Instituto Superior de Engenharia de Lisboa

Computação Distribuída

**Sistema Distribuído para Coleção e Armazenamento de Amostras de Velocidade de Automóveis**

Mestrado em Engenharia Informática de Multimédia

Pedro Gonçalves, 45890

Rodrigo Dias, 45881

Rúben Santos, 49063

Semestre de Inverno, 2021/2022

# Introdução

O objetivo deste projeto consiste na implementação de um sistema distribuído de obtenção e armazenamento de valores de velocidade coletados por sensores dispostos em diversos locais.

Tirar-se-á partido das seguintes tecnologias:

* ***RabbitMQ***
* ***Spread*** ***Daemon***
* ***gRPC***

O código fonte, os executáveis (packages em formato “***.jar***” na pasta **/packages**) e os comandos necessários à execução dos mesmos encontram-se [nesta](https://github.com/Alcachofraz/Velocity-Collecting-Distributed-System) página do **Github**.

# Análise de Requisitos

Será necessário primeiro definir o problema em concreto, os mecanismos que estarão envolvidos e uma possível solução. A análise de requisitos é o primeiro passo no desenvolvimento de qualquer tipo de software, e deve apontar o programador na direção correta.

## Definição do Problema

O sistema deverá apresentar as seguintes funcionalidades fundamentais:

1. Coleção de valores de velocidade (em **km/h**) juntamente com **ID** do sensor correspondente e o local e data em que foi coletado.
2. Os valores coletados deverão ser armazenados num cluster de servidores que suporte distribuição de carga.
3. Os servidores deverão ser capazes de responder a alguns pedidos por parte de aplicações do utilizador, como a velocidade mais alta alguma vez registada ou a média das velocidades numa determinada data.

Diagrama

Descrição gerada automaticamenteNa **figura 1** apresenta-se o diagrama geral do sistema, que ilustra os quatro níveis, cada um com determinados requisitos.

* **1º Nível (Sensores):** Uma aplicação que simule a coleção de dados por parte do sensor.
* **2º Nível (Edge/Fog):** Um servidor ***RabbitMQ*** que suporte o modelo ***Publish/Subscribe***, e que receba os dados dos múltiplos sensores. Neste nível deverá ser feito ***logging*** dos dados recebidos dos sensores.
* **3º Nível (Cloud):** Dois grupos Spread. O ***Event Processing Group*** contará com consumidores cuja função é coletar os dados publicados pelos sensores no servidor ***RabbitMQ***, e ***Front-End Group*** contará com um cluster de servidores que irão receber os dados coletados pelos consumidores, armazenando-os em memória.

Figura - Diagrama geral do sistema

* **4º Nível (Acesso aos dados):** Aplicações para que os utilizadores, por ***gRPC***, possam interagir com os servidores do ***Front-End Group*** para obter determinadas informações acerca dos dados coletados.

