UNIVERSIDADE FEDERAL DE ITAJUBÁ
INSTITUTO DE MATEMÁTICA E COMPUTAÇÃO

COM242 - Sistemas Distribuídos

Prof. Rafael Frinhani

TUTORIAL

Publish-Subscribe

Grupo:

Alef Aparecido de Paula Bispo - alef@unifei.edu.br - 2018008460
Flávio Mota Gomes - flavio.gomes@unifei.edu.br - 2018005379
Rafael Antunes Vieira - rafaelantunesvieira@unifei.edu.br - 2018000980

09 de Junho, 2021

Sumário

---

1 Introdução

Este tutorial realiza uma contextualização sobre o tema **Publish-Subscribe**, um método sobre comunicação em Sistemas Distribuídos. Para isso, apresentar-se-á um panorama sobre esse método, seu conceito e suas características. Em seguida, serão apresentados os passos de instalação necessários para aplicações que utilizam do método, bem como uma aplicação em Python que o apresenta.

2 O método Publish-Subscribe

A troca de mensagens é o meio de comunicação entre processos distribuídos, sendo trocadas entre processos localizados em diferentes máquinas. Para realizar essa comunicação entre plataformas ou mesmo tecnologias diferentes, existem protocolos que garantem o êxito dessa troca [Coulouris et al., 2007]. Essa comunicação, segundo [Coulouris et al., 2007], precisa ser rápida ao ponto que os usuários tenham a impressão de estarem acessando os recursos diretamente.

Nesse sentido, são vários os métodos de comunicação ou coordenação em sistemas distribuídos. Os fatores do meio influenciam na escolha da metodologia mais adequada à situação.

O método *Publish-Subscribe*, objeto de estudo deste trabalho, é um método no qual assinantes (*subscribers*) manifestam seu interesse em receber determinadas informações através de inscrições, que são gerenciadas por um serviço de eventos (*broker*) para o qual os editores (*publisher*) divulgam seus eventos [Coulouris et al., 2007]. A Figura 1 apresenta um organograma do funcionamento desse método. Observe que *Publisher* e *Subscriber* não se relacionam diretamente, e sim têm uma relação mediada por um outro ator, conhecido como *Broker*, que é o responsável por gerenciar a comunicação. As características desses três atores serão explicadas com mais profundidade a seguir.

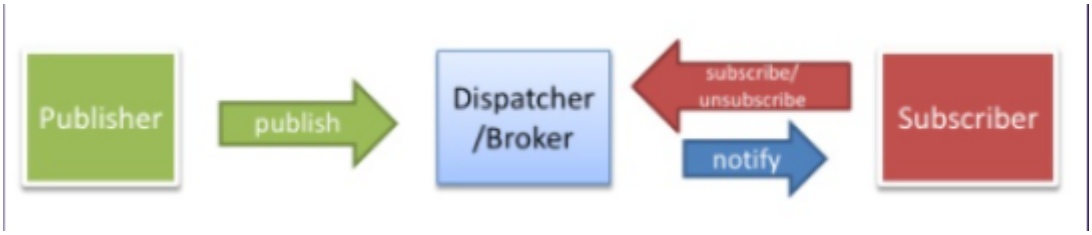

Figura 1: Ilustração de funcionamento do método Publish-Subscribe

.

2.1 Características

Conforme exposto na Figura 1, a estrutura do método *Publish-Subscribe* é composta de três elementos: o *Publisher*, o *Broker* e o *Subscriber*. A seguir, define-se cada um deles.

2.1.1 Publisher

O *Publisher*, também conhecido como publicador, é o agente responsável por mandar as mensagens para o *Broker*. Essa é a unica relação estabelecida, visto que ele não precisa saber quem receberá as mensagens enviadas por ele. Seu trabalho é apenas mandar as informações, e também é responsável por definir o tópico na mensagem.

2.1.2 Broker

O *Broker* é o responsável por armazenar e gerenciar as inscrições. É ele que recebe a mensagem do *Publisher* e redireciona o envio dela para os assinantes (*Subscribers*). O *Broker* é, portanto, o único elo entre publicador e subscrito.

2.1.3 Subscriber

O *Subscriber* é quem realiza suas inscrições para receber determinada informação. Também pode se desinscrever da lista, caso assim deseje. A comunicação sempre é feita com o *Broker*: o *Subscriber* informa ao *Broker* onde quer se inscrever ou se desinscrever e o *Broker* realiza a notificação de que um conteúdo desejado está disponível para ser entregue ao *Subscriber*.

2.2 Exemplo de funcionamento

Este tópico apresentará, de maneira ilustrada, o funcionamento de do método Publish-Subscriber. Em cada uma das dezenove imagens a seguir, estão representados: à esquerda, os *Publishers*; ao centro, o *broker*; à direita, os *Subscribers*.

Os conjuntos P, associados aos *Publishers*, representam o tipo de publicação que eles realizam. Já os conjuntos S, associados aos *Subscribers*, representam as listas nas quais esses

subscribers estão inscritos. Observe, ao decorrer das imagens, como ocorrem as ações de publicação e notificação, e o caminho das mensagens - identificadas em um círculo de cor verde-água - do publicador até o(s) subscrito(s) na lista.

O cenário se inicia com a Figura 2. Nela, estão representados dois *Publishers*: **PUB1** e **PUB2**, o *broker* ao centro, ainda vazio, e dois *Subscribers* à direita; **SUB1** e **SUB2**. **SUB1** está inscrito em **A**, **D**; **SUB2** está inscrito em **A**. Como não há, no broker, **A** ou **D**, os *subscribers* não têm o quê receber. Observe o funcionamento desse sistema a partir da visualização da Figura 2 até a Figura 20. A descrição do que acontece em cada uma das imagens está devidamente destacada em suas legendas.

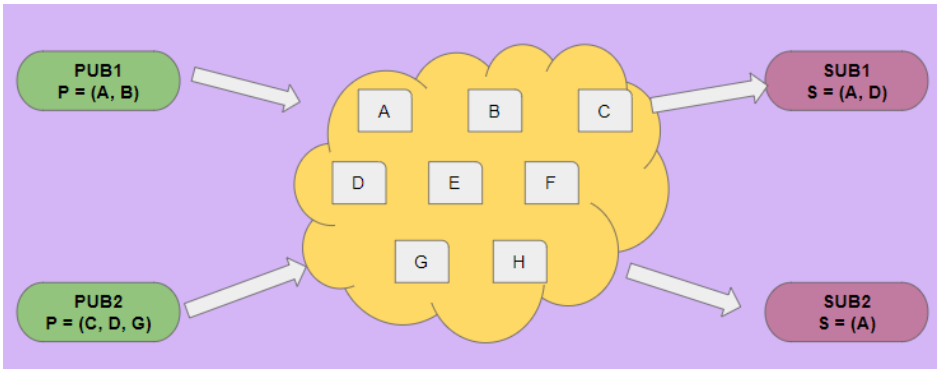

Figura 2: Situação inicial da demonstração.

.

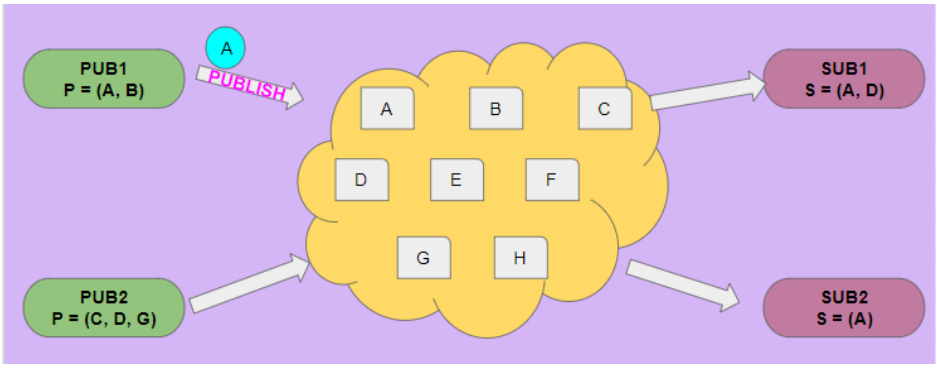

Figura 3: PUB1 realiza PUBLISH na mensagem A.

.

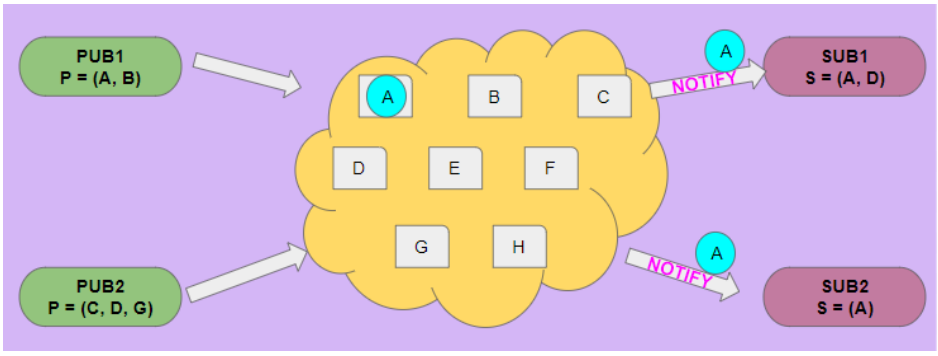

Figura 4: Realizada a publicação, a mensagem A chega ao broker, que notifica os interessados

.

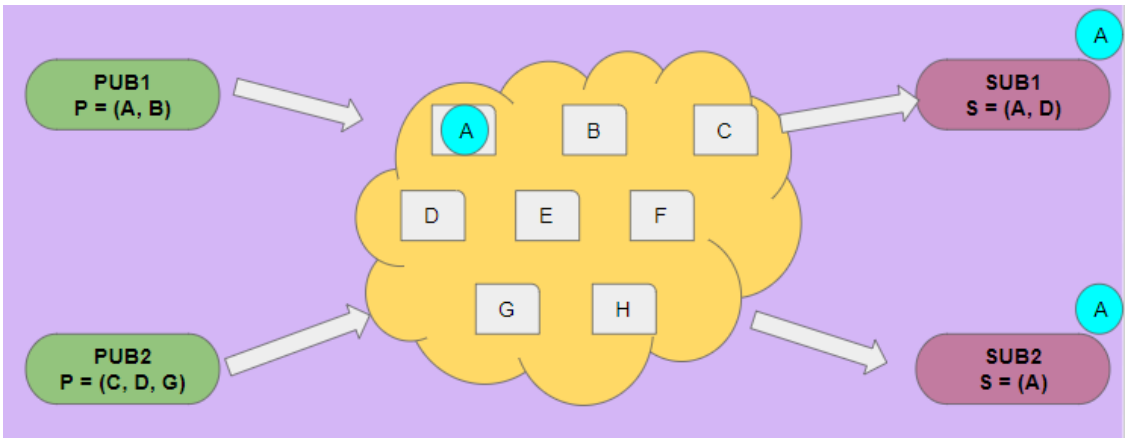

Figura 5: A mensagem A chegou aos SUBs inscritos e permanece presente no broker.

.

