Pade Documentation

Release 1.0

Lucas Melo

ago 06, 2019

Sumário

1 Sistemas Multiagentes para Python! 1
1.1 PADE é simples! 1
1.2 E fácil de instalar! 2
1.3 Funcionalidades 3

2 Guia do Usuário 5
2.1 Instalação 5
2.2 Linha de comando do Pade 6
2.3 Alô Mundo 7
2.4 Agentes Temporais 9
2.5 Enviando Mensagens 10
2.6 Recebendo Mensagens 12
2.7 Um momento Por Favor! 13
2.8 Enviando Objetos 14
2.9 Seleção de Mensagens 15
2.10 Interface Gráfica 16
2.11 Protocolos 16
2.12 Estrutura construtiva do PADE 30

CAPÍTULO **1**

Sistemas Multiagentes para Python!

PADE é um framework para desenvolvimento, execução e gerenciamento de sistemas multiagentes em ambientes de computação distribuída.

PADE é escrito 100% em Python e utiliza as bibliotecas do projeto Twisted para implementar a comunicação entre os nós da rede.

PADE é software livre, licenciado sob os termos da licença MIT, desenvolvido pelo Grupo de Redes Elétricas Inteligentes (GREI) do Departamento de Engenharia Elétrica da Universidade Federal do Ceará.

Qualquer um que queira contribuir com o projeto é convidado a baixar, executar, testar e enviar feedback a respeito das impressões tiradas da plataforma.

1.1 PADE é simples!

```
# agent_example_1.py
# A simple hello agent in PADE!

from pade.misc.utility import display_message, start_loop
from pade.core.agent import Agent
from pade.acl.aid import AID
from sys import argv
```

(continues on next page)

(continuação da página anterior)

```
class AgenteHelloWorld(Agent):
    def __init__(self, aid):
        super(AgenteHelloWorld, self).__init__(aid=aid)
        display_message(self.aid.localname, 'Hello World!')


if __name__ == '__main__':
    agents_per_process = 3
    c = 0
    agents = list()
    for i in range(agents_per_process):
        port = int(argv[1]) + c
        agent_name = 'agent_hello_{}@localhost:{}'.format(port, port)
        agente_hello = AgenteHelloWorld(AID(name=agent_name))
        agents.append(agente_hello)
        c += 1000

    start_loop(agents)
```

Neste arquivo exemplo (que está na pasta de exemplos no repositório PADE) é possível visualizar três sessões bem definidas.

A primeira contém as importações necessárias de classes que se encontram nos módulos do pade.

Na segunda uma classe que herda da classe pade Agent é definida com as pricipais atribuições do agente.

Na terceira parte que está encapsulada em uma estrutura if são realizados os procedimentos para lançar os agentes.

Se você quiser saber mais basta seguir a documentação aqui: *Alô Mundo*.

1.2 E fácil de instalar!

Para instalar o PADE basta executar o seguinte comando em um terminal linux:

```
$ pip install pade
$ pade start-runtime --port 20000 agent_example_1.py
```

Veja mais aqui: *Instalação*.

1.3 Funcionalidades

O PADE foi desenvolvido tendo em vista os requisitos para sistema de automação. PADE oferece os seguintes recursos em sua biblioteca para desenvolvimento de sistemas multi-agentes:

Orientação a Objetos Abstração para construção de agentes e seus comportamentos utilizando conceitos de orientação a objetos;

Ambiente de execução Módulo para inicialização do ambiente de execução de agentes, inteiramente em código Python;

Mensagens no padrão FIPA-ACL Módulo para construção e tratamento de mensagens no padrão FIPA-ACL;

Filtragem de Mensagens Módulo para filtragem de mensagens;

Protocolos FIPA Módulo para a implementação dos protocolos definidos pela FIPA;

Comportamentos Cíclicos e Temporais Módulo para implementação de comportamentos cíclicos e temporais;

Banco de Dados Módulo para interação com banco de dados;

Envio de Objetos Serializados Possibilidade de envio de objetos serializados como conteúdo das mensagens FIPA-ACL.

Além dessas funcionalidades, o PADE é de fácil instalação e configuração, multiplataforma, podendo ser instalado e utilizado em hardwares embarcados que executam sistema operacional Linux, como Raspberry Pi e BeagleBone Black, bem como sistema operacional Windows.

CAPÍTULO **2**

Guia do Usuário

2.1 Instalação

Instalar o PADE em seu computador, ou dispositivo embarcado é bem simples, basta que ele esteja conectado à internet!

2.1.1 Instalação via PIP

Para instalar o PADE via PIP basta digitar em um terminal:

```
$ pip install pade
```

Pronto! O Pade está instalado!

2.1.2 Instalação via GitHub

Se você quiser ter acesso ao fonte do PADE e instala-lo a partir do repositório oficial, basta digitar os seguintes comandos no terminal:

```
$ git clone https://github.com/grei-ufc/pade
$ cd pade
$ python setup.py install
```

Pronto o PADE está pronto para ser utilizado, faça um teste entrando na pasta de exemplos e digitando na linha de comandos:

```
pade start_runtime --port 20000 agent_example_1.py
```

2.1.3 Instalando o PADE em um ambiente virtual

Quando se trabalha com módulos Python é importante saber criar e manipular ambientes virtuais para gerenciar as dependências do projeto de uma maneira mais organizada. Aqui iremos mostrar como criar um ambiente virtual python, ativá-lo e utilizar o pip para instalar o PADE. Para uma visão mais detalhada sobre ambientes virtuais Python, acesse: Python Guide.

Primeiro você irá precisar ter o pacote virtualenv instalado em seu PC, instale-o digitando o comando:

```
$ pip install virtualenv
```

Após instalado o virtualenv é hora de criar um ambiente virtual, por meio do seguinte comando:

```
$ cd my_project_folder
$ virtualenv venv
```

Para ativar o ambiente virtual criado, digite o comando:

```
$ source venv/bin/activate
```

Agora basta instalar o pade, por meio do pip:

```
$ pip install pade
```

Nota: Você pode utilizar também a excelente distribuição python para computação científica Anaconda. Veja como aqui: AnacondaInc.

2.2 Linha de comando do Pade

Para facilitar algumas funções essenciais de gerenciamento de agentes, o framework Pade conta com uma interface de linha de comando (CLI). A estrutura dessa CLI é bem simples:

```
pade [command_name] --[options] [agent_file (.py) or pade_config_file (.json)]
```

Os comandos disponíveis são:

- start-runtime
- start-web-interface
- create-pade-db
- drop-pade-db

2.3 Alô Mundo

PADE foi implementado com um objetivo central: simplicidade!

Depois de ter o PADE instalado em seu computador fica bem simples começar a trabalhar.

