Uma arquitetura baseada em Docker para Sistemas Multiagentes Abertos

Gustavo L. de Lima[1], Marilton S. de Aguiar[1]

[1]Programa de Pós Graduação em Computação (PPGC)
Universidade Federal de Pelotas (UFPEL)
Pelotas – RS – Brazil

{gustavolameirao,marilton}@inf.ufpel.edu.br

Abstract. *In Open Multi-Agent Systems (SMAA), heterogeneous agents (different environments/models) migrate from one system to another, taking their attributes and knowledge with them. The complexity of the opening comes from the dynamic behavior that the change of agents entails, being necessary to formulate techniques to analyze this complexity and understand the general behavior of the system. The article presents an architecture based on Docker to assist in the development of SMAA, acting in the migration of agents between different models running in heterogeneous hardware/software scenarios. We used a simulation scenario with the Open Sugarscape 2 Constant Growback model (NetLogo) and Gold Miners (JaCaMo) to verify the feasibility of the proposal.*

Resumo. *Em Sistemas Multiagentes Abertos (SMAA), agentes heterogêneos (diferentes ambientes/modelos) migram de um sistema para outro, levando seus atributos e conhecimentos. A complexidade da abertura vem do comportamento dinâmico que a mudança de agentes acarreta, sendo necessário formular técnicas para analisar essa complexidade e entender o comportamento geral do sistema. O artigo apresenta uma arquitetura baseada em Docker para auxiliar no desenvolvimento de SMAA, atuando na migração de agentes entre diferentes modelos rodando em cenários heterogêneos de hardware/software. Utilizamos um cenário de simulação com o modelo Open Sugarscape 2 Constant Growback (NetLogo) e o Gold Miners (JaCaMo) para verificar a viabilidade da proposta.*

1. Introdução

Os Sistemas Multiagentes Abertos (SMAA) são um tipo específico de Sistemas Multiagentes (SMA) que permite a interação entre agentes participantes de diferentes modelos. Em SMAA, analisamos agentes heterogêneos sob a perspectiva que estes podem migrar de um sistema para outro, levando seus atributos e conhecimentos [Jamroga et al. 2013]. A heterogeneidade dos agentes pode vir de diferenças entre os modelos, como arquitetura, objetivos ou políticas [Uez 2018].

No entanto, diferentes problemas surgem quando se desenvolve aplicações em SMAA quando comparado a SMA [Dalpiaz et al. 2010]. Primeiro, pode haver diferenças de implementação onde agentes e modelos podem ser criados por equipes diferentes, em outras linguagens de programação ou mesmo em várias plataformas/arquiteturas de agentes. Além disso, frequentemente conflitos de objetivos podem não garantir que os agentes ajam de forma cooperativa e coordenada [Huynh et al. 2004, Kaffille and Wirtz 2006].

Por fim, há ainda a dificuldade gerada pelas incertezas e pelo comportamento dinâmico que a mudança de agentes acarreta.

SMAA precisam lidar com problemas que não estão presentes em sistemas multiagentes fechados (SMA comum). Por exemplo, a migração de agentes entre modelos pode ocorrer em tempo de execução, e a motivação para essa migração de um agente de um sistema para outro pode ser diferente, geralmente de escolha do desenvolvedor, como falhas de execução, vontade própria ou algum gatilho. Além disso, podem ocorrer conflitos de interesse entre os novos agentes quando não projetados para trabalhar naquele conjunto [Uez 2018]. Embora o assunto seja conhecido há muito tempo [van Eijk et al. 1999], ainda há pesquisas na área, como em [Hendrickx and Martin 2016, Franceschelli and Frasca 2018, Houhamdi and Athamena 2020, Noh and Park 2020, Jiang et al. 2021, Franceschelli and Frasca 2021].

Quanto maior o grau de abertura de um SMAA, menor o número de mudanças em um modelo para receber ou enviar agentes [Jamroga et al. 2013]. Desta forma, um sistema multiagentes aberto perfeito não precisaria de transformações (0 passos) para acomodar novos componentes, enquanto, por outro lado, existem sistemas que exigiriam um redesenho completo (muitos passos) quando novos agentes chegassem [Jamroga et al. 2013].

Nesse contexto, o Docker pode auxiliar na resolução de parte dos problemas citados. Docker é uma ferramenta de código aberto que atua no desenvolvimento de aplicações na forma de contêineres. Os contêineres são uma abstração que une o código e suas dependências. Vários contêineres podem ser executados na mesma máquina, compartilhando recursos do kernel do sistema operacional, sendo cada um executado como processos isolados do espaço do usuário [Anderson 2015]. Além disso, o uso do Docker permite uma maior modularização do sistema, onde a implementação de cada módulo dentro de um contêiner possibilita a execução de diferentes ferramentas, em diferentes versões, em diferentes sistemas operacionais, permitindo assim uma fácil substituição de que o contêiner executa [Turnbull 2014].

