
Universidade Federal do Ceará
Campus Quixadá
Engenharia de Computação
Sistema Distríbuidos

# Sistema de agenda de eventos

**Equipe:**
Marianna de Pinho Severo
Marisa do Carmo Silva

**Professor:**
Marcio Espíndola Freire Maia

Junho de 2018
Quixadá - Ceará

# Sumário

# 1 Introdução

Este relatório apresenta a documentação para o *Sistema de agenda de eventos*, em que o objetivo do trabalho é conseguir agendar eventos em dias e horários especificados, como também adicionar os participantes destes. Apresentaremos quais são as funcionalidades do sistema, arquitetura e protocolo de interação utilizados. E os desafios escolhidos para serem tratados neste trabalho.

## 1.1 Descrição do Sistema e Funcionalidades

Um usuário do sistema necessita realizar o cadastro no sistema para que possa adicionar um novo evento, devendo escolher a opção **Cadastrar**. Caso o usuário já tenha um cadastro de participante, este deverá fazer o login para conseguir utilizar as funcionalidades do sistema (opção **Entrar**). Dessa forma, sempre que um usuário executa o cliente, as primeiras funcionalidades a que tem acesso são as de Cadastrar e Entrar no sistema.

As opções possíveis para um participante são:

- **Cadastrar evento por data e hora:** Participantes podem cadastrar um novo evento, definindo o nome deste, data e o horário em que acontecerá.
- **Adicionar participante por evento:** O participante poderá se adicionar à lista de participantes de um determinado evento.
- **Listar eventos por dia:** Listará todos os eventos que acontecerão em uma determinada data.
- **Listar participantes por evento:** Listará todos os participantes de um determinado evento.
- **Remover-se do evento:** O participante irá se remover da lista de participantes de um determinado evento.
- **Fazer checkin no evento:** Confirmará sua ida ao evento, entrando para uma lista de presentes.
- **Listar presentes no evento:** Listará todos os participantes que realizaram checkin no evento.

Entendemos que é responsabilidade de cada participante poder se adicionar em um determinado evento. Como a agenda é pessoal, cada pessoa deve decidir participar ou não de um evento, não podendo ser adicionada por outras. Também é responsabilidade de um determinado participante fazer checkin (comparecendo ou não) e ser capaz de remover-se da participação de um evento, não podendo ser removido por outrem.

# 2 Módulos

Utilizamos a linguagem de programação Python para a implementação dos módulos. A arquitetura implementada é a cliente-servidor, ou seja temos dois módulos: um cliente e um servidor.

## 2.1 Arquitetura cliente-servidor

Em nossa arquitetura, temos os módulos cliente e servidor. O cliente é responsável por fornecer a interface de acesso ao usuário, realizando as requisições ao servidor. Este, por sua vez, processa a requisição - que pode ser qualquer uma das funcionalidades do sistema descritas anteriormente - e retorna a resposta, que pode ser de falha, sucesso e os dados pedidos.

### 2.1.1 Cliente

No cliente, temos os seguintes módulos:

- **Serialização:** a função chamada é a *serializeData* e ela é responsável por transformar todos os dados, recebidos dos outros métodos do cliente, para um formato de representação JSON, para que possam ser enviados. Para fazer isso, ele identifica qual serviço foi solicitado e, então, realiza a serialização dos dados correspondentes.
- **Desserialização:** a função *deserializeData* é chamada e ela é responsável por transformar a mensagem enviada pelo servidor - que está no formato JSON - em uma string de *status*, que informa se a operação falhou ou teve succeso, e uma string de *resposta*, que mostra os dados retornados, caso a operação tenha tido sucesso, ou o tipo de erro, caso ela tenha falhado.

- **Autenticação:** responsável por pedir o *username* e a senha do usuário, seja para entrar no sistema ou para realizar o cadastro. Primeiramente, uma tela de login é mostrada, com as opções **Entrar** e **Cadastrar**. Quando o usuário escolhe uma dessas opções, um método chamado *loginClient* é executado. Ele pede o *nome de usuário* e a *senha* e, ao receber, serializa esses dados e os envia ao servidor. Se o servidor retornar uma resposta de *sucesso*, o usuário pode utilizar os serviços do sistema. Se a resposta for de *falha*, isso é avisado ao usuário e a tela de login é mostrada novamente, para que ele possa tentar outra vez.

