *lambda-calculus* são casos particulares do modelo de atores(HEWITT; ZENIL, 2013). Quando o modelo de atores foi proposto, algumas linguagens implementaram seus princípios como a PLANNER (HEWITT; BISHOP; STEIGER, 1973). Atualmente existem implementações em linguagens modernas mais modernas como Erlang, Scala, Java e C# e que tem tido, inclusive, uso comercial.

Existem outros modelos de programação baseado em troca de mensagens como por exemplo MPI (GROPP et al., 1996), voltados para aplicações mais específicas. O modelo de atores se difere deles por ser uma abstração, e não exige a existência de *threads*, *mailboxes*, fila de mensagens, *brokers*, etc. Com isso, várias aplicações podem ser modeladas e interpretadas a partir dele, como por exemplo serviços de e-mail, podendo ser inclusive implementado diretamente no hardware.

2.2.1 Scala e Java

A linguagem Scala começou a ser desenvolvida entre os anos de 2001 e 2004 no Laboratório de Métodos de Programação da EPFL com o intuito de prover uma linguagem de

programação de tipagem estática, puramente orientada a objetos, funcional e que facilitasse o desenvolvimento de componentes de software reutilizáveis (ODERSKY et al., 2004). Neste sentido o modelo de atores foi incorporado à linguagem como mecanismo de concorrêcia padrão por ser mais escalável que o modelo tradicional de concorrência, uma propriedade desejável no design da linguagem.

Na versão 2.11.0 da linguagem, a implementação do modelo de atores padrão que passa a ser utilizada é a Akka [1] enquanto a implementação anterior foi marcada como *deprecated*. A biblioteca Akka tem a vantagem de ter sido escrita não só para Scala como também para Java, o que não havia na versão anterior presente em Scala. Como ambas as implementações só diferem em termos sintáticos das linguagens e são completamente compatíveis, aqui serão exibidos somente exemplos em Java.

Para criar um ator em Java utilizando o toolkit Akka é necessário extender a classe **UntypedActor**. Essa classe obriga o programador a implementar o método ***onReceive()***, que recebe um parâmetro do tipo **Object** e não tem retorno. Dentro desse método o ator deve tratar e responder a mensagem que recebeu. O Programa 2.2 exibe uma simples implementação que responde a uma **String** com uma mensagem de boas vindas.

```
1 import akka.actor.UntypedActor;
2
3 public class AtorBoasVindas extends UntypedActor {
4
5     public void onReceive(Object msg) {
6         if (msg instanceof String)
7             sender().tell("Seja bem vindo " + ((String) msg) + "\n");
8     }
9 }
```

Programa 2.2 – Classe que define o comportamento de um ator

Todo ator ao receber uma mensagem, trata-a de forma serial, e não compartilha memória com nenhum outro ator, portanto não existem condições de corrida se as premissas do modelo forem respeitadas. Evidentemente, dois atores podem estar executando ***onReceive()*** ao mesmo tempo, isso depende do tipo de escalonamento, do número de *threads* disponíveis e do número de atores prontos para executarem uma mensagem.

Para criar um ator não se usa o construtor da classe: ele deve ser criado dentro do sistema de atores onde será encapsulado, receberá uma *mailbox* e será gerenciado pelo escalonador quando precisar ser executado. A Programa 2.3 exibe a forma correta de criar um sistema de atores e interagir com o ator.

```
1 import akka.actor.ActorSystem;
```

[1] http://docs.scala-lang.org/overviews/core/actors-migration-guide.html

```
2  import akka.actor.ActorRef;
3
4  public class Exemplo1 {
5      public static void main(String [] args) {
6          ActorSystem system = ActorSystem.create("teste");
7          ActorRef  ator = system.actorFor(Props.create(AtorBoasVindas.class),"
       boasVindas");
8
9          ator.tell("Felipe", ActorRef.noSender());
10     }
11 }
```

Programa 2.3 – Programa principal que cria um sistema de atores e instancia um ator

Este programa deve criar um ator (linha 5) dentro do sistema de atores fornecido pelo toolkit Akka (linha 4) e enviar uma mensagem para ele (linha 7). A chamada ao método ***tell()*** é assíncrona, *i.e.*, ela retorna antes da mensagem ser tratada. Todo ator possui uma *mailbox* onde as mensagens são enfileiradas para serem tratadas quando o ator for escalonado. Objetos do tipo **ActorRef** encapsulam o comportamento de todos os atores do sistema e todas as mensagens passadas para um ator são passadas por cópia em dretrimento de passagem por referência, garantindo que um ator não compartilhe memória com seu remetente.

Os atores estão disponíveis sob um caminho lógico que os torna acessíveis a partir de outras instâncias do sistema (a saber, outros processos que podem, inclusive, estar executando em outra máquina na rede). O caminho é estruturado da seguinte maneira:

```
1  protocolo://nomeSistema[@host:porta]/user/nomeAtor
```

Programa 2.4 – Esquema da URI de um ator

Os caminhos lógicos de Akka seguem o padrão URI (Unified Resource Identification). O protocolo para acessar atores remotos deve ser "akka.tcp", caso o ator esteja na mesma máquina o protocolo é somente "akka"e não é necessário especificar o *host* e a porta. Os atores criados pelo programador estão sempre abaixo do ator “user“ que os supervisiona. Caso o ator do Programa 2.2 criasse um outro ator chamado "subAtor", sua URI seria "akka://teste/user/boasVindas/subAtor".

É importante também descrever a semântica de entrega das mensagens que segue a regra *at-most-once*, ou seja, a mensagem será entregue, no máximo, uma vez ao ator. Isso implica em não garantia da entrega das mensagens. Os desenvolvedores do *toolkit* optaram por essa implementação por ser mais simples, pode ser usada de forma assíncrona sem manter estado das mensagens enviadas, e possui melhor desempenho comparado com outras alternativas (*at-least-once* e *exactly-once*). As mensagens trocadas diretamente entre pares

de atores são entregues em ordem, *e.g.* se um ator A1 enviar as mensagens M1, M2 e M3 ao ator A2, se M1 chegar ao destino, chegará antes de M2; se M2 chegar ao destino chegará antes de M3. Esse último atributo é particular da implementação de Akka e não é comum ser encontrado em outras implementações.

Por fim, o modelo de atores elimina os problemas de concorrência em sistemas não triviais permitindo ao software ganhar escalabilidade vertical e horizontal. A implementação de Akka oferece um sistema que encapsula a referência dos atores, permite transparência de localidade através dos *paths*, e um sistema de comunicação que apesar de não garantir a entrega, fornece alto desempenho. Por não compartilharem estado, testar cada ator separadamente é mais fácil que testar *threads* separadamente. Entretanto, para construir um sistema utilizando o paradigma de atores é necessário repensar toda a arquitetura de software. Não basta eliminar os pontos de acesso compartilhado e substituir por troca de mensagem, é necessário pensar em um software que não tem garantias de entrega das mensagens, completamente assíncrono, e que seja tolerante a falhas.

2.3 Considerações finais

ESCREVER ESSA PARTE

Capítulo 3

Trabalhos Relacionados

Durante o desenvolvimento deste trabalho não foi encontrada na literatura nenhuma arquitetura ou sistema multi-agente, que modele o processo cognitivo, e que execute de modo distribuído. Duch, Oentaryo e Pasquier (2008) apresenta algumas arquiteturas reconhecidas na literatura e as classifica em três categorias: simbólicas, emergentes ou hibridas. O autor também distingue as arquiteturas em seus dois principais componentes, a saber, o de memória e o mecanismo de aprendizagem. A classificação oferecida por Duch, Oentaryo e Pasquier (2008) padece de problemas, uma vez que o olhar é voltado somente ao arcabouço tecnológico utilizado, e não diz respeito à abordagem teórica do processo cognitivo.

Arquiteturas simbólicas são baseadas no processamento de um sistema de símbolos de alto nível, e são construídas por uma abordagem *top-down*. As emergentes tem fundamento conexionista, *i.e.*, partem de uma abordagem *bottom-up* e são baseadas em redes neurais. Nesta abordagem espera-se que os fenômenos cognitivos emerjam da interação entre os componentes de uma rede neural. A última classificação proposta pelo autor combina características de ambas as anteriores.

Apesar do presente trabalho se orientar segundo uma abordagem cognitiva situada (ou incorporada) (SANTOS; BORGES, 2004) e não se enquadrar em nenhuma das classificações propostas por Duch, Oentaryo e Pasquier (2008), os trabalhos que mais se aproximam da proposta deste são aqueles classificados como híbridos. Os principais modelos híbridos são o LIDA e CLARION. Eles serão apresentados nas próximas seções, em linhas gerais, bem como o objetivo de cada um dos trabalhos. A seguir será apresentada a arquitetura Artífice que construiu um arcabouço teórico em torno da teoria situada da cognição, e que será utilizado no desenvolvimento deste trabalho.

3.1 LIDA

A arquitetura LIDA (*Learning Distribution Intelligent Agent*) foi baseada em IDA, um agente de software, nas palavras dos autores inteligente, autônomo e "consciente", que auxiliava em tarefas da marinha americana (FRANKLIN; JR, 2006). Ela se baseia na teoria sobre a mente e o consciente chamada *Global Workspace*, proposta por Baars (1997), de base simbolista. Segundo Ramamurthy et al. (2006), a teoria proposta por Baars (1997)

> *[...] associates conscious experience with three basic constructs: a global workspace, a set of specialized unconscious processors, and a set of unconscious contexts that serve to select, evoke, and define conscious contents.* (RAMAMURTHY et al., 2006)

O funcionamento da arquitetura LIDA se resume na execução de ciclos cognitivos, onde o agente deve sensorear o mundo (interno e externo), criar significado baseado na interpretação dos estímulos recebidos, associando-os com processos do "inconsciente", e decidir, já no "consciente"o que é importante para fazer em seguida. Tais processos "inconscientes"são realizados por redes especializadas chamadas *codelets*.

Esse ciclo tem nove etapas que passam pela percepção e associação dessas percepções com o estado inconsciente do agente, e o estado emocional do agente é fundamental em cada uma delas. Os autores entendem sentimentos e emoções como um único conceito que compõe o sistema de valores do agente, *e.g.*, qual ação é proveitosa e em qual situação. As necessidades fisiológicas podem ser vistas, segundo Ramamurthy et al. (2006) reações emocionais positivas à recompensas esperadas.

A arquitetura é composta por três mecanismos de aprendizagem, são eles: episódico, procedural e perceptual. A memória perceptual é composta é a mais básica, composta de *codelets* detectores de características primitivas, indivíduos, categorias ou relações. A aprendizagem episódica emerge de eventos que vêm do consciente e codificam informações de "o que, onde e quando". Neste contexto existem dois tipos de memórias episódicas, as de curto prazo que codificam detalhes do estado sensorial do agente e as de longo prazo ou declarativas, que podem ser autobiográficas ou memórias semânticas, que armazenam fatos. Por fim, memórias procedurais codificam comportamentos em uma estrutura chamada *schema*, que consiste de uma ação, seu contexto e resultado. Comportamentos são escolhidos no ciclo cognitivo baseado em quão bem eles se adéquam ao contexto atual do agente e seu resultado atende algum objetivo ou necessidade do agente (RAMAMURTHY et al., 2006).

3.2 CLARION

CLARION (Connectionist Learning with Adaptative Rule Induction On-Line) é uma arquitetura híbrida para a construção de agentes cognitivos. Sua principal característica é a dicotomia entre processos implícitos e explícitos (SUN, 2016). Em linhas gerais, processos mentais implícitos são menos acessíveis e mais holísticos, enquanto processos mentais explícitos são acessíveis e substanciais. Essa dicotomia está ligada a outras dicotomias recorrentes nos estudos cognitivos, como a do consciente e inconsciente (SUN, 2001).

Um dos fundamentos adotados na construção do modelo é a tríade cognição motivação e interação com o ambiente, onde um não se separa do outro. Partindo do princípio de que as motivações do agente são natas e anteriores à cognição, esta se desenvolve a fim de satisfazer aquelas. Nas palavras do autor:

> *Cognition bridges the needs and motivations of an agent and its environments (be it physical or social), thereby linking all three in a triad* (SUN, 2016)

Um agente da arquitetura CLARION se divide em 3 subsistemas, O ACS (*Action Centered Subsystem*), O NACS (*Non-Action Centered Subsystem*), o MS (*Motivational Subsystem*) e o MCS (*Meta Cognitive Subsystem*). Com o objetivo de contemplar a dualidade implícito/explícito, todo mecanismo é formado por componentes de alto nível, que utilizam um sistema de símbolos, e baixo nível, formado por redes neurais, responsáveis respectivamente por armazenar conhecimento explicito e implícito.