Além disso, no sentido de diminuir a complexidade de transação dos agentes entre modelos, foram implementados módulos específicos de registro (**register**) e roteamento (**router**) de agentes. Estes módulos permitem com que a lógica de criação de agentes e os critérios de escolha de para onde os agentes vão quando saem dos modelos seja abstraída dos modelos. O isolamento destas lógicas permite com que diferentes decisões possam ser tomadas, como por exemplo estender uma estrutura simples de roteamento de agentes por algo mais complexo, como um modelo que retreina os agentes antes de voltarem aos modelos de simulação. Por fim, a modularização permite agregar ao sistema novos comportamentos que os modelos não previram em sua fase de concepção. Em geral, a abordagem visa avançar na generalização do SMAA, simplificando o processo de abertura.

Este artigo apresenta um modelo baseado em Docker para auxiliar no desenvolvimento de SMAA e facilitar a migração de agentes entre diferentes modelos que podem ser executados em cenários heterogêneos de hardware e software, diferentes ambientes de desenvolvimento e ferramentas para SMA ou outros sistemas operacionais.

Organizamos este trabalho da seguinte forma. Primeiramente, apresentamos trabalhos relacionados na Seção 2. Em seguida, a Seção 3 apresenta detalhes da abordagem proposta, como o fluxo de execução e detalhes de implementação. A seguir, a Seção 4 apresenta os resultados obtidos, enfatizando o cenário de simulação desenvolvido, que é de grande importância para verificar a viabilidade da implementação. Por fim, a Seção 5 apresenta as considerações finais do estudo.

2. Trabalhos Relacionados

A Tabela 1 sintetiza os estudos similares encontrados, indicando se é focado no aspecto de organização do SMAA, se utiliza algumas ferramentas DevOps, e se lida com mais de uma plataforma SMA. O aspecto do DevOps é importante porque é uma maneira de implementar várias partes dos sistemas que podem executar diferentes linguagens e sistemas operacionais, possibilitando o uso de todos os tipos de ferramentas SMA. O último aspecto é quantas plataformas/ferramentas SMA estão sendo usadas. Este aspecto é essencial para tornar a ferramenta o mais abrangente possível.

Dos 15 estudos analisados, o foco principal de 9 deles é lidar com os problemas organizacionais advindos da abertura do SMAA. Dos outros 6, 3 são dependentes da linguagem, não permitindo a utilização de ferramentas SMA/SMAA que rodam em um ambiente de desenvolvimento/utilização diferente, como o uso de outra ferramenta que utiliza outra linguagem. Por fim, além do nosso, apenas um estudo trata de mais de uma plataforma SMA. Este estudo trata apenas de SMA, não relacionado a SMAA, portanto não lida com o problema de transporte de agentes entre modelos. Além disso, este estudo conecta duas plataformas SMA que rodam na mesma linguagem (Java), não considerando outras ferramentas que possam utilizar outras linguagens.

Tabela 1. Características dos Trabalhos Relacionados.

Estudo	Ano	Relacionado à organização	Ferramenta de DevOps	Plataforma de SMA
[Jamroga et al. 2013]	2013	Sim	–	Única
[Dalpiaz et al. 2010]	2010	Sim	–	Única
[Demazeau and Costa 1996]	1996	Sim	–	Única
[Gonzalez-Palacios and Luck 2006]	2006	Sim	–	Única
[Artikis 2011]	2012	Sim	–	Única
[Paurobally et al. 2003]	2003	Sim	–	Única
[Singh and Chopra 2009]	2009	Sim	–	Única
[Hattab and Lejouad Chaari 2021]	2021	Sim	–	Única
[Houhamdi and Athamena 2020]	2020	Sim	–	Única
[Ramirez and Fasli 2017]	2017	Outro	–	Múltiplas
[Uez 2018]	2018	Outro	–	Única
[Dähling et al. 2021]	2021	Outro	Docker e Kubernetes	Única
[Pfeifer et al. 2021]	2021	Outro	Docker	Única
[Perles et al. 2018]	2018	Outro	–	Única
Nossa abordagem	2023	Outro	Docker	Múltiplas

Em conclusão, nossa abordagem contribui para o estado da arte em como o programador usará as plataformas. Por exemplo, alguns estudos propõem novas abordagens que exigem que o programador reconstrua seus modelos de acordo com as propostas. Em nossa arquitetura, os programadores não teriam que mudar todo o seu modelo. Em vez disso, eles adicionarão as estruturas necessárias aos seus modelos para se comunicarem com nossa arquitetura (providas pela própria arquitetura), permitindo uma transição mais fácil de um SMA fechado para um SMA aberto. Outro trabalho relacionado já utilizou

abordagens baseadas em contêiner/nuvem relacionadas ao sistema SMA, mostrando que essa abordagem é promissora. Além disso, alguns estudos testaram a comunicação entre agentes NetLogo e Jason.

3. Abordagem Proposta

A abordagem proposta utiliza como base contêineres no Docker. Cada imagem de contêiner é gerada a partir de um arquivo de descrição, denominado *DockerFile*, contendo informações sobre as estruturas necessárias no contêiner, como sistema operacional, linguagens de programação e código a ser executado. Além disso, o Docker possui um banco de imagens, o *Docker Hub Container Image Library*, que contém as imagens frequentemente utilizadas pelos desenvolvedores, ampliando a aplicabilidade da abordagem.