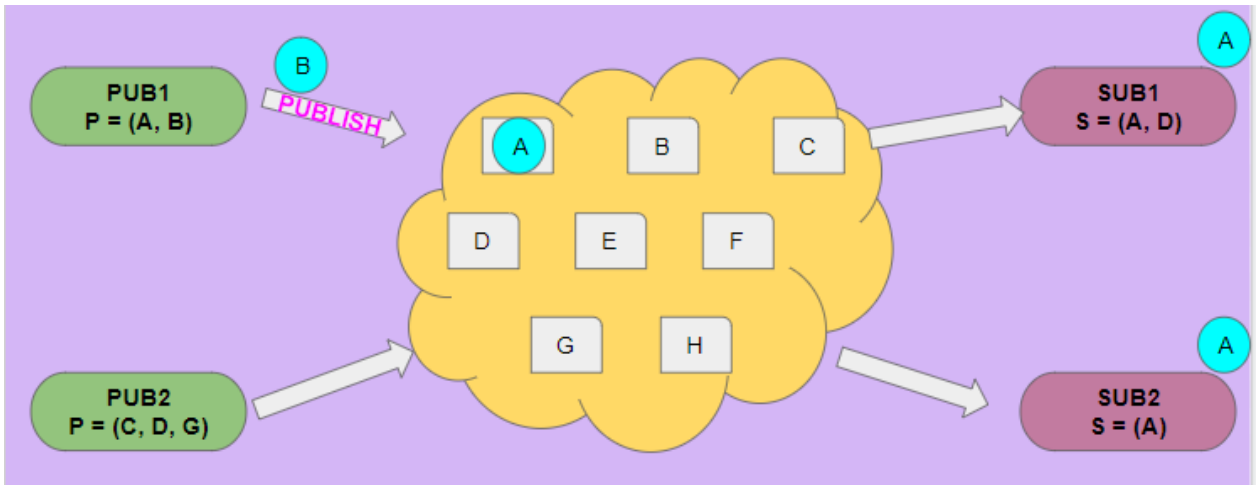

Figura 6: Agora o PUB1 realiza um novo evento de PUBLISH, desta vez com a mensagem B.

.

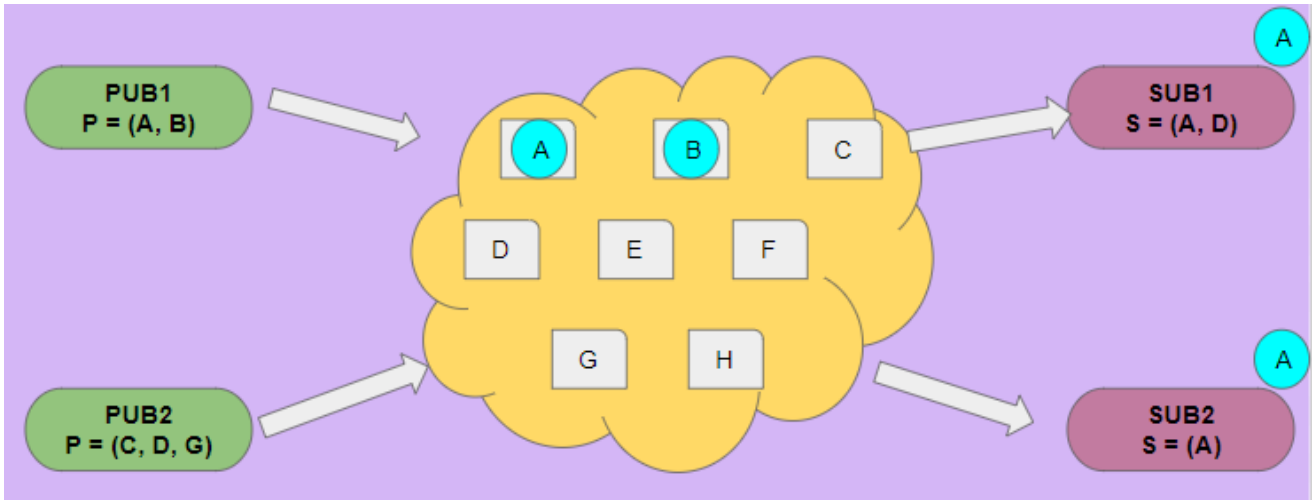

Figura 7: A mensagem B chegou ao broker. Todavia, nenhum dos SUBs atuais está inscrito para recebê-la. Dessa forma, a mensagem permanece apenas no broker.

.

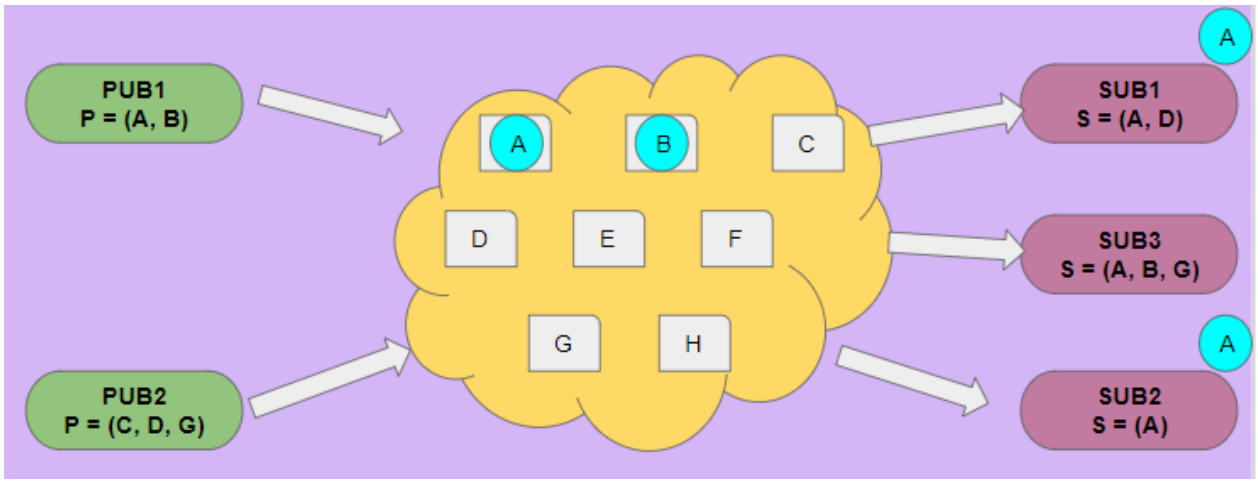

Figura 8: Agregou-se um novo SUB ao broker. O SUB3, recém adicionado, está inscrito para A, B e G. Note que as mensagens A e B já estão no broker.

.

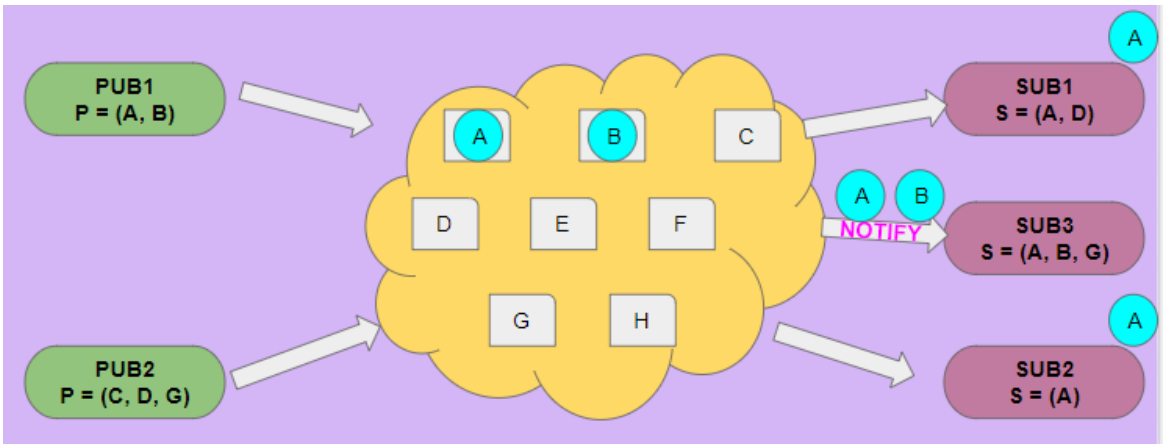

Figura 9: Como as mensagens A e B estão no broker, ele realiza o NOTIFY do SUB3, que deseja recebê-las. Mas, note, não notifica novamente os SUBs que já as têm.

.

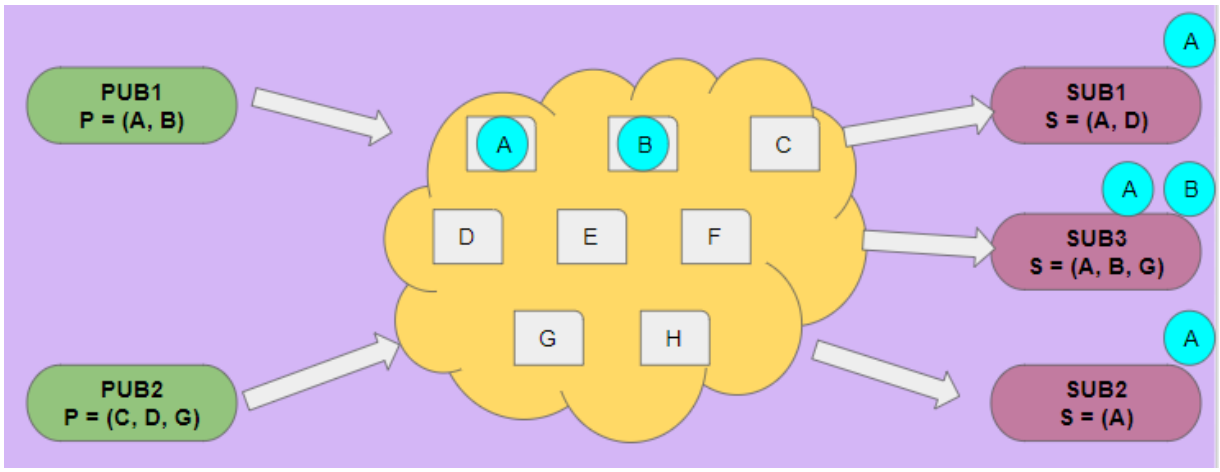

Figura 10: As mensagens A e B são entregues ao SUB3.

.

