ADDETC – Área Departamental de Engenharia Eletrónica e Telecomunicações e de Computadores

MEIM - Mestrado Engenharia informática e multimédia

Computação Distribuída

Trabalho final

Turma:

MEIM-11D

Trabalho realizado por:

Miguel Távora Nº45102

Duarte Domingues Nº45140

Docente:

José Simão

Data: 15/01/2022

Índice

Índice ilustrações

1. Introdução

O projeto final de Computação Distribuída tem como objetivo principal desenvolver um sistema que funciona como simulador de um veículo com um dispositivo IoT, que vai enviando mensagens da velocidade instantânea para um sistema de comunicação por eventos e comunicação por grupos. O sistema possuí então três camadas:

- Primeiro nível – parte do sistema responsável por gerar os eventos aleatoriamente com um identificador único, um local, uma data de aquisição do evento e o valor da velocidade no determinado instante. Após gerados os eventos estes são enviados para o segundo nível.
- Segundo nível – parte do sistema responsável por realizar comunicação por eventos. Para realizar esta comunicação é utilizado o RabbitMQ, no qual existe um *broker* para onde são enviadas as mensagens e cada mensagem é associada a um Exchange. Os Exchanges possuem Binds para enviar as mensagens para determinados Queues dependendo do tipo de Exchange. Os Queues será onde os consumidores irão obter as mensagens.

  Esta parte é responsável por criar os Exchanges, os Queues e os Binds entre o Exchange e os Queues. Este nível é também responsável por receber os eventos e colocá-los num Queue para o terceiro nível ter acesso aos eventos e também deve exibir internamente por meio de *prints* os eventos recebidos.
- Terceiro nível – este nível possui dois grupos de processos que funcionam através de comunicação por grupos, onde no caso será utilizado o *middleware* Spread. O Spread realiza comunicação de aplicações distribuídas por grupos com *multicast* fiável e ordenação total e causal das mensagens, mesmo na presença de falhas.

  1. Primeira parte (Event Processing Group) – possui um consumidor do RabbitMQ no padrão *work-queue*, onde os eventos são distribuídos pelos diversos consumidores, desta forma é possível aliviar a carga de processamento de um processo aumentando o número de processos consumidores. Por cada evento recebido o consumidor verifica se esse evento possuí um valor superior ou igual ao valor de 120km/h. Caso a condição se verifique é enviada uma mensagem

com o evento recebido para um grupo no qual a segunda parte (Front-End Group) participa. Neste grupo deve existir um líder de grupo que deve enviar uma mensagem de notificação sempre que um membro novo entra ou sai do grupo.

2. Segunda parte (Front-End Group) – parte que possui conexão com dois grupos Spread diferentes, primeiro o grupo Spread entre o Event Processing e um grupo Spread somente entre os servidores do Front-End. O grupo do Front-End possui também um líder que realiza a sincronização da informação guardada, nomeadamente dos eventos armazenados, sempre que um membro do grupo reentra no grupo após uma falha. Este grupo possuí também a parte servidor de um contrato gRPC com uma aplicação designada User. O facto de ser um grupo Spread permite tolerância a falhas, visto que caso um servidor falhe pode depois conectar-se a um novo servidor que possui exatamente a mesma informação. Esta arquitetura também permite equilíbrio de carga visto que como existe mais de um servidor, os clientes podem ser distribuídos pelos diversos servidores.

- User – aplicação simples de pedido – resposta aos servidores de Front-End. Este contrato é do tipo gRPC e as operações possíveis são: valor máximo registado, média de todas as velocidades, média de velocidade por data, obter o número total de consumidores no grupo Event Processing e simular a criação de mais um membro no Event Processing.