O ACS é responsável por controlar a execução das ações desempenhadas pelo agente, sejam processos mentais ou físicos, baseado em conhecimento procedural. O NACS mantém conhecimento declarativo usado para realizar inferências. O MS mantém um sistema de motivações e objetivos. Por motivação entende-se as necessidades básicas de um agente, *e.g.*, estar com fome, e por objetivo, uma decisão que satisfaça uma necessidade, *e.g.*, encontrar comida. Por fim, o MCS é responsável por coordenar os outros mecanismos, escolhendo uma motivação e um objetivo a ser cumprido, interrompendo a ação que está em execução no momento, estabelecendo os novos parâmetros para o ACS e NACS funcionarem, e regular a motivação mediante a recompensa recebida.

3.3 Arquitetura Artífice

A arquitetura Artífice foi proposta em seu modelo conceitual por Santos (2003). Sua primeira especificação tinha o objetivo de ser genérica o suficiente para construir agentes inteligentes para qualquer propósito, fundamentando-se na teoria não-objetivista da cognição. Como essa posição filosófica é por demais extensa, não cabe aqui detalhar cada um de

seus aspectos, mas é possível dizer em linhas gerais o que o termo significa. A **teoria não-objetivista** afirma que os sujeitos epistêmicos (aqueles que tem consciência de que sabem) e os objetos do mundo coexistem e estão em constante interação, modificando-se mutuamente. Não obstante, o sujeito não tem conhecimento *a priori* do mundo, mas o conhece a medida que com ele interage e exibe um comportamento coerente, comportamento esse que dependerá de seu estado interno (dos seus componentes cognitivos, parte do sistema nervoso, e dos não cognitivos, que compõem o restante do seu corpo) mas não é determinado por ele.

Definido o arcabouço teórico-conceitual e as premissas de uma arquitetura cognitiva e situada, Santos e Borges (2004) identificam que as interações entre os componentes internos de um ASCS (agente de software cognitivo e situado, mas daqui em diante o termo é substituído por criatura artificial, ou somente criatura, termo atualmente empregado no contexto do projeto) e os componentes do mundo acontecem simultaneamente, e o modelo computacional que, à época, melhor abrigava esse comportamento era o de *threads*.

Baseando-se na versão de Santos e Borges (2004) que também já usava o mecanismo de troca de estímulos entre componentes como modo de comunicação entre as *threads*, Campos (2006) propõe uma versão da arquitetura que engloba o processo cognitivo e emocional. **Emoções** dentro do contexto da arquitetura são necessidades corpóreas primárias, que possuem um nível de excitação (***arousal***) e tem correlação alta com o desempenho do comportamento da criatura. Por **eficiência comportamental** entende-se o quão eficiente a criatura é em encontrar comida ou fugir de alguma ameaça. O *arousal* pode variar de um nível mínimo a um nível máximo, abaixo do mínimo a criatura está em sono profundo e acima do valor máximo ela morre.

As emoções implementadas por Campos (2006) foram fome e sono, e também o reflexo de rubor, contemplando portanto três níveis de resposta: pouco elaborado, semi-elaborado e elaborado. O nível de resposta **não-elaborada** diz respeito sobre as respostas reflexas que a criatura exibe, o **semi-elaborado** são respostas puramente emocionais que não passam por uma avaliação completa da situação, e a **elaborada** é de nível superior e passa por uma avaliação emocional-cognitiva. A criatura então vive em um mundo populado por nutrientes, que podem saciar sua fome, totens que podem estimular seu reflexo de rubor, e deve buscar interagir com o seu meio a fim de manter os níveis de *arousal* entre o mínimo e o máximo, ou seja, conservar seu equilíbrio emocional (**homeostase**). O fenômeno cognitivo emerge dessa regulação do estado interno a fim de manter-se viva por tempo indeterminado.

Outros trabalhos como os de Silva (2008) e Mapa (2009) contribuíram para o aperfeiçoamento do mecanismo de aprendizagem das criaturas artificiais, adicionando memórias de condicionamento e memórias auto-biográficas. Simulações computacionais propostas pelos autores mostraram que ambos os mecanismos são fundamentais para a adaptação

da criatura artificial ao mundo.

3.4 Considerações Finais

As três últimas seções apresentaram três abordagens diferentes para modelar o processo cognitivo. A primeira, implementada na arquitetura LIDA, baseia-se em um modelo psicológico da mente, ignorando, a princípio qualquer particularidade do corpo ou do mundo em que o agente vive. A segunda, apresentada na Seção 3.2, se baseia fortemente na dualidade implícito/explicito da mente, levando em conta um sistema de motivações que guiam as ações, mas também não considera que o agente tenha um corpo, restrições sensórias, motoras, ou que haja algum tipo de indeterminismo em suas ações. Ademais, essas arquiteturas não foram concebidas para operarem como MAS de agentes "inteligentes", mas sim para funcionarem como um único agente.

Na arquitetura Artífice, por sua vez, um agente cognitivo, ou uma criatura, é considerado um construto único, formado de um sistema cognitivo, sistemas sensórios e motores e outros sistemas auxiliares que formam seu corpo. Essa construção, juntamente com as diferentes situações que ele pode enfrentar ao longo da sua história de interações com o mundo, determina as possibilidades de ação que a criatura pode desempenhar. Os eventos durante a simulação acontecem de maneira assíncrona, e não determinam diretamente qual a atuação da criatura. Ela não possuí nenhum mecanismo central de controle, funcionando de maneira independente. O software foi construído utilizando o modelo de programação concorrente baseado em *threads*, e permite que várias criaturas co-existam em uma mesma simulação.

Apesar da possibilidade de simular mais de uma criatura ao mesmo tempo na arquitetura Artífice, esse número é limitado pelos recursos computacionais de uma única máquina. Assim, o objetivo deste trabalho é contornar essa limitação, implementando uma nova arquitetura, baseado no mesmo modelo que inspira o Artífice, utilizando mecanismos de concorrência voltados para programação distribuída, e executar simulações de forrageamento para validar a nova implementação. Dito isto, o próximo capítulo apresenta a metodologia utilizada para desenvolver e testar a implementação da arquitetura utilizando o modelo de atores, e descreve as particularidades da mesma.

Capítulo 4

Desenvolvimento do trabalho

Como abordado no Capítulo 2, o modelo de *threads* é incompatível com o modelo de atores. Enquanto no primeiro a comunicação é feita utilizando compartilhamento de memória, o que pressupõe sincronização no acesso compartilhado, o segundo abre mão dessa possibilidade, estabelecendo a troca de mensagens assíncrona como forma de comunicação, assumindo que as unidades de processamento podem não estar na mesma máquina.

Neste sentido, o presente capitulo apresenta a modelagem de um simulador, baseado no modelo de atores, escolhendo o modelo de sistema nervoso artificial apresentado pela arquitetura Artífice. A próxima seção descreve a metodologia adotada, a modelagem dos sistemas internos da criatura está descrita na Seção 4.2. Estabeleceu-se um protocolo para comunicação entre componentes em diferentes nós, descrito na Seção 4.3.

4.1 Metodologia

A proposta principal do presente trabalho é construir simulador de criaturas artificiais interagentes, cada qual dotada de sistema nervoso, baseado no modelo de atores e que opere como um AD-MAS - *Asynchronous Distributed Multi-Agent System*. Usar esta tecnologia se justifica pelo fato de que assincronia intrínseca do modelo de atores favorece a escalabilidade e se adéqua bem ao modelo de sistema nervoso escolhido. Dado este objetivo, para desenvolver um software distribuído é necessário particioná-lo nos componentes que devem executar no mesmo espaço de endereçamento e nos que podem executar em espaços distintos, projetar os algoritmos de sincronização e propagação de informação no *cluster*.

No modelo de sistema nervoso artificial escolhido, a arquitetura Artífice, existem duas entidades principais: as criaturas artificiais e os componentes do mundo, com as quais a criatura interage. Uma criatura é composta de vários componentes internos que interagem entre si através de troca de estímulos, formando cadeias de estimulação. Portanto o primeiro passo para produzir uma nova implementação utilizando o modelo de atores foi

identificar os componentes e estímulos trocados, e implementar cada uma das cadeias de estimulação. Para cada circuito implementado, uma versão da arquitetura deve ser gerada, e seu funcionamento testado e validado.

Um mecanismo de extração e analise de dados deve ser implementado. Ao finalizar a implementação de todas as cadeias de estimulação, experimentos computacionais devem ser realizados, coletando-se dados do funcionamento, para comparar os resultados passados. Portanto a criatura deve ser capaz de salvar dados da dinâmica interna em banco de dados, de forma que seja possível recuperá-los para analise posterior. Em resumo, a metodologia proposta segue:

1. identificar os componentes da criatura e as trocas de estímulos e as cadeias de estimulação
2. implementar cada uma das cadeias, testando o funcionamento de cada uma delas
3. desenvolver um esquema de banco de dados relacional para salvar os dados da simulação
4. propor e implementar um modelo de distribuição dos componentes, bem como os algoritmos de sincronização
5. propor e executar um conjunto de experimentos computacionais para validar o software produzido, verificando o seu correto funcionamento e a sua escalabilidade

A Seção 4.2 contempla o desenvolvimento dos itens 1, 2 e 3 da metodologia, enquanto a Seção 4.3 contempla o desenvolvimento do item 4. A descrição detalhada dos experimentos realizados e os resultados obtidos encontram-se no Capítulo 5.

4.2 Dos componentes da criatura e da dinâmica interna

Nesta sessão se propõe a discussão da modelagem da dinâmica interna de uma criatura artificial baseando-se na proposta pela arquitetura Artífice. Para que não haja confusão entre o primeiro e o segundo modelo, adotar-se-á o nome DL2L [1] para a arquitetura proposta neste trabalho.

Uma criatura artificial é um conjunto de vários componentes que juntos desempenham a dinâmica interna e podem ser divididos em três grandes categorias: os que fazem parte do sistema somático (SS) e compõe estruturas físicas da criatura (corpo, olho, boca, etc.) e portanto tem uma representação geométrica no mundo artificial e trocam estímulos

[1] L2L é acrônimo para a frase *Learn to live, live to learn*, e a letra "D"de distribuído.

diretamente com este; os que fazem parte do Córtex Sensório-Motor (CSM), que fazem a interface entre o sistema somático e o sistema nervoso central e, por fim, o Sistema Nervoso Central (SNC) que compreende o processo cognitivo-emocional. Os componentes funcionam concorrentemente e trocam mensagens assíncronas.

Existem outros sistemas que compõe a criatura, como o sistema de memória (SM), o sistema de condicionamento (SC), e o sistema emocional (SE) e o seletor de ação (AS), mas estes não atuam diretamente para a dinâmica interna, mas são utilizados pelo SNC para manter a homeostase, decidir e avaliar as ações da criatura.

Alguns componentes precisam esperar um certo conjunto de estímulos para funcionar, *e.g.* os que integram o córtex sensório-motor, só podem enviar estímulos ao sistema nervoso central uma vez que o sistema somático tenha informado haver algum objeto no campo sensório da criatura. Componentes do SNC por sua vez não precisam aguardar estímulos o tempo todo, que é o caso do **PartialAppraisal** que faz a avaliação emocional. A Figura 1 exibe a cadeia de estímulos trocados entre os componentes internos da criatura artificial.