- **Registrar Evento:** uma vez tendo entrado no sistema, o usuário pode escolher cadastrar um novo evento. Se esse serviço for socilitado, um método chamado *registerEvent* é executado. Ele pede ao usuário o *nome*, a *data* e o *horário* do evento, realiza a serialização e envia o resultado para o servidor. Caso um evento com o mesmo nome já exista, o cadastro não é realizado.

- **Adicionar participante a evento:** escolhendo essa opção, um método chamado *client2Event* é chamado. Ele solicita o *nome do evento*, serializa essa informação, juntamente com o *username* do usuário que solicitou esse serviço, e os envia ao servidor. Caso o evento não esteja cadastrado ou o usuário já esteja como participante do evento, essa operação não é realizada.

- **Listar participantes de um evento:** o método *listPartEvent* é chamado e recebe como entrada o *nome do evento*. Caso haja participantes, eles são retornados, juntamente com um status de sucesso. Caso contrário, uma mensagem indicando que não há participantes, juntamente com um status de falha, é recebida. Caso o evento não esteja cadastrado no sistema, uma mensagem de falha também é retornada pelo servidor.

- **Listar presentes em um evento:** assim como listar participantes, o método responsável - nesse caso, *listPresEvent* - recebe o nome do evento. Caso o evento não exista, ou não haja participantes presentes, uma mensagem de falhada é recebida. Se houver presentes, uma mensagem de sucesso, juntamente com a lista de presentes, é retornada pelo servidor.

- **Listar eventos em um dia:** se essa opção for escolhida, o método *listEventDay* é chamado e ele recebe como entrada a data que queremos verificar. Ela deve ser fornecida no formato **DD-MM-YY**. Se não houver eventos cadastrados nesse dia, uma mensagem de falha é retornada. Caso contrário, a lista de eventos cadastrados é enviada pelo servidor.

- **Remover participantes de evento:** essa opção chama o método *removePart*, que recebe como entrada o *nome do evento*. Este e o username do usuário que solicitou o serviço são enviados ao servidor. Caso o evento não exista ou o usuário não seja participante, uma falha é retornada. Pelo contrário, o usuário é removido.

- **Realizar checkin:** essa opção executa um método chamado *doCheckin*, que recebe como entrada o *nome do evento* e o serializa, juntamente com o username do usuário que solicitou esse serviço, enviando-os ao servidor. Se o evento não estiver cadastrado ou o usuário não for participante daquele evento, uma mensagem de falha é retornada. Caso contrário, o usuário é adicionado à lista de presentes.

- **Logout:** por último, o usuário pode escolher sair do sistema, realizando logout. Então, a tela de login volta a ser mostrada.

Após a execução de qualquer um dos métodos, a tela com todos os serviços é mostrada, assim como o resultado da operação solicitada. Entretanto, se a opção *doLogout* tiver sido a escolhida, a tela mostrada passa a ser a de *login*. Todos os dados enviados ao servidor são serializados, assim como todos os que são recebidos são desserializados.

### 2.1.2 Servidor

No servidor, temos os seguintes módulos:

- **Serialização:** a função chamada é a *serializeData*. Ela é responsável por transformar o *status* da operação e a *resposta* do servidor para o formato JSON.

- **Autenticação:** ao receber uma requisição do usuário, solicitando a autenticação de dados, o servidor busca em um arquivo, que contém todos os usuários, o *username* e a *senha* recebidos. Se esses dados forem encontrados, caso o serviço solicitado tenha sido *Entrar* no sistema, uma mensagem de sucesso é retornada; caso o serviço tenha sido de *Cadastrar*, uma resposta de falha é enviada, pois não podem existir dois usuários com o mesmo *username*. Caso o usuário não seja encontrado, se o serviço solicitado tiver sido *Entrar*, uma mensagem de falha é retornada; caso tenha sido de *Cadastro*, o novo usuário é adicionado ao arquivo e uma resposta de sucesso é retornada.

- **Registrar Evento:** caso esse tenha sido o serviço solicitado, o método *registerEvent* é chamado. Ele verifica se o evento já está cadastrado no sistema, lendo de um arquivo que contém todos os eventos cadastrados. Se estiver, uma mensagem de falha é retornada. Caso não esteja, o evento é adicionado ao arquivo e uma resposta de sucesso é enviada ao cliente.