2. Desenvolvimento

2.1 IoTProducer

O IoTProducer corresponde ao primeiro nível do sistema. Esta aplicação tem a função de simular um sensor de velocidade de um veículo com um sensor IoT. Este simulador gera valores aleatoriamente da velocidade instantânea de um veículo, os valores variam entre 30 e 150km/h. Por cada valor gerado é gerada uma mensagem que possui os seguintes campos:

- sid - que corresponde a um identificador único do sensor
- local – distrito de Portugal no qual foi gerado o evento
- data – data no qual foi gerado o evento no formado de dd-MM-yyyy
- value – velocidade entre 30 e 150 km/h

Este produtor possui duas formas de funcionar uma forma onde é gerado automaticamente as mensagens e outra onde é o utilizador que seleciona os valores pretendidos de acordo com as restrições das mensagens. A geração automática é realizada pela classe IoTCar, onde são gerados valores num intervalo de tempo entre 3 a 5 segundos. O identificador sid é gerado um número aleatorio entre 1 e 99999, o local é selecionado um distrito de Portugal aleatóriamente a data é selecionado um dia e um mês aleatório e o valor um número aleatório entre 30 e 150. A geração manual é feita pela classe Input e possui as mesmas restrições do que a geração automática. Ambas as formas de geração de mensagens utilizam a classe Message para guardar as informações geradas e poder assim enviar a mensagem em *string* para o Exchange do RabbitMQ.

A classe responsável por realizar o envio das mensagens para o Exchange é a classe designada IoTProducer. Esta classe cria uma conexão com o *broker* do RabbitMQ e possui um método para publicar mensagens num Exchange com uma *routing key* bem conhecida entre o primeiro e o segundo nível. A partir do método de publicar mensagens é utilizado o método de gerar automaticamente ou manualmente as mensagens conforme o que é selecionado pelo utilizador.

2.2 EdgeFog

O EdgeFog corresponde ao segundo nível de implementação do sistema. Este nível é o responsável por estabelecer conectividade entre o primeiro, o segundo nível e o terceiro nível. Esta conectividade é feita por comunicação por eventos pelo modelo publish/subscribe. O modelo publish/subscribe é um modelo onde assume que existem três entidades: uma geradora de eventos, uma onde são guardados os eventos e uma que consome os eventos. Para criar esta comunicação foi utilizado o *middleware* RabbitMQ, este *middleware* funciona através de Exchanges, Queues e Binds entre Queues para Exchanges. O Exchange é para onde o produtor deve enviar as mensagens com uma determinada *routing key*, a *routing key* é utilizada para identificar quais são os Queues para onde a mensagem deve ir dependendo do tipo do Exchange. Os Queues é onde as mensagens ficam guardadas à espera que um consumir as receba e devolva um *acknowledge*. Os Binds entre Queue e Exchange é um conexão a ligar os dois.

A classe que possibilita estas ações é designada por RabbitMQConf. Para isso inicialmente a classe cria uma conexão e um canal, através destas instâncias é possível realizar as operações enumeradas anteriormente. Visto que é necessário o segundo nível e o terceiro nível aceder aos eventos vindos do primeiro nível foi criado um Exchange, dois Queues e dois Binds para os Queues se conectarem ao Exchange. Visto que é pretendido propagar as mesmas mensagens por ambos os Queues o Exchange é do tipo FANOUT, este modo permite que todos os Queues que tenham um Bind para o Exchange vão receber as mensagens do Exchange. Desta forma o segundo nível acede a um Queue e o terceiro nível acede a outro e assim todos recebem as mensagens. Para exibir os eventos recebidos pelo primeiro nível este nível possuí um consumidor de mensagens do RabbitMQ conectado a um dos Queues. A classe consumidora de eventos é designada RabbitMQConsumer. Existe também uma classe designada Message que permite através da mensagem recebida pelo consumidor obter o identificador, local, data e valor.

A comunicação por eventos do RabbitMQ implementado segue o seguinte esquema:

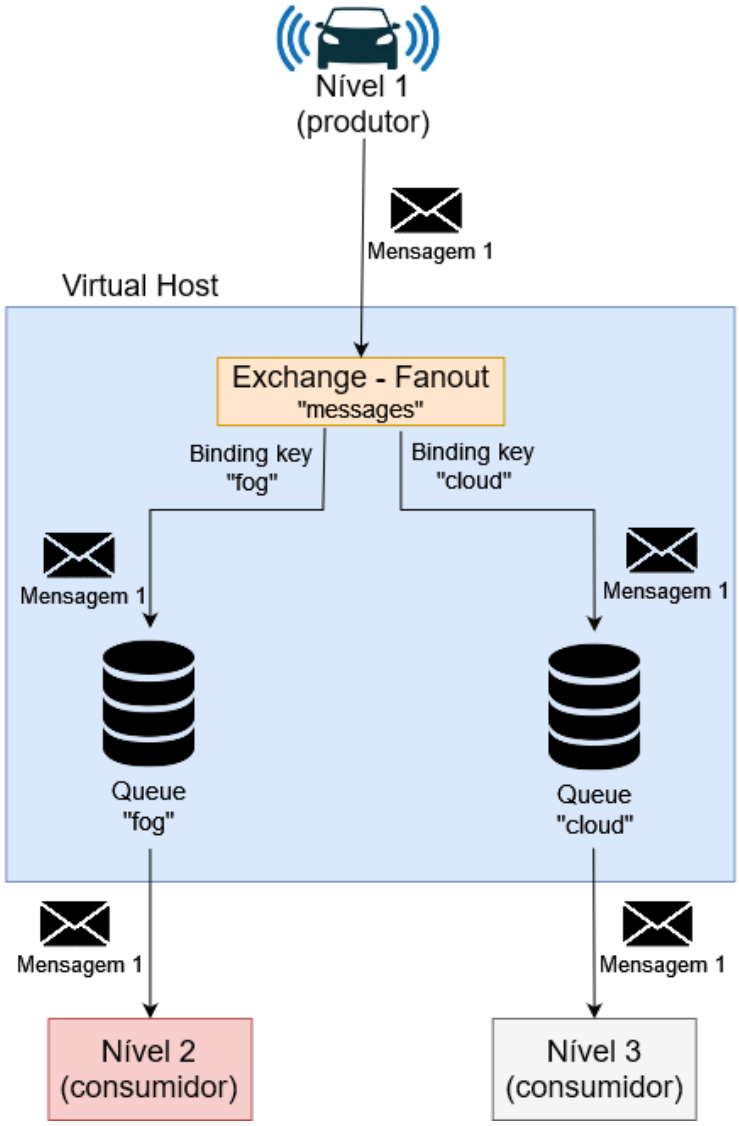

Figura 1 – Interação implementada do RabbitMQ

2.3 EventProcessing

O EventProcessing representa a primeira parte do terceiro nível. Este nível é responsável por obter os eventos do primeiro nível guardados num Queue. Para isso este nível irá então possuir um consumidor do RabbitMQ, a classe representativa desse consumidor é designada RabbitMQConsumer que funciona de forma semelhante ao consumidor do segundo nível e possui também uma classe Message para obter os campos gerador no primeiro nível. De todos os eventos recebidos somente os que possuem um valor superior ou igual a 120km/h é que serão posteriormente enviados para o grupo Spread. Este consumo funciona no padrão *work-queue*, onde por cada processo que se inicia é criado um novo consumidor e as mensagens guardadas no Queue vão assim ser distribuídas pelos diferentes consumidores. Esta metodologia permite o balanceamento de carga porque assim não fica um processo sobrecarregado com todos os eventos, pois este trabalho é distribuído por diversos processos.

Esta primeira parte também pertence a um grupo Spread. O Spread funciona como *middleware* de comunicação de grupos que requerem alta fiabilidade, alto desempenho e comunicação robusta entre vários subconjuntos de membros. O Spread proporciona comunicação por grupos com *multicast*, com ordenação total e causal das mensagens mesmo na presença de falhas, isto é, possui tolerância a falhas. A utilização deste *middleware* começa por uma aplicação se conectar a um deamon, esse deamon pode ou não estar conectado a outros deamons. Caso um deamon esteja conectado um ou mais deamons e se dois membros conectam por deamons diferentes a um grupo com o mesmo nome é possível os dois membros contatar-se mesmo estando conectados a deamons diferentes. Isto deve-se a que o Spread proporciona uma conexão transparente tanto ao utilizador como ao programador que devenvolve o sistema.