A criatura artificial interage com o mundo artificial, através dos seus componentes do sistema somático, a fim de manter a sua regulação homeostática, *i.e.*, manter o nível de excitação de suas emoções em um valor estável. Para ilustrar o funcionamento da dinâmica interna uma descrição hipotética da sequência de eventos que acontecem quando um objeto entra em seu campo de visão será oferecida. Supondo que a criatura está caminhando pelo mundo quando dois nutrientes A e B entram em seu campo de visão, e olfativo, ambos em contato com a sua boca:

1. o componente **Eye** receberá dois estímulos do tipo Luminous, um para cada nutriente. E enviará, para cada um deles, um estímulo do tipo Visual para o VisionSensor;
2. O componente **Nose** receberá, concorrentemente um estímulo do tipo **Smell** para cada nutriente, e enviará para o **OlfactiveSensor** dois estímulos do tipo **Olfactive**;
3. o componente **Mouth** receberá dois estímulos do tipo **Mechanical** e enviará dois estímulos do tipo **Tactile** ao **MouthSensor**;
4. Os componentes sensores (MouthSensor, VisionSensor e OlfactiveSensor) ao receberem os estímulos citados, produziram para cada um deles um estímulo do tipo Proprioceptive, que será enviado ao PartialAppraisal, esses estímulos podem chegar juntos ao componente Partial ou não;
5. Supondo que o PartialAppraisal recebe todos os seis estímulos proprioceptivos, ele criará uma Situação, que representa o conjunto de sensores ativados e por quais componentes. Essa situação será enviada juntamente com a emoção mais

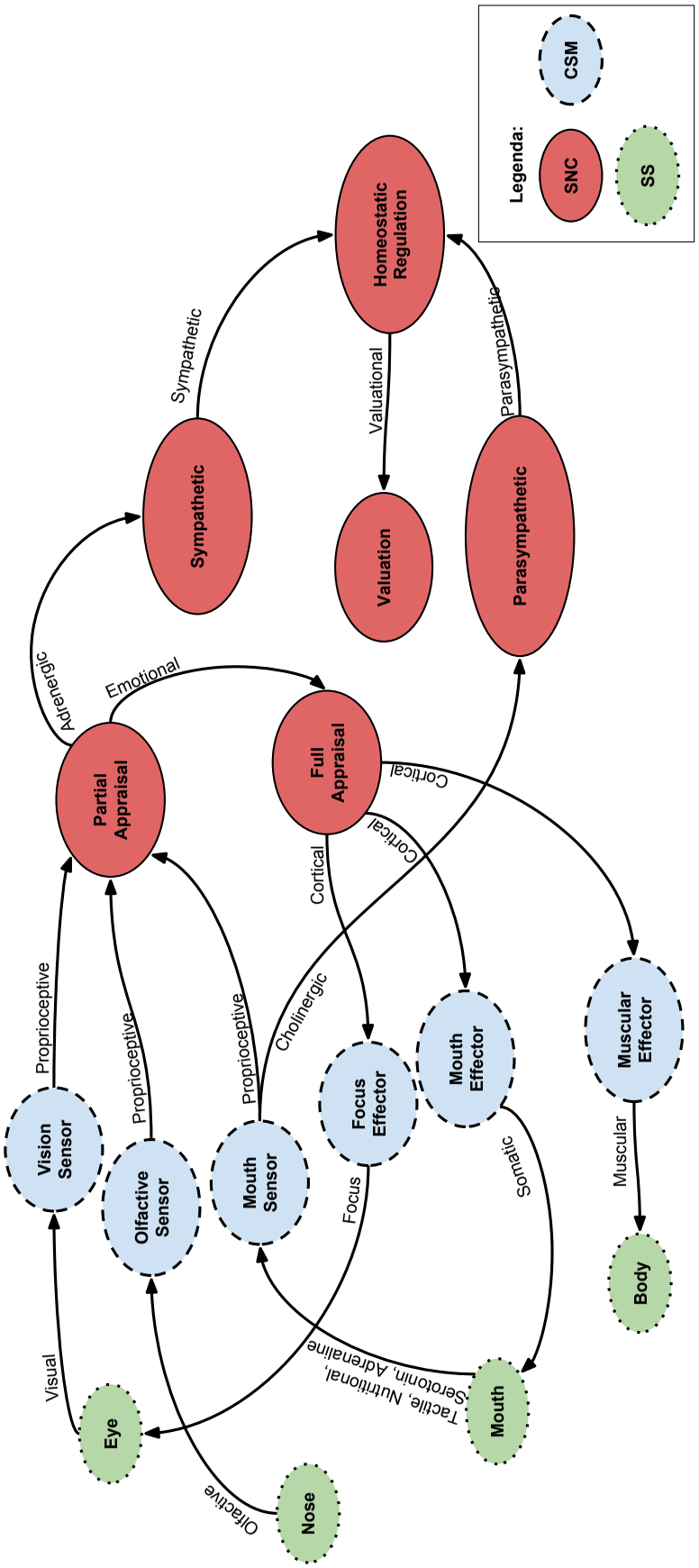


Figura 1 – Ciclos de estimulação internos da criatura artificial. O tipo do estímulo emitido pelo componente está descrito no lado esquerdo da seta. Os componentes estão discriminados pela sua classe, segundo a legenda.

desregulada ao componente FullAppraisal, essa emoção é utilizada para calcular a eficiência comportamental da criatura. Neste exemplo supõe que o *arousal* mais desregulado seja o da fome. O PartialAppraisal também envia um estímulo do tipo Adrenergic ao componente Sympathetic, dizendo que o arousal das emoções deve ser atualizado;

6. O componente Sympathetic envia um estímulo ao HomeostaticRegulation para atualizar o *arousal* de todas emoções, somando uma constante que será chamada de delta simpático Δ_s;

7. o FullAppraisal, ao receber o estímulo do tipo Emotional, criará uma lista de possibilidades de ações, ou *affordances*, que no caso são Comer o nutriente A, comer o nutriente B, evitar os nutrientes, andar em qualquer direção ou dormir. Essa lista, juntamente com a emoção mais desregulada, será enviada ao AS para escolher a próxima ação a ser executada que melhor regula a emoção fome. o AS utiliza os sistemas condicionamento e memória para desambiguar o conjunto de ações, caso não seja possível ele escolhe uma ação aleatória do conjunto de *affordances*. Assumindo que a ação de comer o nutriente A será escolhida, o FullAppraisal envia para os componentes do córtex efetor (FocusEffector, MouthEffector e MuscularEffector) em um estímulo cortical essa informação;

8. O estímulo cortical recebido no MouthEffector é transmitido ao componente Mouth por uma mensagem do tipo Somatic e ao ser recebida no orgão efetor, desencadeia um estímulo destrutivo ao nutriente A;

9. Os outros componentes do córtex efetor atualizam os outros membros do sistema somático o valor da eficiência comportamental, que é utilizada para calcular os parâmetros de sensibilidade (dimensão do campo visual, olfativo, tamanho do passo, etc.);

10. Ao receber uma mensagem destrutiva, o nutriente se remove do mundo e envia à criatura que o distruíu um estímulo nutritivo;

11. O componente Mouth, recebendo o estímulo nutritivo envia um estímulo do tipo Nutritional para o MouthSensor;

12. O sensor, ao receber o estimulo nutricional, envia ao Parasympathetic um estímulo Cholinergic;

13. O estímulo Cholinergic é recebido no Parasympathetic que encaminha ao HomeostaticRegulation informando que o *arousal* da emoção fome deve ser diminuído. Essa diminuição é igual ao valor nutritivo do estímulo recebido;

14. O HomeostaticRegulation envia uma mensagem ao Valuation informando que o nutriente já foi comido e a emoção já regulada, e a ação que produziu essa regulação deve ser valorada;

15. O Valuation atualiza a memória de condicionamento e a memória de longo prazo ao receber uma mensagem do HomeostaticRegulation e o cíclo de estímulação termina neste passo.

É importante ressaltar que o processamento dos estímulos acontece de maneira concorrente, e essa é só uma das possíveis ordens de execução. Ademais, o início de um novo ciclo em algum outro componente não influencia o que já está em execução, produzindo comportamentos distintos.

Dada a organização entre os componentes da criatura, a estratégia escolhida para implementá-la foi criar um ator tipado[2] cujo sub-atores criados são seus componentes internos. A hierarquia dos objetos é de um único nível, de forma a facilitar a um componente acessar a referência para outro componente.

Como as mensagens em Akka são tratadas de forma individual e sequencial, foi necessário estender a *mailbox* default do *toolkit*, mudando o seu comportamento ao desenfileirar. Na nova implementação, ao ser solicitada a próxima mensagem, o método empacota todas as mensagens que estão na fila em uma lista e entrega essa lista ao ator. Esse novo comportamento não permite ao ator saber o **ActorRef** de cada remetente a partir do método **sender()**, uma vez que os estímulos são desempacotados de seus envelopes que contém essa informação, e empacotados em um novo envelope sem remetente. Entretanto, todo estímulo enviado possui dois identificadores do tipo **SequentialId** que mantêm o *id* do remetente e do destinatário. Para recuperar o ator que enviou a mensagem, o ator da criatura, que supervisiona os componentes, mantêm uma tabela *hash* que mapeia **SequentialId**s para o **ActorRef** do componente.

O **PartialAppraisal** é um tipo especial de componente, que precisam exibir um comportamento ativo, enviando estímulos ainda que não tenha recebido nenhum. Como os atores de Akka são reativos, no sentido de que só são executados ao receber uma mensagem de outro ator, foi criada uma tarefa no *scheduler* de cada criatura para uma mensagem periódica para o **PartialAppraisal**. Assim, o componente é executado periodicamente para tratar, ao menos, essa mensagem.

[2] Atores em Akka podem ser não tipados (como nos exemplos do capítulo anterior) ou tipados. Atores tipados devem implementar alguma interface e são acessíveis através e somente dela, como objetos convencionais, mas com as mesmas propriedades de atores convencionais. Para maiores detalhes de implementação, a documentação se encontra em .

Por fim, um ponto importante da implementação das criaturas artificiais é a comunicação com o banco de dados. Ela deve acontecer sempre que um componente terminar sua execução, persistindo dados importantes para analise posterior. Esses dados em geral são o estado de cada componente, o conjunto de estímulos que ele recebeu e que alterou seu estado interno, e o conjunto de estímulos que ele emitiu ao alterar esse estado. A tecnologia utilizada para criar as tabelas do banco de dados e fazer o acesso durante a simulação foi a *Java Persistence API* (JPA). Ela foi escolhida para manter compatibilidade com versões anteriores da arquitetura Artífice.

O diagrama de classes que são persistidas em banco de dados está representado na Figura 2. É possível ver na figura que todos os objetos que representam algum estado relevante durante a simulação estão associados a um **ChangeStimulusState**. Esta entidade de banco de dados é de grande importância pois é responsável por manter as transduções de estímulos. A saber, uma transdução é a transformação de estímulo que recebido em um estímulo emitido, produzindo uma mudança de estado interno em um componente. Toda alteração de estado é salva com o *timestamp* em que ela aconteceu e juntamente com a entidade **StimulusState**, forma um grafo pelo qual é possível reconstruir a cadeia de estimulação que produziu um comportamento qualquer da criatura artificial.

É relevante saber o instante em que os componentes alteraram seu estado uma vez que este modelo cognitivo se comporta como um sistema dinâmico no tempo, e a maioria das analises são feitas baseadas nele. Além disto, como este é um sistema inteligente que deve exibir um comportamento coerente, é necessário também correlacionar a dinâmica externa da criatura artificial com seu estado interno. A título de exemplo, é preciso saber se a criatura come quando sua emoção mais desregulada é a fome, ou qual mecanismo de decisão ele usa para desambiguar uma ação.