- **Adicionar participante a evento:** para tratar essa requisição, o método *client2Event* é chamado. Ele lê um arquivo que contém todos os eventos cadastrados, procurando o evento passado como parâmetro. Uma vez encontrado, ele verifica se o usuário já não está associado ao evento, verificando o *username* recebido. Se o usuário já estiver associado ou o evento não for encontrado, uma mensagem de falha é retornada. Caso o evento seja encontrado, mas o usuário ainda não esteja associado, o evento é atualizado - adicionando-se o nome do usuário, que agora é participante - e salvo no arquivo. Além disso, uma resposta de sucesso é enviada para o cliente.

- **Listar participantes de um evento:** para essa requisição, o método *listParticipants* é chamado. Ele procura, no arquivo, o nome do evento recebido. Se ele encontrar, ele verifica se há participantes cadastrados. Se houver, ele retorna uma mensagem de sucesso, juntamente com a lista de participantes. Se não houver participantes ou o evento não for encontrado, uma mensagem de falha é enviada ao cliente.

- **Listar presentes em um evento:** o método *listPresEvent* é chamado. Da mesma forma que *listar participantes*, ele procura pelo nome do evento no arquivo que contém todos. Se ele for encontrado, ele verifica se há participantes do evento que estiveram presentes no evento. Se houver, ele retorna uma mensagem de sucesso, juntamente com a lista de presentes. Se não houver presentes ou o evento não for encontrado, ele retorna uma mensagem de falha.

- **Listar eventos em um dia:** o servidor chama o método *listEventDay*. Ele percorre todos os eventos armazenados no arquivo de persistência, capturando aqueles que estão cadastrados com a data recebida através da requisição. Após percorrer todos os eventos, ele retorna aqueles que estavam associados à data recebida. Se não houver eventos cadastrados na data recebida, o servidor retorna uma mensagem de falha.

- **Remover participantes de evento:** para tratar essa requisição, o servidor chama o método *removeParts*, o qual procura pelo nome do evento no arquivo de persistência. Uma vez encontrado o evento, ele procura pelo nome do participante dentro desse evento. Se o participante for encontrado, ele é removido, o evento é atualizado e uma mensagem de sucesso é enviada. Caso o participante ou o evento não sejam encontrados, uma mensagem de falha é retornada.

- **Realizar checkin:** para tratar essa requisição, o servidor chama o método *doCheckin*, o qual procura pelo evento. Se encontrado, ele verifica se o usuário já é um participante do evento. Se ele for, seu nome também é adicionado à lista de presentes e o evento é atualizado no arquivo de persistência. Se o usuário não for participante, uma mensagem de falha é enviada. Caso o evento não seja encontrado, o servidor também retorna uma resposta de falha.

Todos os dados recebidos através das requisições são desserializados, para então ser tratados. E todas as respostas a serem enviadas aos usuários são serializadas, conforme descrito acima.

### 2.1.3 Interação entre módulos

Todas as mensagens trocadas entre clientes e servidores contêm a *operação*, que é o serviço solicitado, e os parâmetros correspondentes a cada tipo de serviço.

Na Figura 1 está apresentado o diagrama de classes do nosso sistema com os dois módulos e suas funcionalidades que foram descritas acima.