A classe responsável por criar a conexão com um deamon do Spread, juntar-se a um grupo, deixar um grupo e desconectar-se do daemon é GroupMember. No caso esta parte conecta-se somente a um grupo designado “EventProcessing”. A classe responsável por lidar com as mensagens recebida pelos grupos é designada por MessageHandler. Esta classe vai então receber mensagens de si própria ou de outros membros quando o evento recebido pelo consumidor do RabbitMQ possui um valor superior ou igual a 120km/h.

Este grupo também possui um líder que possui a responsabilidade de informar o grupo que representa a segunda parte no nível 3(Front-End) da entrada ou saída de membros consumidores do RabbitMQ. A classe que realiza todo este algoritmo é a classe designada por LeaderGroup. O algoritmo de implementação do líder foi feita da seguinte maneira:

- O líder eleito é o primeiro membro a conectar-se ao grupo.
- Quando algum membro se conecta ao grupo que não é o primeiro ele pergunta quem é o líder enviando uma mensagem do tipo “leader?2”, onde 2 representa o número de membros conectados ao grupo.
- Todos os membros recebem esta mensagem e enviam uma mensagem do tipo “request_leader:membrox:2”, onde membrox representa o nome do líder para o determinado grupo e 2 o número de membros.
- Caso um membro não tenha líder atribuído envia uma mensagem do tipo “request_leader:1456789:2”, onde 1456789 representa o seu número para ser líder e 2 o número de membros conectado no grupo. Caso receba uma mensagem com o nome de um líder, que é identificado por ser só caracteres, é assumido que esse é o líder. A verificação de quem é o líder é feita quando recebe todas as mensagens de todos os membros do grupo, obtida a partir do número de membros.
- Caso dois membros enviem um líder diferente é considerado assincronia e todos os membros removem o seu atual líder e enviam todos uma mensagem do tipo “request_leader:1456789:2” e quem tiver o identificador maior é considerado o líder. O mesmo acontece quando o líder do grupo se desconecta por deixar o grupo, se desconectar ou existir um problema no deamon do líder.

Além de líder o grupo envia um tipo de mensagens de entrada ou saída de consumidores no grupo Spread que significa também a paragem ou entrada de um consumidor do RabbitMQ. Estas mensagens são geradas a partir de *strings* no formato json. Esta transformação é feita pela classe MessageConsumers. Esta mensagem possui somente uma lista com os todos os nomes dos consumidores existentes no grupo, que será depois guardado pela segunda parte do terceiro nível. Quem envia a mensagem com todos os consumidores é somente o líder e todos os membros obtém os consumidores. Apesar do Event Processing gerar mensagens tanto de

velocidade como de notificação da entrada e saída de membros não guarda nenhuma das mensagens, visto que as informações só são relevantes para o Front-End.

Um detalhe importante é a resiliência a falhas que o sistema possui, visto que sempre que um membro deixa o grupo, desconecta ou o servidor com o deamon falha o sistema consegue dinamicamente reeleger novo líder e continuar a funcionar sem os membros que saíram. Desta forma é garantido que a informação é sempre consistente em todos os instantes e mesmo quando um novo membro se junta ao grupo.

