

UNIVERSIDADE FEDERAL DO MARANHÃO – CIDADE UNIVERSITÁRIA
BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO
SISTEMAS DISTRIBUÍDOS

LUIS FELIPE FERREIRA SOARES
JOSE ROBERTO DE ARAUJO ARANHA JUNIOR

1 PYGROUPS

Este documento descreve a execução do trabalho proposto.

1.1 Descrição do Projeto

Este projeto se destina a criação de uma biblioteca em Python destinada à comunicação de 1 para N processos na mesma máquina e/ou máquinas distintas (grupo), liberando o programador da tarefa de gerenciar N - 1 conexões ponto-a-ponto.

Na criação da biblioteca, foram assumidas algumas premissas:

- Os processos registram o interesse de entrar ou sair do grupo através de diretivas “join” ou “leave”, respectivamente, se tornando ou deixando de ser “membros” do grupo, assim existindo um acordo sobre quem pertence ou não ao grupo;
- Os processos não-falham e não são “mortos”, isto é, não entram/saem do grupo enquanto a comunicação está em curso;
- Os processos recebem as mensagens em ordem total;
- Os processos identificam a falta de mensagens no recebimento de outras mensagens; - Os processos requisitam mensagens faltosas a todos os membros do grupo, ou seja, os processos mantêm um registro das mensagens enviadas.

1.2 Implementação

Um Grupo é representado como uma classe. Uma instância do Grupo pode se tornar um membro através do método “join”. As mensagens trocadas entre membros do grupo são estruturadas. Esta estrutura é destinada a identificar qual o objetivo da mensagem recebida. Mensagens (estruturadas ou não) enviadas por processos externos ao grupo são recusadas, salvo casos especiais, como o “join”. Uma vez que está no grupo, o membro poderá receber e enviar mensagens para o grupo. O envio é feito diretamente, através do método “send”. O recebimento é feito através do registro de callbacks: Toda vez que uma mensagem é recebida, as funções de “callback” são chamadas para decidir o que fazer com a mensagem (armazenar em uma

estrutura/objeto/arquivo, mostrar no console/GUI, etc.). Apenas tipos de mensagens específicos podem registrar callbacks, pois certas mensagens são apenas meros acordos entre as instâncias. Uma vez que o processo não deseje mais participar da dinâmica da comunicação em grupo, ele pode optar por sair do grupo através do método “leave”, se tornando apenas uma instância vazia da classe Grupo. Uma segunda opção é apenas pausar o recebimento de mensagens, sem prejuízo ao envio.

1.2.1 DA ENTRADA DE MEMBROS

A entrada de um membro se dá através do “join”. Este método recebe como argumento o nome do grupo e faz a inicialização de atributos básicos de membro. Este método executa um broadcast na rede para determinar se já existem grupos com o nome passado como argumento da função. Caso exista, ele faz uma solicitação de entrada no grupo para quem lhe respondeu. Eventualmente, ele receberá a resposta desta solicitação, descobrindo os dados (atualizados) do grupo já existente e se tornando efetivamente um membro. Caso não exista, ele cria um novo grupo com esse nome. Uma vez efetivada como membro do grupo, a instância do Grupo é capaz de receber e enviar mensagens.

1.2.2 DO ENVIO E RECEBIMENTO DE MENSAGENS

A estrutura das mensagens define se uma mensagem é destinada a aplicação ou se é uma negociação entre instâncias do Grupo. Os métodos “públicos” da classe permitem o envio apenas de mensagens destinadas a aplicação. Toda mensagem é estruturada em “cabeçalho”, onde estão informações acerca da negociação de comunicação, e “corpo”, que contém o texto referente a mensagem.

As mensagens são recebidas através de uma “thread”, para evitar a característica bloqueante do socket no processo principal, e tratadas por “handlers” de acordo com as “tags” contidas no “cabeçalho” da mensagem. Os membros podem registrar “callbacks”, funções que recebem o “corpo” da mensagem, após a mensagem ser devidamente tratada na etapa anterior, e decidem o que fazer com ela. As mensagens de negociação não são enviadas para a aplicação, exceto aquelas de “join” e “leave”. É possível pausar o recebimento de mensagens através do método “receiverOff”, que simplesmente cessa o recebimento no socket. Devido à natureza dos sockets, as mensagens destinadas a estes durante o período de pausa chegarão assim que o recebimento for solicitado novamente através do método “receiverOn” ou serão completamente perdidas caso a conexão seja encerrada, ou seja, execute um “leave”.

O envio de mensagens é uma tarefa bem simples. O método “send” das instâncias membros do grupo recebe uma string e a envia para todos os membros do grupo. As mensagens são armazenadas em um buffer para o eventual reenvio em caso de falha na entrega.

1.2.3 DA SAÍDA DE MEMBROS

A saída do grupo se dá através do método “leave”. Este método não recebe nenhum argumento, pois somente pode ser executado por um membro de um grupo. O método executa o envio de uma mensagem informando sua saída do grupo e realiza um “cleanup” na instância do grupo, impossibilitando a utilização da instância para recebimento/envio de mensagens após indicar seu interesse em deixar o grupo.