4.3 Dos componentes da simulação em ambiente distribuído

Ao particionar o funcionamento da arquitetura Artífice. foram identificadas quatro papeis importantes que podem funcionar de modo distribuído, elas são: o gerenciamento da simulação, gerenciamento dos componentes da simulação (criaturas e nutrientes), o fornecimento de identificadores, e a manutenção física da simulação. Para cada uma das responsabilidades apresentadas foram projetados um ator e suas peculiaridades serão descritas a diante. A Figura **??** mostra um esquema de como os atores se organizariam com K holders, M criaturas e N nutrientes. Este esquema é um dos muitos possíveis que poderiam ser adotados e foi escolhido tendo em vista o balanceamento de criaturas e nutrientes ao longo dos nós de processamento, entretanto, outras possibilidades de organização devem ser estudadas em trabalhos futuros, analisando a performance em termos de sistemas distribuídos, bem como diferentes esquemas influenciam o comportamento do sistema

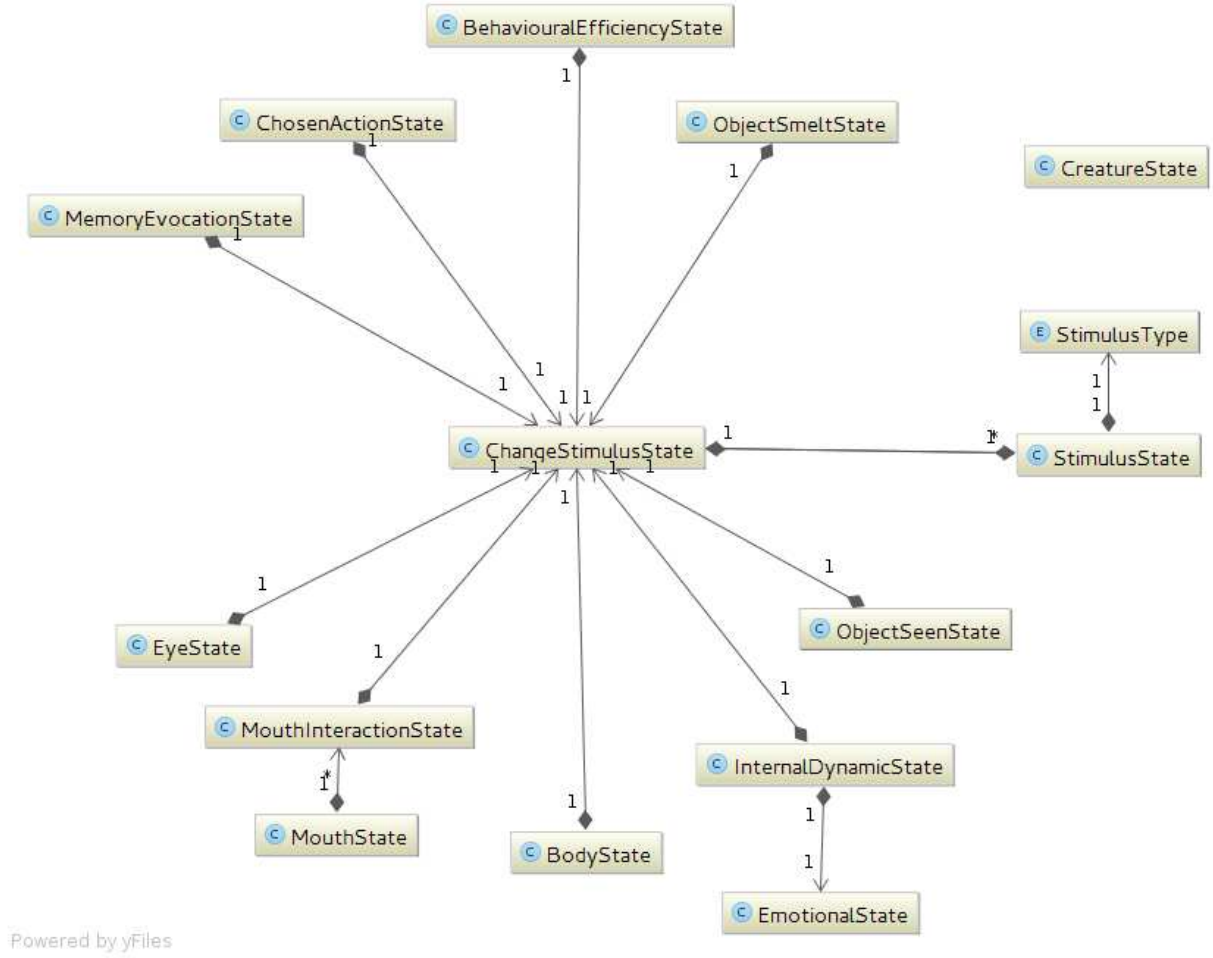


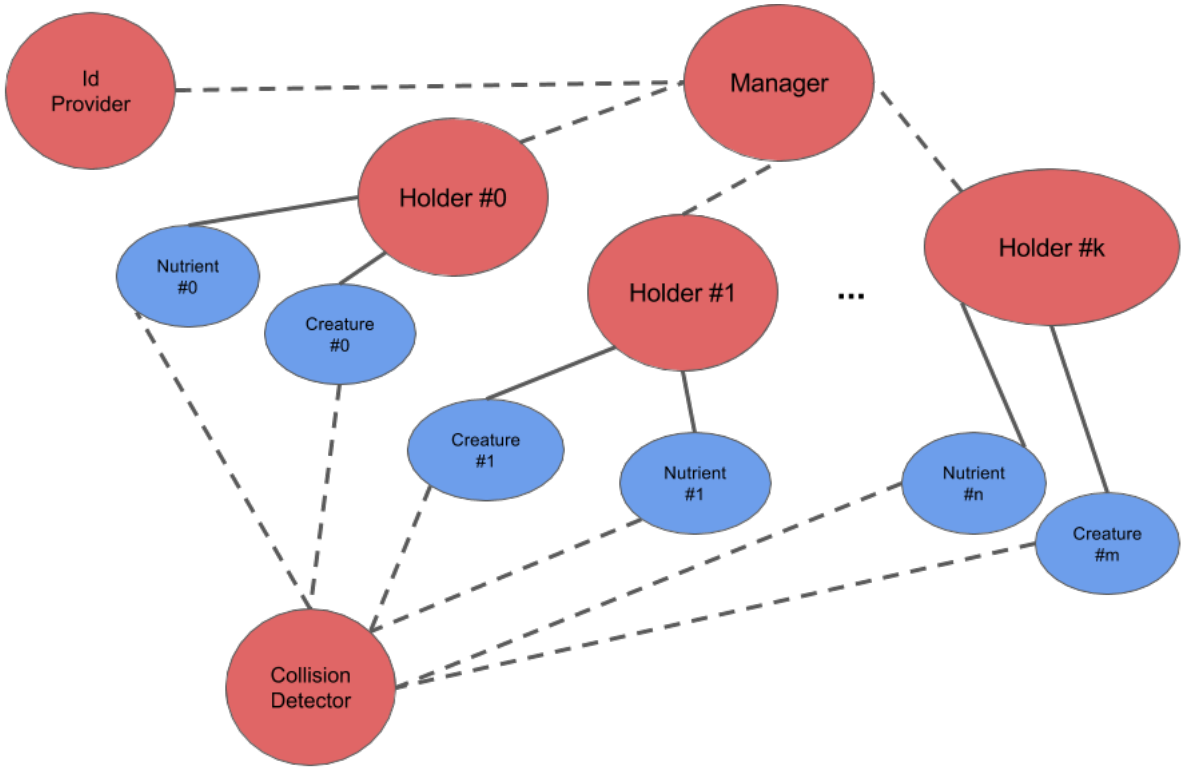
Figura 2 – Diagrama de classes do banco de dados

modelado.

Para auxiliar na comunicação de atores em diferentes processos foi utilizada a biblioteca *akka-cluster* [3] que faz parte do *toolkit* Akka. Ela oferece recursos para identificar membros e propagar informação entre atores distribuídos diferentes processos de maneira transparente. É baseada no protocolo *gossip* e possuí um detector automático de falhas, oferecendo um serviço sem ponto único de falha ou ponto único de gargalo.

O ator responsável pelo gerenciamento da simulação, o *manager*, tem o papel de iniciar e interromper a simulação, fazendo interface com o usuário final. Ele mantém uma tabela com a referência de todos os nós associados a uma simulação, e é responsável por entregar um identificador a todo novo nó do tipo *holder* que entra no *cluster*. Esse identificador será necessário para encontrar de forma eficiente um membro da simulação em um nó do *cluster* partindo de outro nó qualquer. Ao receber uma notificação que um nó importante falhou, a simulação é imediatamente interrompida pelo *manager*.

[3] http://doc.akka.io/docs/akka/current/java/common/cluster.html



Como os identificadores do tipo **SequentialId** são globais para toda a simulação, optou-se por gerá-los de forma centralizada, garantindo que não existam dois identificadores repetidos. Essa estratégia foi adotada por ser a que teria menor *overhead* de implementação. Tratar a falha do *id-provider*, ator responsável por gerar os identificadores, é simples de corrigir: reinicia-se o ator em outro nó. O *manager* deve ter conhecimento do *id-provider* mas não é necessário mantê-lo indexado.

As colisões entre os objetos da simulação continua sendo verificada por um nó centralizado chamado *Collision Detector*. Esse nó recebe mensagens constantemente, sempre que uma criatura atualiza sua posição ou um nutriente é destruído, e deve responder, a cada nova mensagem, com quais objetos o remetente colidiu. O algoritmo de pesquisa que encontra o conjunto de objetos com o qual o remetente da mensagem colidiu é linear no número de objetos no mundo, o que pode ser um gargalo, a medida que o tamanho do mundo cresce. Esta tarefa é de crucial importância para um AD-MAS como este posto que toda a dinâmica do mundo (a interação entre criaturas e objetos do mundo) depende dela. Estando ela centralizada em um nó, além de caracterizar um ponto único de falha, o que é um aspecto negativo do ponto de vista de sistemas distribuídos podendo ser prejudicial para o desempenho e escalabilidade, pode também degradar a dinâmica interna das criaturas. Neste sentido, é importante reforçar a necessidade de avaliar outros modelos de distribuição, levando em conta uma melhor abordagem para o problema de verificar colisões em um sistema distribuído que depende do tempo.

Finalmente, o *holder* é responsável por manter as entidades da simulação, criaturas e nutrientes. Ele também mantém uma tabela com os outros *holders* que fazem parte do *cluster*, índice que é construído dinamicamente à medida que as mensagens vão sendo trocadas. Ao entrar no *cluster*, um *holder* deve receber um identificador inteiro sequencial do *manager*. Ao receber a informação do usuário que um novo membro deve ser criado, é papel do *manager* solicitar um novo **SequentialId** ao *id-provider*. O resto da divisão inteira da super-chave do identificador pelo número de *holders* no *cluster* informa em qual *holder* o novo membro deve ser criado. Assim o *manager* ordena a criação do novo membro no nó apropriado.

As criaturas dentro de um *holder* podem eventualmente querer se comunicar com membros em outro *holder* a partir dos identificadores trocados nas mensagens, sem saber exatamente onde está fisicamente o ator, *e.g.* uma criatura que está no *holder-1* pode querer enviar um estímulo destrutivo para um nutriente que esteja no *holder-2*. Para tanto, a criatura encaminha a mensagem ao seu *holder*: caso ele conheça o nó do destinatário, a mensagem é encaminhada a ele. Caso o *holder* do destinatário seja desconhecido, a mensagem é enviada ao *manager* que a encaminha ao *holder* apropriado e responde com o endereço dele ao *holder* remetente, para que ele seja indexado. Esse algoritmo é baseado no *plugin akka-cluster-sharding*, que oferece a funcionalidade de endereçar um ator por um identificador lógico sem conhecer sua localização física. Optou-se por adotar uma versão simplificada do algoritmo em detrimento do *plugin* uma vez que o segundo permite a migração de entidades entre nós do cluster, o que não é uma funcionalidade desejável para a simulação de vida artificial proposta, pois não se sabe o impacto que a transição de um nó para outro pode causar a dinâmica interna da criatura artificial.

4.4 Considerações finais

Até aqui foi apresentada uma proposta de de modelagem e implementação de um simulador de criaturas artificiais, dotadas de um sistema nervoso artificial que se comporta como um AD-MAS. Foram apresentados os sistemas e sub-sistemas da criatura, bem como os componentes desses subsistemas e suas interações. Uma proposta de organização dos atores em rede

Capítulo 5

Análise e Discussão dos Resultados

Apresentada no capítulo Capítulo 4 a modelagem de um simulador de criaturas artificiais assíncrono e distribuído, baseado no arquitetura Artífice, é necessário validar dois aspectos do software proposto. Primeiro, no que diz respeito ao comportamento das criaturas artificiais, se ele é qualitativamente comparável com a arquitetura Artífice. O segundo aspecto a se verificar é quão escalável é o sistema, horizontal e verticalmente.

Os experimentos foram realizados no *cluster* do Laboratório de Sistemas Inteligentes (LSI) do CEFET, composto de oito máquinas com processador Intel i7, 32GB de RAM e 2TB de HD. O sistema operacional utilizado é o CentOS 6, e o SLURM como escalonador de tarefas. Cada experimento foi executado 30 vezes. Esse número de repetições foi escolhido de de modo que os valores de erros relativo às médias seja aceitável.