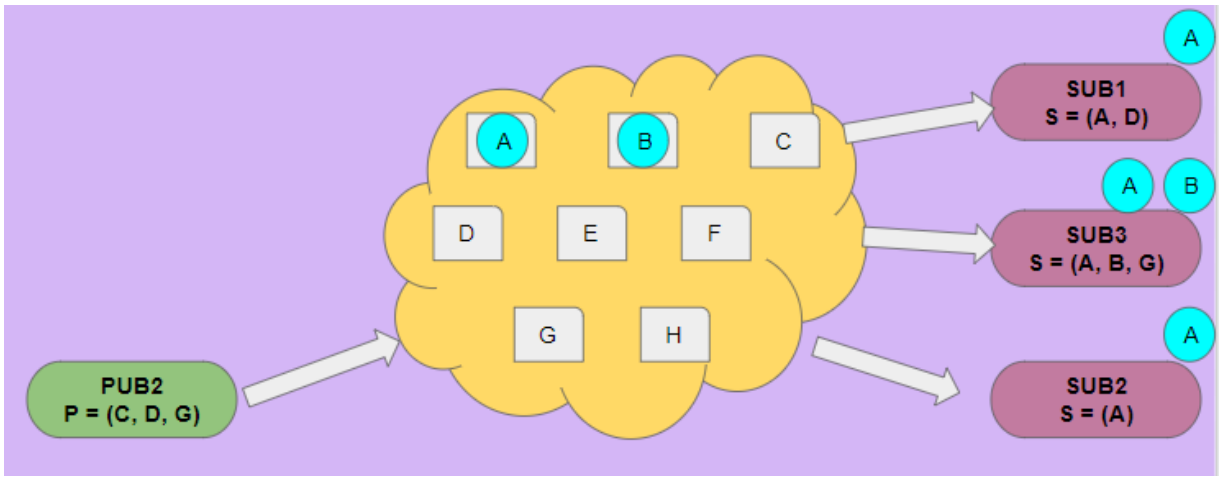

Figura 11: O PUB1 foi removido da figura, visto que já realizou todos seus eventos de PUBLISH.

.

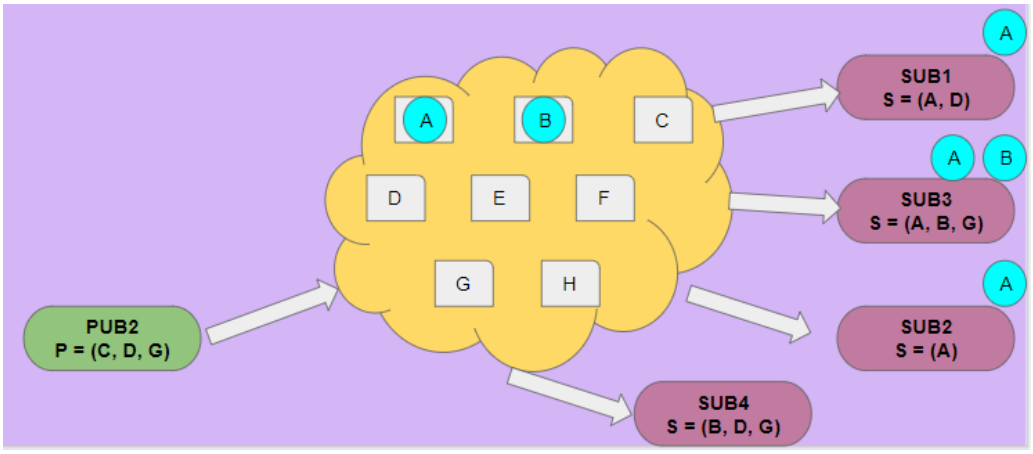

Figura 12: Um novo SUB foi adicionado. Trata-se do SUB4, inscrito para B, D e G. Note que o broker já possui B.

.

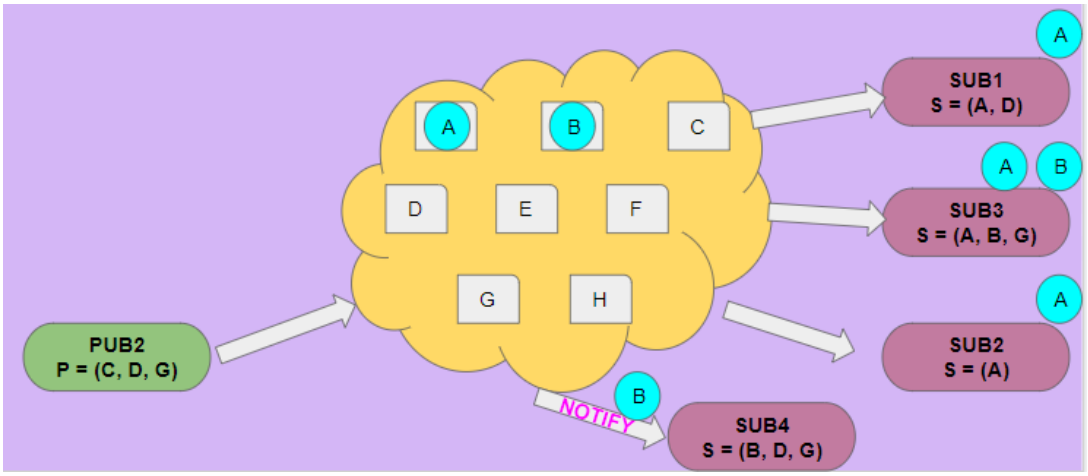

Figura 13: O broker realiza o NOTIFY de B, já que ele já o tem.

.

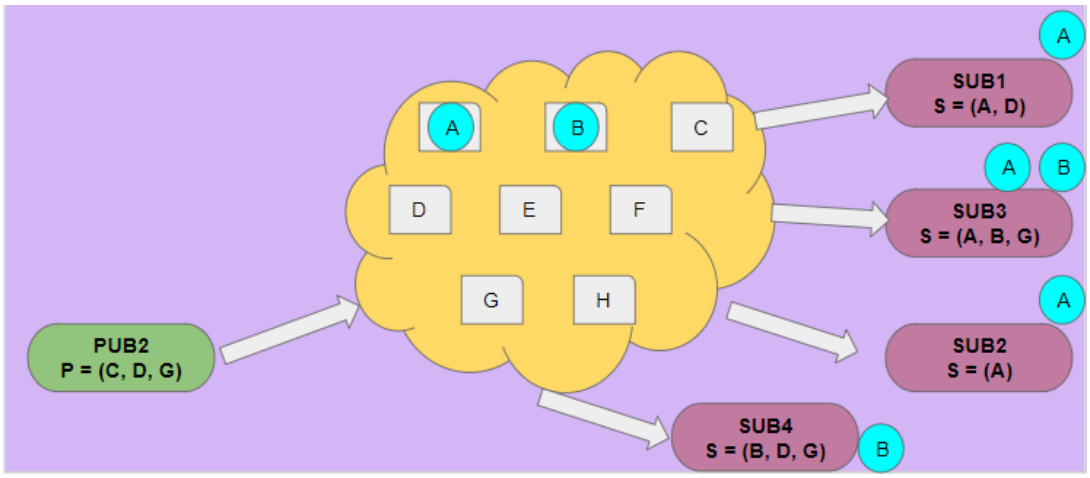

Figura 14: O recém adicionado SUB4 recebe a mensagem B.

.

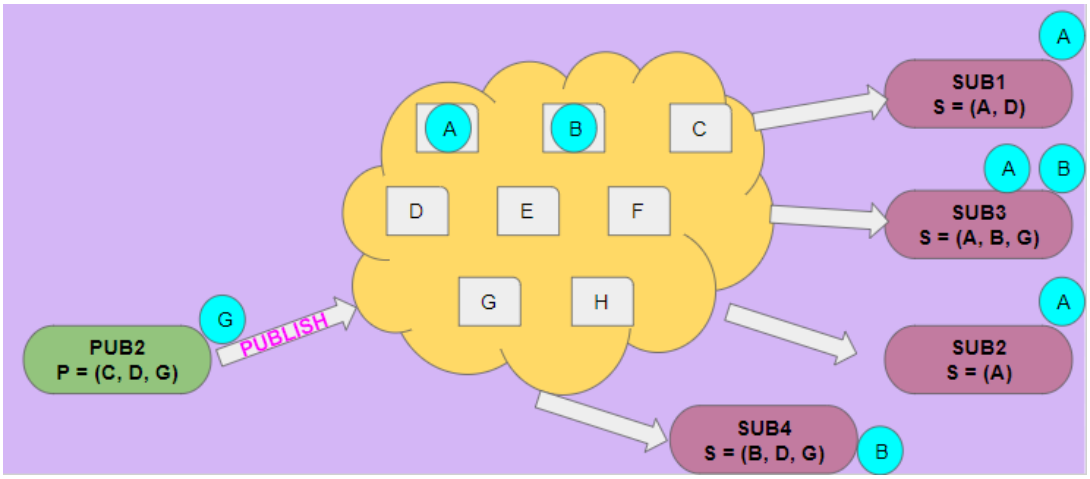

Figura 15: O PUB2 realiza o PUBLISH da mensagem G.

.

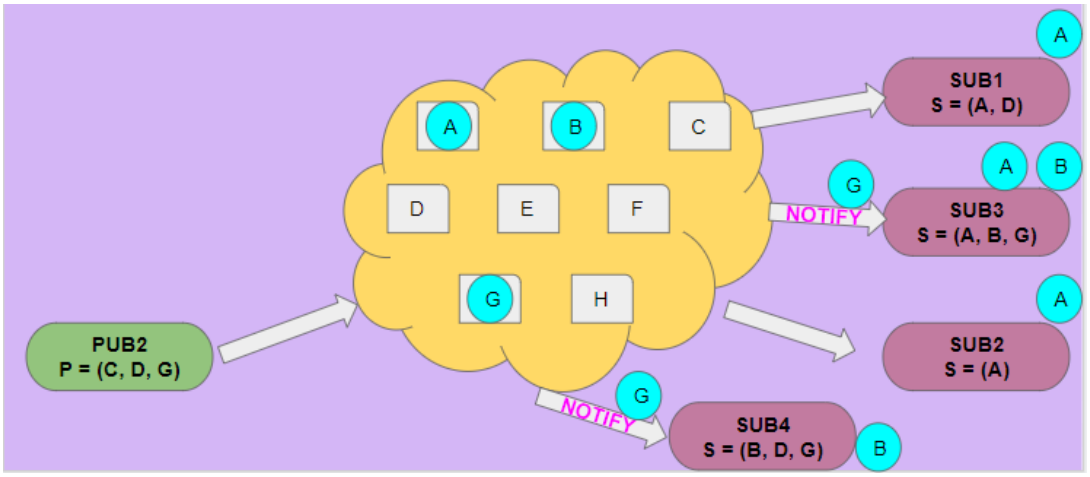

Figura 16: Tendo recebido G, o broker realiza o NOTIFY para os SUBs inscritos.

.