1.2.4 DA ORDENAÇÃO TOTAL

Segundo Tanenbaum (2007, p. 213), a “entrega totalmente ordenada” ou “ordenação total” significa que: “[...] independentemente de a entrega ser não ordenada, ordenada em Fifo ou ordenada por causalidade, exige-se adicionalmente que, quando as mensagens forem entregues, devam ser entregues na mesma ordem a todos os membros do grupo”. Para tanto, utilizou-se a ideia de sequenciador. O sequenciador é responsável por determinar um número de sequência. Antes do envio de cada mensagem, os membros solicitam este número para o sequenciador e adicionam esta informação no cabeçalho da mensagem. Cada membro possui o número da última mensagem recebida, assim, no recebimento da mensagem, eles comparam o número que possuem, que chamaremos aqui de “ n ”, e o número de sequência da mensagem, “ k ”. Quando $k=n+1$, a mensagem é aceita corretamente e incrementa-se o valor de n . O caso $k<n+1$ implica mensagem já foi anteriormente recebida, sendo assim descartada. Enquanto que $k>n+1$ indica que existem $k-n-1$ mensagens atrasadas, solicitando por flooding no grupo as mensagens que estão faltando.

A respeito do sequenciador, este é escolhido da seguinte forma:

- (1) na criação do grupo, o sequenciador é quem criou o grupo;
- (2) caso o sequenciador saia do grupo, o segundo membro no início da lista de membros é eleito o novo sequenciador (pois o primeiro é o antigo sequenciador).

1.2.5 DA ORDENAÇÃO CAUSAL

Segundo Tanenbaum (2007, p. 213), um *multicast* confiável ordenado por causalidade entrega mensagens de forma que a potencial causalidade entre mensagens diferentes seja preservada. Isso implica dizer que, se uma mensagem m_1 for enviada antes de uma mensagem m_2 e estas tiverem relação de causalidade, independentemente de elas terem sido enviadas em

multicast pelo mesmo remetente, a camada de comunicação em cada receptor sempre entregará m_2 após ter recebido e entregado m_1 . Para tal, utilizou-se os conceitos de relógios de Lamport, generalizado para relógios vetoriais.

Nos relógios vetoriais, cada membro do grupo mantém um vetor de números de sequência, denotado aqui por $S = [S_1, S_2, S_3, \dots, S_n]$, onde cada S_k representa a quantidade de mensagens recebidas do membro k . Quando um membro i envia uma mensagem ao grupo, ele incrementa S_i e anexa seu vetor S ao cabeçalho da mensagem. Por exemplo, se tivermos o vetor $S = [6, 4, 3]$, significa que o grupo possui 3 membros, e que o membro 1 já enviou 6 mensagens ao grupo, e já recebeu 4 mensagens do membro 2, e 3 mensagens do membro 3.

Quando uma nova mensagem chega ao grupo, ela contém um cabeçalho que contém o vetor $T = [T_1, T_2, T_3, \dots, T_n]$, que representa o relógio vetorial atual do grupo, além do id do membro que enviou a mensagem.

Ao receber essa nova mensagem, enviada por um membro i , o membro j pode aceitar a mensagem, pedir o reenvio da mesma (caso ela esteja adiantada), ou descartá-la, seguindo às seguintes regras:

- (1) Se $T_i = S_i + 1$ e $T_k \leq S_k$ para todo $k \neq i$, então a mensagem é aceita.
- (2) Se $T_i \geq S_i + 1$ ou existir um $k \neq i$ tal que $T_k \geq S_k$, então é solicitado o reenvio da mensagem.
- (3) Se $T_i \leq S_i$, então a mensagem é repetida, e por esse motivo é ignorada.

1.2.6 EXEMPLO

Exemplo do funcionamento da biblioteca.

