Pade Documentation

Versão 1.0

06 mar, 2019

Conteúdo

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 no ambito da Universidade Federal do Ceará pelo Grupo de Redes Elétricas Inteligentes (GREI) que pertence ao departamento de Engenharia Elétrica.

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!

```
# este e o arquivo start_ams.py
from pade.misc.common import set_ams, start_loop

if __name__ == '__main__':
    set_ams('localhost', 8000)
    start_loop(list(), gui=True)
```

1.2 E fácil de instalar!

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

```
$ pip install pade
$ python start_ams.py
```

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!

> **Aviso:** Atenção! O PADE é oficalmente testado no ambiente Ubuntu 14.04 LTS. Sendo necessário a instalação dos pacotes python-twisted-qt4reactor e pyside

2.1.2 Instalação via GitHub

Se você quiser ter acesso ao fonte do PADE e instala-lo a partir do fonte, basta digitar as seguintes linhas no terminal:

```
$ git clone https://github.com/lucassm/Pade
$ cd Pade
$ python setup.py install
```

Pronto o PADE está pronto para ser utilizado, faça um teste no interpretador Python digitando:

```
from pade.misc.utility import display_message
```

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
```

2.1.4 Ativando a interface gráfica

Para que a interface gráfica do PADE funcione é necessário que o pacote PySide, um binding do Qt, esteja instalado. Como não existem binários do PySide disponíveis para Linux e o procedimento de compilação é bastante demorado, é necessário fazer o seguinte procedimento:

1. Instale o PySide via linha de comando:

```
$ sudo apt-get install python-pyside
```

2. Copie a pasta com a instalação do PySide da pasta site-packages, onde fica a instalação padrão do Python em seu sistema operacional, e então coloque dentro da pasta onde ficam instalados os pacotes padrões do ambiente virtual, no nosso caso: venv/lib/python2.7/site-packages.

Pronto a instalação do PySide no ambiente virtual está concluída, mas outro procedimento que deve ser realizado é a instalação do reactor que interage com o loop de eventos do PySide, para isso, digite:

```
$ sudo apt-get install python-qt4reactor
```

2.2 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 start_ams.py com o seu editor de texto preferido, e copie e cole o seguinte trecho de código dentro dele:

```
# este e o arquivo start_ams.py
from pade.misc.common import set_ams, start_loop

if __name__ == '__main__':
    set_ams('localhost', 8000)
    start_loop(list(), gui=True)
```

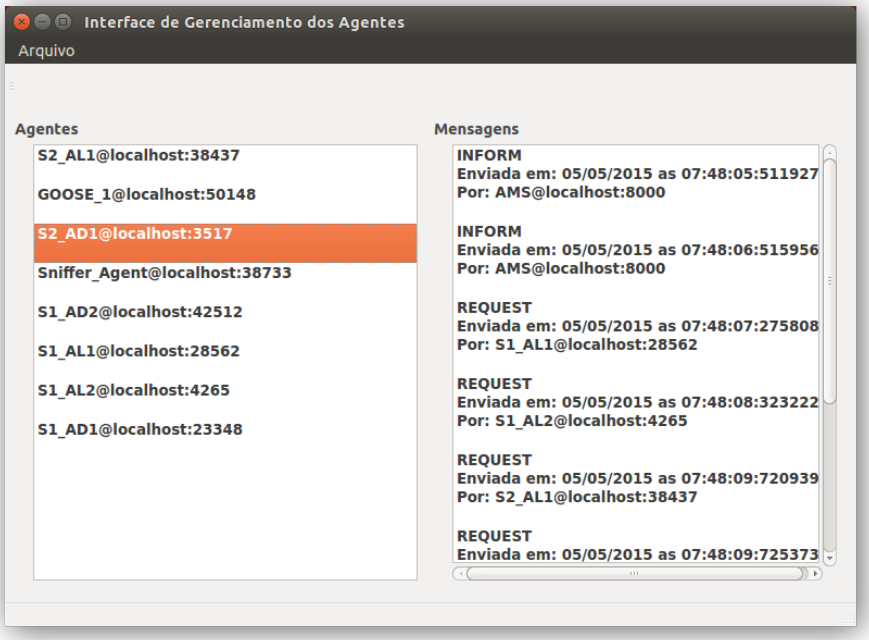
A essa altura você já deve estar vendo a interface gráfica de monitoramento dos agentes em Python, assim como essa:

2.3 Meu Primeiro Agente

Agora já está na hora de construir um agente de verdade!

2.3.1 Construindo meu primeiro agente

Para implementar seu primeiro agente abra seu editor de textos preferido e digite o seguinte código:




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


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

if __name__ == '__main__':

    set_ams('localhost', 8000, debug=False)

    agents = list()

    agente_hello = AgenteHelloWorld(AID(name='agente_hello'))
    agente_hello.ams = {'name': 'localhost', 'port': 8000}
    agents.append(agente_hello)

    start_loop(agents, gui=True)
```