Os contêineres são responsáveis pela execução de cada bloco essencial da estrutura da arquitetura. Portanto, contêineres podem ser facilmente substituídos ou adicionados, tornando a arquitetura mais modularizada e adaptável a novos cenários não previstos na concepção, tornando-a mais robusta. Também lidamos com questões de segurança, criando redes virtuais separadas para que cada contêiner tenha uma visão parcial do sistema, limitando o acesso a códigos confidenciais.

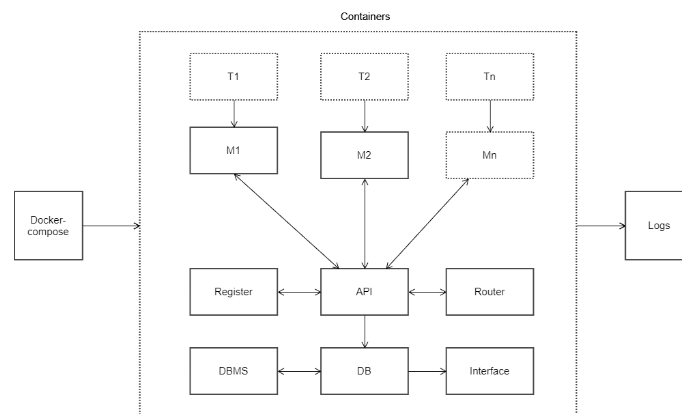

Figura 1. Organização da arquitetura proposta.

A Figura 1 apresenta uma descrição geral da arquitetura proposta em formato de diagrama de blocos. No bloco **Docker-compose**, temos a geração de imagens e serviços baseados no arquivo *docker-compose.yaml* e no *Dockerfile* de cada serviço. Cada *Dockerfile* contém a sequência de instruções necessárias para montar a imagem com todos os requisitos que cada serviço necessita. Então, com base nas imagens, o **Docker-compose** usa os parâmetros de cada serviço (nome do contêiner, portas expostas, rede, volumes, comando/arquivo a ser executado) e constrói os contêineres referentes. Por fim, após todos os serviços descritos no **Docker-compose** serem montados, eles podem ser executados usando apenas um comando.

Os blocos $\mathbf{M}_1$, $\mathbf{M}_2$, ..., $\mathbf{M}_n$ representam os contêineres responsáveis por executar os modelos. No arquivo de configuração principal, o designer do sistema pode escolher um parâmetro chamado *auto-run*. Quando definimos este parâmetro como *True*, os modelos são executados automaticamente quando o contêiner é montado. Se for *False*, devem ser implementados contêineres que irão determinar quando a execução dos modelos começa (contêineres de disparo $\mathrm{T}_1, \mathrm{T}_2, \ldots, \mathrm{T}_n$). Além disso, é necessário implementar as

comunicações entre a arquitetura e os modelos, como os gatilhos *triggers* de entrada/saída e as funções de envio/recebimento de agentes. Até este estágio de desenvolvimento, existem contêineres, gatilhos e funções prontos para oferecer suporte aos modelos NetLogo (contêiner Java) e JaCaMo (Java via Gradle).

$\mathbf{T}_1$, $\mathbf{T}_2$, ..., $\mathbf{T}_n$ são contêineres de gatilhos de execução, responsáveis por enviar uma mensagem aos modelos para iniciar sua execução. Usamos esses contêineres quando existe uma lógica mais robusta para iniciar a execução dos modelos. Por exemplo, eles podem executar códigos que leem um sensor, aguardar a medição ou executar um determinado modelo somente quando os agentes são atribuídos.

O bloco **API** (*Application Programming Interface*) é o contêiner responsável por gerenciar o acesso dos modelos ao banco de dados. Esse contêiner atua como um intermediário quando qualquer parte do sistema deve gravar, ler, atualizar ou excluir informações do banco de dados. Atualmente, a API é implementada em Python, usando o *framework* Flask [Grinberg 2018].

O bloco **Database** (DB) representa o contêiner responsável pelo banco de dados onde armazenamos, para cada modelo, todas as informações de entrada/saída de agentes para serem acessadas por outros contêineres que possam necessitar dessas informações. Além disso, esse contêiner é responsável por realizar as operações do banco de dados (ler, gravar, atualizar e excluir).

O bloco **Database Management System** (DBMS) representa o contêiner que facilita o acesso ao DB. Ele fornece uma interface da Web (exposta por padrão à máquina *host*) que pode importar/exportar conteúdo/arquivos SQL (*Structured Query Language*) e visualizar as informações em tempo real.

O bloco **Register** é um contêiner responsável por gerenciar a entrada de todos os agentes na plataforma. Todo agente deve possuir uma identificação para ser utilizado pela arquitetura, portanto as primeiras requisições de inserção de novos agentes vindas de qualquer modelo de contêineres necessitam de uma identificação única. Assim, o **Register** é responsável por gerar uma identificação única para cada agente, e então encaminhá-los de volta aos ambientes.

O bloco **Router** é responsável por receber todos os agentes que saíram de um determinado modelo, analisando e julgando a qual modelo enviar o agente. Esta etapa especifica os protocolos de entrada e saída do agente de diferentes contêineres/modelos. Este bloco é parte essencial da proposta, pois os modelos delegam ao roteador a tarefa de distribuição dos agentes entre os modelos. Quando o ambiente de simulação não compartilha (ou apenas parcialmente) informações sobre o mundo, o roteador pode lidar com diversos problemas relacionados à ausência dessas informações. O julgamento pode ocorrer de forma diferente, como analisar os agentes mais promissores, códigos de aprendizado de máquina e executar um novo modelo SMA que retreine os agentes.