Da dinâmica externa das criaturas artificiais foram escolhidos os dados de tempo de vida, distância percorrida, nutrientes comidos. Dos resultados da dinâmica interna é importante analisar três medidas relevantes ao longo do tempo, a saber: a quantidade de estímulos trocados, a utilização dos mecanismos de aprendizagem e seleção de ação e a quantidade de presas ingeridas. A primeira delas é relevante para verificar se todos os estímulos estão sendo corretamente enviados e recebidos, e se a taxa com que eles são trocados é coerente. O segundo e terceiro aspecto dizem da adaptação e do correto funcionamento dos mecanismos de aprendizagem, se as escolhas que a criatura faz (e como ela as faz) guarda correlação com sua história de interações passadas. Dito de outro modo, é necessário saber se a criatura aprende que comer uma presa sem valor nutricional não vale a pena, que dormir quando está com fome também não vale a pena e que comer um elemento de maior valor nutricional é melhor que comer um de menor valor.

Como descrito no capitulo anterior, os dados da simulação são gravados em banco de dados relacional, e para todo evento é mantido o *timestamp* (em milisegundos) em que ele aconteceu. Para calcular as médias temporais, os eventos são ordenados e agrupados pelo

tempo, convertido em minutos com precisão de três casas decimais. Para cada criatura, em cada instante de tempo, a média aritimética naquele instante é calculada. O erro relativo é calculado segundo a Equação 1.

$$ER(X) = Z_\alpha \frac{S}{\bar{X}\sqrt{n}} \tag{1}$$

Onde X é a amostra de tamanho n, $\bar{X}$ é a média amostral, S é o desvio padrão amostral e Z_α é a constante para um intervalo confiança de 95%. O erro relativo $ER(X)$ é dado em percentual e vale como uma medida da qualidade da média da amostra X.

Desta feita, a próxima seção apresenta os resultados de validação do modelo, contrapondo com os resultados da arquitetura Artífice para configuração semelhante. A Seção 5.2 apresenta o experimento feito em uma única máquina para verificar a escalabilidade vertical do software. A Seção 5.3 apresenta os resultados do experimento executado com mais de uma máquina a fim de verificar a escalabilidade horizontal.

5.1 Validação do modelo

Este experimento foi realizado para verificar a compatibilidade do comportamento das criaturas entre a arquitetura Artífice, que inspirou este trabalho, e a versão aqui produzida. Foi executada uma simulação de forrageamento (busca por alimento) com uma única criatura em cada arquitetura e 50 presas de três tipos diferentes totalizando 150 nutrientes, que são: maçãs vermelhas (ra), maçãs verdes (ga), e maçãs cinzas (gra). O valor nutricional N em ambos os experimentos respeita a relação $N_{gra} = 0 < N_{ra} < N_{ga}$. Cada experimento foi repetido por 30 vezes. A arquitetura Artífice foi configurada com um $\Delta_{sym} = 3 \times 10^{-3}$, enquanto a arquitetura DL2L foi configurada com um $\Delta_{sym} = 1.5 \times 10^{-3}$. Esses parâmetros foram escolhidos de forma que as criaturas pudessem interagir com o mundo artificial por um tempo aceitável, sendo possível observar o seu comportamento.

Para a configuração descrita a arquitetura Artífice obteve um tempo de vida médio de $12.6 \pm 9.96\%$ em minutos e comeu um número de $25.3 \pm 10\%$ presas. O teste de normalidade para o tempo de vida e presas comidas resultou em p-valores de 1.4% e 3.7% respectivamente, o que indica que não permite dizer que a distribuição é normal. Já a arquitetura DL2L obteve nesse experimento um tempo de vida médio de $31.4 \pm 6.79\%$ em minutos e comeu, em média, $304.2 \pm 4.35\%$ presas. O teste de normalidade para o tempo de vida e presas comidas resultou em p-valores de 48.07% e 65.72%, o que indica que as distribuições são normais para um intervalo de confiança de 95%.

A média temporal da troca de estímulos entre as arquiteturas está apresentada na Figura 3.

Vale dizer que a arquitetura DL2L tem um número menor de tipos de estímulos comparada à Artifice, e isso se deva a uma simplificação dos componentes e do fato de que nem todas as funcionalidades foram implementadas por completo.

A Figura 4 exibe a média da soma acumulada de escolhas feitas no tempo, separadas por mecanismo. Nesta simulação foram utilizados somente os três mecanismos mais básicos: a criatura seleciona primeiro os alvos que estão à menor distância, depois seleciona a ação mais provável para cada alvo baseado na memória de condicionamento, e por fim, caso não tenha restado uma ação única, ela faz uma escolha aleatória.

A eficiência comportamental da criatura artificial é uma função do máximo *arousal* e da quantidade de objetos no campo sensório. Ela é calculada durante a execução da simulação e é utilizada para determinar a velocidade do passo da criatura e o foco atencional. Existem duas fórmulas para a eficiência comportamental, uma para tarefas simples, onde a decisão envolve um único objeto, e outra para tarefas complexas que envolvem mais de um objeto. A relação da eficiência comportamental em função do *arousal* máximo A e o número de objetos no campo sensor N é apresentada na Equação 2:

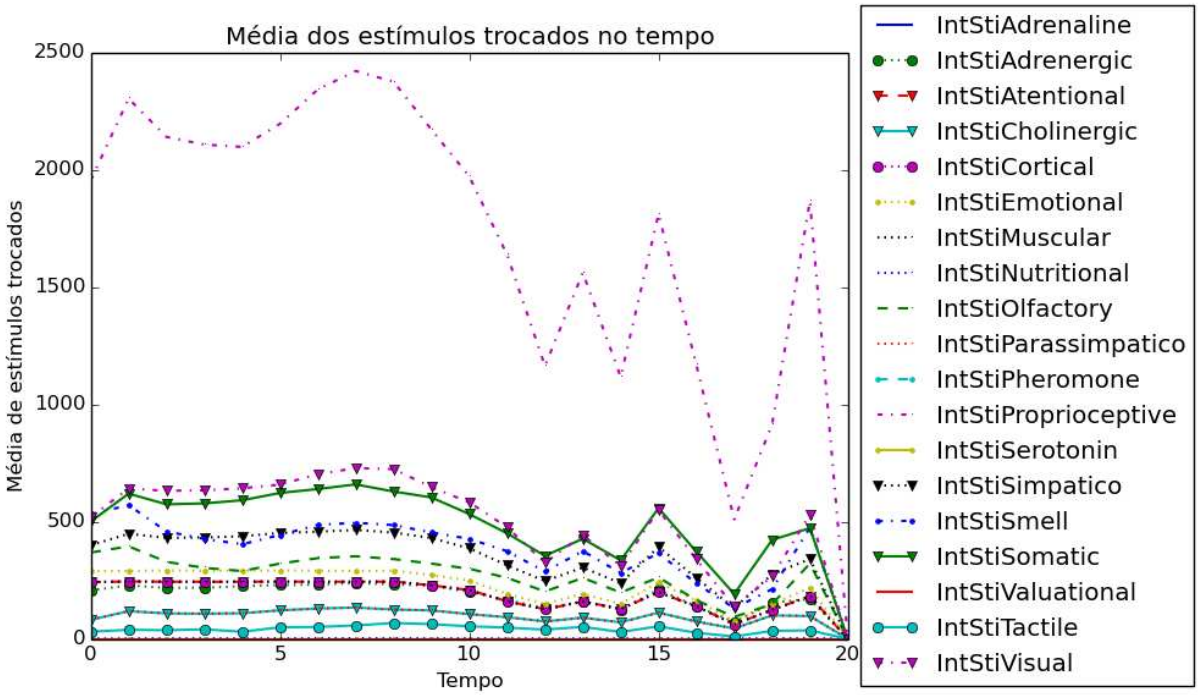
$$BE(A,N) = \begin{cases} 16 - 16e^{-0.4A} & \text{se } N < 2 \\ 5.714A - 0.816A^2 & \text{se } N \geq 2 \end{cases} \tag{2}$$

A Figura 5 exibe a média temporal normalizada da eficiência comportamental para as duas arquiteturas.

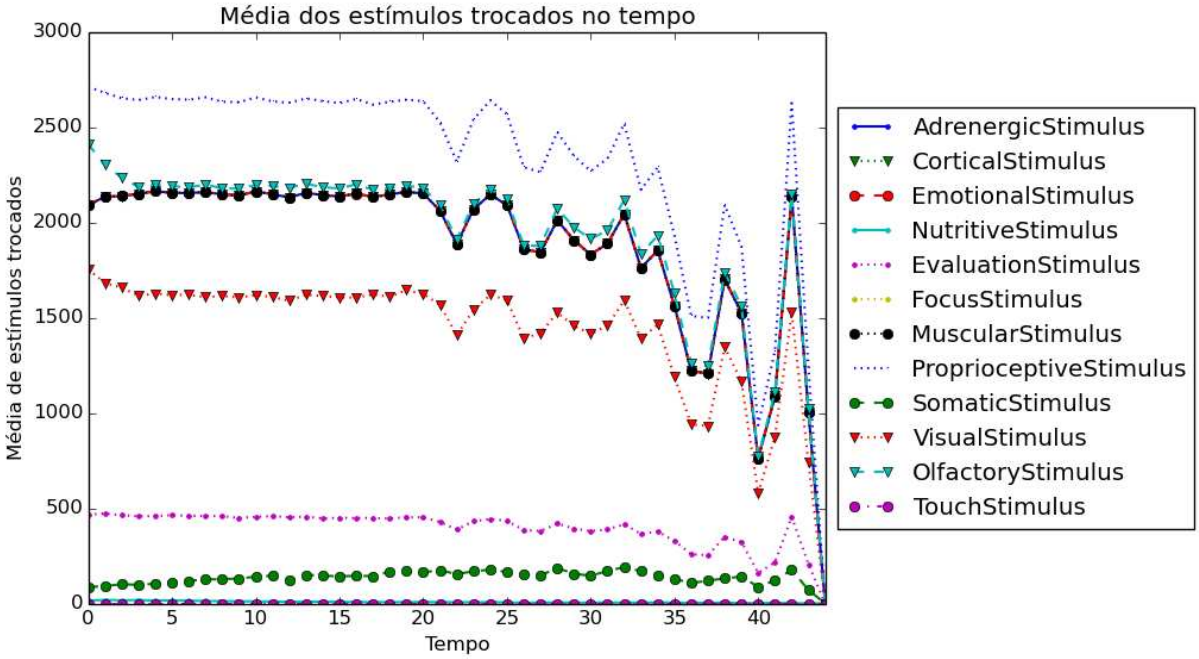
5.2 Escalabilidade vertical

A fim de verificar a influência de mais de uma criatura em um mesmo *holder* na arquitetura DL2L, foi planejado este segundo experimento.Foram executadas cinco configurações diferentes, iniciando com uma criatura em um nó até cinco criaturas em um nó. Cada configuração é repetida 30 vezes para garantir o mínimo de confiabilidade estatística. A densidade do mundo é mantida constante entre as configurações, iniciando em 50 presas de cada tipo (assim como no experimento anterior) em um mundo de 4.8×10^5 pixeis.

A Tabela **??** expõe os resultados da média de tempo de vida, distância percorrida e nutrientes comidos. O erro foi calculado de acordo com a Equação 1.