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:

```
#!coding=utf-8
# Hello world temporal in PADE!
#
# Criado por Lucas S Melo em 21 de julho de 2015 - Fortaleza, Ceará - Brasil

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


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)
```

(continues on next page)

(continuação da página anterior)

```
if __name__ == '__main__':

    set_ams('localhost', 8000, debug=False)

    agents = list()

    agente_1 = AgenteHelloWorld(AID(name='agente_1'))
    agente_1.ams = {'name': 'localhost', 'port': 8000}

    agents.append(agente_1)

    start_loop(agents, gui=True)
```

2.4.2 Execução de dois agentes temporais

Mas e se eu quiser dois agentes com o mesmo comportamento!? Não tem problema, basta instanciar um outro agente com a mesma classe!

```
if __name__ == '__main__':

    set_ams('localhost', 8000, debug=False)

    agents = list()

    agente_1 = AgenteHelloWorld(AID(name='agente_1'))
    agente_1.ams = {'name': 'localhost', 'port': 8000}

    agente_2 = AgenteHelloWorld(AID(name='agente_2'))
    agente_2.ams = {'name': 'localhost', 'port': 8000}

    agents.append(agente_1)
    agents.append(agente_2)

    start_loop(agents, gui=True)
```

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>
        <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
from pade.misc.common import set_ams, start_loop
from pade.core.agent import Agent
from pade.acl.aid import AID
from pade.acl.messages import ACLMessage


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):
        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__':
```

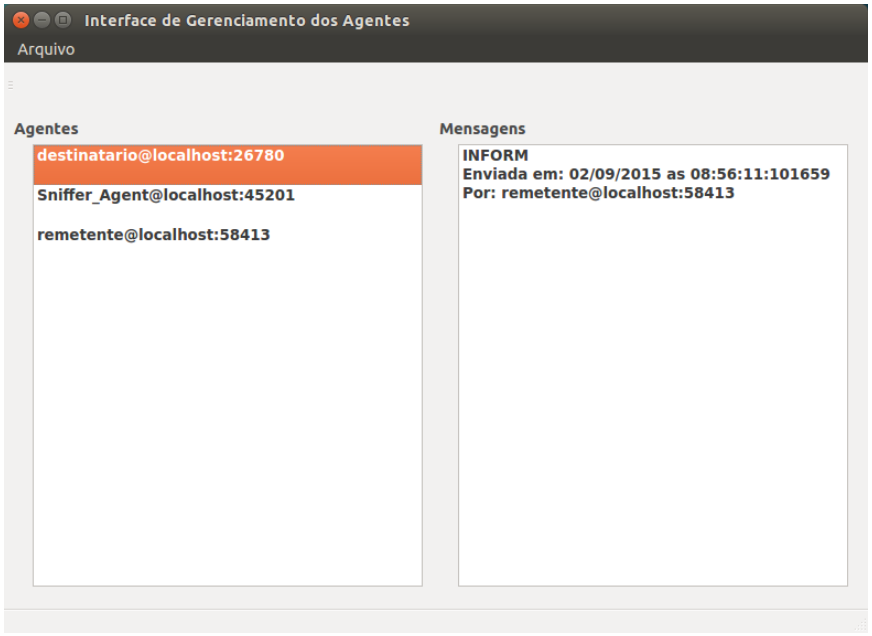
(continues on next page)

(continuação da página anterior)

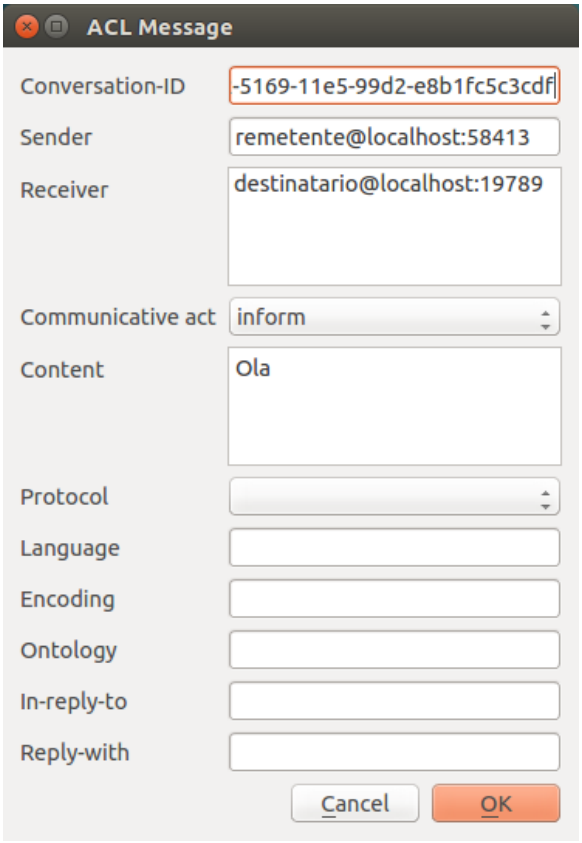
```
    set_ams('localhost', 8000, debug=False)