## Requisitos da Solução

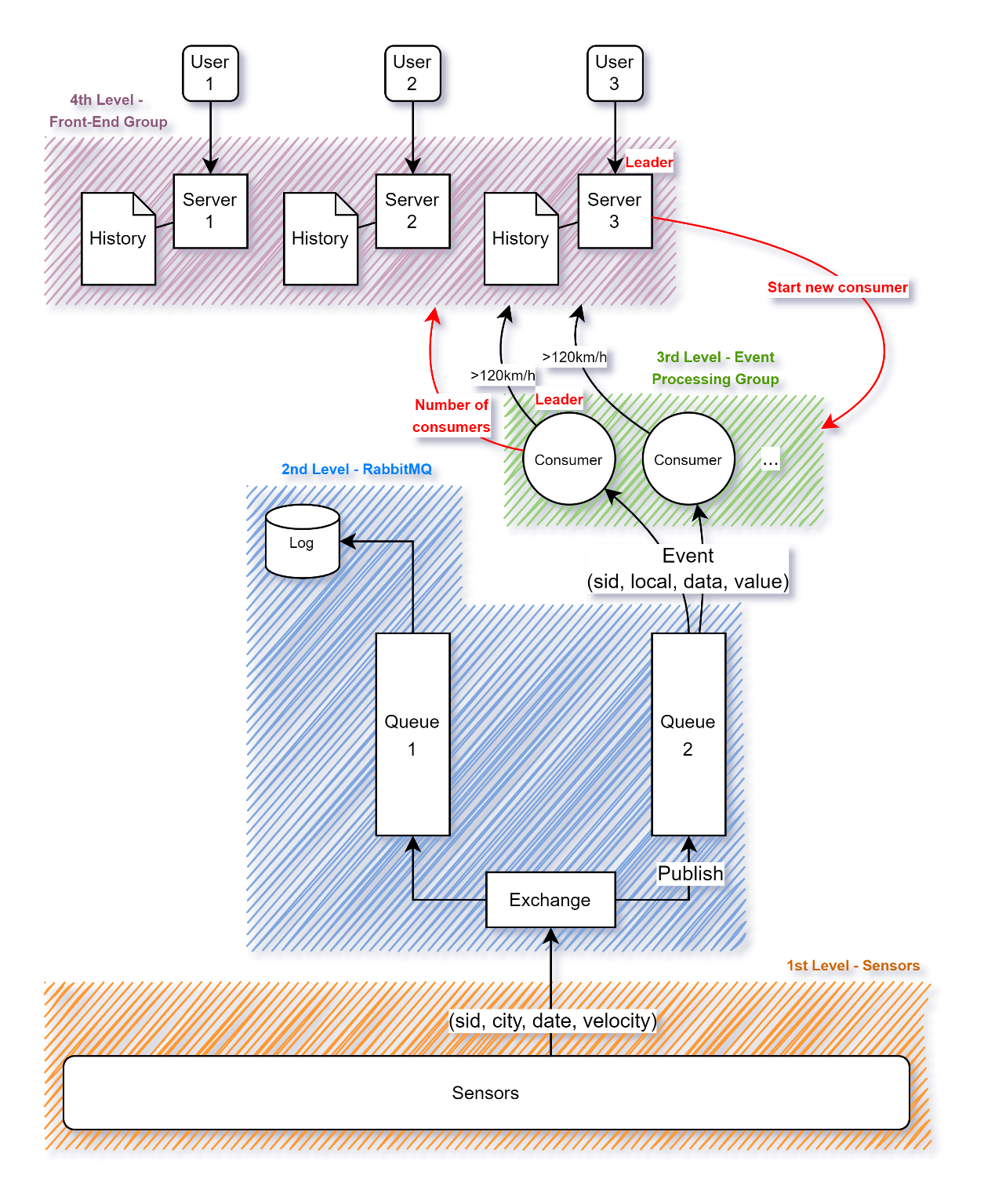
Perante os requisitos do capítulo anterior, começar-se-á por definir, com mais rigor, os diferentes níveis do sistema. Na **figura 2**, verifica-se uma ilustração do sistema mais detalhada.

Figura - Diagram detalhado do sistema

### Primeiro Nível

No primeiro nível do sistema, no qual se substituirão os sensores por aplicações que simulam a coleção de dados, dever-se-á considerar alguns requisitos adicionais:

* A aplicação deverá aceitar como parâmetros:
  + O ***Endpoint*** (**<IP>:<PORT>**) do servidor ***broker RabbitMQ***;
  + O ritmo a que serão gerados dados;
  + O ID do sensor;
  + A cidade onde o sensor se situa;
  + Um parâmetro que permita influenciar a data de coleção dos dados (para efeitos de teste).
* Deverá gerar valores de velocidade aleatórios, compreendidos entre **0 km/h** e **270 km/h**.
* Aquando da geração de um novo valor, deverá publicá-lo para o ***Exchange*** do servidor ***RabbitMQ*** no segundo nível.

### Segundo Nível

No segundo nível, deverá existir uma aplicação capaz de configurar o servidor ***RabbitMQ*** devidamente. Essa aplicação deverá aceitar como parâmetros o Endpoint do servidor ***broker RabbitMQ***. Deverá criar um **Exchange** com duas **Queues** associadas (***bind***), uma visível para os consumidores do terceiro nível do sistema, e outra para ***logging***.

Devido à necessidade de fazer logging, deverá ser criada uma segunda aplicação neste nível, capaz de coletar os dados da **Queue** de ***logging*** e de os guardar num ficheiro.

### Terceiro Nível

O terceiro nível será o mais complexo, visto que se responsabiliza por processar, armazenar e disponibilizar os dados, e irá interagir com os outros níveis.