Figura 1 - Um cliente do PyGroups

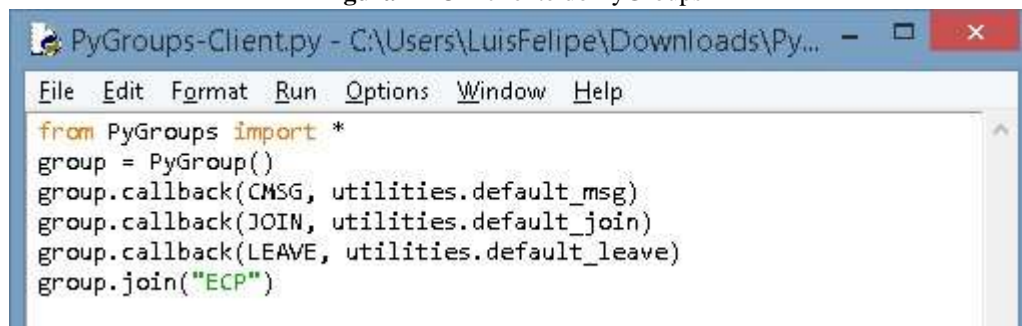
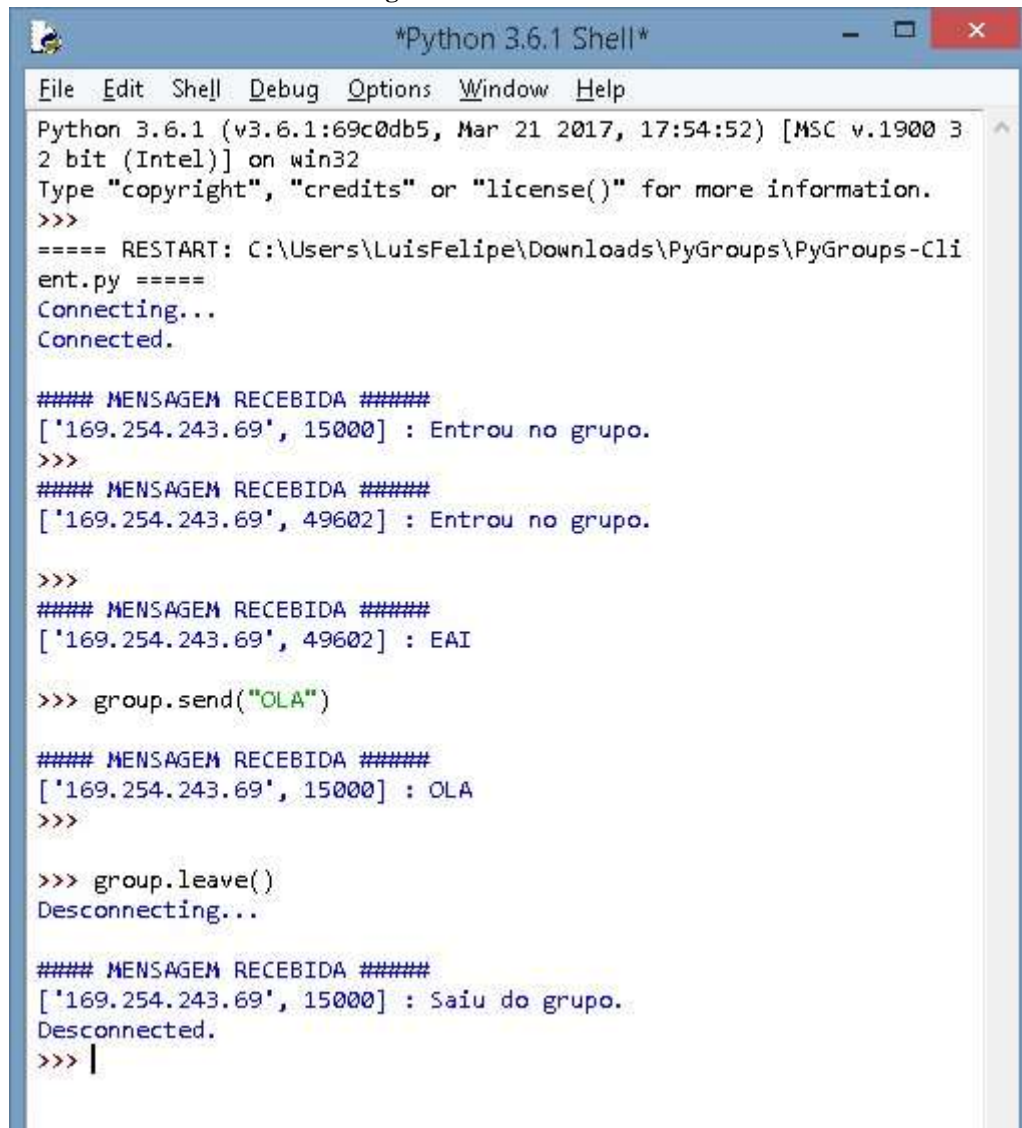


Figura 2 - Cliente Rodando



```
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\LuisFelipe\Downloads\PyGroups\PyGroups-Client.py =====
Connecting...
Connected.

#### MENSAGEM RECEBIDA ####
['169.254.243.69', 15000] : Entrou no grupo.
>>>
#### MENSAGEM RECEBIDA ####
['169.254.243.69', 49602] : Entrou no grupo.

>>>
#### MENSAGEM RECEBIDA ####
['169.254.243.69', 49602] : EAI

>>> group.send("OLA")

#### MENSAGEM RECEBIDA ####
['169.254.243.69', 15000] : OLA
>>>

>>> group.leave()
Disconnecting...

#### MENSAGEM RECEBIDA ####
['169.254.243.69', 15000] : Saiu do grupo.
Disconnected.
>>> |
```

REFERÊNCIAS

TANENBAUM, A. S.; STEEN, M. V. Sistemas distribuídos: princípios e paradigmas. 2. ed. São Paulo: Pearson Prentice Hall, 2007.