    agentes = list()

    destinatario = Destinatario(AID(name='destinatario'))
    destinatario.ams = {'name': 'localhost', 'port': 8000}
    agentes.append(destinatario)

    remetente = Remetente(AID(name='remetente'))
    remetente.ams = {'name': 'localhost', 'port': 8000}
    agentes.append(remetente)

    start_loop(agentes, gui=True)
```

2.6.2 Visualização via Interface Gráfica

A seguir é possível observar a interface gráfica do PADE que mostra os agentes cadastrados no AMS.



Ao clicar na mensagem recebida pelo agente *destinatario* é possível observar todos os dados contidos na mensagem:



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 utiliza *call_later()* é utilizado na classe *Remetente()* no método *on_start()* para assegurar que todos os agentes já foram lançados na plataforma, chamando o método *send_message()* 4,0 segundos após a inicialização dos agentes:

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

(continues on next page)

(continuação da página anterior)

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

    def on_start(self):
        self.call_later(4.0, self.send_message)

    def send_message(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):
        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__':

    set_ams('localhost', 8000, debug=False)

    agentes = list()

    destinatario = Destinatario(AID(name='destinatario'))
    destinatario.ams = {'name': 'localhost', 'port': 8000}
    agentes.append(destinatario)

    remetente = Remetente(AID(name='remetente'))
    remetente.ams = {'name': 'localhost', 'port': 8000}
    agentes.append(remetente)

    start_loop(agentes, gui=True)
```

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

Para ativar a funcionalidade de interface gráfica do PADE é bem simples, basta passar o parâmetro *gui=True* na função

```
start_loop(agentes, gui=True)
```

A interface gráfica do PADE ainda está bem simples e sem muitas funcionalidades, implementada com base no framework para desenvolvimento de GUI Qt/PySide, isso gera ulgumas complicações. Para a versão 2.0 será implementada uma interface web com base no framework Flask, mais completa e funcional.

2.11 Protocolos

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

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

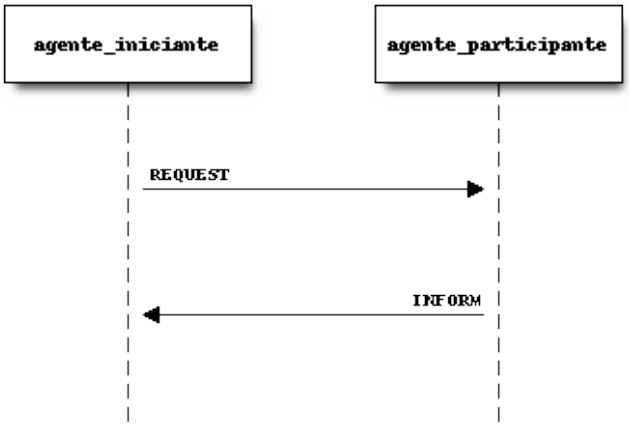
No PADE qualquer protocolo deve ser implementado como uma classe que extende a classe do protocolo desejado, por exemplo para implementar o protocolo FIPA-Request, deve ser implementada uma classe que implementa a herança da classe FipaRequestProtocol:

```
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:

```
from pade.misc.common import start_loop, set_ams
from pade.misc.utility import display_message
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


class CompRequest(FipaRequestProtocol):
    """Comportamento FIPA Request
    do agente Horario"""
    def __init__(self, agent):
        super(CompRequest, self).__init__(agent=agent,
                                          message=None,
                                          is_initiator=False)

    def handle_request(self, message):
        super(CompRequest, self).handle_request(message)
        display_message(self.agent.aid.localname, 'mensagem request recebida')
        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):
    """Comportamento FIPA Request
    do agente Relogio"""
    def __init__(self, agent, message):
        super(CompRequest2, self).__init__(agent=agent,
                                           message=message,
                                           is_initiator=True)
```

(continues on next page)

(continuação da página anterior)

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


class ComportTemporal(TimedBehaviour):
    """Comportamento FIPA Request
    do agente Relogio"""
    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 AgenteHorario(Agent):
    """Classe que define o agente Horario"""
    def __init__(self, aid):
        super(AgenteHorario, self).__init__(aid=aid, debug=False)

        self.comport_request = CompRequest(self)

        self.behaviours.append(self.comport_request)


class AgenteRelogio(Agent):
    """Classe que define o agente Relogio"""
    def __init__(self, aid):
        super(AgenteRelogio, self).__init__(aid=aid)

        # mensagem que requisita horario do horario
        message = ACLMessage(ACLMessage.REQUEST)
        message.set_protocol(ACLMessage.FIPA_REQUEST_PROTOCOL)
        message.add_receiver(AID(name='horario'))
        message.set_content('time')

        self.comport_request = CompRequest2(self, message)
        self.comport_temp = ComportTemporal(self, 1.0, message)

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


def main():
```

(continues on next page)

(continuação da página anterior)

```
    agentes = list()
    set_ams('localhost', 8000, debug=False)

    a = AgenteHorario(AID(name='horario'))
    a.ams = {'name': 'localhost', 'port': 8000}
    agentes.append(a)

    a = AgenteRelogio(AID(name='relogio'))
    a.ams = {'name': 'localhost', 'port': 8000}
    agentes.append(a)

    start_loop(agentes, gui=True)

if __name__ == '__main__':
    main()
```

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.

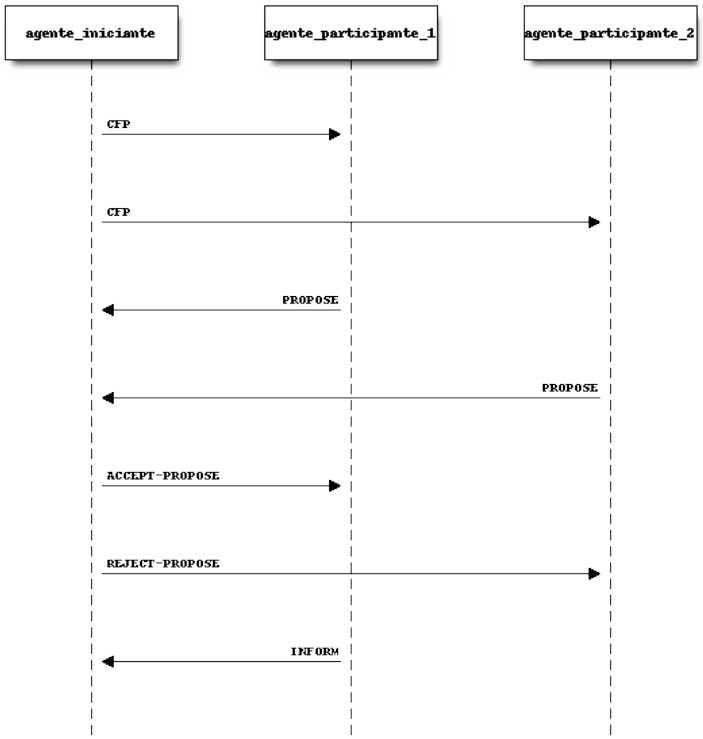
Em seguida, os agente propriamente ditos, são definidos nas classes AgenteHoraio e AgenteRelogio que estendem a classe Agent, nessas classes é que os comportamentos e protocolos são associados a cada agente.