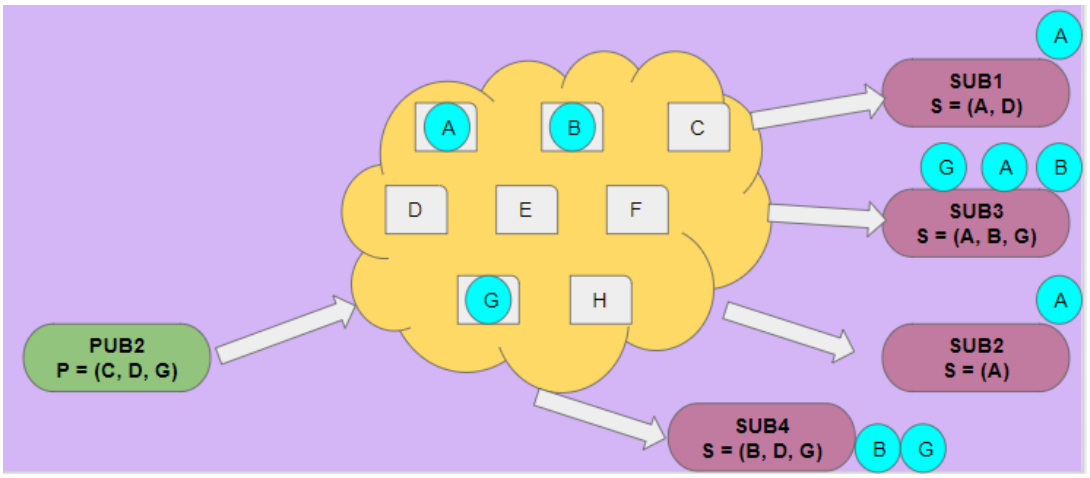

Figura 17: As mensagem G é entregue ao SUB3 e ao SUB4.

.

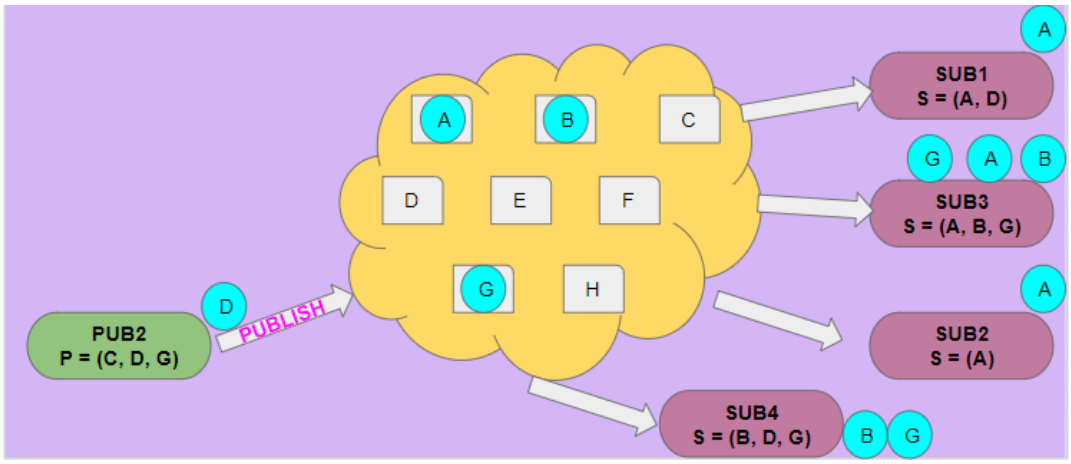

Figura 18: PUB2 realiza um novo PUBLISH, agora da mensagem D.

.

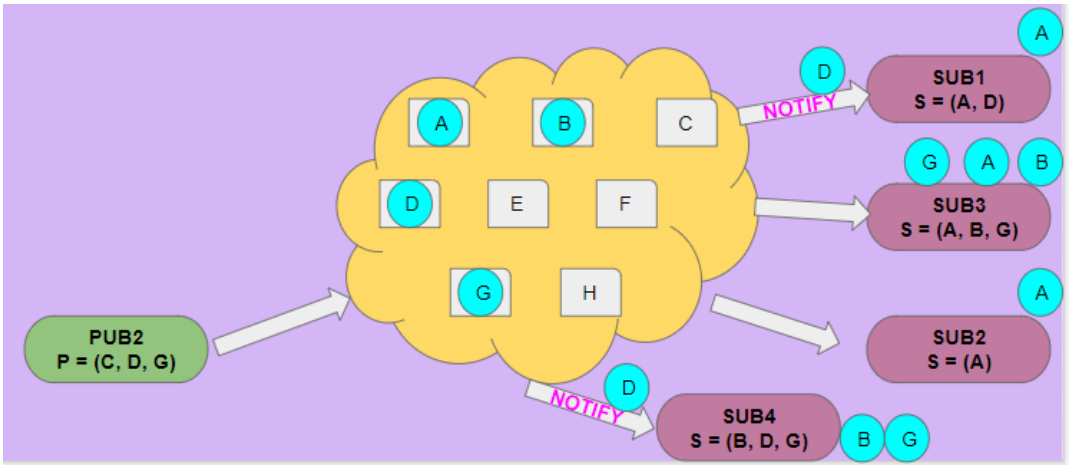

Figura 19: Mensagem D chega ao broker, que, imediatamente, realiza o NOTIFY para os inscritos SUB1 e SUB4.

.

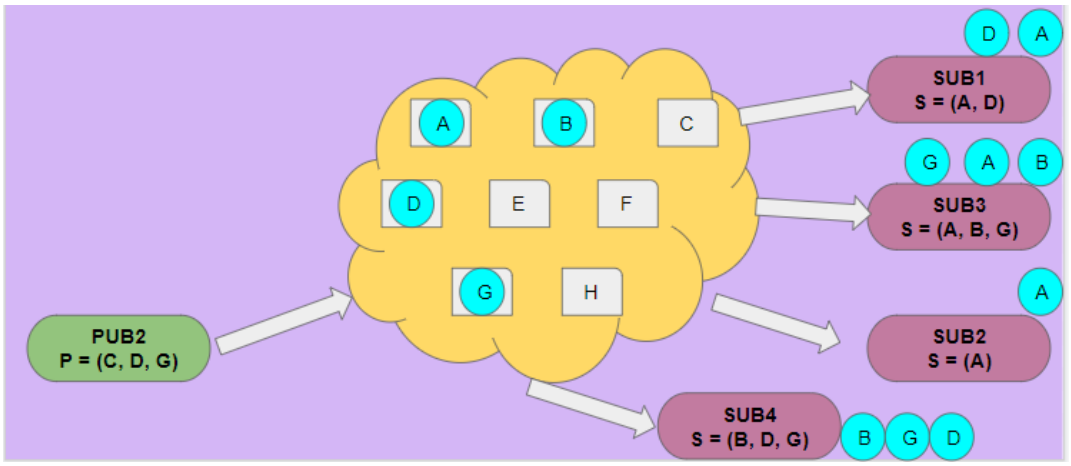

Figura 20: SUB1 e SUB4 recebem a mensagem D.

.

Ao final dessa execução demonstrada, todos os *subscribers* receberam as informações para as quais estavam inscritos.

A síntese desse exemplo de funcionamento está representada na Figura 21, uma representação mais detalhada da Figura 1 apresentada anteriormente. Nela, observe a existência de quatro *Publishers* e quatro *Subscribers*. Veja também o fluxo de informação: o *Publisher* envia ao *Broker* o evento de PUBLISH. Por sua vez, o *Broker* armazena e gerencia as subscrições, bem como as informações passadas pelo *Publisher*, enviando aos *Subscribers* as ações NOTIFY com aquilo que esses solicitaram pelas ações de SUBSCRIBE receber. Em adição a isso, os *Subscribers* também podem solicitar deixar de receber. Novamente, destaque para a não existência de interações diretas entre o *Publisher* e o *Subscriber*.

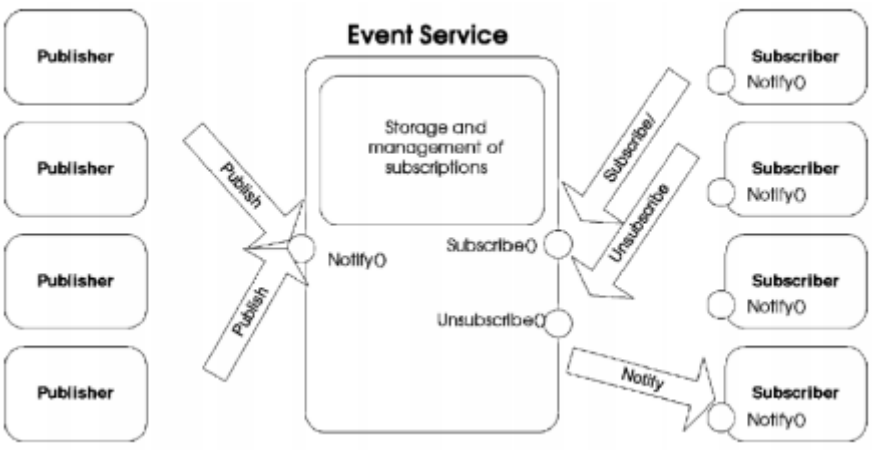

Figura 21: Ilustração de funcionamento do método Publish-Subscribe

.

2.3 Comunicação Indireta

De acordo com [Righi, 2019], são quatro os fundamentos de espaço e de tempo:

- **Acoplamento de tempo e de espaço**: a comunicação é dirigida a um determinado receptor e o receptor precisa existir quando acontece a comunicação;

- **Desacoplamento de tempo e acoplamento de espaço**: a comunicação é direta e transmissor e receptor podem existir em espaços diferentes de tempo;
- **Acoplamento de tempo e desacoplamento de espaço**: neste caso, o transmissor não conhece o receptor. Além disso, o receptor deve existir quando ocorre a transmissão;
- **Desacoplamento de tempo e de espaço**: neste caso, transmissor e receptor não se conhecem e podem até mesmo existir em diferentes espaços de tempo.

A comunicação indireta está associada ao desacoplamento de espaço e de tempo. Nela, não existe uma ligação direta entre o emissor da mensagem e o receptor (ou os receptores) dessa mensagem. Para que a comunicação seja possível, ela ocorre mediante um intermediário, conhecido como *broker* [Righi, 2019].

É nesse tipo de comunicação que o método *Publish-Subscribe* se encaixa. Conforme apresentado nas seções anteriores, esse método não apresenta uma ligação direta entre *publisher* e *subscriber*, respectivamente, emissor e receptor. Sequer é necessário que esses dois existam ao mesmo tempo, haja vista que toda a comunicação é gerenciada por um terceiro, o *broker*.

2.4 Modelos de Inscrição

Existem diferentes maneiras de especificar as notificações de interesse, e isso nos leva a formas diferentes de identificação das variantes do modelo pub/sub. Temos alguns modelos de inscrição que aparecem na literatura, todos eles possuem diferentes poderes de expressividade [Virgillito, 2003]. Esses modelos oferecem aos inscritos a possibilidade de combinar precisamente seus interesses, nesse caso, estamos falando de receber apenas as notificações que para eles importam. Entretanto, o poder de expressividade dos modelos de inscrição não está somente relacionado com a flexibilidade de interação com os clientes, mas tem forte influência sobre todo o processo de notificação. Nessa seção, serão apresentados alguns dos modelos de inscrição do pub/sub mais populares, destacando as compensações entre expressividade e facilidade de realizar implementações escalonáveis.

2.4.1 Topic-based Model

No modelo baseado em tópicos, as notificações são agrupadas em tópicos (ou assuntos), em outras palavras, um subscribe declara seu interesse por determinado tópico no qual ele tenha interesse e a partir disso ele passará a receber todas as notificações relacionadas a esse tópico. O filtro de uma assinatura é simplesmente a especificação de um tópico. Cada um dos tópicos corresponde de uma maneira lógica a um canal de evento, conectando de maneira ideal cada editor possível a todos os assinantes interessados [Virgillito, 2003]. Ou seja, existe uma associação estática entre um canal e todos os seus inscritos, então quando uma notificação é publicada, o sistema não precisa calcular todos os receptores. Esse modelo tem sido a solução adotada em todas as primeiras encarnações de pub/sub.