O bloco **Interface** é um contêiner que recebe as informações de movimentação do agente através do sistema e gera relatórios expostos e formatados para visualizar métricas de execução da arquitetura. Na implementação atual, este contêiner possui um Apache Webserver rodando código PHP (*Hypertext Preprocessor*) que lê todas as informações dos agentes que já passaram pelo **Router** e gera um relatório com todos os agentes, seus atributos e o caminho percorrido por eles, em cada passagem. A forma geral do

relatório é por meio de um front-end HTML (*HyperText Markup Language*), CSS (*Cascading Style Sheet*) e código JS (JavaScript)), que é acessível através da exposição da porta do contêiner ao *host* (porta 80 por padrão). Cada tupla do relatório contém todos os atributos essenciais para cada interação do agente até o momento.

Por fim, o bloco **Logs** não é um contêiner e sim um módulo responsável por gerar *logs* das saídas de cada contêiner do sistema, *logs* regulares do docker (via comando docker log), tuplas de banco de dados via exportação SQL e impressões em arquivos txt (usando operadores de terminal ao executar o contêiner no docker-compose). Esses *logs* podem auxiliar na depuração de tarefas ou na geração de dados para análise de execução.

4. Resultados

Para o cenário de simulação proposto, construímos contêineres para dar suporte a duas diferentes ferramentas de desenvolvimento para SMA: NetLogo [Tisue and Wilensky 2004] e JaCaMo [Boissier et al. 2013]. Escolhemos um modelo da documentação de cada ferramenta: NetLogo's Open Sugarscape 2 Constant Growback [Epstein and Axtell 1996] e JaCaMo's Gold Miners [Bordini et al. 2006], com adaptações. Usamos três contêineres modelo, executando duas cópias isoladas do Open Sugarscape e uma do Gold Miners. O principal objetivo deste cenário de simulação é verificar a viabilidade da abordagem, permitindo que os agentes se movam livremente entre os modelos, mesmo que as duas ferramentas utilizem arquiteturas de agentes completamente diferentes.

4.1. NetLogo – Open Sugarscape (Modelo e Adaptações)

O modelo Open Sugarscape simula uma população com recursos limitados onde cada agente é representado como uma formiga que, para sobreviver, deve se deslocar pelo ambiente em busca de alimento (açúcar). Cada formiga nasce com uma quantidade de açúcar (de 5 a 25), metabolismo (de 1 a 4) e visão (de 1 a 6). O metabolismo define quanta energia a formiga perde ao se mover. Se a energia da formiga chega a zero, ela morre e sai do modelo. A visão determina quantas posições de distância a formiga pode ver o açúcar de sua posição atual.

A Figura 2(a) e a Figura 2(b) mostram, respectivamente, o código-fonte necessário para que o NetLogo possa receber e enviar dados da arquitetura. Nossa abordagem fornece as funções *check_new_agent_on_api* e *send_agent_to_api* ao usuário para definir os gatilhos para enviar (e quais informações devem ser enviadas) e receber um agente. Para este modelo específico, quando um agente morre (segundo o modelo original, um agente morre quando seu alimento chega a zero), o modelo envia este agente para a arquitetura e remove este agente da simulação.

Fizemos outro ajuste quando o agente volta para a simulação; em vez de usar suas últimas informações sobre comida, geramos esse parâmetro aleatoriamente, enquanto os outros parâmetros são os mesmos da última simulação. Fizemos esse ajuste por causa da maneira como a simulação original funciona. Se o agente sair da simulação quando sua comida for zero, e usarmos essa mesma informação novamente quando o agente retornar, isso faria o agente chegar à simulação com zero comida e morreria novamente.

Os dois únicos atributos adicionados aos agentes NetLogo são *agent_id* e *historic*. Usamos o *agent_id* guardar o id único do agente para a arquitetura, enquanto o id normal é usado apenas para a simulação do NetLogo. O atributo *historic* mostra o caminho de quais

```
to check_new_agent_on_api
  py:setup py:python
  py:run "from db_python_netlogo import receive_agent"
  let result py:runresult (word "receive_agent('" ("m1") "')")
  ifelse(length result > 0)
  [
    foreach result
    [
      x ->
      let tuple read-from-string item 1 x

      create-turtles 1
      [
        set agent_id item 0 x
        set sugar random-in-range 5 25
        set metabolism item 1 tuple
        set vision item 2 tuple
        set historic item 2 x

        set shape "circle"
        move-to one-of patches with [not any? other turtles-here]
        set vision-points []
        foreach (range 1 (vision + 1)) [ n ->
          set vision-points sentence vision-points (
            list (list 0 n) (list n 0) (list 0 (- n)) (list (- n) 0))
        ]
        run visualization
        print "Agent created"
      ]
    ]
  ]
  [
    print ("There is no new agent on DB")
  ]
end
```