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:




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


class CompContNet1(FipaContractNetProtocol):
    '''CompContNet1

       Comportamento FIPA-ContractNet Iniciante que envia mensagens
       CFP para outros agentes alimentadores solicitando propostas
       de restauração. Este comportamento também faz a analise das
       das propostas e analisa-as selecionando a que julga ser a
       melhor'''

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

(continues on next page)

(continuação da página anterior)

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

        super(CompContNet1, self).handle_all_proposes(proposes)

        melhor_propositor = None
        maior_potencia = 0.0
        demais_propositores = list()
        display_message(self.agent.aid.name, 'Analisando propostas...')

        i = 1

        # lógica de seleção de propostas pela maior potência disponibilizada
        for message in proposes:
            content = message.content
            potencia = float(content)
            display_message(self.agent.aid.name,
                            'Analisando proposta {i}'.format(i=i))
            display_message(self.agent.aid.name,
                            'Potencia Ofertada: {pot}'.format(pot=potencia))
            i += 1
            if potencia > maior_potencia:
                if melhor_propositor is not None:
                    demais_propositores.append(melhor_propositor)

                maior_potencia = potencia
                melhor_propositor = message.sender
            else:
                demais_propositores.append(message.sender)

        display_message(self.agent.aid.name,
                        'A melhor proposta foi de: {pot} VA'.format(
                            pot=maior_potencia))

        if demais_propositores != []:
            display_message(self.agent.aid.name,
                            'Enviando respostas de recusa...')
            resposta = ACLMessage(ACLMessage.REJECT_PROPOSAL)
            resposta.set_protocol(ACLMessage.FIPA_CONTRACT_NET_PROTOCOL)
            resposta.set_content('')
            for agente in demais_propositores:
                resposta.add_receiver(agente)

            self.agent.send(resposta)
```

(continues on next page)

(continuação da página anterior)

```
        if melhor_propositor is not None:
            display_message(self.agent.aid.name,
                            'Enviando resposta de aceitacao...')

            resposta = ACLMessage(ACLMessage.ACCEPT_PROPOSAL)
            resposta.set_protocol(ACLMessage.FIPA_CONTRACT_NET_PROTOCOL)
            resposta.set_content('OK')
            resposta.add_receiver(melhor_propositor)
            self.agent.send(resposta)

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

        display_message(self.agent.aid.name, 'Mensagem INFORM recebida')

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

        display_message(self.agent.aid.name, 'Mensagem REFUSE recebida')

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

        display_message(self.agent.aid.name, 'Mensagem PROPOSE recebida')


class CompContNet2(FipaContractNetProtocol):
    '''CompContNet2

       Comportamento FIPA-ContractNet Participante que é acionado
       quando um agente recebe uma mensagem do Tipo CFP enviando logo
       em seguida uma proposta e caso esta seja selecinada realiza as
       as análises de restrição para que seja possível a restauração'''

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

(continues on next page)