2.4.2 Content-based Model

Na variante baseada em conteúdo, os assinantes expressam seu interesse especificando condições sobre o conteúdo das notificações que desejam receber. Ou seja, um filtro em uma assinatura é uma consulta composta por um conjunto de restrições sobre os valores dos atributos da notificação. Restrições que podem ser impostas caso desejadas dependem do tipo de atributo e do idioma da assinatura. A maioria dos modelos de inscrição compreende operadores de igualdade e comparação, bem como expressões regulares. Normalmente as restrições podem ser unidas dentro de filtros através de expressões do tipo AND/OR [Virgillito, 2003].

2.4.3 Type-based Content

O terceiro modelo de inscrição a ser apresentado é o baseado em tipos. Essa variante aprimora o pub/sub comum com conceitos derivados da programação orientada a objetos: as notificações são declaradas como objetos pertencentes a um tipo específico, que podem, portanto, encapsular atributas, e métodos. Em relação aos modelos simples e não estruturados, os de tipo representam uma forma de dados mais robustos para o desenvolvedor de aplicativos, fornecendo algum tipo de segurança a ser verificado pelo serviço de notificação, ao invés de dentro da aplicação [Virgillito, 2003]. É válido ressaltar que nesse modelo a declaração de um tipo desejado é o principal atributo discriminante. Em outras palavras, com relação aos modelos mencionados anteriormente, o pub/sub que é baseado em tipos se posiciona de alguma maneira no meio, dando uma estrutura grosseira nas notificações, na qual as restrições podem ser expressas sobre atributos ou sobre métodos (como uma consequência da abordagem orientada a objetos), portanto, esse modelo se assemelha ao de Topic-based.

2.5 Tipos de Implementação

O sistema *Publish/Subscribe* possui como objetivo garantir que os eventos sejam entregues de maneira eficiente para todos os assinantes com os filtros correspondentes em um determinado evento que foi gerado. [Tarkoma, 2012] discorre sobre as maneiras de implementar um sistema deste tipo. Ver-se-á, a seguir, os tipos de implementação que podem ser feitas no *Publish/Subscribe*.

2.5.1 Implementação Centralizada

Nesta implementação, teremos apenas um nó que aturá como broker e fará a intermediação do evento. De maneira geral, é uma mais simples, entretanto pode apresentar alguns problemas, retratados nas Figuras seguintes.

Veja, na Figura 22, que a existência de um único *Broker* da implementação centralizada, somada à existência de diversos *Subscribers* pode acarretar gargalos de tempo. Sozinho, o *Broker* pode ter problemas para distribuir as informações aos subscritos, demorando muito tempo para isso, o que gera o problema de escalabilidade.

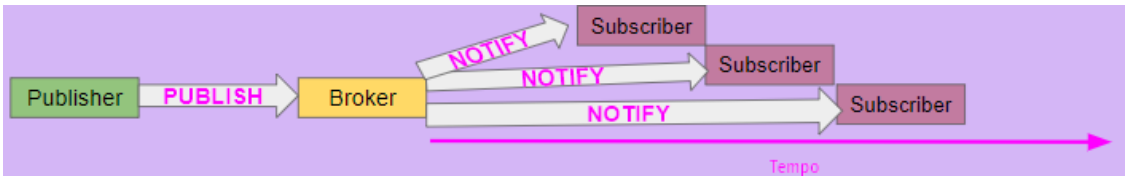

Figura 22: Um único *broker*, muitos *subscribers* → problema de escalabilidade, gargalo de tempo.

.

Veja agora, na Figura 23 que se houver um problema nesse único *Broker*, compromete-se todo o sistema, visto que o sistema é totalmente dependente dele. No exemplo da referida figura, um problema no *Broker* após a ação de PUBLISH faria com que não ocorresse a ação de NOTIFY, ou seja, os *Subscribers* não recebem a informação, neste exemplo.

2.5.2 Implementação Distribuída

Nesta implementação, há uma rede de *brokers* interligados cooperando entre si. Os nós agem como *brokers* e trabalham em cooperação para o roteamento dos eventos. Assim, se um deles apresenta falha, os outros conseguem compensá-lo. Portanto, é uma implementação menos suscetível a falhas. Veja, na Figura 24 a esquematização de funcionamento desta implementação. Note a interligação entre os *brokers* na rede de *brokers* presente no centro da imagem, em amarelo.

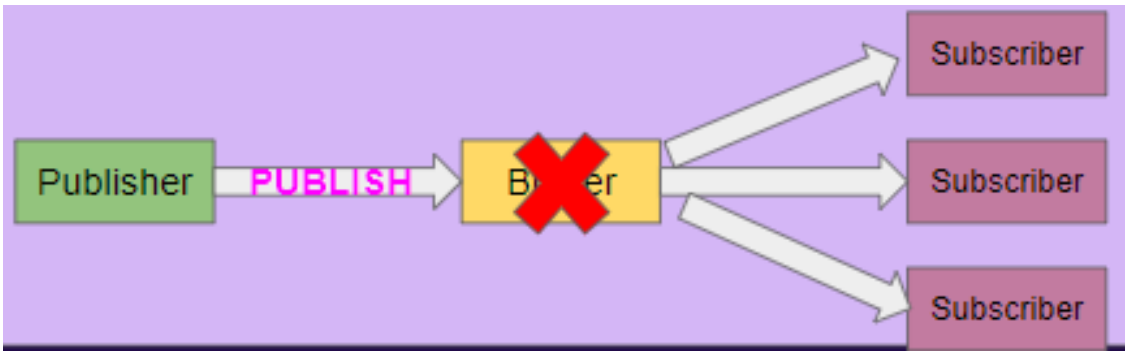

Figura 23: Se o único *broker* apresentar um problema, falha em todo o funcionamento. *Subscribers*, neste exemplo, não receberão a informação.

.

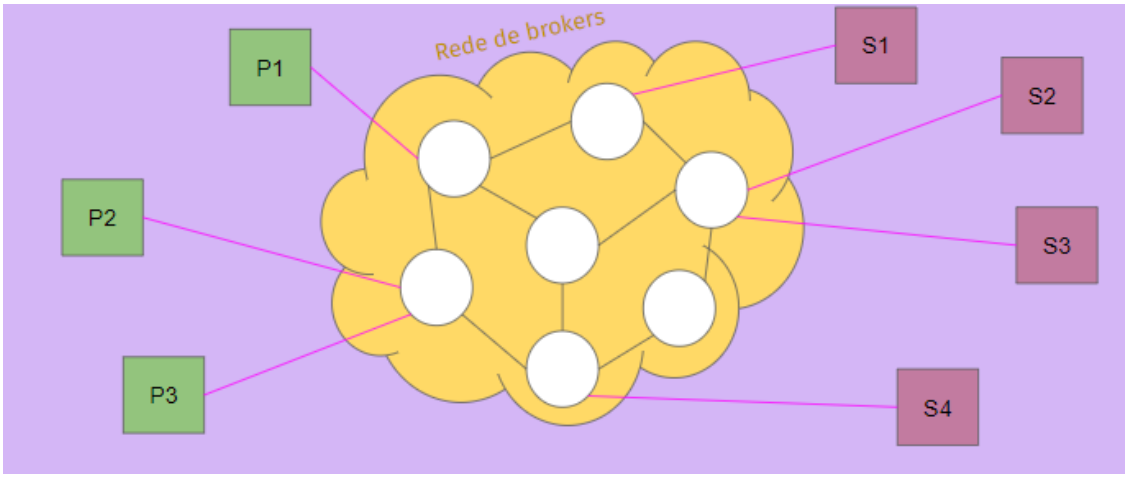

Figura 24: Implementação Distribuída.

.

2.6 Roteamento de eventos

Uma discussão importante por trás de um sistema *Publish-Subscrbe* é o roteamento de eventos. De uma maneira informal, roteamento de eventos diz respeito ao processo de entrega de um evento a todos os assinantes que emitiram uma assinatura correspondente antes da publicação. Estamos tratando de uma visita aos nós do serviço de notificação de maneira, a encontrar para qualquer evento publicado, todos os clientes cuja assinatura registrada estejam presentes no sistema no momento da publicação. Isso pode ser feito de maneiras distintas, conforme será abordado a seguir.

2.6.1 Flooding

O Flooding é um dos tipos de roteamento possíveis em implementações distribuídas. Nele, todos os nós de *broker* recebem a notificação do evento publicado e esses nós são responsáveis por identificar os subscritos para entregar a mensagem. Esse tipo de roteamento resulta em um alto tráfego [Baldoni et al., 2009]. Esse algoritmo pode ser implementado de forma simples em todas as arquiteturas: uma solução baseada em redes consiste na difusão de cada evento a todos os processos conhecidos. A desvantagem óbvia é que nesse mecanismo de roteamento não é dimensionado em termos de sobrecarga de mensagem. Entretanto, a inundação de eventos apresenta sobrecarga mínima de memória (nenhuma informação de roteamento precisa ser armazenada em um nó) e não há limitações de idioma. Também podemos especificar a solução onde temos a

inundação de assinatura: cada assinatura é enviada a todos os nós junto com o identificador do assinante. Ou seja, cada nó tem o conhecimento completo de todo o sistema, portanto os destinatários podem ser alcançados em um único salto, e os eventos não interessantes podem ser imediatamente filtrados nos publishers.

2.6.2 Flitering

No Filtering, existe uma aplicação de filtros em redes de *brokers*. Os *publishers* podem apenas transmitir quando existe um caminho válido até os assinantes. Para isso, tem-se uma função de correspondência que é responsável por tomar uma notificação de evento e uma lista de nós com inscrições e retorna um conjunto de nós onde a notificação tem correspondência com a inscrição do *subscriber* [Baldoni et al., 2009]. É importante que cada nó mantenha uma lista dos vizinhos na rede de *brokers*, uma lista de assinantes e uma tabela de roteamento.A construção de caminhos de difusão requer que as informações de roteamento sejam armazenadas e mantidas nos nós. As informações de roteamento em um nó estão associadas a cada um de seus vizinhos na sobreposição e consistem no conjunto de assinaturas que podem ser acessadas por meio desse broker. Isso permite construir caminhos reversos para assinantes seguidos por eventos. Na prática, cópias de todas as assinaturas devem ser difundidas para todos

3 Implementação

Este tópico descreve o passo a passo da implementação autoral do *Publish-Subscribe*. A implementação foi feita em quatro arquivos distintos, são eles: **servidor.py**, **cliente1.py**, **cliente2.py** e **cliente3.py**. Além deles, criou-se três arquivos txt para persistência de dados. Cada um deles corresponde a um dos clientes da aplicação e contêm as listas nas quais esses crientes estão inscritos. Para critérios de explicação da implementação, apresentam-se a seguir as características de **servidor.py** e **cliente1.py**. Os demais clientes são cópias do primeiro, e foram criados com o intuito de simular um ambiente verdadeiro, com múltiplos clientes. Faz-se necessário destacar que os clientes, nesta aplicação, correspondem aos *subscribers*. Vale destacar também que o numero de *subscribers* é escalável, isto é, pode-se criar quantos *subscribers* o usuário entender. É no arquivo **servidor.py** que estão inseridos os *publishers* e o *broker*, conforme apresentado a seguir. A Figura 25 a seguir ilustra o esquema de implementação adotado. A seguir a ela, explica-se com detalhe toda essa implementação.