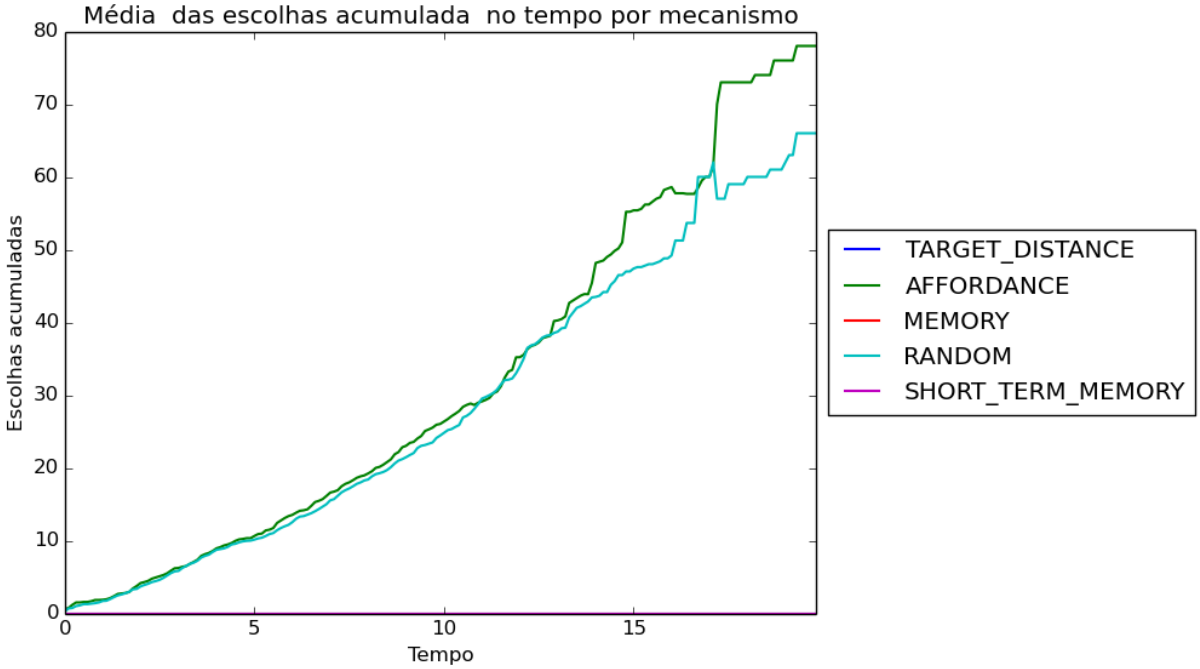


(a) Troca de estímulos na arq. Artífice

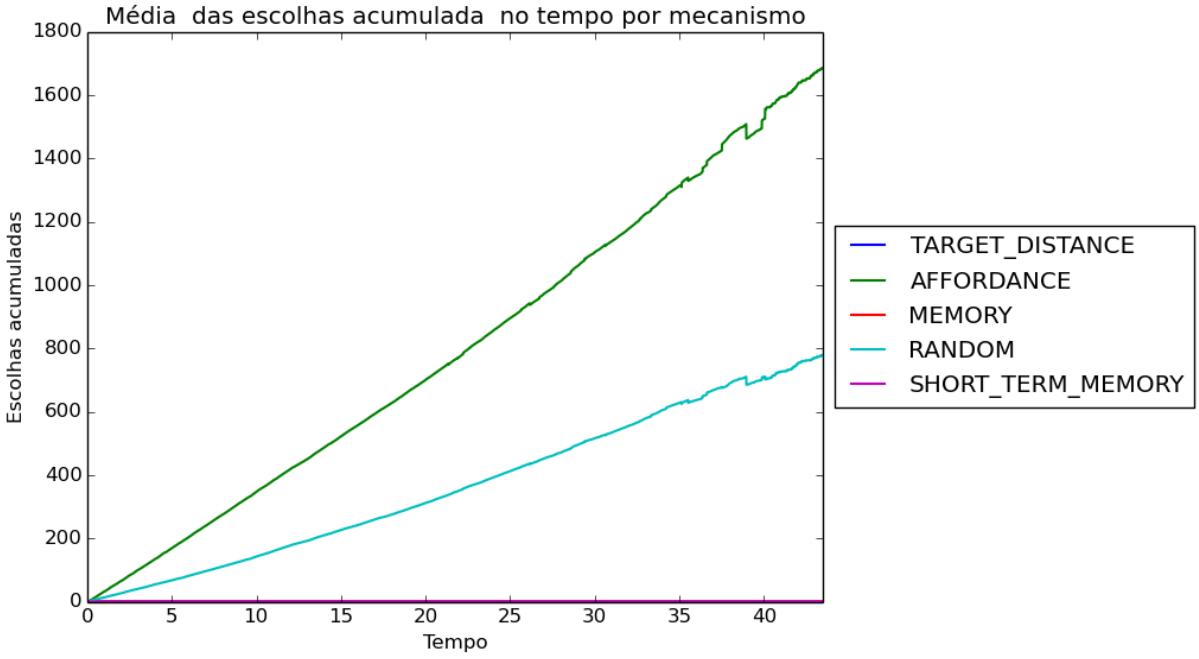


(b) Troca de estímulos na arq. DL2L

Figura 3 – Média da troca de estímulos no tempo para ambas as arquiteturas. Cada curva representa um tipo de estímulo diferente. Ambas as médias foram calculadas com precisão de minutos.

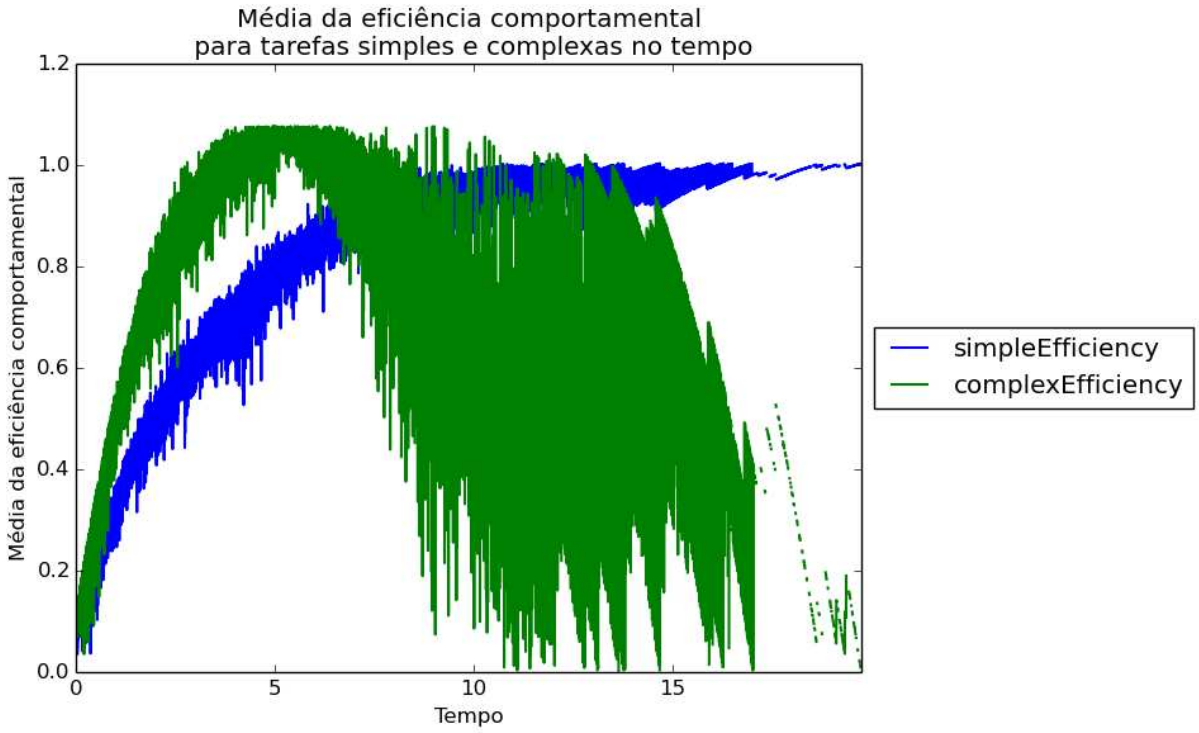


(a) Arquitetura Artífice

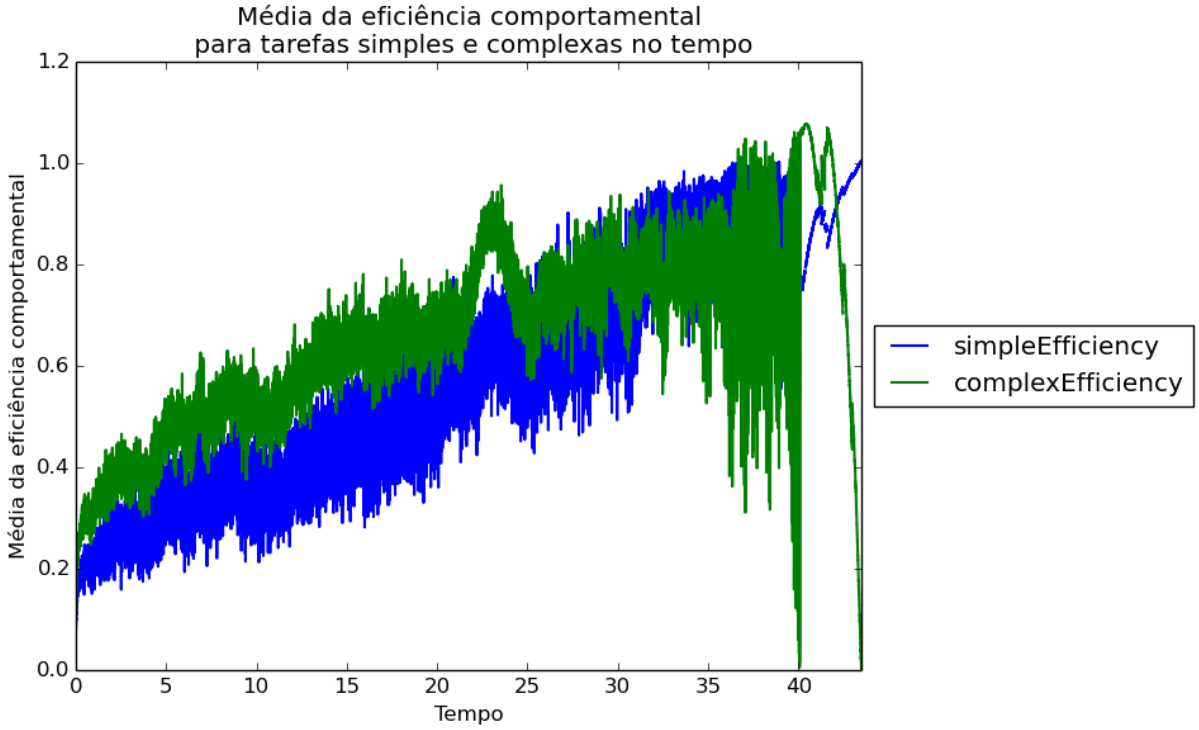


(b) Arquitetura DL2L

Figura 4 – Gráficos da média de escolhas acumuladas no tempo para a arquitetura Artífice e DL2L



(a) Arquitetura Artífice



(b) Arquitetura DL2L

Figura 5 – Gráficos da média da eficiência comportamental no tempo para a arquitetura Artífice e DL2L

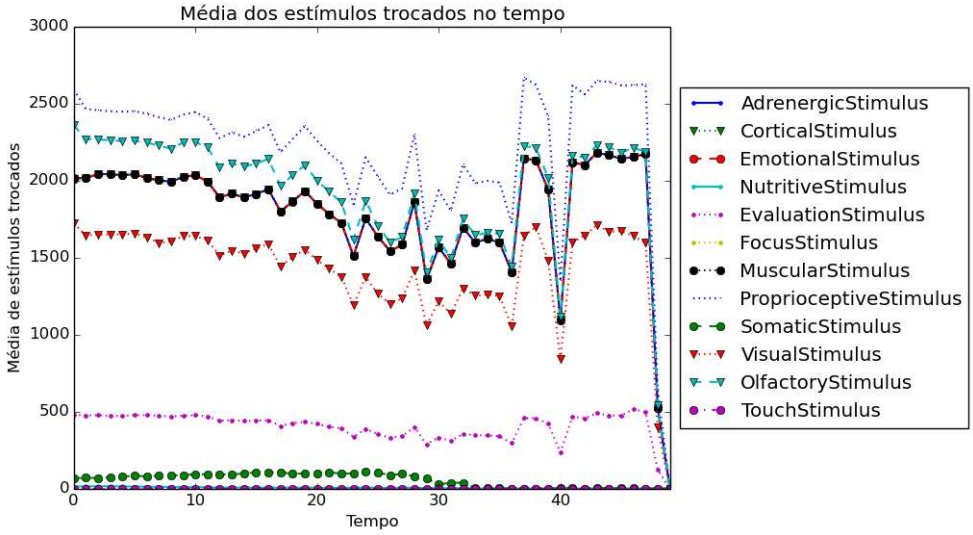


Figura 6 – Média temporal dos estímulos trocados em simulação utilizando 2 criaturas