Em uma pasta crie um arquivo chamado agent_example_1.py com o seu editor de texto preferido, e copie e cole o seguinte trecho de código dentro dele. Ou vá na pasta de exemplos do repositório pade no GitHub (*repositório pade <https://github.com/grei-ufc/pade>*) e procure o arquivo agent_example_1.py que já contém o código listado abaixo:

```
# Hello world in PADE!
#
# Criado por Lucas S Melo em 21 de julho de 2015 - Fortaleza, Ceará - Brasil

from pade.misc.utility import display_message, start_loop
from pade.core.agent import Agent
from pade.acl.aid import AID
from sys import argv

class AgenteHelloWorld(Agent):
    def __init__(self, aid):
        super(AgenteHelloWorld, self).__init__(aid=aid)
        display_message(self.aid.localname, 'Hello World!')


if __name__ == '__main__':

    agents_per_process = 3
    c = 0
    agents = list()
    for i in range(agents_per_process):
        port = int(argv[1]) + c
        agent_name = 'agent_hello_{}@localhost:{}'.format(port, port)
        agente_hello = AgenteHelloWorld(AID(name=agent_name))
        agents.append(agente_hello)
        c += 1000
```

(continues on next page)

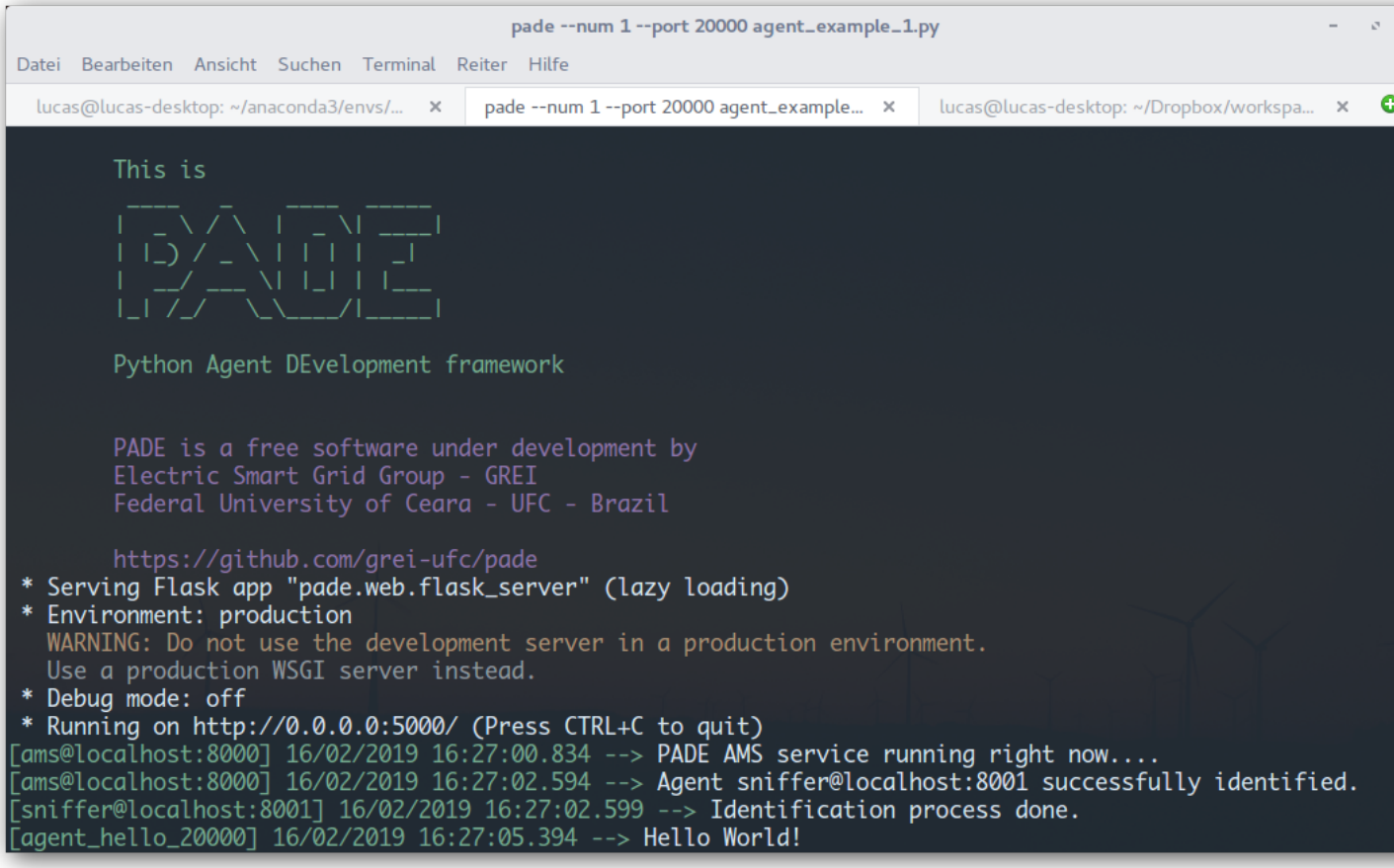
(continuação da página anterior)

```
    start_loop(agents)
```

Na linha de comando do terminal digite:

```
$ pade start_runtime --port 20000 agent_example_1.py
```

Pronto! Se tudo der certo você deve estar vendo na tela do seu terminal a splash screen do pade, parecida com esta:



Este já é um agente, mas não tem muita utilidade, não é mesmo! Executa apenas uma vez :(

Então como construir um agente que tenha seu comportamento executado de tempos em tempos?

2.4 Agentes Temporais

Em aplicações reais é comum que o comportamento do agente seja executado de tempos em tempos e não apenas uma vez, mas como fazer isso no Pade? :(

2.4.1 Execução de um agente temporal

Este é um exemplo de um agente que executa indefinidamente um comportamento a cada 1,0 segundos. O código fonte deste agente de comportamento temporal pode ser encontrado no diretório de exemplos no repositório do PADE no GitHub, no arquivo agent_example_2.py.

```
#!coding=utf-8
# Hello world temporal in Pade!

from pade.misc.utility import display_message, start_loop
from pade.core.agent import Agent
from pade.acl.aid import AID
from pade.behaviours.protocols import TimedBehaviour
from sys import argv

class ComportTemporal(TimedBehaviour):
    def __init__(self, agent, time):
        super(ComportTemporal, self).__init__(agent, time)

    def on_time(self):
        super(ComportTemporal, self).on_time()
        display_message(self.agent.aid.localname, 'Hello World!')


class AgenteHelloWorld(Agent):
    def __init__(self, aid):
        super(AgenteHelloWorld, self).__init__(aid=aid, debug=False)

        comp_temp = ComportTemporal(self, 1.0)

        self.behaviours.append(comp_temp)


if __name__ == '__main__':
    agents_per_process = 2
    c = 0
    agents = list()
    for i in range(agents_per_process):
```

(continues on next page)

(continuação da página anterior)

```
        port = int(argv[1]) + c
        agent_name = 'agent_hello_{}@localhost:{}'.format(port, port)
        agente_hello = AgenteHelloWorld(AID(name=agent_name))
        agents.append(agente_hello)
        c += 1000

    start_loop(agents)
```

2.5 Enviando Mensagens

Para enviar uma mensagem com o PADE é muito simples! As mensagens enviadas pelos agentes desenvolvidos com PADE seguem o padrão FIPA-ACL e têm os seguintes campos:

- *conversation-id:* identidade única de uma conversa;
- *performative:* rótulo da mensagem;
- *sender:* remetente da mensagem;
- *receivers:* destinatários da mensagem;
- *content:* conteúdo da mensagem;
- *protocol:* protocolo da mensagem;
- *language:* linguagem utilizada;
- *encoding:* codificação da mensagem;
- *ontology:* ontologia utilizada;
- *reply-with:* Expressão utilizada pelo agente de resposta a identificar a mensagem;
- *reply-by:* A referência a uma ação anterior em que a mensagem é uma resposta;
- *in-reply-to:* Data/hora indicando quando uma resposta deve ser recebida..