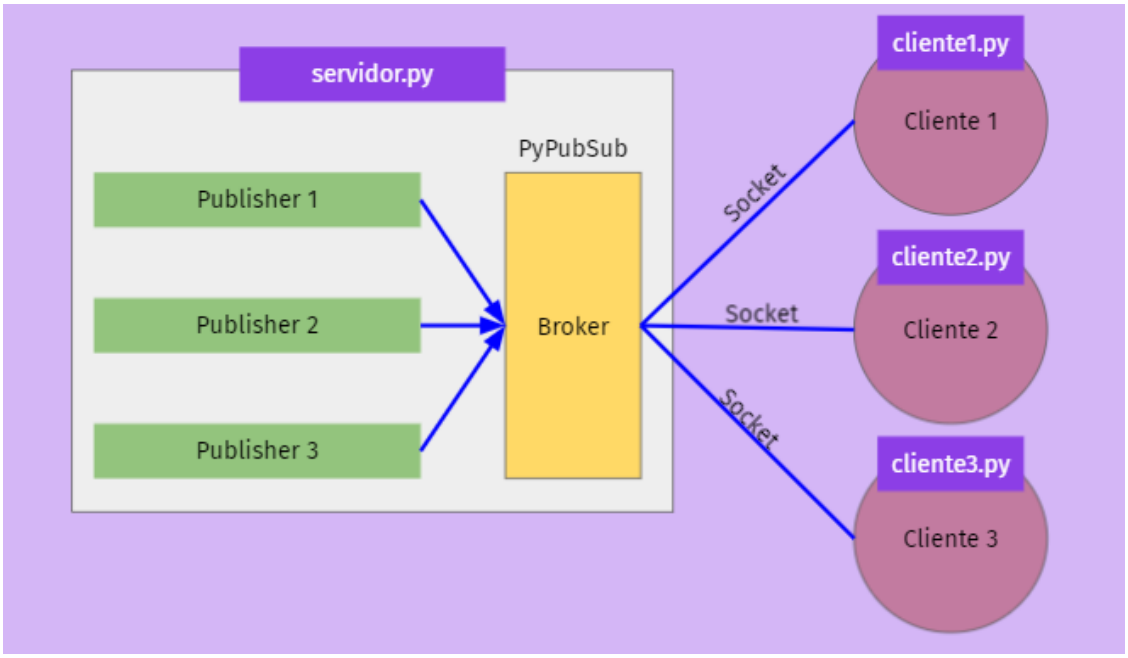

Figura 25: Esquema de implementação utilizado.

Servidor

O arquivo **servidor.py** contém as atribuições dos *Publishers* e do *Broker*. A aplicação utiliza de *sockets* para realizar o envio das mensagens e dados de comunicação entre *Publishers, Broker* e *Subscribers*. Também é utilizada a biblioteca **PyPubSub** [Schoenborn, 2019] para gerenciar toda a lógica do sistema de *Publishe-Subscribe*, fazendo o papel do *broker* na implementação.

No trecho de código a seguir, demonstra-se o trecho de código onde são importadas as bibliotecas *socket* [McMillan, 2021] e *PyPubSub* [Schoenborn, 2019]. Além disso, também demonstra-se a definição do HOST da aplicação, bem como três portas para os três clientes que serão utilizados. Caso houver o desejo aumentar o numero de *subscribers* conectados diretamente ao *broker* e indiretamente aos *publishers*, basta disponibilizar mais portas para a conexão.

```
#   Importando a biblioteca do Socket
import socket

#   Importando a biblioteca do PyPubSub
from pubsub import pub

#   Definindo o Host e a Portas
HOST = 'localhost'
PORT1 = 5001
PORT2 = 5002
PORT3 = 5003
```

A seguir, consta as listas de *publishers*. Uma para *publishers* de mensagens e outra para *publishers* de cálculo. Elas correspondem aos tópicos nos quais os *subscribers* podem se inscrever. Trata-se de uma implementação baseada em tópico, onde os *subscribers* elegem os tópicos dos quais desejam receber informações. Neste exemplo, a lista de *publishers* contém três *publishers* para mensagens: futebol, vôlei e basquete, além de três *publishers* para realização de cálculos matemáticos no servidor.

```
#   Lista de Publichers cadastrados no sistema
global listaPub
listaPub = ['Futebol', 'Volei', 'Basquete']

global listaPubCalc
listaPubCalc = ['Diametro da Circunferencia',
                'Area da Circunferencia', 'Comprimento da Circunferencia']
```

Foi criada uma classe cliente, que contém as listas nas quais os *subscribers* estão inscritos. Veja, a seguir, que ela separa a lista de tópicos e a de cálculos. Criou-se também um *socket* para cada cliente, representado pela variável **sock**.

```
# Classe cliente, contem a estrutura que cada cliente tera
class Cliente:
    def __init__(self, HOST, PORT):
        self.ListaInscricaoPub = []
        self.ListaInscricaoPubCalc = []
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.bind((HOST, PORT))
```

No trecho de código a seguir, instancia-se três objetos da classe Cliente. Eles representam, vale destacar, os *subscribers* desta aplicação.

```
#   Criando clientes
Cliente1 = Cliente(HOST, PORT1)
Cliente2 = Cliente(HOST, PORT2)
Cliente3 = Cliente(HOST, PORT3)
```

Depois de instanciados os clientes, precisa-se colocá-los em modo de escuta para que consigam ouvir as conexões em cada porta.

```
#   Colocando o socket em modo de escuta
print('\nSERVIDOR LIGADO!\n\nAguardando conexões......\n')
Cliente1.sock.listen()
Cliente2.sock.listen()
Cliente3.sock.listen()
```

Agora que os *sockets* de cada cliente estão ouvindo as portas, é preciso confirmar a conexão do *broker* com cada *subscriber*. O código a seguir se encarrega disso.

```
#   Retorno da confirmação da conexão pelo Subscriber
connexao1, endereco1 = sub1.accept()
print('Conectado com: ', endereco1)

connexao2, endereco2 = sub2.accept()
print('Conectado com: ', endereco2)

connexao3, endereco3 = sub3.accept()
print('Conectado com: ', endereco3)
```

Nesta etapa, a conexão com os *subscribers* foi concluída. Agora o *socket* está pronto para enviar as mensagens que a biblioteca PyPubSub requisitar. A seguir estão as funções que se encarregam de enviar a mensagem.

```
#   Função que envia dados para os Subscribers inscritos
def EnviaMensagemCliente1(data):
    connexao1.sendall(str.encode(data))

def EnviaMensagemCliente2(data):
    connexao2.sendall(str.encode(data))

def EnviaMensagemCliente3(data):
    connexao3.sendall(str.encode(data))
```

A função a seguir é responsável por persistir os dados de inscrição dos clientes em arquivos. Cada cliente têm um arquivo próprio onde são salvas as inscrições.

```
def PersistirDados(sub, Lista):
    #   Persistencia dos dados em arquivos
    if (sub == 1):
        arquivo1 = open("PersistenciaDados/DadosCliente1.txt", "w+")
        arquivo1.writelines(Lista)
        arquivo1.close()
    elif(sub == 2):
        arquivo2 = open("PersistenciaDados/DadosCliente2.txt", "w+")
        arquivo2.writelines(Lista)
        arquivo2.close()
    elif(sub == 3):
        arquivo3 = open("PersistenciaDados/DadosCliente3.txt", "w+")
        arquivo3.writelines(Lista)
        arquivo3.close()
```

A seguir, apresenta-se a função que inscreve o *subscriber* em algum dos *publishers*. Essa função captará a intenção do *subscriber* em realizar alguma inscrição, intenção essa manifestada diretamente por ele. Nela, verificar-se-á qual dos *subscribers* é o responsável pelo pedido de inscrição. O *broker* envia a lista de *publishers* nos quais é possível se inscrever e aguarda que o usuário devolva a opção escolhida. Em seguida, chama a função **Subscriber**, do *PyPubSub*, que realiza, de fato, a inscrição deste *subscriber* no tópico desejado. Ao realizar a inscrição, é chamado o método de persistência que salva a inscrição no arquivo, persistindo o dado.

```
# Função responsavel por inscrever o usuario em algum Publisher
def Inscrição(connexao, cliente, sub, lista):
    # Envia a lista de Publishers:
    for n in lista:
        connexao.sendall(str.encode(n))
    # Recebe Publisher escolhido pelo usuario
    MensagemRecebida = connexao.recv(1024)
    # Salva na lista o Publisher escolhido
    cliente.ListaInscricaoPub.append(MensagemRecebida.decode()+'\n')
    if(MensagemRecebida.decode() == 'Diametro da Circunferencia' or MensagemRecebida.decode(
        cliente.ListaInscricaoPubCalc.append(MensagemRecebida.decode()+'\n')

    if(sub == 1):
        PersistirDados(1, cliente.ListaInscricaoPub)
        # imprime a escolha do usuario
        print('Subscriber ', sub, ' se inscreveu no:', MensagemRecebida.decode())
```

```
        #   Função responsavel por inscrever os Subscriber nos grupos
        pub.subscribe(EnviaMensagemCliente1, MensagemRecebida.decode()+'\n')
    elif(sub == 2):
        PersistirDados(2, cliente.ListaInscricaoPub)
        # imprime a escolha do usuario
        print('Subscriber ', sub, ' se inscreveu no:', MensagemRecebida.decode())
        #   Função responsavel por inscrever os Subscriber nos grupos
        pub.subscribe(EnviaMensagemCliente2, MensagemRecebida.decode()+'\n')
    elif(sub == 3):
        PersistirDados(3, cliente.ListaInscricaoPub)
        # imprime a escolha do usuario
        print('Subscriber ', sub, ' se inscreveu no:', MensagemRecebida.decode())
        #   Função responsavel por inscrever os Subscriber nos grupos
        pub.subscribe(EnviaMensagemCliente3, MensagemRecebida.decode()+'\n')
```