Será necessária uma aplicação que faça o papel de consumidor, que terá os seguintes objetivos:

* Ingressar no **Event Processing Spread Group**;
* Coletar dados da **Queue** do servidor ***RabbitMQ*** do segundo nível;
* Dos dados que coleta, deverá partilhar com o **Front-End Group** aqueles que apresentam uma velocidade maior que **120 km/h**;
* Se o consumidor em questão for o líder do grupo, partilhar também com o **Front-End Group** atualizações acerca de alterações no número de membros, isto é, consumidores, do **Event Processing Group**.

O consumidor líder, como referido, possui tarefas adicionais. Mas como é que um consumidor sabe se é o líder ou não? Isso será abordado no capítulo de **Implementação**.

Será também necessária uma aplicação que faça o papel de servidor, e que deverá concretizar o seguinte:

* Ingressar no **Front-End Spread Group**;
* Receber e armazenar dados que vêm dos consumidores;
* Receber atualizações do número de membros do **Event Processing Group**;
* Através de um serviço ***gRPC***, disponibilizar determinadas ***queries*** para que a aplicação do utilizador possa interagir com os dados armazenados no servidor.
* Se o servidor em questão for o líder do grupo, deverá, quando um novo servidor se juntar ao Front-End Group, partilhar com ele o seu histórico de dados armazenado em memória. Essa partilha será feita através de um evento no Front-End Group, que deverá ser interpretado apenas pelo servidor que acabou de se juntar.

O algoritmo de eleição do líder, e de partilha privada do histórico (no caso de ser o líder), será abordado no capítulo de **Implementação**.

As ***queries*** a disponibilizar serão:

* Maior valor de velocidade registado;
* Menor valor (acima de 120 km/h) de velocidade registado;
* Dados coletados numa determinada cidade;
* Dados coletados numa determinada data;
* Média das velocidades coletadas numa determinada cidade;
* Média das velocidades coletadas numa determinada data;
* Verificar o número de consumidores ativos no **Event Processing Group**.

### Quarto Nível

O quarto nível representa o acesso aos dados por uma aplicação de utilizador. Essa aplicação, implementando um contrato de serviço do servidor a que se irá conectar, deverá ser capaz de pedir dados, consoante as ***queries*** disponíveis nesse mesmo contrato.

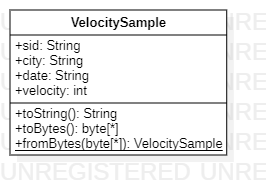
Os resultados das queries deverão ser imprimidos no formato **“SID: <sid>, CITY: <city>, DATE: <date>, VELOCITY: <velocity>”.**

# Implementação

## Configurador RabbitMQ

A aplicação **RabbitConfigurator** deverá configurar um ***Exchange*** (**Fanout**) no servidor **RabbitMQ**, juntamente com duas filas associadas. O ***Exchange*** terá nome **VELOCITY\_SAMPLES** e ***routing key*** **VELOCITY\_SAMPLES**. O nome das filas será **VELOCITY\_QUEUE** para a fila a que os consumidores estarão subscritos, e **VELOCITY\_QUEUE\_LOG** para a fila para efeitos de ***logging***.

## Sensor

A aplicação **Sensor** deverá apresentar duas funcionalidades fundamentais:

* Gerar números aleatórios de forma periódica.
* Publicar eventos no servidor **RabbitMQ**.

Para executar código periodicamente, tirar-se-á partido da classe **ScheduledExecutorService** para criar uma thread pool que se responsabilizará por agendar e executar instruções após um determinado ***delay***.

Figura - Estrutura que representa uma velocidade

Na **figura 3** apresenta-se a estrutura de dados que será convertida para um array de ***bytes***, e que será publicada no servidor **RabbitMQ** (Exchange de nome **VELOCITY\_SAMPLES**), bem como será transportada nos eventos do servidor **Spread**.