2.5.1 Mensagens FIPA-ACL no PADE

Uma mensagem FIPA-ACL pode ser montada no pade da seguinte forma:

```
from pade.acl.messages import ACLMessage, AID
message = ACLMessage(ACLMessage.INFORM)
message.set_protocol(ACLMessage.FIPA_REQUEST_PROTOCOL)
message.add_receiver(AID('agente_destino'))
message.set_content('Ola Agente')
```

2.5.2 Enviando uma mensagem com PADE

Uma vez que se está dentro de uma instância da classe *Agent()* a mensagem pode ser enviada, simplesmente utilizando o comando:

```
self.send(message)
```

2.5.3 Mensagem no padrão FIPA-ACL

Realizando o comando *print message* a mensagem no padão FIPA ACL será impressa na tela:

```
(inform
    :conversationID b2e806b8-50a0-11e5-b3b6-e8b1fc5c3cdf
    :receiver
        (set
            (agent-identifier
                :name agente_destino@localhost:51645
                :addresses
                    (sequence
                    localhost:51645
                    )
            )

        )
    :content "Ola Agente"
    :protocol fipa-request protocol
)
```

2.5.4 Mensagem no padrão XML

Mas também é possível obter a mensagem no formato XML por meio do comando *print message.as_xml()*

```
<?xml version="1.0" ?>
<ACLMessage date="01/09/2015 as 08:58:03:113891">
        <performative>inform</performative>
        <sender/>
        <receivers>
                <receiver>agente_destino@localhost:51645</receiver>
        </receivers>
        <reply-to/>
        <content>Ola Agente</content>
```

(continues on next page)

(continuação da página anterior)

```
        <language/>
        <enconding/>
        <ontology/>
        <protocol>fipa-request protocol</protocol>
        <conversationID>b2e806b8-50a0-11e5-b3b6-e8b1fc5c3cdf</conversationID>
        <reply-with/>
        <in-reply-to/>
        <reply-by/>
</ACLMessage>
```

2.6 Recebendo Mensagens

No Pade para que um agente possa receber mensagens, basta que o método react() seja implementado, dentro da classe que herda da classe *Agent()*.

2.6.1 Recebendo mensagens FIPA-ACL com PADE

No exemplo a seguir são implementados dois agentes distintos, o primeiro é o agente `remetente` modelado pela classe *Remetente()*, que tem o papel de enviar uma mensagem ao agente *destinatario* modelado pela classe *Destinatario*, que irá receber a mensagem enviada pelo agente *remetente* e por isso tem o método *react()* implementado.

```
from pade.misc.utility import display_message, start_loop
from pade.core.agent import Agent
from pade.acl.messages import ACLMessage
from pade.acl.aid import AID
from sys import argv


class Remetente(Agent):
    def __init__(self, aid):
        super(Remetente, self).__init__(aid=aid, debug=False)

    def on_start(self):
        display_message(self.aid.localname, 'Enviando Mensagem')
        message = ACLMessage(ACLMessage.INFORM)
        message.add_receiver(AID('destinatario'))
        message.set_content('Ola')
        self.send(message)

    def react(self, message):
```

(continues on next page)

(continuação da página anterior)

```
        pass


class Destinatario(Agent):
    def __init__(self, aid):
        super(Destinatario, self).__init__(aid=aid, debug=False)

    def react(self, message):
        display_message(self.aid.localname, 'Mensagem recebida')


if __name__ == '__main__':

    destinatario_agent = Destinatario(AID(name='destinatario'))
    agents.append(destinatario_agent)

    remetente_agent = Remetente(AID(name='remetente'))
    agents.append(remetente_agent)

    start_loop(agents)
```

2.7 Um momento Por Favor!

Com PADE é possível adiar a execução de um determinado trecho de código de forma bem simples! É só utilizar o método *call_later()* disponível na classe *Agent()*.

2.7.1 Como utilizar o método call_later()

Para utilizar *call_later()*, os seguintes parâmetros devem ser especificados: tempo de atraso, metodo que deve ser chamado após este tempo e argumento do metodo utilizado, *call_later(tempo, metodo, *args)*.

No código a seguir o *call_later()* é utilizado na classe *HelloAgent()* no método *on_start()* chamando o método *say_hello()* 10,0 segundos após a inicialização do agente:

```
from pade.misc.utility import display_message, start_loop, call_later
from pade.core.agent import Agent
from pade.acl.messages import ACLMessage
from pade.acl.aid import AID
from sys import argv

class HelloAgent(Agent):
```

(continues on next page)

(continuação da página anterior)

```
    def __init__(self, aid):
        super(HelloAgent, self).__init__(aid=aid, debug=False)

    def on_start(self):
        super().on_start()
        self.call_later(10.0, self.say_hello)

    def say_hello(self):
        display_message(self.aid.localname, "Hello, I\'m an agent!")


if __name__ == '__main__':

    agents = list()

    hello_agent = HelloAgent(AID(name='hello_agent'))
    agents.append(hello_agent)

    start_loop(agents)
```

2.8 Enviando Objetos

Nem sempre o que é preciso enviar para outros agentes pode ser representado por texto simpes não é mesmo!

Para enviar objetos encapsulados no content de mensagens FIPA-ACL com PADE basta utilizar o módulo nativo do Python *pickle*.

2.8.1 Enviando objetos serializados com pickle

Para enviar um objeto serializado com piclke basta seguir os passos:

```
import pickle
```

pickle é uma biblioteca para serialização de objetos, assim, para serializar um objeto qualquer, utilize *pickle.dumps()*, veja:

```
dados = {'nome' : 'agente_consumidor', 'porta' : 2004}
dados_serial = pickle.dumps(dados)
message.set_content(dados_serial)
```

Pronto! O objeto já pode ser enviado no conteúdo da mensagem.

2.8.2 Recebendo objetos serializados com pickle

Agora para receber o objeto, basta carregá-lo utilizando o comando:

```
dados_serial = message.content
dados = pickle.loads(dados_serial)
```

Simples assim ;)

2.9 Seleção de Mensagens

Com PADE é possível implementar filtros de mensagens de maneira simples e direta, por meio da classe Filtro:

```
from pade.acl.filters import Filter
```

2.9.1 Filtrando mensagens com o módulo filters

Por exemplo para a seguinte mensagem:

```
from pade.acl.messages import ACLMessage
from pade.acl.aid import AID

message = ACLMessage(ACLMessage.INFORM)
message.set_protocol(ACLMessage.FIPA_REQUEST_PROTOCOL)
message.set_sender(AID('remetente'))
message.add_receiver(AID('destinatario'))
```

Podemos criar o seguinte filtro:

```
from pade.acl.filters import Filter

f.performative = ACLMessage.REQUEST
```

Em uma sessão do interpretador Python é possível observar o efeito da aplicação do filtro sobre a mensagem:

```
>> f.filter(message)
False
```

Ajustando agora o filtro para outra condição:

```
f.performative = ACLMessage.INFORM
```

E aplicando o filtro novamente sobre a mensagem, obtemos um novo resultado:

```
>> f.filter(message)
True
```

2.10 Interface Gráfica

A interface web do Pade é desenvolvida utilizando o framework para desenvolvimento de aplicações web Flask.