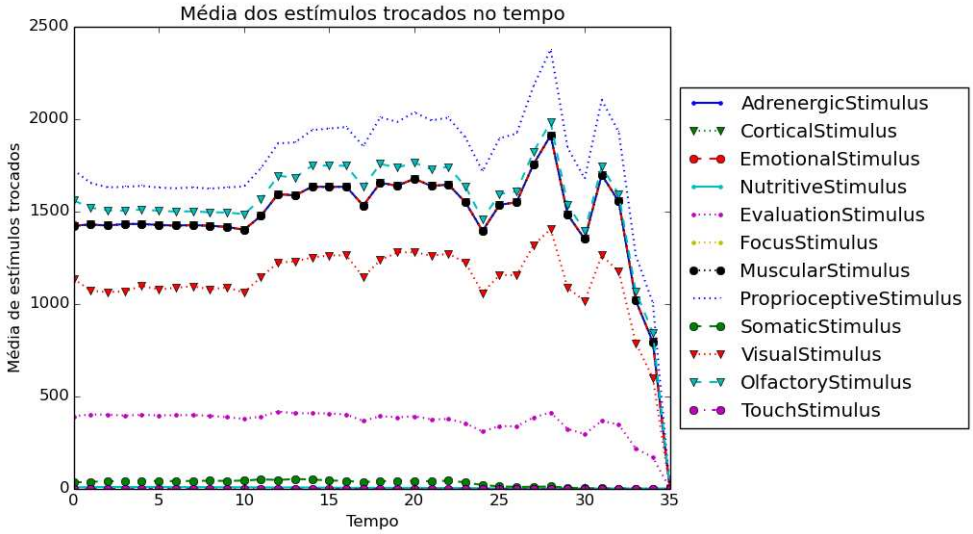


Figura 7 – Média temporal dos estímulos trocados em simulação utilizando 3 criaturas

Tabela 1 – Média do tempo de vida, distância percorrida e nutrientes comidos para o experimento 2

Criaturas	Tempo de vida		Distância percorrida		Nutrientes comidos	
	Média	ER (%)	Média	ER (%)	Média	ER (%)
1	31.40	6.79	2.72E+05	9.33	304.20	4.35
2	21.71	8.43	1.83E+05	10.39	189.82	7.61
3	18.81	7.24	1.30E+05	9.90	115.38	7.79
4	13.51	6.99	9.59E+04	9.20	62.73	8.77
5	10.38	4.73	5.43E+04	7.39	31.24	8.97

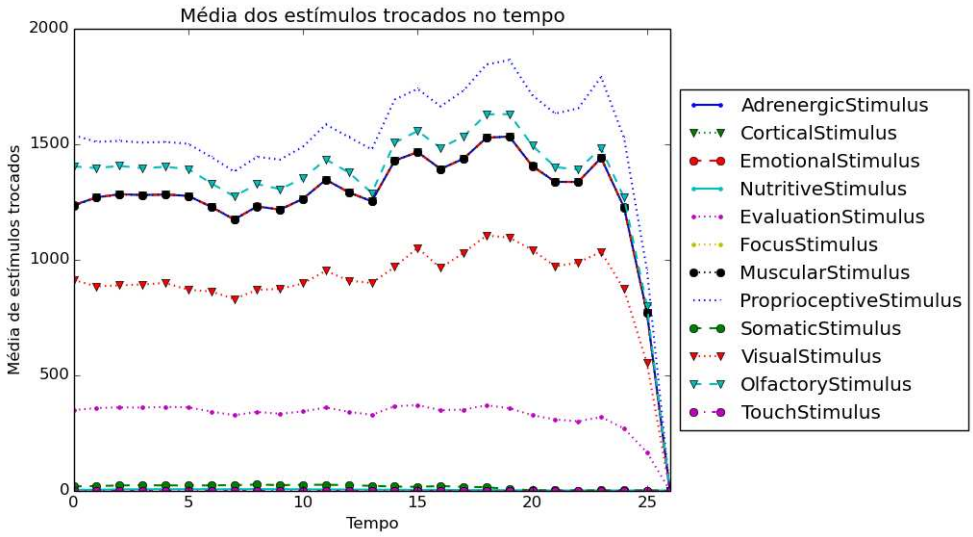


Figura 8 – Média temporal dos estímulos trocados em simulação utilizando 4 criaturas

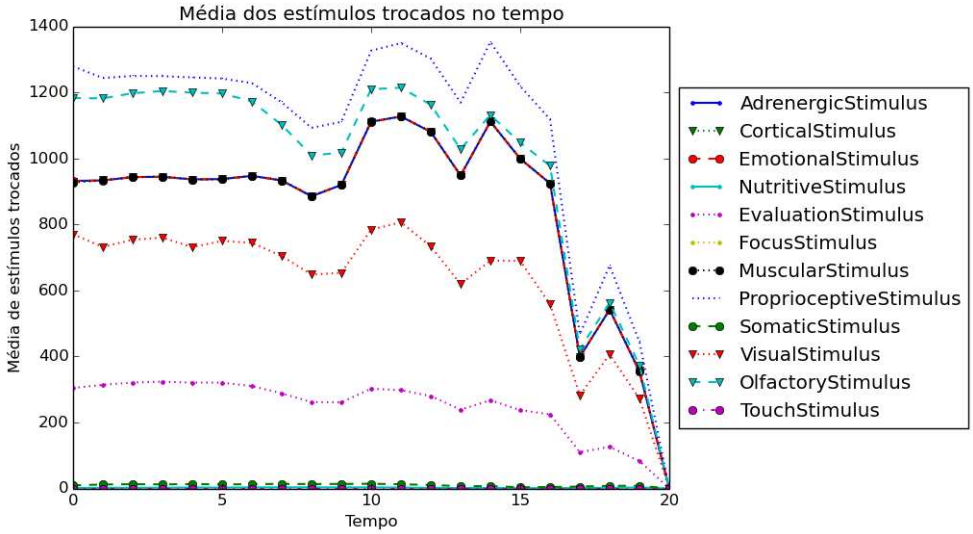


Figura 9 – Média temporal dos estímulos trocados em simulação utilizando 5 criaturas

5.3 Escalabilidade horizontal

Este experimento foi executado com intuito de verificar a influência de mais de um *holder* em uma simulação de forrageamento. Foram executadas cinco configurações diferentes, começando com um *holder* até cinco. Foi mantido o número de uma criatura por nó, e a densidade do mundo foi mantida constante, iniciando com 50 nutrientes de cada tipo em um mundo de 4.8×10^5 pixeis.

Na Tabela **??** estão apresentados os resultados da média de tempo de vida, distância percorrida e nutrientes comidos. O erro foi calculado de acordo com a Equação 1. É notável que a medida que o número de holders aumenta, há uma diminuiçao em todas as médias.

Dos resultados da dinâmica interna não houve alteração significativa , com exeção da média

Tabela 2 – Média do tempo de vida, distância percorrida e nutrientes comidos para o experimento 3

No. Holders	Tempo de vida	Distância percorrida	Nutrientes comidos
1	31.40 ± 6.79%	2.72E+05 ± 9.33%	304.20 ± 4.35%
2	24.69 ± 8.69%	2.24E+05 ± 11.27%	229.36 ± 7.03%
3	16.85 ± 6.74%	1.64E+05 ± 7.31%	121.67 ± 6.21%
4	6.75 ± 5.93%	7.06E+04 ± 6.63%	38.03 ± 6.60%
5	4.82 ± 3.75%	4.93E+04 ± 4.41%	15.83 ± 7.44%

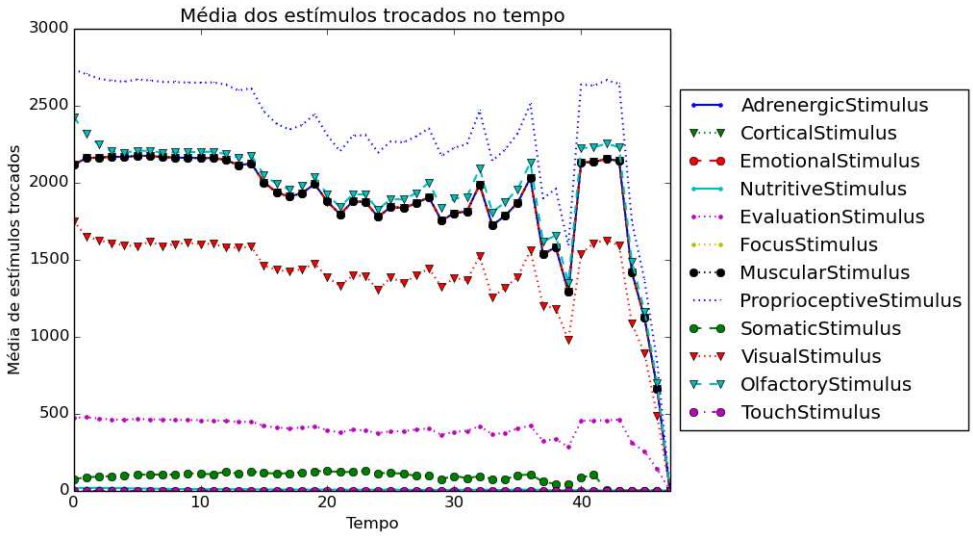


Figura 10 – Média temporal dos estímulos trocados em simulação utilizando 2 *holders*

de estimulos trocados no tempo. Para cada experimento estão apresentados os resultados abaixo, salvo o experimento com um holder, cujo resultado é o mesmo apresentado na Figura 3b.

A medida que o tempo evolui e as criaturas morrem o erro relativo aumenta, e os dados próximos ao final do eixo temporal não tem confiabilidade estatística. Os gráficos da evolução do erro relativo no tempo estão apresentados no **??**.

5.4 Síntese dos resultados

A respeito do primeiro experimento, não é possível fazer uma comparação direta entre as duas arquiteturas pois há uma mudança de parâmetro que foi justificada pelo tempo de execução dos experimentos. As dinâmica interna da criatura é extremamente sensível ao Δ_{sym}, e um decremento de centésimos pode aumentar o tempo de vida em horas. Entretanto, qualitativamente o comportamento das criaturas é parecido, principalmente no que diz respeito a dinâmica interna. Por exemplo, na Figura 4 é possível ver que o estímulo que tem a maior média temporal é o que o proprioceptivo, cuja origem é dos componentes sensores.

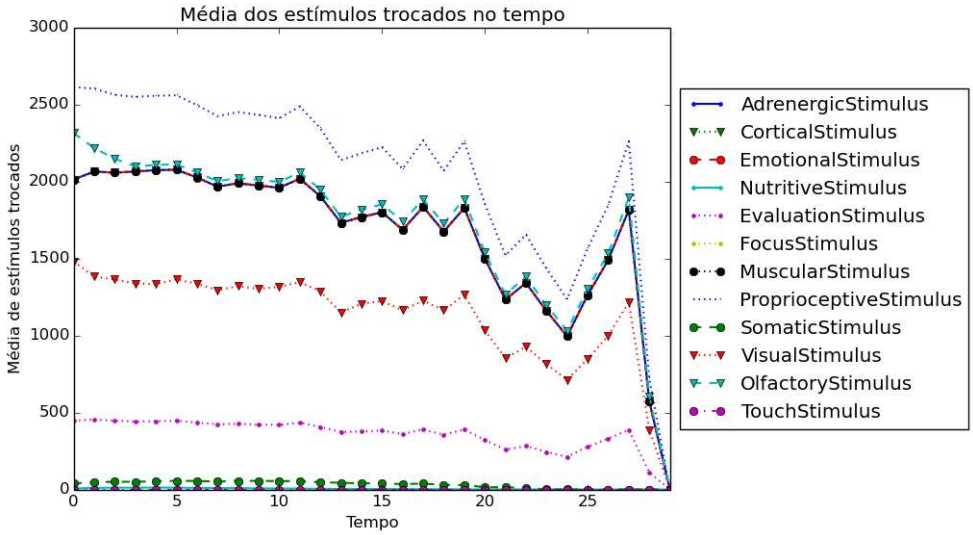


Figura 11 – Média temporal dos estímulos trocados em simulação utilizando 3 *holders*