(a)

```
to send_agent_to_api
  py:setup py:python
  py:run "from db_python_netlogo import send_function"
  ;//Example: "send_function('12', '[1 2 3]', '1-1')"

  let updated_historic ""
  ifelse (historic = "")
  [
    set updated_historic 1
  ]
  [
    set updated_historic (word "" (historic) "-1")
  ]
  let tuple []
  set tuple lput sugar tuple
  set tuple lput metabolism tuple
  set tuple lput vision tuple


  let result py:runresult (word
    "send_function('" (agent_id) "', '" tuple "', '" (updated_historic) "')"
  )

  print("Agent sent?")
  print(result)
end
```

(b)

Figura 2. Funções do NetLogo responsáveis por (a) receber e (b) enviar agentes entre a arquitetura e o modelo no NetLogo.

modelos os agentes passaram. Para adaptar qualquer modelo NetLogo à nossa plataforma, a única dependência necessária é a extensão do Py NetLogo. Precisamos desta extensão para enviar/receber informações da API através do código Python. Temos funções simples para isso, bastando o programador incluir essas funções e utilizá-las sempre que o modelo precisar enviar/receber informações sobre os agentes.

4.2. JaCaMo – Gold Miners (Modelo e a Adaptações)

Este modelo simula um conjunto de agentes representando mineradores cujo papel é navegar no ambiente. Por exemplo, quando um agente encontra um nó de ouro, ele interrompe seu objetivo atual, obtém o ouro e o traz de volta ao depósito central. No modelo original, há a representação dos agentes, do depósito, de barreiras do ambiente (posições não acessíveis aos agentes) e os nós de ouro.

Um trecho do código fonte (Java e ASL – *AgentSpeak Language*) necessário para que o JaCaMo possa enviar e receber dados da arquitetura é mostrado na Figura 3. A Figura 3(a) e a Figura 3(b) mostram o código Java para receber e deletar agentes. Figura 3(c) e Figura 3(d) mostram o código ASL para verificar novos agentes e remover agentes excluídos da arquitetura. A arquitetura apresenta dois agentes para envio/recebimento de informações: *killer_agent* e *check_new_agents*. O *killer_agent* espera uma mensagem para que um agente seja removido da simulação. Quando isso acontece, este agente salva todas as informações do agente a ser removido em um arquivo *.asl* (que será utilizado no caso deste agente voltar para a simulação) e então retira o agente da simulação.

A função *check_new_agents* verifica se um agente está esperando para entrar no modelo atual. Se sim, o agente é inserido no modelo. Este agente utiliza código Java e ASL para se comunicar com a API e inserir o agente na simulação. Também podemos adaptar este código para inserir informações iniciais extras ao agente, como crenças, objetivos ou foco nos artefatos do CArtAgO. Quando o modelo inclui um agente na simulação,

```java
System.out.println("Sugar: " + sugar);
System.out.println("Metabolism: " + metabolism);
System.out.println("Vision: " + vision);
char ch='"';
String bels = "agent_id("+agent_id+")";
bels = bels + ",path("+ ch + agent_path + ch + ")";
bels = bels + ", sugar("+sugar+"), metabolism("+metabolism+"), vision("+vision+")";
System.out.println("Agent bels: "+bels);
Settings s = new Settings();
s.addOption(Settings.INIT_BELS, bels);
s.addOption(Settings.INIT_GOALS,
    "jcm::focus_env_art([art_env(mining,m2view,default)],5)");
try {
    String asl_file_name = "list/"+agent_id+".asl";

    if (!Files.exists(Paths.get("src/agt/"+asl_file_name))){
        asl_file_name = "miner3.asl";
    }

    System.out.println("asl_file_name: "+asl_file_name);

    rs.createAgent(agent_id, asl_file_name, null, null, null, s, ts.getAg());
    rs.startAgent(agent_id);
    System.out.println("Agent created by custom file");
} catch (Exception e) {
    e.printStackTrace();
}
```

(a)

```java
String tuple_agent_id = String.valueOf(args[0]);
String tuple_data = "[" + String.valueOf(args[3]) +
" " + String.valueOf(args[4]) + " " + String.valueOf(args[5]) + "]";
String tuple_path = String.valueOf(args[1]) == "" ? "3" :
String.valueOf(args[1]).replace("\"", "")+"-3";

System.out.println("Tuple - agent_id: "+tuple_agent_id);
System.out.println("Tuple - data: "+tuple_data);
System.out.println("Tuple - path: "+tuple_path);

if (rs.killAgent(tuple_agent_id, null, 0)){
    System.out.println("Agent "+tuple_agent_id+" removed from the simulation...");

    String postUrl = "http://"+host+":5000/api/v1/resources/model_to_router";
    JSONObject json_obj = new JSONObject();
    json_obj.put("agent_id",tuple_agent_id);
    json_obj.put("data",tuple_data);
    json_obj.put("path",tuple_path);

    HttpClient httpClient = HttpClientBuilder.create().build();
    HttpPost post = new HttpPost(postUrl);
    StringEntity postingString = new StringEntity(json_obj.toString());
    post.setEntity(postingString);
    post.setHeader("Content-type", "application/json");
    HttpResponse  response = httpClient.execute(post);
} else {
    System.out.println("Error while removing agent from simulation...");
}
```

(b)

```
+!check_new_agent : true
<-
    .print("Checking and creating agent if it exists on API");
    mylib.check_new_agent;
    .wait(1000);
    !check_new_agent.
```