A interface web do Pade está em processo de evolução com a adição de novas funcionalidades. Atualmente estão disponíveis as seguintes funcionalidades:

Em construção

2.11 Protocolos

O PADE tem suporte aos protocolos mais utilizados estabelecidos pela FIPA, são eles:

- *FIPA-Request*
- *FIPA-Contract-Net*
- *FIPA-Subscribe*

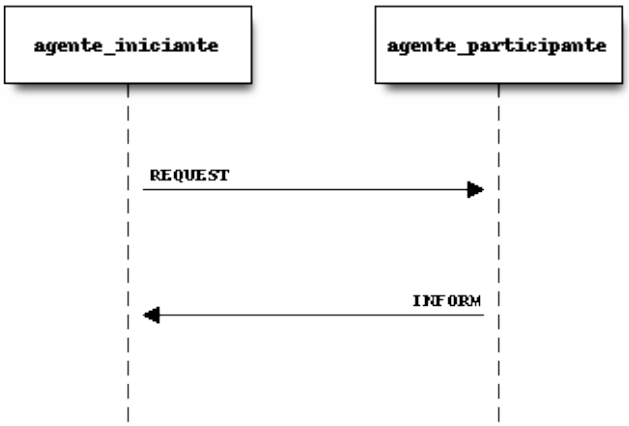
No PADE qualquer protocolo deve ser implementado como uma classe que extende a classe do protocolo desejado, por exemplo para implementar o protocolo FIPA-Request, deve ser implementada uma classe que implementa a herança da classe FipaRequestProtocol:

```
from pade.behaviours.protocols import FipaRequestProtocol


class ProtocoloDeRequisicao(FipaRequestProtocol):

    def __init__(self):
        super(ProtocoloDeRequisicao, self).__init__()
```

2.11.1 FIPA-Request

O protocolo FIPA-Request é o mais simples de se utilizar e constitui uma padronização do ato de requisitar alguma tarefa ou informação de um agente iniciador para um agente participante.

O diagrama de comunicação do protocolo FIPA-Request está mostrado na Figura abaixo:



Para exemplificar o protocolo FIPA-Request, iremos utilizar como exemplo a interação entre dois agentes, um agente relogio, que a cada um segundo exibe na tela a data e o horário atuais, mas com um problema, o agente relogio não sabe calcular nem a data, e muito menos o horário atual. Assim, ele precisa requisitar estas informações do agente horario que consegue calcular estas informações.

Dessa forma, será utilizado o protocolo FIPA-Request, para que estas informações sejam trocadas entre os dois agentes, sendo o agente relógio o iniciante, no processo de requisição e o agente horário, o participante, segue o código do exemplo, que corresponde ao arquivo exemplo agent_example_3.py:

```
from pade.misc.utility import display_message, start_loop
from pade.core.agent import Agent
from pade.acl.messages import ACLMessage
from pade.acl.aid import AID
from pade.behaviours.protocols import FipaRequestProtocol
from pade.behaviours.protocols import TimedBehaviour

from datetime import datetime
from sys import argv

class CompRequest(FipaRequestProtocol):
    """FIPA Request Behaviour of the Time agent.
    """
    def __init__(self, agent):
        super(CompRequest, self).__init__(agent=agent,
                                          message=None,
                                          is_initiator=False)

    def handle_request(self, message):
```

(continues on next page)

(continuação da página anterior)

```
        super(CompRequest, self).handle_request(message)
        display_message(self.agent.aid.localname, 'request message received')
        now = datetime.now()
        reply = message.create_reply()
        reply.set_performative(ACLMessage.INFORM)
        reply.set_content(now.strftime('%d/%m/%Y - %H:%M:%S'))
        self.agent.send(reply)


class CompRequest2(FipaRequestProtocol):
    """FIPA Request Behaviour of the Clock agent.
    """
    def __init__(self, agent, message):
        super(CompRequest2, self).__init__(agent=agent,
                                           message=message,
                                           is_initiator=True)

    def handle_inform(self, message):
        display_message(self.agent.aid.localname, message.content)


class ComportTemporal(TimedBehaviour):
    """Timed Behaviour of the Clock agent"""
    def __init__(self, agent, time, message):
        super(ComportTemporal, self).__init__(agent, time)
        self.message = message

    def on_time(self):
        super(ComportTemporal, self).on_time()
        self.agent.send(self.message)


class TimeAgent(Agent):
    """Class that defines the Time agent."""
    def __init__(self, aid):
        super(TimeAgent, self).__init__(aid=aid, debug=False)

        self.comport_request = CompRequest(self)

        self.behaviours.append(self.comport_request)


class ClockAgent(Agent):
    """Class thet defines the Clock agent."""
    def __init__(self, aid, time_agent_name):
```

(continues on next page)

(continuação da página anterior)

```
        super(ClockAgent, self).__init__(aid=aid)

        # message that requests time of Time agent.
        message = ACLMessage(ACLMessage.REQUEST)
        message.set_protocol(ACLMessage.FIPA_REQUEST_PROTOCOL)
        message.add_receiver(AID(name=time_agent_name))
        message.set_content('time')

        self.comport_request = CompRequest2(self, message)
        self.comport_temp = ComportTemporal(self, 8.0, message)

        self.behaviours.append(self.comport_request)
        self.behaviours.append(self.comport_temp)


if __name__ == '__main__':

    agents_per_process = 1
    c = 0
    agents = list()
    for i in range(agents_per_process):
        port = int(argv[1]) + c
        time_agent_name = 'agent_time_{}@localhost:{}'.format(port, port)
        time_agent = TimeAgent(AID(name=time_agent_name))
        agents.append(time_agent)

        clock_agent_name = 'agent_clock_{}@localhost:{}'.format(port - 10000, port -␣
↪10000)
        clock_agent = ClockAgent(AID(name=clock_agent_name), time_agent_name)
        agents.append(clock_agent)

        c += 500

    start_loop(agents)
```