A função a seguir tem o papel de enviar mensagens aos *subscribers*. Basicamente, sua ação dá-se em recolher as novas mensagens que devem ser enviadas em cada lista. Observe que o parâmetro que se refere a qual lista será enviada a mensagem é justamente **listaPub**[], que, conforme supracitado, corresponde à lista de tópicos cadastrados. A opção que o cliente passa indica se a finalidade da mensagem é um tópico ou um cálculo.

```
# Função responsavel por enviar a mensagem para cada Subscribers
def EnviarMensagem(opcao):
    #  Variavel contadora para contar as requisições de mensagem
    cont1 = 0
    cont2 = 0
    cont3 = 0
    #   Verifica se a mensagem é de texto ou é de numero para executar calculos
    #  1 para mensagens 2 para calculos
    if(opcao == 1):
        #  Laço para contar quantos Publishers o usuario esta inscrito
        for n in Cliente1.ListaInscricaoPub:
            if(n == 'Futebol'+'\n' or n == 'Volei'+'\n' or n == 'Basquete'+'\n'):
                cont1 = cont1+1
        #  Envia mensagem
        connexao1.sendall(str.encode(str(cont1)))
        for n in Cliente2.ListaInscricaoPub:
            if(n == 'Futebol'+'\n' or n == 'Volei'+'\n' or n == 'Basquete'+'\n'):
                cont2 = cont2+1
        connexao2.sendall(str.encode(str(cont2)))
        for n in Cliente3.ListaInscricaoPub:
            if(n == 'Futebol'+'\n' or n == 'Volei'+'\n' or n == 'Basquete'+'\n'):
                cont3 = cont3+1
        connexao3.sendall(str.encode(str(cont3)))
        # Recebe a mensagem e envia ao subscribers
        print('\nEscreva uma mensagem para os Subscribers do Publisher',
              listaPub[0], ':')
        data1 = input()
        pub.sendMessage(listaPub[0]+'\n', data=data1)

        print('\nEscreva uma mensagem para os Subscribers do Publisher',
              listaPub[1], ':')
```

```
        data2 = input()
        pub.sendMessage(listaPub[1]+'\n', data=data2)

        print('\nEscreva uma mensagem para os Subscribers do Publisher',
              listaPub[2], ':')
        data3 = input()
        pub.sendMessage(listaPub[2]+'\n', data=data3)
        print('\nMensagem enviada aos Subscribers!\n')

    elif(opcao == 2):
        #  Laço para contar quantos Publishers o usuario esta inscrito
        for n in Cliente1.ListaInscricaoPubCalc:
            cont1 = cont1+1
        #  Envia o numero de Publishers inscritos
        connexao1.sendall(str.encode(str(cont1)))
        for n in Cliente2.ListaInscricaoPubCalc:
            cont2 = cont2+1
        #  Envia o numero de Publishers inscritos
        connexao2.sendall(str.encode(str(cont2)))
        for n in Cliente3.ListaInscricaoPubCalc:
            cont3 = cont3+1
        #  Envia o numero de Publishers inscritos
        connexao3.sendall(str.encode(str(cont3)))
        #   Colhendo o numero do calculo
        print('\nEscreva um valor do raio para calculo ',
              listaPubCalc[0], ':')
        data1 = input()
        #   Envia o numero do calculo
        pub.sendMessage(listaPubCalc[0]+'\n', data=data1)
        #   Indica qual função utilizar
        pub.sendMessage(listaPubCalc[0]+'\n', data='1')

        print('\nEscreva um valor do raio para calculo ',
              listaPubCalc[1], ':')
        data2 = input()
        #   Envia o numero do calculo
        pub.sendMessage(listaPubCalc[1]+'\n', data=data2)
        #   Indica qual função utilizar
        pub.sendMessage(listaPubCalc[1]+'\n', data='2')

        print('\nEscreva um valor do raio para calculo ',
              listaPubCalc[2], ':')
        data3 = input()
        #   Envia o numero do calculo
        pub.sendMessage(listaPubCalc[2]+'\n', data=data3)
        #   Indica qual função utilizar
        pub.sendMessage(listaPubCalc[2]+'\n', data='3')

        print('\nMensagem enviada aos Subscribers!\n')
```

A função a seguir é responsável por desinscrever o *subscriber* do *publisher*. Primeiramente, o servidor manda aos *subscribers* os *publishers* nos quais ele está inscrito e pergunta em qual

deles deseja se desinscrever. Recebendo a resposta, o servidor aciona o *PyPubSub* que realizará a desinscrição. Além disso, realiza-se a atualização do arquivo de persistência, isto é, persiste-se a informação da desinscrição.

```
# Função responsavel por desinscrever o Subscriber
def Desinscrever(sub, cliente, connexao):
    # Envia a quantidade de Publishers o subscriber esta inscrito
    connexao.sendall(str.encode(str(len(cliente.ListaInscricaoPub))))
    # Envia a lista de Publishers o subscriber esta inscrito
    for n in cliente.ListaInscricaoPub:
        connexao.sendall(str.encode(n))
    # Recebe o dado com a informação de qual Pubisher o Subscribe quer se desinscrever
    RespMensagem = connexao.recv(1024)
    # Salva na lista
    cliente.ListaInscricaoPub.remove(RespMensagem.decode()+'\n')
    if(RespMensagem.decode() == 'Diametro da Circunferencia' or RespMensagem.decode() == 'Ar
        cliente.listaPubCalc.remove(RespMensagem.decode()+'\n')
    if(sub == 1):
        PersistirDados(1, cliente.ListaInscricaoPub)
        # Chama a biblioteca PyPubSub para executar a desinscrição
        pub.unsubscribe(EnviaMensagemCliente1, RespMensagem.decode()+'\n')
        print('Subscriber ', sub, 'se desinscreveu do:',
              RespMensagem.decode())
    elif(sub == 2):
        PersistirDados(2, cliente.ListaInscricaoPub)
        # Chama a biblioteca PyPubSub para executar a desinscrição
        pub.unsubscribe(EnviaMensagemCliente2, RespMensagem.decode()+'\n')
        print('Subscriber ', sub, 'se desinscreveu do:',
              RespMensagem.decode())
    elif(sub == 3):
        PersistirDados(3, cliente.ListaInscricaoPub)
        # Chama a biblioteca PyPubSub para executar a desinscrição
        pub.unsubscribe(EnviaMensagemCliente3, RespMensagem.decode()+'\n')
        print('Subscriber ', sub, 'se desinscreveu do:',
              RespMensagem.decode())
```

O trecho de código a seguir refere-se à rotina de execução do servidor. Toda vez que o servidor é inicializado, faz-se necessário recuperar os dados persistidos. Isso é necessário para recuperar o estado anterior do sistema. Além disso, para cada um dos *subscribers* cadastrados (no trecho a seguir deixou-se apenas um), a rotina espera por uma resposta do *subscriber*, que pode ser tanto a de esperar uma resposta, ou seja, uma atualização, quanto a de se inscrever em uma lista ou se desinscrever. O trecho se repete para cada *subscriber* cadastrado no sistema.

```
#   Lendo o arquivo e aplicando os dados persistidos caso haja.
#   Restaurando os dados persistidos
arquivo1 = open("PersistenciaDados/DadosCliente1.txt", "r")
arquivo2 = open("PersistenciaDados/DadosCliente2.txt", "r")
arquivo3 = open("PersistenciaDados/DadosCliente3.txt", "r")

conteudo1 = arquivo1.readlines()
conteudo2 = arquivo2.readlines()
conteudo3 = arquivo3.readlines()
```

```
Cliente1.ListaInscricaoPub = conteudo1
Cliente2.ListaInscricaoPub = conteudo2
Cliente3.ListaInscricaoPub = conteudo3

for n1 in conteudo1:
    pub.subscribe(EnviaMensagemCliente1, n1)
    if(n1 == 'Diametro da Circunferencia'+'\n' or n1 == 'Area da Circunferencia'+'\n' or n1
        Cliente1.ListaInscricaoPubCalc.append(n1)

for n2 in conteudo2:
    pub.subscribe(EnviaMensagemCliente2, n2)
    if(n2 == 'Diametro da Circunferencia'+'\n' or n2 == 'Area da Circunferencia'+'\n' or n2
        Cliente2.ListaInscricaoPubCalc.append(n2)

for n3 in conteudo3:
    pub.subscribe(EnviaMensagemCliente3, n3)
    if(n3 == 'Diametro da Circunferencia'+'\n' or n3 == 'Area da Circunferencia'+'\n' or n3
        Cliente3.ListaInscricaoPubCalc.append(n3)

arquivo1.close()
arquivo2.close()
arquivo3.close()


# Rotina de execução do servidor.
while True:
    # ouvindo os Subscribers para saber se eles querem se inscrever ou desinscrever
    resposta1 = connexao1.recv(1024)
    if resposta1.decode() == '1':
        EnviarMensagem(1)
    elif resposta1.decode() == '2':
        EnviarMensagem(2)
    elif resposta1.decode() == '3':
        Inscrição(connexao1, Cliente1, 1, listaPub)
    elif resposta1.decode() == '4':
        Inscrição(connexao1, Cliente1, 1, listaPubCalc)
    elif resposta1.decode() == '5':
        Desinscrever(1, Cliente1, connexao1)

    resposta2 = connexao2.recv(1024)
    if resposta2.decode() == '3':
        Inscrição(connexao2, Cliente2, 2, listaPub)
    elif resposta2.decode() == '4':
        Inscrição(connexao2, Cliente2, 2, listaPubCalc)
    elif resposta2.decode() == '5':
        Desinscrever(2, Cliente2, connexao2)

    resposta3 = connexao3.recv(1024)
    if resposta3.decode() == '3':
        Inscrição(connexao3, Cliente3, 3, listaPub)
```

```
    elif resposta3.decode() == '4':
        Inscrição(connexao3, Cliente3, 3, listaPubCalc)
    elif resposta3.decode() == '5':
        Desinscrever(3, Cliente3, connexao3)

#   Fecha as conexao do socket
connexao1.close()
connexao2.close()
connexao3.close()
```

Cliente

A seguir, descreve-se o código dos arquivos **cliente1.py**, **cliente2.py** e **cliente3.py**. Destaca-se, mais uma vez, que o **cliente** desta aplicação corresponde ao *subcriber* e que utilizou-se aqui três, mas que basta uma adaptação para inserção de quantos mais forem necessários.