(c)

```
+kill(V0, V1, X, B0, B1, B2) : true
  <- .print("I've received a message to kill agent ",X);
     .print("Killing agent ",X);
     mylib.my_delete_ag(V0, V1, X, B0, B1, B2);
     .wait(2000);
     .print("This agent has being removed from the simulation: ",X).
```

(d)

Figura 3. Funções Java e ASL inseridos no código JaCaMo para prover as funções responsáveis por (a) receber, (b) excluir, (c) checar, e (d) remover agentes da/para a arquitetura.

o *check_new_agents* verifica se existe um arquivo *.asl* com o id do agente, ou seja, se este agente já esteve no modelo. Se o arquivo existir, o modelo cria o agente com suas informações anteriores (utilizando o arquivo *.asl* anterior). Caso contrário, o modelo cria o agente com um novo arquivo *.asl* usado como modelo.

Na atual adaptação do modelo original, utilizamos apenas um agente minerador, juntamente com os agentes da arquitetura (*check_new_agents* e *killer_agent*) e o agente líder. Assim, quando o agente entra na simulação, ele assume o controle do agente minerador. Este agente é inserido com base nas informações armazenadas anteriormente (se tiver), navega pelo mapa, pega uma unidade de ouro, traz de volta ao depósito e então sai da simulação (envia uma mensagem para o *killer_agent* para ser removido da simulação). Finalmente, quando o agente sai da simulação, o modelo verifica o próximo agente que entrará na simulação e assume o controle do agente minerador para que a simulação possa continuar sua execução.

Para usar o modelo JaCaMo na nossa plataforma, precisamos rodar o projeto via Gradle (que é a opção padrão), adaptando-o para isso [Boissier et al. 2022a]. As etapas necessárias estão apresentadas na documentação do GitHub do JaCaMo [Boissier et al. 2022b]. Depois, precisamos incluir duas bibliotecas no arquivo Gradle de compilação: JSON-simple (para lidar com informações no formato JSON) e cliente HTTP (para lidar com solicitações HTTP). O último passo é incluir os agentes *killer/check* (com seus arquivos ASL) que lidam com a API. Com essas etapas concluídas, o modelo está pronto para se comunicar com a API. Agora, o programador pode usar as funções do sistema para enviar e receber agentes entre a API e o modelo.

4.3. Cenário de Simulação

No cenário de simulação criado, os contêineres $\mathbf{M}_1$ e $\mathbf{M}_2$ executam simultaneamente duas réplicas do modelo Sugar Scape NetLogo, e o contêiner $\mathbf{M}_3$ executa o modelo Gold Mi-

ners, no JaCaMo. Todos os três modelos são executados indefinidamente e usam a opção *auto-run* (modelos executados automaticamente quando os contêineres são montados). Quando um agente morre, o modelo atual é responsável por enviá-lo para o **Router** (e depois para o **DB** via **API**).

No início da execução, ao invés de criar os agentes diretamente pelo modelo, os modelos $\mathbf{M}_1$ e $\mathbf{M}_2$ pedem ao contêiner **Register** para criar os agentes e receberem uma identificação única no SMAA. Na implementação atual do cenário de simulação, apenas os modelos NetLogo solicitam novos agentes. O modelo JaCaMo está recebendo e enviando agentes, mas não os criando na inicialização, embora seja possível.

Após a etapa inicial de criação e cadastro de todos os agentes, os modelos $\mathbf{M}_1$, $\mathbf{M}_2$ e $\mathbf{M}_3$ continuam sua execução, juntamente com o processamento do **Router**. Sempre que um agente sai de algum dos modelos, ele vai para o contêiner **DB** (via API). Uma vez que uma nova informação está no **DB**, o **Router** sabe que existem novos agentes a serem processados, então ele lê os agentes e os envia para um modelo de acordo com o tipo de roteamento (por padrão, aleatoriamente). Ambos os contêineres **Log** ou **Interface** podem acessar o fluxo desta informação.

É fundamental ressaltar que a maioria das adaptações não são substanciais, que impliquem a descaracterização os modelos originais para serem utilizados na arquitetura; ao contrário, as alterações são devidas porque esses modelos precisam estar preparados para receber agentes de fora do modelo. Fornecemos exemplos das funções de gatilho para enviar e receber informações para a arquitetura. Os usuários devem adicioná-lo ao seu modelo, adaptar as informações que desejam enviar ou receber e usar os gatilhos quando quiserem que o modelo envie ou receba agentes.

Por fim, é importante ressaltar que ambas plataformas têm acesso a todas as informações do agente. Por exemplo, o modelo do JaCaMo poderia acessar qualquer atributo do agente do NetLogo (açúcar, metabolismo e visão) para tornar algo útil na simulação. O contrário é verdadeiro: o NetLogo tem acesso ao arquivo *.asl* de cada agente, possibilitando o uso de algumas informações, como uma crença, para algo útil na simulação. Disponibilizamos vídeos e tutoriais na página do GitHub [de Lima and de Aguiar 2022] sobre o cenário testado que são essenciais para verificar os pontos fortes da arquitetura.