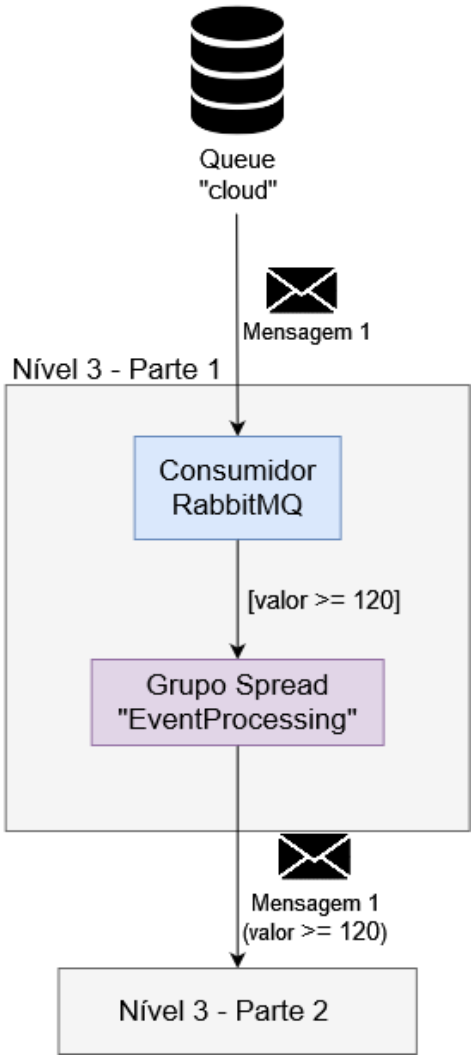

Figura 2 - Interação de mensagens da primeira parte do terceiro nível

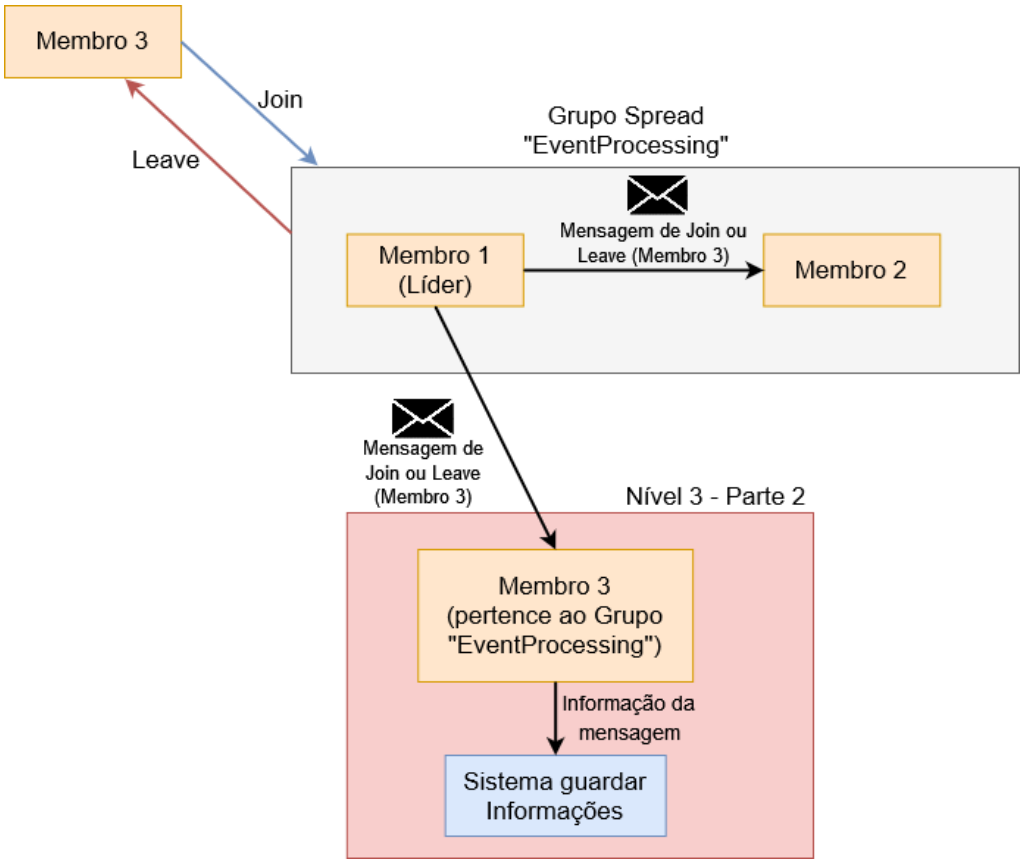

Figura 3 - Interação entre mensagens de entrada ou saída de membros no grupo Spread