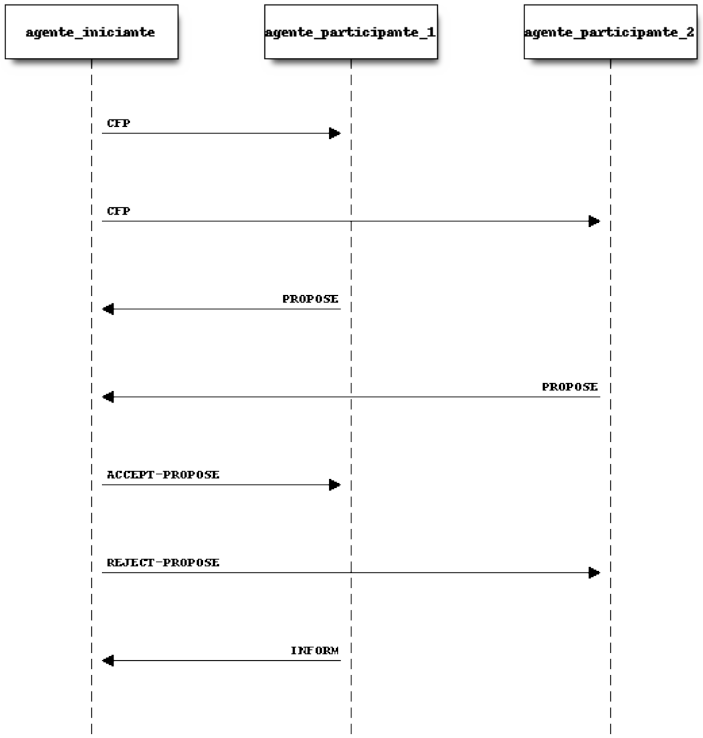
Na primeira parte do código são importadas todos módulos e classes necessários à construção dos agentes, logo em seguida as classes que implementam o protocolo são definidas, as classes ComptRequest e ComptRequest2 que serão associadas aos comportamentos dos agentes horario e relogio, respectivamente. Como o agente relogio precisa, a cada segundo enviar requisição ao agente horario, então também deve ser associado a este agente um comportamento temporal, definido na classe ComportTemporal que envia uma solicitação ao agente horario, a cada segundo.

Em seguida, os agente propriamente ditos, são definidos nas classes AgenteHoraio e AgenteRelogio que estendem a classe Agent, nessas classes é que os comportamentos e protocolos são associados a cada agente.

Na ultima parte do código, é definida uma função main que indica a localização do agente ams, instancia os agentes e dá inicio ao loop de execução.

2.11.2 FIPA-Contract-Net

O protocolo FIPA-Contract-Net é utilizado para situações onde é necessário realizar algum tipo de negociação entre agentes. Da mesma forma que no protocolo FIPA-Request, no protocolo FIPA-ContractNet existem dois tipos de agentes, um agente que inicia a negociação, ou agente iniciante, fazendo solicitação de propostas e um ou mais agentes que participam da negociação, ou agentes pasticipantes, que repondem às solicitações de propostas do agente iniciante. Veja:



Um exemplo de utilização do protocolo FIPA-ContractNet na negociação é mostrado abaixo, com a solicitação de um agente iniciante por potência elétrica a outros dois agentes participantes. Este código corresponde ao arquivo exemplo agent_example_4.py:

```
from pade.misc.utility import display_message, start_loop
from pade.core.agent import Agent
from pade.acl.aid import AID
from pade.acl.messages import ACLMessage
from pade.behaviours.protocols import FipaContractNetProtocol
from sys import argv
```

(continues on next page)

(continuação da página anterior)

```
from random import uniform

class CompContNet1(FipaContractNetProtocol):
    '''CompContNet1

       Initial FIPA-ContractNet Behaviour that sends CFP messages
       to other feeder agents asking for restoration proposals.
       This behaviour also analyzes the proposals and selects the
       one it judges to be the best.'''

    def __init__(self, agent, message):
        super(CompContNet1, self).__init__(
            agent=agent, message=message, is_initiator=True)
        self.cfp = message

    def handle_all_proposes(self, proposes):
        """
        """

        super(CompContNet1, self).handle_all_proposes(proposes)

        best_proposer = None
        higher_power = 0.0
        other_proposers = list()
        display_message(self.agent.aid.name, 'Analyzing proposals...')

        i = 1

        # logic to select proposals by the higher available power.
        for message in proposes:
            content = message.content
            power = float(content)
            display_message(self.agent.aid.name,
                            'Analyzing proposal {i}'.format(i=i))
            display_message(self.agent.aid.name,
                            'Power Offered: {pot}'.format(pot=power))
            i += 1
            if power > higher_power:
                if best_proposer is not None:
                    other_proposers.append(best_proposer)

                higher_power = power
                best_proposer = message.sender
            else:
                other_proposers.append(message.sender)
```

(continues on next page)

(continuação da página anterior)

```
        display_message(self.agent.aid.name,
                        'The best proposal was: {pot} VA'.format(
                            pot=higher_power))

        if other_proposers != []:
            display_message(self.agent.aid.name,
                            'Sending REJECT_PROPOSAL answers...')
            answer = ACLMessage(ACLMessage.REJECT_PROPOSAL)
            answer.set_protocol(ACLMessage.FIPA_CONTRACT_NET_PROTOCOL)
            answer.set_content('')
            for agent in other_proposers:
                answer.add_receiver(agent)

            self.agent.send(answer)

        if best_proposer is not None:
            display_message(self.agent.aid.name,
                            'Sending ACCEPT_PROPOSAL answer...')

            answer = ACLMessage(ACLMessage.ACCEPT_PROPOSAL)
            answer.set_protocol(ACLMessage.FIPA_CONTRACT_NET_PROTOCOL)
            answer.set_content('OK')
            answer.add_receiver(best_proposer)
            self.agent.send(answer)

    def handle_inform(self, message):
        """
        """
        super(CompContNet1, self).handle_inform(message)

        display_message(self.agent.aid.name, 'INFORM message received')

    def handle_refuse(self, message):
        """
        """
        super(CompContNet1, self).handle_refuse(message)

        display_message(self.agent.aid.name, 'REFUSE message received')

    def handle_propose(self, message):
        """
        """
        super(CompContNet1, self).handle_propose(message)
```

(continues on next page)

(continuação da página anterior)

```
        display_message(self.agent.aid.name, 'PROPOSE message received')


class CompContNet2(FipaContractNetProtocol):
    '''CompContNet2

       FIPA-ContractNet Participant Behaviour that runs when an agent
       receives a CFP message. A proposal is sent and if it is selected,
       the restrictions are analized to enable the restoration.'''

    def __init__(self, agent):
        super(CompContNet2, self).__init__(agent=agent,
                                           message=None,
                                           is_initiator=False)

    def handle_cfp(self, message):
        """
        """
        self.agent.call_later(1.0, self._handle_cfp, message)

    def _handle_cfp(self, message):
        """
        """
        super(CompContNet2, self).handle_cfp(message)
        self.message = message

        display_message(self.agent.aid.name, 'CFP message received')

        answer = self.message.create_reply()
        answer.set_performative(ACLMessage.PROPOSE)
        answer.set_content(str(self.agent.pot_disp))
        self.agent.send(answer)

    def handle_reject_propose(self, message):
        """
        """
        super(CompContNet2, self).handle_reject_propose(message)

        display_message(self.agent.aid.name,
                        'REJECT_PROPOSAL message received')

    def handle_accept_propose(self, message):
        """
        """
        super(CompContNet2, self).handle_accept_propose(message)
```

(continues on next page)

(continuação da página anterior)

```
        display_message(self.agent.aid.name,
                        'ACCEPT_PROPOSE message received')

        answer = message.create_reply()
        answer.set_performative(ACLMessage.INFORM)
        answer.set_content('OK')
        self.agent.send(answer)


class AgentInitiator(Agent):

    def __init__(self, aid, participants):
        super(AgentInitiator, self).__init__(aid=aid, debug=False)

        message = ACLMessage(ACLMessage.CFP)
        message.set_protocol(ACLMessage.FIPA_CONTRACT_NET_PROTOCOL)
        message.set_content('60.0')

        for participant in participants:
            message.add_receiver(AID(name=participant))

        self.call_later(8.0, self.launch_contract_net_protocol, message)

    def launch_contract_net_protocol(self, message):
        comp = CompContNet1(self, message)
        self.behaviours.append(comp)
        comp.on_start()


class AgentParticipant(Agent):

    def __init__(self, aid, pot_disp):
        super(AgentParticipant, self).__init__(aid=aid, debug=False)

        self.pot_disp = pot_disp

        comp = CompContNet2(self)

        self.behaviours.append(comp)

if __name__ == "__main__":
    agents_per_process = 2
    c = 0
    agents = list()
```