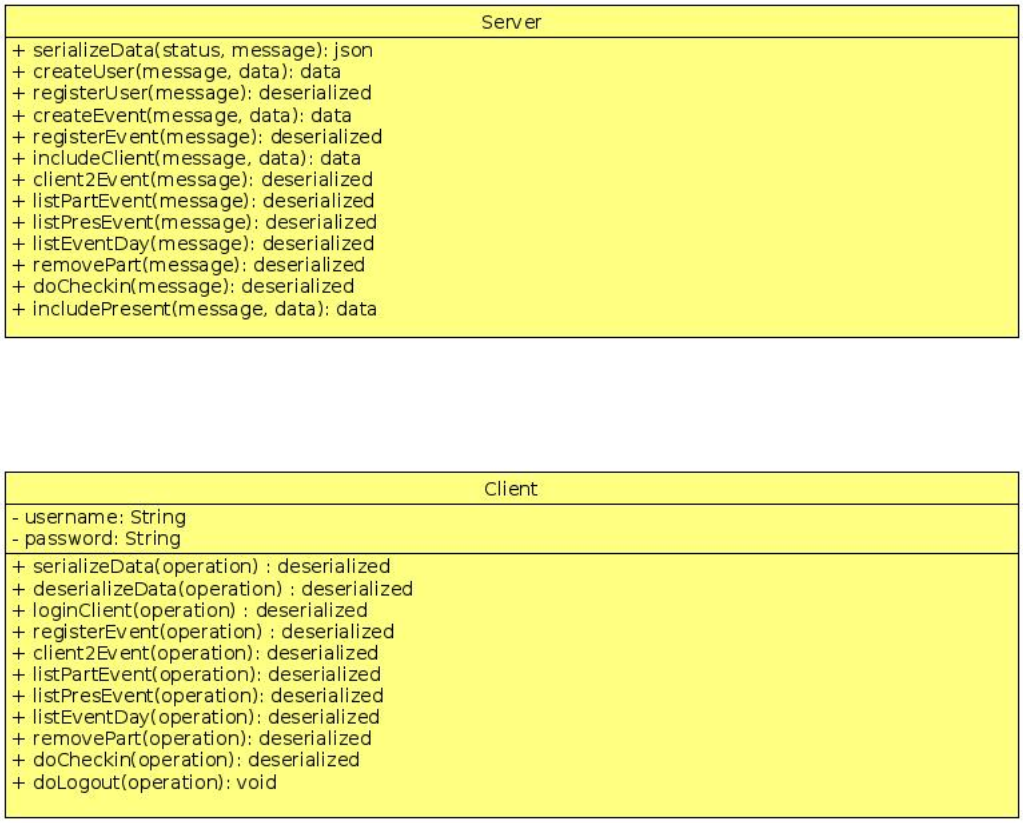


Figura 1: Diagrama de Classes

# 3 Protocolo de interação

Escolhemos um protocolo de interação baseado em filas, ele atende as nossas demandas, onde todas as requisições dos clientes serão atendidas em algum momento por estarem armazenadas em buffers (filas).

## 3.1 RabbitMQ

O RabbitMQ consegue lidar com o tráfego de mensagens, sendo um intermediário (aceitando e encaminhando mensagens). ELe utiliza o protocolo de mensagens AMQP *(Advanced Message Queuing Protocol)*.

As mensagens são armazenadas em filas. Uma fila é limitada pelos limites de memória no disco do *host*. Os produtores podem enviar mensagens para uma fila e os consumidores podem tentar receber os dados desta.

Essa forma de interação possibilita o desacoplamento tanto no espaço como no tempo, pois cliente e servidor não precisam estar ativos ao mesmo tempo. Além disso, o cliente não sabe para qual servidor ele está mandando sua requisição.

O RabbitMQ faz um tratamento muito interessante das requisições nas filas que é, caso um servidor que esteja tratando uma requisição pare de funcionar antes de retornar a resposta e avisar à fila que terminou o processamento, a requisição não é retirada da fila e é enviada para o próximo servidor disponível, até que possa ser tratada. Além disso, podemos configurar as filas de forma que, caso haja vários servidores consumindo requisições delas, as mensagens sejam distribuídas entre os servidores, não sobrecarregando-os.

## 3.2 Funcionalidades

Em nosso sistema, estamos utilizando o RabbitMQ na configuração RPC (*Remote Procedure Call*). Assim, ao executarmos um cliente, ele é associado a uma fila, que chamamos de *rpc_queue*, responsável por armazenar todas as requisições. Todo novo cliente executado será associado a essa mesma fila. Além disso, para cada novo cliente, fila auxiliar é criada, para recebimento das respostas dos servidores (*callback*). Assim, ao executarmos os servidores, eles

também são associados à fila *rpc_queue*, mas como consumidores, e são associados às filas correspondentes de *callback*. Outro detalhe importante é que estamos passando um parâmetro, chamado *correlation_id*, em todas as requisições enviadas pelos clientes e servidores. Isso permite que o cliente saiba à qual requisição a resposta recebida corresponde, uma vez que pode haver várias respostas em sua fila de *callback*.

# 4 Representação dos dados

Para representarmos os dados externamente, de maneira a se tornarem independentes da linguagem de programação e plataforma adotadas, utilizamos o formato de dados JSON. Assim, sempre que queremos trocar dados entre clientes e servidores, serializamos os dados, colocando-os nesse formato. Da mesma forma, quando a mensagem é recebida, ela é desserializada. O servidor sabe como desserializar os dados devido ao campo *operação*, que indica o serviço solicitado e, consequentemente, os parâmetros passados na requisição. Da mesma forma, o cliente sabe como desserializar a resposta enviada pelo servidor, pois ela possui um formato definido.