2.4 FrontEnd

O FrontEnd representa a segunda parte do terceiro nível. Esta parte é responsável por receber os eventos das mensagens do primeiro nível e das notificações de membros a entrar ou sair do grupo. Este grupo como foi referido anteriormente é designado por "EventProcessing". A classe responsável por isso é GroupMember, que é similar à da parte um, mas como também é utilizada no grupo Spread da parte dois do terceiro nível, designado "FrontEnd", possui ajustes em conformidade. Estes ajustes são nomeadamente poder instanciar a classe que recebe mensagens tanto do grupo "EventProcessing" como do grupo "FrontEnd". A classe que recebe as mensagens do grupo "EventProcessing" é designada MessageHandlerProcessing e a única diferença para a primeira parte é o facto de possuir métodos para guardar as diversas informações trocadas neste grupo. A classe utilizada para guardar as informações recebidas neste grupo é designada StoreInformation. Esta classe possui somente métodos de *get* e *set* para poder ser convertido o seu conteúdo para json e utilizado no grupo "FrontEnd". A classe MessageHandlerProcessing utiliza também a classe MessageConsumers para obter a quantidade de consumidores, que neste caso estão no grupo "EventProcessing". Apesar de esta parte também pertence ao grupo mas não incrementa o número de consumidores. Esta parte também pode representar o líder do grupo "EventProcessing" e não somente a primeira parte. Sempre que é lançada uma aplicação desta parte é lançada uma mensagem a perguntar quem são os consumidores do grupo.

Como foi referido anteriormente, existe mais um grupo Spread além do "EventProcessing" que é partilhado somente pelas instâncias que fazem parte da segunda parte do terceiro nível, designado por "FrontEnd". O grupo "FrontEnd" também possui um líder dentro do grupo eleito de igual forma que o grupo "EventProcessing" e a classe responsável por isso é LeaderGroup. Este grupo serve para caso um membro realize uma reentrada no grupo, o líder do grupo envia uma mensagem com toda a informação que possui guardada e o membro que entrou recebe a informação e guarda-a, os outros membros ignoram a mensagem. Desta forma todos os membros do grupo possuem o mesmo estado armazenado. Esta metodologia permite tolerância a falhas, visto que caso um servidor falhe existe um outro com o mesmo estado ao qual o utilizador pode realizar os seus pedidos e também equilíbrio de carga podendo assim dividir

os clientes pelos diversos servidores. A classe que recebe as mensagens do grupo “FrontEnd” é designada MessageHandlerFront. Esta classe utiliza a classe MessageSync, utilizada pelo líder para enviar a sua informação guardada em formato json e para o membro que se conectou obter as informações a partir do json.

Esta parte também implementa a parte servidora para uma aplicação designada User. Quem realiza este comportamento é a classe ServerUser. Essencialmente esta classe tem acesso às informações que vão sendo guardadas na classe StoreInformation pelo grupo “EventProcessing”. A partir destas informações e de um contrato gRPC ele responde à aplicação User.

Da mesma forma que o grupo “EventProcessing” o grupo “FrontEnd” também possui uma grande robustez no que diz respeito a tolerância a saídas, desconexão ou falha de um deamon de membros. Pois é detetada a falha e caso o líder seja um dos membros que se conectou é eleito novo líder e o sistema continua a funcionar.

Apesar de o sistema estar a funcionar, não foi tido em conta o tamanho das mensagens no mecanismo do spread. Quer isto dizer que caso a mensagem seja muito grande esta não será enviada pelo sistema e este deixa de funcionar. Isto poderia acontecer nomeadamente na sincronização das mensagens de um novo servidor que entra. Uma maneira de resolver este problema seria dividir a mensagem em diversos blocos com um *header* identificador de quantas mensagens seriam enviadas, conforme o sistema receberia as mensagens estas seriam guardadas. Desta forma o sistema funcionaria para qualquer tamanho de mensagem.