O código começa com o *subscriber* estabelecendo a conexão com o servidor. A lógica é a mesma do servidor para o *subscriber*, apresentada anteriormente.

```
#   Importando a biblioteca do Socket
import socket

#   Importando biblioteca de matematica para o calculo
import math

#   Definindo meu Host e porta
HOST = 'localhost'
PORT = 5001

#   Criando o objeto socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

print('\nConectando.......\n')

#   Pedindo a conexao ao Broker
s.connect((HOST, PORT))

print('Conexão concluida!\n')
```

Após efetuar a conexão com o servidor, é preciso inscrever o cliente a algum *Publisher*, a função a seguir é responsável por fazer essa tarefa. Primeiramente, ela recebe os dados dos *Publishers* disponíveis para inscrição. Depois, ela apresenta as opções ao cliente e logo em seguida recolhe a informação de escolha do cliente e manda para o servidor. O servidor tornara esse cliente um subscriber de um dos Publishers escolhido.

```
def Inscrição():
    #   Recebendo a lista de Publishers
    cont = 0
    Publishers = []
    while cont < (3):
        value = s.recv(1024)
        Publishers.append(value.decode())
        cont += 1
```

```
    #   Imprimindo a lista de Publishers para o Subscriber escolher:
    print('\nPublishers cadastrados:')
    for n in Publishers:
        print('->', n)

    #   Perguntar oa Subscriber qual Publisher ele quer se inscrever
    print('Qual Publisher voce quer se inscrever?')
    resposta = input()

    #   Envia a resposta
    print('\nInscrição concluida em: ', resposta)
    s.sendall(str.encode(resposta))
```

Neste trecho, foi criada uma lista chamada "inscricoes"que armazena os *publishers* em que o *subscriber* está inscrito.

```
#   Lista de inscrições do Subscriber
inscricoes = ['']
```

A seguir, apresenta-se a função responsável por receber respostas do *broker*. Esta função comunica-se diretamente com o **broker**. É através dela que a ação de NOTIFY, direcionada do *broker* ao *subscriber*, é efetivamente aplicada.

```
#   Função responsavel por receber a mensagem do Broker
def RecebeMensagem():
    i = 0
    numEscricao = s.recv(1024)
    while i < int(numEscricao):
        i = i+1
        #   Recebendo resposta da comunicacao
        data = s.recv(1024)
        print('\nMensagem recebida: ', data.decode(), '\n')
```

A função a seguir é a responsável por enviar do *subscriber* ao *broker* a intenção do primeiro em se desinscrever em determinada lista. Para isso, verifica-se em quantas e quais listas o cliente (*subscriber*) está inscrito, elas são impressas na tela e solicita-se a informação de qual delas será feita a ação de *unsubscribe*. O *broker* receberá essa informação e, conforme já exposto anteriormente, retirará este *subscriber* da lista em questão. Por fim, nesta função que se segue, o *subscriber* apaga sua lista de inscrições para receber do *broker* a lista atualizada, já com a desinscrição consolidada.

```
#   Função responsavel por enviar a requisição de desinscrição do Publisher
def Desinscrever():
    numinscricoes = s.recv(1024)
    cont = 0
    while cont < (int(numinscricoes.decode())):
        value = s.recv(1024)
        inscricoes.append(value.decode())
        cont += 1

    #   Imprimindo a lista:
    print('\nVocê está inscrito nos Publichers:')
```

```
    for n in inscricoes:
        print(n)

    #   Perguntar oa usuario qual Publisher ele quer se desinscrever
    print('\nQual Publisher voce quer se desinscrever?')
    resposta = input()

    #   Envia a resposta
    print('\nDesinscrito do Publisher: ', resposta, '\n')
    s.sendall(str.encode(resposta))

    #   Limpa a lista aqui para receber uma nova lista atualizada do Broker
    inscricoes.clear()
```

Conforme supracitado, o servidor também possui a capacidade de realizar cálculos matemáticos. Para concretização do *Publish-Subscribe*, há também no cliente uma função responsável por receber os resultados desses cálculos. Ela se comunica com o servidor e, usando o parâmetro que indica em qual lista de cálculos o cliente está inscrito, é capaz de receber o resultado, conforme apresenta-se no código a seguir.

```
#   Recebe os calculos e processa ele
def RecebeCalculo():
    i = 0
    numEscricao = s.recv(1024)
    while i < int(numEscricao):
        i = i+1
        #   Recebendo resposta da comunicacao
        data = s.recv(1024)
        funcao = s.recv(1024)
        print('\nCalculo Recebido: ', data.decode(), '\n')
        print('Calculando......')
        #   Calcula Diametro
        if(funcao.decode() == '1'):
            diametro = 2*int(data.decode())
            print('Calculo do Diametro da Circunferencia executado, valor= ', diametro)
        #   Calcula da Área
        elif(funcao.decode() == '2'):
            area = math.pi*int(data.decode())
            print('Calculo da Area da Circunferencia executado, valor= ', area)
        #   Calcula da Comprimento
        elif(funcao.decode() == '3'):
            comprimento = 2*math.pi*int(data.decode())
            print(
                'Calculo do Comprimento da Circunferencia executado, valor= ', comprimento)
```

O trecho de código a seguir refere-se à rotina de execução do *subscriber*. A rotina exibe um menu para o usuário e logo em seguida envia uma resposta do *subscriber* para o servidor, que pode ser tanto a de esperar uma resposta, ou seja, uma atualização, quanto a de se inscrever em uma lista ou se desinscrever.

```
# Rotina de execução do Subscriber
while True:
```

```
#   Lendo opção que o Subscriber deseja fazer:
print('\n----Menu Subscribe ----\n')
print('Digite 1 para receber mensagens')
print('Digite 2 para receber calculos')
print('Digite 3 para se inscrever para receber mensagens')
print('Digite 4 para se inscrever para receber calculos')
print('Digite 5 para se desinscrever')
print('Digite 6 para sair')
opcao = input()
#   Comunica o Broker da intenção do Subscriber
s.sendall(str.encode(opcao))
# Executa as funç˜eos de acordo com o escolhido pelo Subscriber
if(opcao == '1'):
    RecebeMensagem()
elif(opcao == '2'):
    RecebeCalculo()
elif(opcao == '3'):
    Inscrição()
elif(opcao == '4'):
    Inscrição()
elif(opcao == '5'):
    Desinscrever()
else:
    s.close()
    break
```

4 Conclusão

O método *Publish-Subscribe* é um método eficiente para distribuição de informação em Sistemas Distribuídos. Conforme apresentado ao longo deste trabalho, possui diversas características que lhe possibilitam a correta distribuição de informações do publicador ao subscrito, em um gerenciamento eficiente de caminho da informação.

Não a toa, esse método é empregado em diversos serviços na internet. Um dos mais proeminentes deles é o Google Notícias. Presente em ambiente *desktop* e em aplicação *mobile*, este serviço do Google, utilizando a conta Google do usuário, permite-lhe a inscrição em serviços de notícias e em tópicos. O usuário pode assinalar, por exemplo, sua preferência por receber informações de determinado jornal que gosta e de seu esporte favorito, e recebe-as diretamente na aplicação do Google. Por exemplo, um torcedor de determinado time de futebol pode subscrever-se para receber notícias desse time, e a aplicação do Google realiza esse papel de *broker* entre os serviços de notícias que publicaram determinada notícia do time - portanto, *publishers* - com esse usuário, o *subscriber*. A notícia postada pelo jornal chega nas mãos desse torcedor utilizando o conceito do *Publish-Subscribe*.

Outro exemplo prático de aplicação de *Publish-Subscribe* é o RSS (*Rich Site Summary*). Ele permite que os utilizadores se inscrevam em feeds de sites de notícias ou blogs. O usuário, então, recebe atualizações das listas que se inscreveu sem ter a necessidade de acessar as páginas originais. Diversos provedores de notícias fornecem aos seus usuários a opção de utilização do RSS. Um exemplo disso é o site de notícias G1. Na Figura 26, ao lado, apresenta-se a página desse portal de notícias onde constam os links para seus usuários se inscreverem em listas de RSS por assunto, para receberem suas notícias via RSS.

Figura 26: Página de inscrição em feeds RSS do G1.

O modelo *Publish-Subscribe*, na busca pela entrega rápida e eficiente das mensagens, apresenta diversos benefícios, mas não está imune a problemas.

Começa-se citando alguns dos problemas que esse padrão apresenta. O primeiro deles refere-se à segurança: a conexão com qualquer canal de mensagem deve ser restrita por política de segurança, para que, dessa maneira, a interceptação por usuários ou aplicativos não autorizada possa ser impedida. Podem ocorrer problemas com relação a assinantes coringa, onde deve-se analisar a possibilidade de permitir que assinantes se inscrevam em vários tópicos por meio de curingas. Mensagens suspeitas são outro ponto a ser discutido, uma mensagem mal formada ou apenas uma tarefa que exige acesso a recursos que não estão disponíveis, pode causar alguma falha em uma instância de serviço. E por último podemos discutir sobre as mensagens repetidas, a mesma mensagem pode ser enviada mais de uma vez, isso pode causar problemas no tráfego das mensagens e sobrecarga dos serviços.

Após a apresentação de problemas, pode-se discorrer também sobre a ampla gama de benefícios que esse padrão apresenta. Um desses benefícios é a escalabilidade. Sistemas deste tipo são escalonáveis, ou seja, não perdem sua funcionalidade com o acréscimo ou decréscimo de novos *publishers* e *subscribers*. Essa característica foi demonstrada no exemplo citado no tópico 2.2 deste relatório, quando foi demonstrado um exemplo de funcionamento do padrão. Na Figura 8, por exemplo, um novo *subscriber* foi adicionado, sem prejuízo ao sistema. Somada à escalabilidade vem a confiabilidade. Este sistema traz essa vantagem ao permitir suporte à entrega de mensagens assíncronas, o que garante entrega confiável das mensagens mesmo com alterações na arquitetura do tópico. Por fim, destaca-se aqui o acoplamento fraco entre os componentes do sistema. Como este sistema permite o isolamento dos componentes, isso o torna robusto e mantém seu código simples para manutenção.

Diante de tudo o que foi exposto ao longo deste relatório, conclui-se que o método *Publish-Subscribe* é um método eficiente de comunicação em Sistemas Distribuídos e possui aplicabilidade funcional no mundo real, demonstrada em seu uso por agregadores de notícias e feeds RSS. Demonstrou-se ainda que há um leque de possibilidades de implementação desse método e que, dessa forma, ele pode ser adaptado às necessidades do sistema que pretende usá-lo. Encerra-se este relatório concluindo que os objetivos deste minicurso foram alcançados.

Referências

[Baldoni et al., 2009] Baldoni, R., Querzoni, L., Tarkoma, S., and Virgillito, A. (2009). *Distributed Event Routing in Publish/Subscribe Communication Systems.*

[Coulouris et al., 2007] Coulouris, G., Dollimore, J., and Kindberg, T. (2007). *Sistemas Distribuídos - Conceitos e Projetos*, volume 4. Bookman.

[McMillan, 2021] McMillan, G. (2021). Howto sobre a programação de soquetes. Disponível em: `https://docs.python.org/pt-br/3/howto/sockets.html`. Acesso em: 10 mai. 2021.

[Righi, 2019] Righi, R. (2019). Sistemas distribuídos: notas de aula. Disponível em: `https://www.dropbox.com/s/p1n8cj5pfazws9w/aula4-sistemas.pdf?dl=0`. Acesso em: 04 mai. 2021.

[Schoenborn, 2019] Schoenborn, O. (2019). Pypubsub v4.0.3. Disponível em: `https://pypubsub.readthedocs.io/en/v4.0.3/`. Acesso em: 10 mai. 2021.

[Tarkoma, 2012] Tarkoma, S. (2012). *Publish/Subscribe systems: design and principles.* John Wiley Sons.

[Virgillito, 2003] Virgillito, A. (2003). *Publish/Subscribe Communication Systems: from Models to Applications.*

A Apêndice 1 - GitHub

Os códigos desta aplicação demonstrada neste relatório, juntamente de um tutorial de execução do mesmo, estão disponíveis para acesso no GitHub. O acesso é feito a partir do seguinte link: `https://github.com/Rav98/Publisher-Subscribe`.

B Apêndice 2 - Vídeo explicativo

Um vídeo de explicação deste método, em conjunto com a demonstração de execução desta aplicação pode ser encontrado neste link: `https://youtu.be/nizucEyP6ws`.