5. Conclusões

Este artigo apresentou uma proposta de arquitetura para auxiliar o desenvolvimento de SMAA utilizando Docker. Essa arquitetura permite a migração de agentes entre modelos que podem rodar em cenários heterogêneos. Além disso, a estrutura permite que o código seja executado dentro de contêineres que contenham a mesma estrutura de operação onde foram desenvolvidos, evitando problemas como programas que rodam na fase de desenvolvimento mas na fase de produção apresentam problemas.

O protótipo apresentado permitiu que os agentes se movimentassem livremente entre os modelos, compartilhando todas as informações do agente com os respectivos modelos e reduzindo a complexidade da adaptação do código, demonstrando que é suficiente, mas simples para entender melhor a abordagem proposta. A migração de agentes entre modelos ocorre em tempo de execução.

Em trabalhos futuros, queremos explorar novos gatilhos que fazem os agentes mudarem de modelo, como limites geográficos e modelos paralelos. Limites geográficos tratam de modelos onde, quando o agente atinge a borda do ambiente, ele passa para o outro modelo. Modelos paralelos são modelos onde o mesmo agente participa de mais de um modelo simultaneamente, mas cada modelo evolui atributos particulares do agente. Além disso, mesmo que a plataforma esteja atualmente rodando em uma máquina local, diversas plataformas (como o AWS Docker da Amazon) permitem que toda a estrutura que está sendo desenvolvida com base em Docker rode na nuvem, possibilitando a portabilidade de aplicações e ampliando o universo de aplicações na arquitetura. Finalmente, a implementação da arquitetura atual suporta duas plataformas de agentes relevantes, NetLogo e JaCaMo. No entanto, queremos ir além e oferecer suporte a outras plataformas de agentes, como JADE (baseado em Java) ou Mesa (baseado em Python).

Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001.

Referências

Anderson, C. (2015). Docker [software engineering]. *Ieee Software*, 32(3):102–c3.

Artikis, A. (2011). Dynamic specification of open agent systems. *Journal of Logic and Computation*, 22(6):1301–1334. doi: 10.1093/logcom/exr018.

Boissier, O., Bordini, R. H., Hübner, J. F., Ricci, A., and Santi, A. (2013). Multi-agent oriented programming with jacamo. *Science of Computer Programming*, 78(6):747–761. doi: 10.1016/j.scico.2011.10.004.

Boissier, O., Bordini, R. H., Hübner, J. F., Ricci, A., and Santi, A. (2022a). Jacamo's github page. Available in: `https://github.com/jacamo-lang/jacamo`. Accessed in 26 jul. 2022.

Boissier, O., Bordini, R. H., Hübner, J. F., Ricci, A., and Santi, A. (2022b). Jacamo's github page. Available in: `https://github.com/jacamo-lang/jacamo/blob/master/doc/install.adoc`. Accessed in 26 jul. 2022.

Bordini, R. H., Hübner, J. F., and Tralamazza, D. M. (2006). Using jason to implement a team of gold miners. In *International Workshop on Computational Logic in Multi-Agent Systems*, pages 304–313. Springer. doi: 10.1007/978-3-540-69619-3_18.

Dähling, S., Razik, L., and Monti, A. (2021). Enabling scalable and fault-tolerant multi-agent systems by utilizing cloud-native computing. *Autonomous Agents and Multi-Agent Systems*, 35(1):1–27. doi: 10.1007/s10458-020-09489-0.

Dalpiaz, F., Chopra, A. K., Giorgini, P., and Mylopoulos, J. (2010). Adaptation in open systems: Giving interaction its rightful place. In Parsons, J., Saeki, M., Shoval, P., Woo, C., and Wand, Y., editors, *Proc. of the Conceptual Modeling – ER 2010*, pages 31–45, Berlin, Heidelberg. Springer Berlin Heidelberg. doi: 10.1007/978-3-642-16373-9_3.

de Lima, G. L. and de Aguiar, M. S. (2022). Architecture's github page. Available in: `https://github.com/GustavoLLima/open_mas_docker_opt`. Accessed in 26 oct. 2022.

Demazeau, Y. and Costa, A. R. (1996). Populations and organizations in open multi-agent systems. In *Proceedings of the 1st National Symposium on Parallel and Distributed AI (PDAI'96)*, pages 1–13, India. University of Hyderabad.

Epstein, J. M. and Axtell, R. L. (1996). *Growing artificial societies: Social Science from the Bottom Up*. Complex Adaptive Systems. Bradford Books, Cambridge, MA.

Franceschelli, M. and Frasca, P. (2018). Proportional dynamic consensus in open multi-agent systems. In *Proc. of the IEEE Conference on Decision and Control (CDC)*, pages 900–905, Miami, FL, USA. IEEE. doi: 10.1109/CDC.2018.8619639.

Franceschelli, M. and Frasca, P. (2021). Stability of open multiagent systems and applications to dynamic consensus. *IEEE Transactions on Automatic Control*, 66(5):2326–2331. doi: 10.1109/TAC.2020.3009364.

Gonzalez-Palacios, J. and Luck, M. (2006). Towards compliance of agents in open multi-agent systems. In *Proc. of the International Workshop on Software Engineering for Large-Scale Multi-agent Systems*, pages 132–147, Shanghai, China. Springer. doi: 10.1007/978-3-540-73131-3_8.