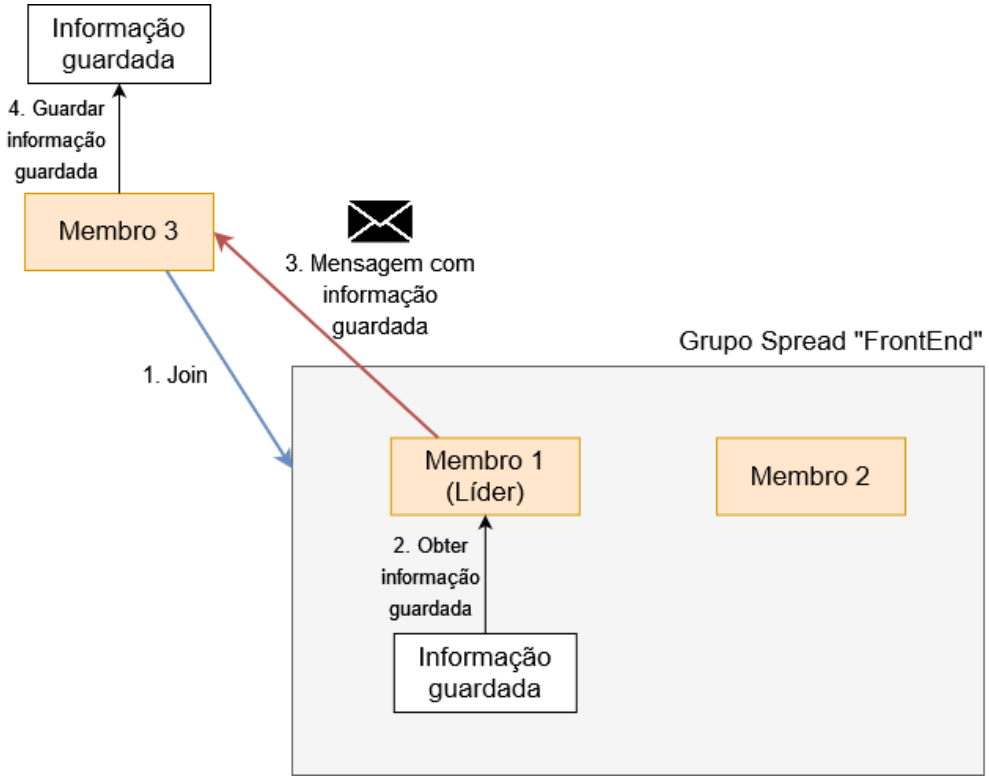

Figura 4 - Interação entre o grupo Spread "FrontEnd"

2.5 FrontEndUserContract

Esta parte representa o contrato entre o servidor que se encontra no FrontEnd e a aplicação User. O contrato possui as operações se segue:

```
service Questions {
    rpc maxValueVelocity(google.protobuf.Empty) returns (VelocityComplete);
    rpc meanVelocity(google.protobuf.Empty) returns (Velocity);
    rpc meanVelocityBetweenDates(Date) returns (Velocity);
    rpc numConsumers(google.protobuf.Empty) returns (Consumer);
    rpc executeConsumer(ConsumerName) returns (Confirm);
}
```

Figura 5 – Contrato implementado entre FrontEnd e User

O contrato possui no total 5 operações sendo elas:

- Operação: maxValueVelocity – o servidor devolve de todos os valores o mais alto que possui guardado, adicionalmente também envia qual foi o identificador que enviou essa velocidade, o local e a data de aquisição.
- Operação: meanVelocity – o servidor devolve a media de todas as velocidades que tem guardado.
- Operação: meanVelocityBetweenDates – o User envia duas datas, uma data de início e uma data de fim, o servidor irá calcular a média das velocidades entre as datas enviadas pelo User.
- Operação: numConsumers – o servidor envia o número de consumidores do RabbitMQ que estão a consumir mensagens no terceiro nível.
- Operação: executeConsumer – uma operação de simulação onde deveria executar mais um consumidor no terceiro nível. Contudo esta operação só devolve se o utilizador pode ou não realizar esta operação.

2.6 User

A aplicação User é uma aplicação simples responsável por se conectar ao servidor do FrontEnd com o contrato FrontEndUserContract. Por isso esta aplicação possui então uma classe com o main e uma classe que recebe os *inputs* do utilizador. A classe executada será então o User e a classe responsável por receber os *inputs* e validar se estes estão corretos é a classe UserInput. Esta aplicação também deveria poder criar mais consumidores do RabbitMQ na primeira parte do terceiro nível, para isso possui um método rpc para simular essa situação.