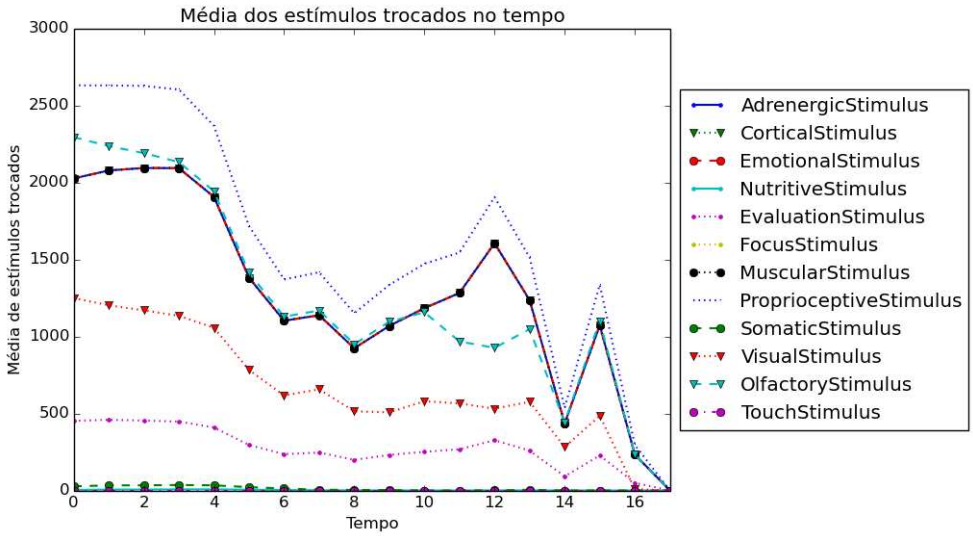


Figura 12 – Média temporal dos estímulos trocados em simulação utilizando 4 *holders*

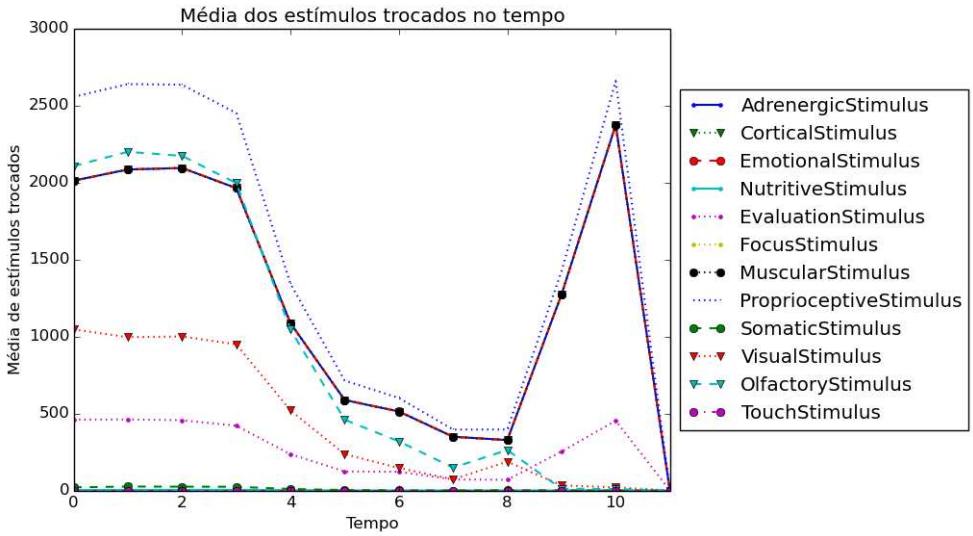


Figura 13 – Média temporal dos estímulos trocados em simulação utilizando 5 *holders*

No uso dos mecanismos de escolha há uma discrepância entre as arquiteturas, enquanto na Figura 4a as escolhas aleatórias praticamente acompanham as escolhas por *affordances*, na Figura 4b as escolhas aleatórias estabilizam muito abaixo do outro mecanismo. Para explicar esse comportamento pode-se levantar uma hipotese sobre a velocidade e quantidade de estímulos que são recebidos e tratados pelos componentes simultaneamente. No Artífice, os componentes do sistema nervoso são escalonados para tratar estímulos à uma taxa fixa, o que leva a uma acumulação de estímulos no *buffer* compartilhado por eles. Já a arquitetura DL2L tem um caráter reativo, *i.e.*, os componentes reagem somente na presença de estímulo, que não tem uma frequência fixa, com exceção dos estímulos enviados pelo *PartialAppraisal*. Portanto, tão logo um componente recebe um estímulo, ele é escalonado para tratá-lo. Isso diminuí o número de estímulos simultâneos, o que reduz o número de *affordances* que a criatura precisa escolher possibilitando uma desambiguação mais efetiva, diminuindo o número de escolhas aleatórias.

Sobre a diferença entre as curvas de eficiência comportamental, é possível que ela aconteça em razão da diferença entre as constantes internas da criatura e também da frequência com que os componentes são escalonados em cada arquitetura, mas isso deve ser melhor investigado.

Da escalabilidade horizontal da arquitetura DL2L.

Capítulo 6

Conclusão

A proposta deste trabalho foi remodelar uma arquitetura multi-agente utilizando o modelo de atores. Como foi apresentado no Capítulo 1, a arquitetura Artífice é um sistema multi-agente baseado no modelo de concorrência bloqueante utilizando *threads*. O que motiva tal tarefa é a dificuldade de se executar grandes simulações graças ao modelo de concorrência por estado compartilhado, e o fato dos conceitos do modelo de atores estarem em conformidade com a modelagem do sistema nervoso das criaturas artificiais adotadas no Artífice. O trabalho se justifica pelos resultados já alcançados com a arquitetura.

O Capítulo 2 faz uma revisão dos conceitos de programação concorrente baseado em *threads* e memória compartilhada, mostrando como surgem os principais problemas de sincronização, como condições de corrida e *deadlocks*. Logo em seguida o modelo de atores é introduzido, ressaltando seu comportamento assíncrono, e exemplos do funcionamento e utilização do *toolkit* Akka, implementação do modelo de atores adota, são mostrados. Um breve histórico da arquitetura Artífice é feito mostrando os seus mecanismos principais de aprendizagem e adaptação, e como o modelo de concorrência adotado na sua construção perdurou. O modelo de atores então é proposto como solução para esses problemas visando a escalabilidade da arquitetura.

A metodologia do trabalho é apresentada no Capítulo 4 e uma descrição do desenvolvimento do trabalho é feita. O funcionamento interno da criatura é baseado em ciclos de estimulação e para reconstruir seu funcionamento é necessário começar por esses ciclos. O software também foi particionado em diferentes funcionalidades, que foram distribuídas entre atores que podem executar em processos diferentes e a comunicação entre eles deve ser feita pela rede.

O trabalho até aqui conta com uma versão preliminar da arquitetura Artífice, cujo ciclo de estimulação sensório-motor já está em funcionamento. Esta versão foi testada em uma única máquina e exibiu um funcionamento coerente com o esperado. O próximo passo é

inserir um mecanismo de persistência em banco de dados para que essa versão possa ser validada no *cluster* do CEFET-MG.

6.1 Cronograma

O Quadro 1 exibe as próximas ativdades do projeto e seus respectivos prazos.

Quadro 1 – Cronograma para as próximas atividades do projeto

Atividades	Jul	Ago	Set	Out	Nov	Dez
Revisão da literatura	X					
Impl. dos demais circuitos	X	X	X			
Testes do mundo artificial			X	X		
Simulação de forrageamento				X	X	
Redação do texto final				X	X	X

Referências

AGHA, G. A. **Actors: A model of concurrent computation in distributed systems.** [S.l.], 1985. Citado na página 7.

BAARS, B. J. In the theatre of consciousness. global workspace theory, a rigorous scientific theory of consciousness. **Journal of Consciousness Studies**, Imprint Academic, v. 4, n. 4, p. 292–309, 1997. Citado na página 12.

BEDAU, M. A. et al. Open problems in artificial life. **Artificial life**, MIT Press, v. 6, n. 4, p. 363–376, 2000. Citado na página 1.

CAMPOS, L. M. d. A. et al. A concurrent, minimalist model for an embodied nervous system. In: **International Brazilian Meeting on Cognitive Science**. [S.l.: s.n.], 2015. Citado na página 1.

CAMPOS, L. M. de A. **Modelagem do processo cognitivo-emocional de um organismo artificial numa perspectiva dinâmico-interacionista**. 2006. Citado na página 14.

DUCH, W.; OENTARYO, R. J.; PASQUIER, M. Cognitive architectures: Where do we go from here? In: **AGI**. [S.l.: s.n.], 2008. v. 171, p. 122–136. Citado 2 vezes nas páginas 1 e 11.

FRANKLIN, S.; JR, F. P. The lida architecture: Adding new modes of learning to an intelligent, autonomous, software agent. **pat**, v. 703, p. 764–1004, 2006. Citado na página 12.

GROPP, W. et al. A high-performance, portable implementation of the mpi message passing interface standard. **Parallel computing**, Elsevier, v. 22, n. 6, p. 789–828, 1996. Citado na página 7.

HALLER, P. On the integration of the actor model in mainstream technologies: the scala perspective. In: ACM. **Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions**. [S.l.], 2012. p. 1–6. Citado na página 2.

HANSEN, P. B. **The origin of concurrent programming: from semaphores to remote procedure calls**. [S.l.]: Springer Science & Business Media, 2013. Citado 2 vezes nas páginas 2 e 4.

HEWITT, C. Actor model of computation: scalable robust information systems. **arXiv preprint arXiv:1008.1459**, 2010. Citado na página 2.

HEWITT, C.; BISHOP, P.; STEIGER, R. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In: STANFORD RESEARCH INSTITUTE. **Advance Papers of the Conference**. [S.l.], 1973. v. 3, p. 235. Citado 2 vezes nas páginas 2 e 7.

HEWITT, C.; ZENIL, H. **What is computation? Actor model versus Turing's model**. [S.l.]: World Scientific Publishing Singapore, 2013. Citado na página 7.

MAPA, S. **Modelagem de Organismos Artificiais Cognitivo-Emocionais Dotados de Memória Experimenal de Longo Prazo**. 2009. Citado na página 14.

MÜLLER, J. P.; FISCHER, K. Application impact of multi-agent systems and technologies: a survey. In: SPRINGER. **Agent-oriented software engineering**. [S.l.], 2014. p. 27–53. Citado na página 1.

NIAZI, M.; HUSSAIN, A. Agent-based computing from multi-agent systems to agent-based models: a visual survey. **Scientometrics**, Akadémiai Kiadó, co-published with Springer Science+ Business Media BV, Formerly Kluwer Academic Publishers BV, v. 89, n. 2, p. 479–499, 2011. Citado na página 1.

ODERSKY, M. et al. **An overview of the Scala programming language**. [S.l.], 2004. Citado na página 7.

PRASAD, M. N.; LESSER, V. R.; LANDER, S. E. Learning organizational roles in a heterogeneous multi-agent system. In: **Adaptation, Coevolution and Learning in Multiagent Systems: Papers from the 1996 AAAI Spring Symposium**. [S.l.: s.n.], 1996. p. 72–77. Citado na página 1.

RAMAMURTHY, U. et al. Lida: A working model of cognition. 2006. Citado na página 12.

SANTOS, B. A. **Aspectos Conceituais e Arquiteturais para a Criação de Linhagens de Agentes de Software Cognitivos e Situados**. Junho 2003. 130 f. Dissertação (Mestrado) — CEFET-MG, Belo Horizonte, 2003. Citado na página 13.

SANTOS, B. A.; BORGES, H. E. Biologically inspired architecture for creating cognitive situated software agents. In: **Workshop on Architecture and Methodology for Building Agent-Based Learning Environments**. [S.l.: s.n.], 2004. p. 111–116. Citado 2 vezes nas páginas 11 e 14.

SILVA, V. A. **Modelagem e Desenvolvimento de um Mecanismo de Condicionamento para a Arquitetura Artífice**. 2008. Citado na página 14.

SUN, R. **Duality of the mind: A bottom-up approach toward cognition**. [S.l.]: Psychology Press, 2001. Citado na página 13.

SUN, R. The clarion cognitive architecture: Toward a comprehensive theory of the mind. **The Oxford Handbook of Cognitive Science**, Oxford University Press, p. 117, 2016. Citado 2 vezes nas páginas 12 e 13.

TASHAROFI, S.; DINGES, P.; JOHNSON, R. E. Why do scala developers mix the actor model with other concurrency models? In: SPRINGER. **European Conference on Object-Oriented Programming**. [S.l.], 2013. p. 302–326. Citado na página 2.