## 4.1 Interfaces de acesso

Na Figura 2, podemos observar um dos métodos que fornecem interface com o cliente. Ele recebe os parâmetros do cliente, especificando o formato em que deseja recebê-los; transforma-os em um objeto JSON e os envia ao servidor. Ao receber a resposta do servidor, ela é desserializada e retornada para ser tratada.

```
def registerEvent(self, operation):
    print("================= REGISTER EVENT ================")
    eventName = raw_input('Event Name:')
    eventDate = raw_input('Date of the event: [format DD-MM-YY]\n')
    eventHour = raw_input('Hour of the event: [format hh:mm]\n')
    os.system('clear')

    jsonOBJ = serializeData(operation, eventName, eventDate, eventHour)
    response = self.sendMessage(jsonOBJ)
    return deserializeData(response)
```

Figura 2: Exemplo - Interface de Serviço

Já na Figura 3, podemos observar um exemplo de como a função de serialização funciona. Ela recebe um parâmetro especificando o serviço solicitado e um parâmetro argumento, de tamanho variável. De acordo com a operação, ela sabe quantos argumentos foram passados e em que ordem.

```
elif(operation == "addEvent"):
    data = {"operation": operation,
            "eventName": args[0],
            "eventDate": args[1],
            "eventHour": args[2]}
```

Figura 3: Exemplo - Serialização de dados

Quando as mensagens chegam ao servidor, este identifica o serviço solicitado, descobrindo como interpretá-las.

# 5 Desafios abordados

Entre os desafios de sistemas distribuídos existentes, escolhemos tratar os seguintes desafios: *heterogeneidade* e *balanceamento de carga*. A seguir apresentaremos de que forma lidamos com estes desafios na arquitetura e no código.

## 5.1 Heterogeneidade

A heterogeneidade diz respeito ao funcionamento do sistema em um ambiente diversificado, em que os dispositivos com os quais ele irá se comunicar podem ter sido construídos utilizando diferentes linguagens de programação, com implementações de diferentes desenvolvedores, podem estar hospedados em diferentes plataformas de hardware, operando sob variados sistemas operacionais e os dados podem estar circulando dentro de diferentes tipos de redes.

Assim, lidamos com este desafio utilizando uma representação de dados (JSON) que simplifica o formato das mensagens trocadas entre os elementos do sistema, padronizando o modo como são construídas e tornando-as independentes das características de cada plataforma e linguagens utilizadas.

### 5.2 Balanceamento de Carga

Balanceamento de carga consiste em dividir o processamento de um sistema entre dois ou mais servidores, podendo, assim, atender a um número maior de requisições com tempo de resposta menor.

Tratamos o balanceamento de carga utilizando o RabbitMQ. Ele consegue criar vários servidores e separar as requisições entre eles. Assim quando um servidor estiver ocupado, e houver uma nova requisição, o novo cliente será direcionado para um servidor livre. Assim, não há congestionamento na rede, maximizando o desempenho do sistema e minimizando o tempo de resposta.

## 6 Considerações finais

Realizamos vários testes com as funcionalidades do sistema, obtendo os resultados esperados. Consideramos que o trabalho foi de grande aprendizado, trabalhamos com muitas tecnologias e plataformas que não havíamos tido nenhum contato. Também foi um desafio interessante realizar a implementação em Python, pois foi nosso primeiro contato com esta linguagem, escolhemos ela para simplificar a criação das classes e por haver muito conteúdo disponível na internet sobre alguns tópicos requeridos no trabalho.

## Referências


[1] RabbitMQ Tutorials. Disponível em: `https://www.rabbitmq.com/getstarted.html`. Acesso em 10 de de junho de 2018.

[2] Reading and Writing JSON to a File in Python: `http://stackabuse.com/reading-and-writing-json-to-a-file-in-python/`. Acesso em 20 de de junho de 2018.

[3] Protocol Buffer Basics: Java. Disponível em: `https://developers.google.com/protocol-buffers/docs/javatutorial`. Acesso em 20 de de junho de 2018.

[4] RabbitMQ Message Broker Desacoplando Workflows – Parte 1. Disponível em: `https://www.mundipagg.com/blog/rabbitmq-message-broker-desacoplando-workflows-parte-1/`. Acesso em 20 de junho de 2018.