Grinberg, M. (2018). *Flask web development: developing web applications with python*. O'Reilly Media, Inc. doi: 10.5555/2621997.

Hattab, S. and Lejouad Chaari, W. (2021). A generic model for representing openness in multi-agent systems. *The Knowledge Engineering Review*, 36:e3. doi: 10.1017/S0269888920000429.

Hendrickx, J. M. and Martin, S. (2016). Open multi-agent systems: Gossiping with deterministic arrivals and departures. In *Proc. of the 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 1094–1101, Monticello, IL, USA. IEEE. doi: 10.1109/ALLERTON.2016.7852357.

Houhamdi, Z. and Athamena, B. (2020). Collaborative team construction in open multi-agents system. In *Proc. of the 21st International Arab Conference on Information Technology (ACIT)*, pages 1–7, Giza, Egypt. IEEE. doi: 10.1109/ACIT50332.2020.9300116.

Huynh, T. D., Jennings, N. R., and Shadbolt, N. (2004). Developing an integrated trust and reputation model for open multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 13:119–154. doi: 10.1007/s10458-005-6825-4.

Jamroga, W., Meski, A., and Szreter, M. (2013). Modularity and openness in modeling multi-agent systems. *Electronic Proceedings in Theoretical Computer Science*, 119:224–239. doi: 10.4204/eptcs.119.19.

Jiang, W., Chen, Y., and Charalambous, T. (2021). Consensus of general linear multi-agent systems with heterogeneous input and communication delays. *IEEE Control Systems Letters*, 5(3):851–856. doi: 10.1109/LCSYS.2020.3006452.

Kaffille, S. and Wirtz, G. (2006). Modeling the static aspects of trust for open mas. In *2006 International Conference on Computational Intelligence for Modelling Control and Automation and International Conference on Intelligent Agents Web Technologies and International Commerce (CIMCA'06)*, pages 186–186, Australia. IEEE, IEEE. doi: 10.1109/CIMCA.2006.150.

Noh, S. and Park, J. (2020). System design for automation in multi-agent-based manufacturing systems. In *Proc. of 20th International Conference on Control, Automation and Systems (ICCAS)*, pages 986–990, Busan, Korea (South). IEEE. doi: 10.23919/ICCAS50221.2020.9268357.

Paurobally, S., Cunningham, J., and Jennings, N. R. (2003). Ensuring consistency in the joint beliefs of interacting agents. In *Proc. of 2nd International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2003, July 14-18, 2003, Melbourne, Victoria, Australia, Proceedings*, pages 662–669, Melbourne, Australia. ACM. doi: 10.1145/860575.860682.

Perles, A., Crasnier, F., and Georgé, J.-P. (2018). Amak - a framework for developing robust and open adaptive multi-agent systems. In Bajo, J., Corchado, J. M., Navarro Martínez, E. M., Osaba Icedo, E., Mathieu, P., Hoffa-Dabrowska, P., del Val, E., Giroux, S., Castro, A. J., Sánchez-Pi, N., Julián, V., Silveira, R. A., Fernández, A., Unland, R., and Fuentes-Fernández, R., editors, *Proc. of International Conference on Practical Applications of Agents and Multi-Agent Systems – Highlights of Practical Applications of Agents, Multi-Agent Systems, and Complexity: The PAAMS Collection*, pages 468–479, Cham. Springer International Publishing. doi: 10.1007/978-3-319-94779-2_40.

Pfeifer, V., Passini, W. F., Dorante, W. F., Guilherme, I. R., and Affonso, F. J. (2021). A multi-agent approach to monitor and manage container-based distributed systems. *IEEE Latin America Transactions*, 20(1):82–91. doi: 10.1109/TLA.2022.9662176.

Ramirez, W. A. L. and Fasli, M. (2017). Integrating netlogo and jason: a disaster-rescue simulation. In *2017 9th Computer Science and Electronic Engineering (CEEC)*, pages 213–218. IEEE. doi: 10.1109/CEEC.2017.8101627.

Singh, M. P. and Chopra, A. K. (2009). Programming multiagent systems without programming agents. In *Proc. of the International Workshop on Programming Multi-Agent Systems*, pages 1–14, Budapest, Hungary. Springer. doi: 10.1007/978-3-642-14843-9_1.

Tisue, S. and Wilensky, U. (2004). Netlogo: A simple environment for modeling complexity. In *International conference on complex systems*, volume 21, pages 16–21. Citeseer.

Turnbull, J. (2014). *The Docker Book: Containerization Is the New Virtualization*. James Turnbull, Melbourne, Australia.

Uez, D. M. (2018). *Open AEOlus: um método para especificação de sistemas multiagentes abertos*. Phd thesis, phd on automation and systems engineering, Federal University of Santa Catarina, Centro Tecnológico, Florianópolis. Available in: https://repositorio.ufsc.br/handle/123456789/205584.

van Eijk, R. M., de Boer, F. S., Van Der Hoek, W., and Meyer, J.-J. C. (1999). Open multiagent systems: Agent communication and integration. In *Proc. of the International Workshop on Agent Theories, Architectures, and Languages*, pages 218–232, Orlando, Florida, USA. Springer. doi: 10.1007/10719619_16.