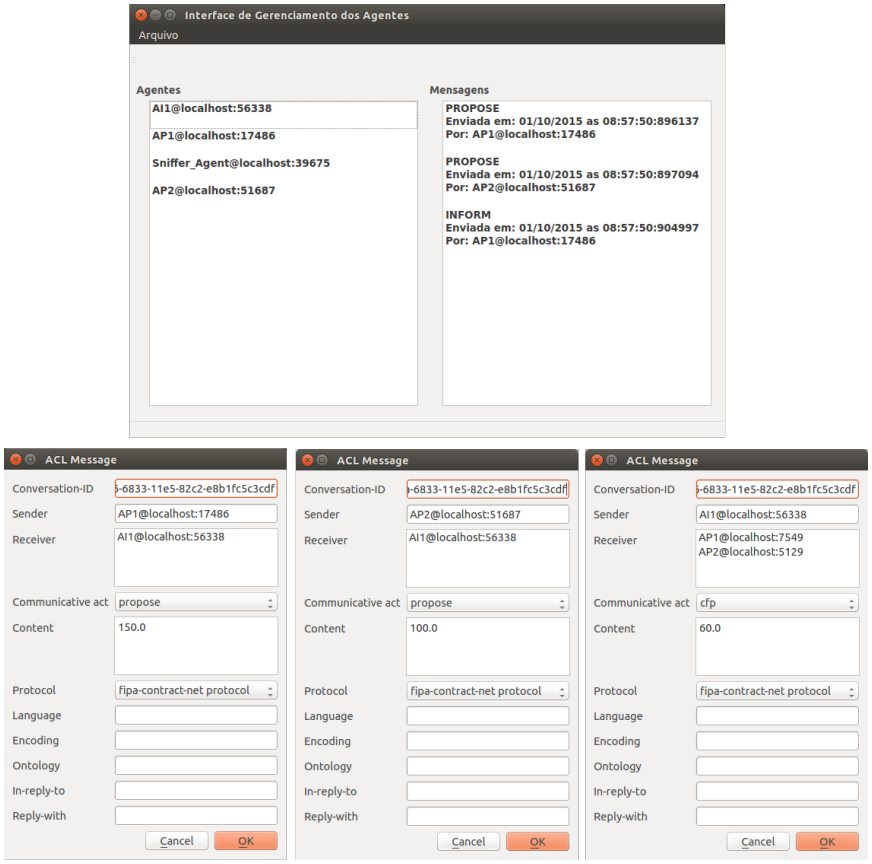
(continues on next page)

(continuação da página anterior)

```
    for i in range(agents_per_process):
        port = int(argv[1]) + c
        k = 10000
        participants = list()

        agent_name = 'agent_participant_{}@localhost:{}'.format(port - k, port - k)
        participants.append(agent_name)
        agente_part_1 = AgentParticipant(AID(name=agent_name), uniform(100.0, 500.0))
        agents.append(agente_part_1)

        agent_name = 'agent_participant_{}@localhost:{}'.format(port + k, port + k)
        participants.append(agent_name)
        agente_part_2 = AgentParticipant(AID(name=agent_name), uniform(100.0, 500.0))
        agents.append(agente_part_2)

        agent_name = 'agent_initiator_{}@localhost:{}'.format(port, port)
        agente_init_1 = AgentInitiator(AID(name=agent_name), participants)
        agents.append(agente_init_1)

        c += 1000

    start_loop(agents)
```

O código que implementa os agentes que se comunicam utilizando o protocolo FIPA-ContractNet, definine as duas classes do protocolo, a primeira implementa o comportamento do agente Iniciante (CompContNet1) e a segunda implementa o comportamento do agente participante (CompContNet2). Note que para a classe iniciante é necessário que uma mensagem do tipo CFP (call for proposes) seja montada e o método on_start() seja chamado, isso é feito dentro da classe que implementa os agente iniciante, AgenteIniciante(), já a classe AgenteParticipante(), implementa os agentes que participarão da negociação como propositores.

É possível observar as mensagens da negociação na intergace gráfica do PADE, veja:



2.11.3 FIPA-Subscribe

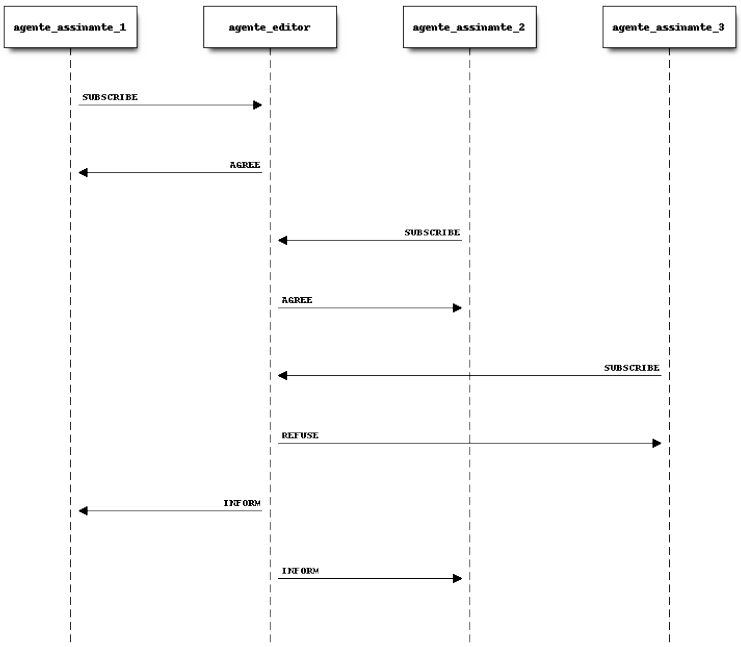
O protocolo FIPA-Subscribe, implementa o comportamento de editor-assinante, que consiste na presença de um agente editor que pode aceitar a associação de outros agentes interessados, agentes assinantes, em algum tipo de informação que este agente possua, assinando a informação e recebendo mensagem sempre que esta informação for disponibilizada pelo agente editor. Veja:

Para assinar a informação o agente precisa enviar uma mensagem SUSBCRIBE para o agente editor. Que por sua vez pode aceitar ou recusar a assinatura (AGREE/REFUSE). Quando uma informação é atualizada, então o editor publica esta informação para todos os seus assinantes, enviando-os mensagens INFORM.