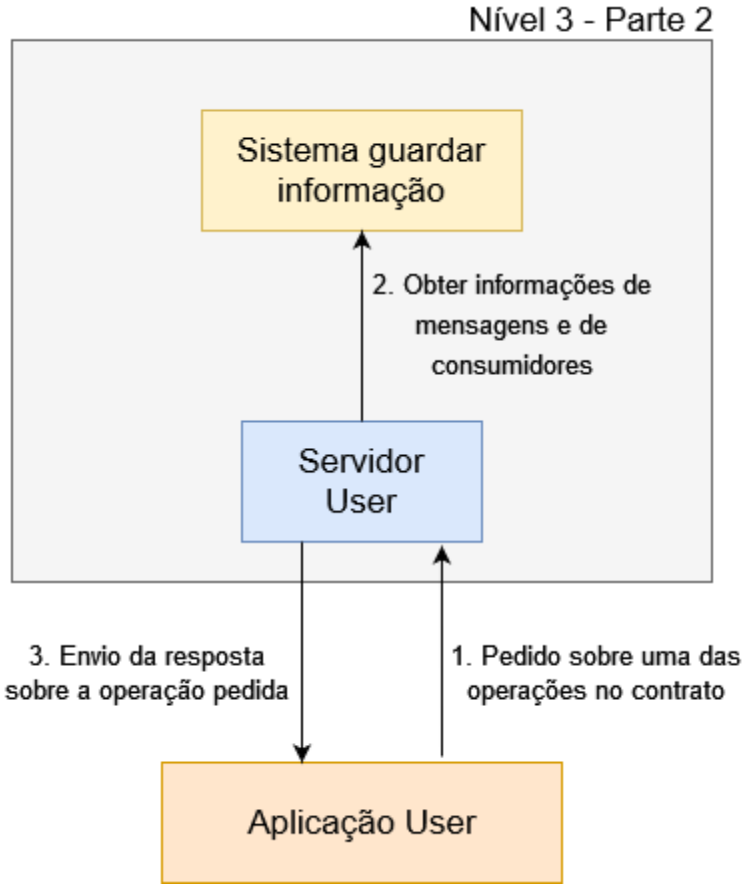

Figura 6 - Interação realizada pelo sistema User

3. Conclusões

Em suma com a realização do presente trabalho prático o grupo adquiriu os seguintes conhecimentos:

- Utilizar o Google RPC para fazer troca de mensagens entre um cliente e um servidor, onde através da implementação de um contrato feito por um ficheiro proto é possível criar uma abstração ao nível da aplicação facilitando toda a implementação do sistema de troca de mensagens.
- Utilização de containers numa VM para executar uma pequena parte do sistema implementado.
- Utilização do *middleware* RabbitMQ para comunicação entre eventos do tipo publish/subscribe.
- Utilização do *middleware* Spread para realizar comunicação por grupos, onde é utilizado tempo, coordenação e consenso em sistemas distribuídos.
- Vantagens de utilização de comunicação do tipo publish/subscribe, para obter balanceamento de carga por as mensagens ficarem repartidas por todos os *subscribers* e tolerância a falhas por as mensagens ficarem guardadas e mesmo que um produtor falhe a mensagem fica guardada até ser consumida.
- Vantagens de utilização de comunicação por grupos para garantir consistência de estado entre servidores e assim garantir a tolerância a falhas por poder contactar qualquer servidor no caso de um falhar e equilíbrio de carga por ficar vários clientes distribuídos pelos servidores.

Em termos de implementação do sistema este ficou todo ele funciona e é possível utilizar este sistema como pretendido pelo enunciado com a VM para o RabbitMQ e com três servidores para o FrontEnd. Esta arquitetura permite melhorar a tolerância a falhas e disponibilidade.

4. Bibliografia

[1] Slides CD-04 Vms e Contains, José Simão e Luís Assunção, 2021

[2] Slides CD-05 Comunicação por eventos e RabbitMQ, José Simão e Luís Assunção, 2021

[3] Slides CD-06 Tempo, coordenação e consensos em sistemas distribuídos (Comunicação por grupos e Spread Toolkit), José Simão e Luís Assunção, 2021

http://www.spread.org/