(continuação da página anterior)

```
    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, 'Mensagem CFP recebida')

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

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

        display_message(self.agent.aid.name,
                        'Mensagem REJECT_PROPOSAL recebida')

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

        display_message(self.agent.aid.name,
                        'Mensagem ACCEPT_PROPOSE recebida')

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


class AgenteIniciante(Agent):

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

(continues on next page)

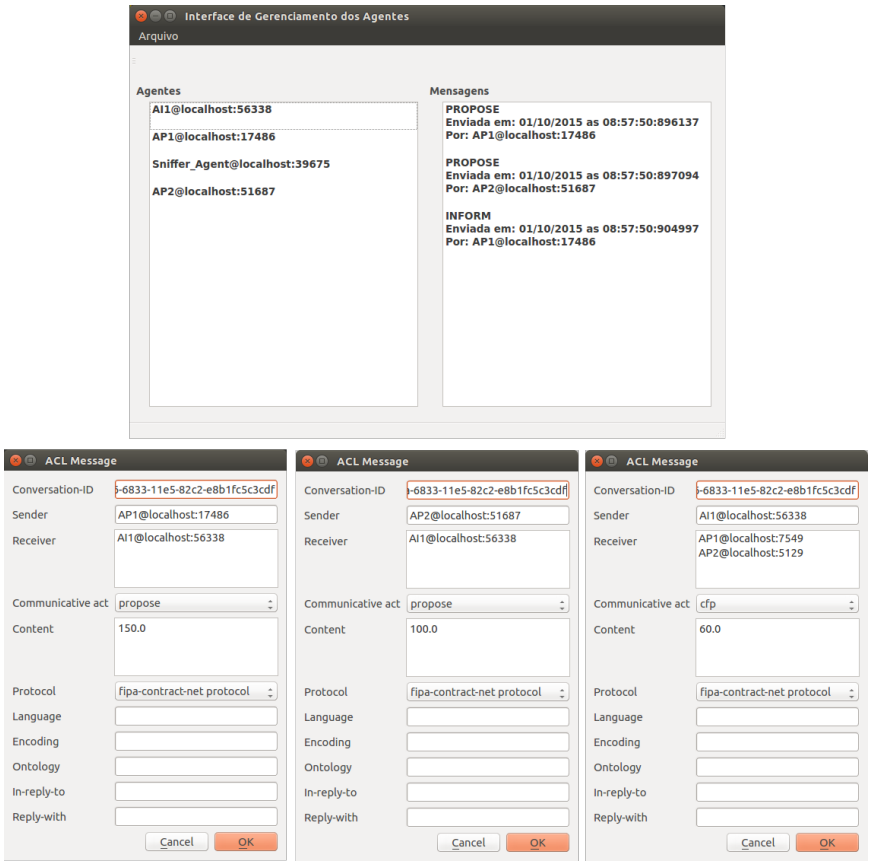
(continuação da página anterior)

```
        message = ACLMessage(ACLMessage.CFP)
        message.set_protocol(ACLMessage.FIPA_CONTRACT_NET_PROTOCOL)
        message.set_content('60.0')
        message.add_receiver(AID('AP1'))
        message.add_receiver(AID('AP2'))

        comp = CompContNet1(self, message)
        self.behaviours.append(comp)
        self.call_later(2.0, comp.on_start)


class AgenteParticipante(Agent):

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

        self.pot_disp = pot_disp

        comp = CompContNet2(self)

        self.behaviours.append(comp)

if __name__ == "__main__":

    set_ams('localhost', 5000, debug=False)

    aa_1 = AgenteIniciante(AID(name='AI1'))
    aa_1.ams = {'name': 'localhost', 'port': 5000}

    aa_2 = AgenteParticipante(AID(name='AP1'), 150.0)
    aa_2.ams = {'name': 'localhost', 'port': 5000}

    aa_3 = AgenteParticipante(AID(name='AP2'), 100.0)
    aa_3.ams = {'name': 'localhost', 'port': 5000}

    agents_list = list([aa_1, aa_2, aa_3])

    start_loop(agents_list, gui=True)
```

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, AgenteI-

niciante(), 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:

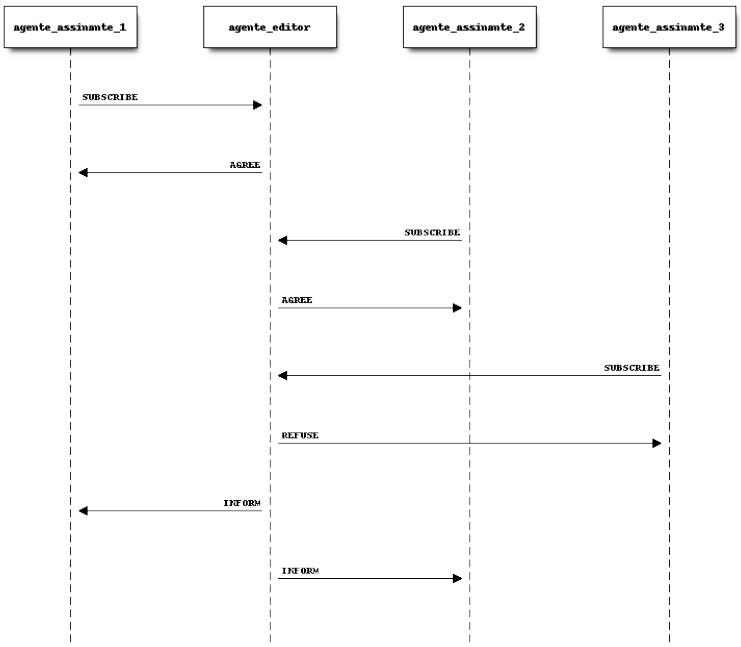


2.11.3 FIPA-Subscribe

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:




```
from pade.misc.common import start_loop, set_ams
from pade.misc.utility import display_message
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
from numpy import sin


class SubscribeInitiator(FipaSubscribeProtocol):

    def __init__(self, agent, message):
        super(SubscribeInitiator, 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 SubscribeParticipant(FipaSubscribeProtocol):
```

(continues on next page)

(continuação da página anterior)

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

    def handle_subscribe(self, message):
        self.register(message.sender)
        display_message(self.agent.aid.name, message.content)

        resposta = message.create_reply()
        resposta.set_performative(ACLMessage.AGREE)
        resposta.set_content('Pedido de subscricao aceito')
        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(SubscribeParticipant, 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(sin(self.inc)))

        self.notify(message)
        self.inc += 0.1


class AgenteInitiator(Agent):

    def __init__(self, aid, message):
        super(AgenteInitiator, self).__init__(aid)
        self.protocol = SubscribeInitiator(self, message)
        self.behaviours.append(self.protocol)
```

(continues on next page)

(continuação da página anterior)

```
class AgenteParticipante(Agent):

    def __init__(self, aid):
        super(AgenteParticipante, self).__init__(aid)

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

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

if __name__ == '__main__':

    set_ams('localhost', 5000, debug=False)

    editor = AgenteParticipante(AID('editor'))
    editor.ams = {'name': 'localhost', 'port': 5000}

    msg = ACLMessage(ACLMessage.SUBSCRIBE)
    msg.set_protocol(ACLMessage.FIPA_SUBSCRIBE_PROTOCOL)
    msg.set_content('Pedido de subscricao')
    msg.add_receiver('editor')

    ass1 = AgenteInitiator(AID('assinante_1'), msg)
    ass1.ams = {'name': 'localhost', 'port': 5000}

    ass2 = AgenteInitiator(AID('assinante_2'), msg)
    ass2.ams = {'name': 'localhost', 'port': 5000}

    agentes = [editor, ass1, ass2]

    start_loop(agentes, gui=True)
```

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 ati-

vidade 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():

    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')
```

(continues on next page)

(continuação da página anterior)

```
    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.

CAPÍTULO **3**

Referência da API do PADE

3.1 API PADE

Aqui estão todos os módulos que são de interesse para utilização do PADE, tais como o módulo que contém a classe Agent, módulo de construção de mensagens e de construção de comportamentos.

3.1.1 FIPA-ACL message creation and handling module

This module contains a class which implements an ACLMessage type object. This object is the standard FIPA message used in the exchange of messages between agents.

class pade.acl.messages.**ACLMessage**(*performative=None*)
Class that implements a ACLMessage message type

add_receiver(*aid*)
Method used to add recipients for the message being created.

Parâmetros **aid** – AID type object that identifies the agent that will receive the message.

add_reply_to(*aid*)
Method used to add the agents that should receive the answer of the message.

Parâmetros **aid** – AID type object that identifies the agent that will receive the answer of this message.

create_reply()
Creates a reply for the message Duplicates all the message structures exchanges the "from" AID with the "to" AID

set_performative(*performative*)
Method to set the Performative parameter of the ACL message.

Parâmetros **performative** – performative type of the message.

It can be any of the attributes of the ACLMessage class.

set_sender(*aid*)
Method to set the agent that will send the message.

Parâmetros **aid** – AID type object that identifies the agent that will send the message.

Índice de Módulos do Python

Índice