O código que implementa um agente editor e dois agentes assinantes utilizando PADE pode ser visualizado abaixo e corresponde ao arquivo exemplo agent_example_5.py:

```
from pade.misc.utility import display_message, start_loop
from pade.core.agent import Agent
from pade.acl.aid import AID
from pade.acl.messages import ACLMessage
from pade.behaviours.protocols import FipaSubscribeProtocol, TimedBehaviour
```

(continues on next page)



(continuação da página anterior)

```
from sys import argv
import random

class SubscriberProtocol(FipaSubscribeProtocol):

    def __init__(self, agent, message):
        super(SubscriberProtocol, self).__init__(agent,
                                                 message,
                                                 is_initiator=True)

    def handle_agree(self, message):
        display_message(self.agent.aid.name, message.content)

    def handle_inform(self, message):
        display_message(self.agent.aid.name, message.content)


class PublisherProtocol(FipaSubscribeProtocol):

    def __init__(self, agent):
        super(PublisherProtocol, self).__init__(agent,
                                                message=None,
                                                is_initiator=False)

    def handle_subscribe(self, message):
```

(continues on next page)

(continuação da página anterior)

```
        self.register(message.sender)
        display_message(self.agent.aid.name, message.content)
        resposta = message.create_reply()
        resposta.set_performative(ACLMessage.AGREE)
        resposta.set_content('Subscribe message accepted')
        self.agent.send(resposta)

    def handle_cancel(self, message):
        self.deregister(self, message.sender)
        display_message(self.agent.aid.name, message.content)

    def notify(self, message):
        super(PublisherProtocol, self).notify(message)


class Time(TimedBehaviour):

    def __init__(self, agent, notify):
        super(Time, self).__init__(agent, 1)
        self.notify = notify
        self.inc = 0

    def on_time(self):
        super(Time, self).on_time()
        message = ACLMessage(ACLMessage.INFORM)
        message.set_protocol(ACLMessage.FIPA_SUBSCRIBE_PROTOCOL)
        message.set_content(str(random.random()))
        self.notify(message)
        self.inc += 0.1


class AgentSubscriber(Agent):

    def __init__(self, aid, message):
        super(AgentSubscriber, self).__init__(aid)

        self.call_later(8.0, self.launch_subscriber_protocol, message)

    def launch_subscriber_protocol(self, message):
        self.protocol = SubscriberProtocol(self, message)
        self.behaviours.append(self.protocol)
        self.protocol.on_start()


class AgentPublisher(Agent):
```

(continues on next page)

(continuação da página anterior)

```
    def __init__(self, aid):
        super(AgentPublisher, self).__init__(aid)

        self.protocol = PublisherProtocol(self)
        self.timed = Time(self, self.protocol.notify)

        self.behaviours.append(self.protocol)
        self.behaviours.append(self.timed)

if __name__ == '__main__':

    agents_per_process = 2
    c = 0
    agents = list()
    for i in range(agents_per_process):
        port = int(argv[1]) + c
        k = 10000
        participants = list()

        agent_name = 'agent_publisher_{}@localhost:{}'.format(port, port)
        participants.append(agent_name)
        agent_pub_1 = AgentPublisher(AID(name=agent_name))
        agents.append(agent_pub_1)

        msg = ACLMessage(ACLMessage.SUBSCRIBE)
        msg.set_protocol(ACLMessage.FIPA_SUBSCRIBE_PROTOCOL)
        msg.set_content('Subscription request')
        msg.add_receiver(agent_pub_1.aid)

        agent_name = 'agent_subscriber_{}@localhost:{}'.format(port + k, port + k)
        participants.append(agent_name)
        agent_sub_1 = AgentSubscriber(AID(name=agent_name), msg)
        agents.append(agent_sub_1)

        agent_name = 'agent_subscriber_{}@localhost:{}'.format(port - k, port - k)
        agent_sub_2 = AgentSubscriber(AID(name=agent_name), msg)
        agents.append(agent_sub_2)

        c += 1000

    start_loop(agents)
```

2.12 Estrutura construtiva do PADE

Este guia está sendo desenvolvido com o objetivo de auxiliar o desenvolvimento do PADE, fornecendo compreenssão a respeito da estrutura construtiva do PADE.

PADE passará por alterações profundas entre suas versões 1.0 e 2.0. Como a versão 1.0 já está em sua versão estável tendo sido utilizada em diversas aplicações com exito, e não sendo mais objeto de desenvolvimento, essa documentação irá focar nas nova estrutura do PADE que, como mencionado, passou por profundas alterações internas.

2.12.1 Características do PADE 1.0

Apesar de muitas alterações estarem sendo implementadas para a versão 2.0, é necessário verificar e revisar alguns aspectos da versão 1.0 que fornecem a base do desenvolvimento do PADE e não deverão ser alterados.

Primeiramente é necessário mencionar que o PADE é desenvolvido utilizando o framework twisted, que segundo descrição encontrada no site do twisted:

> Twisted is an event-driven networking engine written in Python and licensed under the open source

Twisted é um framework para implementação de aplicações distribuídas que utilizem protocolos padrões, como http, ftp, ssh, ou protocolos personalizados. PADE faz uso do twisted por meio da segunda opção, implementando protocolos personalizados, tanto para troca de mensagens de controle interno, quanto para fornecer uma interface aos protocolos descritos pelo padão FIPA.

2.12.2 Pacote core

A essencia do desenvolvimento do Twisted está contida no pacote *core*. O pacote core, por sua vez possui quatro módulos:

- __init__.py: torna uma pasta um pacote;
- agent.py: contém as classes que implementam as funcionalidades necessárias para a definição dos agentes e para possibilitar registro no AMS, verificação de conexão e troca de mensagens;
- new_ams.py: implementa o agente AMS e seus comportamentos;
- peer.py: implementa comportamentos básicos do protocolo twisted para troca de mensagens.

Sendo assim, como PADE é desenvolvido com Twisted, precisa obdecer sua estrutura de classes. No Twisted, duas classes são essenciais:

1. Uma classe para implementar o protocolo propriamente dito, que trata as mensagens recebidas por um ponto comunicante e envia as mensagens para outro ponto comunicante.
2. Uma classe para implementar o protocolo definido na primeira classe, essa classe é chamada de classe Factory.

Na versão 1.0 do pade, a definição dessas classes estavam presentes no módulo agent.py, mas por uma questão de organização do código, a classe que implementa o protocolo foi transferida para o módulo peer.py e chamasse agora **PeerProtocol** extendendo a classe do twisted **LineReceiver**.

Módulo agente.py

No módulo agent.py estão presentes as seguintes classes:

- **AgentProtocol**: Essa classe herda a classe PeerProtocol que é importada do módulo peer.py e extende alguns de seus métodos, tais como connectionMade, connectionLost, send_message e lineReceived.
- **AgentFactory**: Essa classe herda a classe ClientFactory do módulo protocol importado do Twisted, implementando o método buildProtocol, clientConnectionFailed e clientConnectionLost, muito importantes para funcionamento do sistema multiagente. Nessa classe temos importantes atributos, tais como messages, react, ams_aid, on_start e table.
- **Agent_**: Essa classe estabelece as funcionalidades essenciais de um agente como: conexão com o AMS; configurações iniciais; envio de mensagens e adição de comportamentos. Os atributos presentes nesta classe são: aid, debug, ams, agentInstance, behaviours, system_behaviours.

Os metodos presentes nesta classe são: send: utilizado para o envio de mensagens entre agentes call_later: utilizado para executar metodos apos periodo especificado; send_to_all: envia mensagem a todos os agentes registrados na tabela do agente; add_all: adiciona todos os agentes presentes na tabela dos agentes; on_start: a ser utilizado na implementação dos comportamentos iniciais; react: a ser utlizado na implementação dos comportamentos dos agentes quando recebem uma mensagem.

Para a descrição das demais classes cabe aqui uma breve explicação. Na versão 1.0 as mensagens de controle internas do PADE eram enviadas sem nenhuma padronização, aproveitando somente a infraestrutura que o twisted monta para executar suas aplicações distribuídas. Isso deixava o código sujo, commuitos ifs e elses, dificultando seu entendimento e manutenabilidade.

Uma das alterações mais profundas na versão 2.0 é que toda essa estrutura de troca de mensagens internas agora faz uso e obedece as classes que implementam os protocolos de comunicação FIPA. Isso deixou o código mais organizado e criou a oportunidade de implementar algumas melhorias na lógica de execução. Sendo assim, o comportamento

de atualização das tabelas que cada agente tem com os endereços dos agentes presentes na plataforma foi implementado com um protocolo FIPASubscribe que realiza o paradigma de comunicação editor-assinante, em que o agente AMS é o agente editor e todos os outros agentes da plataforma são os assinantes. Também o comportamento de verificação se o agente está ativo ou não foi modelado como um protocolo FIPA request, em que o AMS solicita a cada um dos agentes sua resposta, caso a resposta não venha, significa que o agente não está mais ativo.

- **SubscribeBehaviour**: Classe que implementa o lado subscriber do protocolo Publisher/Subscriber para atualização das tabelas de agentes.
- **CompConnection**: Classe que implementa o comportamento de verificação de atividade dos agentes.
- **Agent**: Pode ser considerada a classe mais importante do módulo agent.py pois é extendendo esta classe que um agente é definido no PADE. Dois métodos são definidos nesta classe: o primeiro é o método de inicialização padão de classes em Python, onde são instanciadas as classes dos protocolos padrões do PADE que irão realizar a identificação do agente junto ao AMS e também inscrever este agente como assinante do AMS em seu comportamento de verificação de atividade dos agente (se o agente está ou não em funcionamento). O segundo método, não menos importante, envia todas as mensagens que recebe ao AMS, para que sejam armazenas em um banco de dados e estejam disponíveis para consulta pelo administrador do sistema multiagente por meio da interface web ou do acesso direto ao banco de dados.

Módulo new_ams.py

Este módulo é uma adição do PADE 2.0. Na versão 1.0 o comportamento do AMS estava contido dentro do módulo ams.py que implementava um protocolo com as classes do Twisted exclusivamente para o agente AMS e também para o agente Sniffer que deixou de existir na versão 2.0, pois o comportamento de envio de mensagens para armazenamento já está embutido no comportamento do próprio agente que não precisa mais ser solicitado por mensagens, uma vez que assim que uma mensagem é recebida pelo agente ela é enviada automaticamente para o AMS, que armazena a mensagem em um banco de dados especificado.

No módulo new_ams as seguintes classes são implementadas:

- **ComportSendConnMessages**: Conportamento temporal que envia mensagens a cada 4,0 segundos para verificar se o agente está conectado.
- **ComportVerifyConnTimed**: Comportamento temporal que a cada 10,0 segundos verifica o intervalo de tempo de respostas do comportamento ComportSendConnMessages. Se o intervalo de tempo da ultima resposta for maior que 10,0 segundos o agente AMS considera que o agente não está mais ativo e retira o agente da tabela de agentes vigente.

- **CompConnectionVerify**: Protocolo Request que recebe as mensagens de resposta dos agentes que têm sua conexão verificada.
- **PublisherBehaviour**: Protocolo Publisher-Subscribe em que o AMS é o publisher e publica uma nova tabela de agentes sempre que um novo agente ingressa na plataforma, ou que algum agente existente tem sua desconexão detectada.
- **AMS**: Classe que declara todos os comportamentos relacionados ao agente AMS, como verificação de conexão e publicação das tabelas de agentes, além de receber e armazenar as mensagens recebidas por todos os agentes presentes na plataforma.

2.12.3 Pacote misc

O pacote misc armazena módulos com diferentes propósitos gerais. Até o presente desenvolvimento, o pacote misc tem dois módulos principais, são eles: - common.py - utility.py

Começando pelo mais simples, o módulo utility.py tem como propópito oferecer métodos com facilidades para o desenvolvimento, o único método disponível é o método display_message() que imprime uma mensagem na tela, passada como parâmetro para o método, com data, horário e nome do agente.

O segundo módulo do pacote misc é o módulo common.py que fornece ao usuário do PADE métodos e objetos de propósito geral que têm relação com a inicialização de uma sessão multiagente PADE. Neste módulo estão presentes as classes FlaskServerProcess que inicializa o serviço web desenvolvido com o framework Flask. A outra classe do módulo é a classe PadeSession, que define as sessões a serem lançadas no ambiente de execução PADE. Essa classe tem uma das mais importantes funções no PADE que é a de inicializar o loop de execução do PADE. Ou seja, quando se pretende lançar agentes no ambiente de execução do PADE, o procedimento é o seguinte:

```
# importações necessarias:
from pade.misc.utility import display_message
from pade.misc.common import PadeSession
from pade.core.agent import Agent
from pade.acl.aid import AID

# definicao dos agentes por meio das classes Agent e de
# classes associadas a comportamentos ou protocolos

class AgenteHelloWorld(Agent):
    def __init__(self, aid):
        super(AgenteHelloWorld, self).__init__(aid=aid, debug=True)
        display_message(self.aid.localname, 'Hello World!')

# definicao do metodo para inicializacao dos agentes:
def config_agents():
```

(continues on next page)

(continuação da página anterior)

```
    agents = list()

    agente_hello = AgenteHelloWorld(AID(name='agente_hello'))
    agents.append(agente_hello)

    # declaracao do objeto PadeSession
    s = PadeSession()
    # adicao da lista de agentes
    s.add_all_agents(agents)
    # registro de usuarios na plataforma
    s.register_user(username='lucassm', email='lucas@gmail.com', password='12345')

    return s

# inicializacao dos ambiente de execucao
# propriamente dito:
if __name__ == '__main__':

    s = config_agents()
    # inicializacao do loop de execucao Pade
    s.start_loop()
```

Como pode ser observado por meio deste exemplo, é no método config_agents() que os agentes são instanciados, assim como o objeto PadeSession(), que logo em seguida tem seu método add_all_agents() chamado e recebe como parâmetro uma lista com as instancias dos agentes. Em seguida é chamado o método register_user() para registrar um usuario. O objeto PadeSession é retornado pelo método config_agents(), que é chamado no corpo principal desse script e atribuído a variável s, que logo em seguida, chama o método start_loop() do objeto PadeSession retornado.

Nesse simples exemplo é possível observar a importancia da classe PadeSession, que recebe as instancias dos agentes a serem lançados, registra usuarios da sessão e inicializa o loop de execussão do PADE.

Aqui cabe um questionamento essencial para o ambiente de execução de agentes PADE. E se quisermos lançar agentes em uma sessão PADE já iniciada, como faremos?

A primeira coisa a ser definida aqui é que após iniciada a sessão, não será possível registrar usuários e os agentes lançados, só poderão entrar na plataforma se forem utilizadas credenciais de usuários já registrados.