## Consumidor

Diagrama

Descrição gerada automaticamenteA aplicação **Consumer** deverá adotar uma abordagem de subscrição à fila **VELOCITY\_QUEUE** sem ***Acknowledge***. Deverá ser a própria aplicação a decidir se o **ACK** é enviado ou não: Se o envio dos dados coletados para o **Front-End Group** falhar, deve ser enviado **NACK** para os dados sejam repostos na fila, de forma a que não sejam esquecidos e possam voltar a ser coletados. Caso os dados sejam enviados para o **Front-End Group** com sucesso, **ACK** deverá ser enviado. A **figura 4** ilustra os dois casos possíveis

Figura - Sistema de Acknowledge

Outro aspeto importante da implementação do consumidor, incide na necessidade de eleger um consumidor líder que se responsabilizará por enviar atualizações acerca do número de consumidores ativos para o grupo de **Front-End**. O algoritmo de eleição tem início no nome dado a cada um dos consumidores quando ingressam no grupo **Spread** de **Processamento de Eventos**. Deverá corresponder aos segundos desde o **EPOCH**, de forma a que todos os consumidores tenham nomes únicos, e o mais antigo apresento o nome lexicamente menor. Isto permite que ao obter o nome dos membros do **Event Processing Group**, ao ordenar, o primeiro da lista será o mais antigo, e portanto, o líder. Quando um consumidor abandona o grupo (originando um evento do qual tirar-se-á partido), o consumidor com o nome lexicamente menor (ou seja, o mais antigo) será eleito o novo líder.

Diagrama

Descrição gerada automaticamenteA **figura 5** representa um esquema temporal, que exemplifica um cenário em que três consumidores entram em funcionamento, e quando o **Consumidor 1** se disconecta, ocorre necessidade de reeleição de um líder.

Figura - Cenário com três consumidores

## Server

A aplicação **Server** deverá implementar um contrato de serviço que permite disponibilizar à aplicação **User** algumas queries acerca da informação armazenada sobre as velocidades coletadas pelos sensores. O contrato implementado é o seguinte:

Texto

Descrição gerada automaticamente com confiança média

Também apresentará um algoritmo de eleição do líder semelhante à aplicação **Consumer**. Neste caso, o mecanismo discutido previamente que elege sempre o servidor mais antigo, é particularmente interessante, visto que o servidor mais antigo terá sempre os dados mais atualizados (histórico e número de consumidores ativos).

Além disso, existirá outro algoritmo importante a desenvolver relativamente à sincronização do histórico de velocidades entre os servidores do grupo ***Spread*** **Front-End**.

Imagine-se um cenário em que estão dois servidores a trabalhar, e que já possuem, em memória, diversos dados recebidos dos consumidores. Se um terceiro servidor se juntar ao grupo **Front-End**, o líder deverá atualizá-lo com o seu histórico, de modo a manter todos os servidores ativos sincronizados.

Os dados serão enviados pelo líder através de uma única mensagem, com um prefixo “**ANSWER**”, seguido da lista de velocidades, seguido de uma ***keyword*** “**CONSUMERS**”, seguido do número de consumidores ativos. O problema surge quando, a meio desse processo (em que o líder já percebeu que tem de atualizar o novo servidor e prepara os dados para o envio) o servidor líder falha e desconecta-se. Neste caso, o novo servidor acaba por não receber a mensagem de atualização do histórico, e o novo líder (que foi eleito em consequência da desconexão do líder anterior) não sabe que existe um novo servidor.

De modo precaver esta situação, todos os servidores devem ser iniciados com uma variável booleana ***pending*** colocada a **true**. Enquanto esta variável for verdadeira, sempre que ocorrer qualquer evento no grupo, o servidor em questão enviará para os restantes servidores uma mensagem “**REQUEST**”, indicando que está à espera que o líder lhe envie o histórico. Por sua vez, o líder, recebendo esta mensagem, procede ao envio dos dados com o prefixo “**ANSWER**”. Quando esta mensagem é recebida e serializada pelo novo servidor, a variável ***pending*** é colocada a **false**.

Para evitar duplicação de dados de velocidades, um servidor que possua a variável ***pending*** a **true**, ignorará novos valores de velocidades fornecidos por consumidores.

Na **figura 6** ilustra-se um cenário em que existem dois servidores ativos, que já receberam dados de um consumidor. Depois, junta-se um terceiro servidor que admite uma variável ***pending*** a **true**. Neste cenário o **servidor 1**, que é o líder, recebe a mensagem de ***request***, mas falha e desconecta-se antes que possa enviar a mensagem de resposta, com o histórico. Contudo, o terceiro servidor deteta isso, e reenvia a mensgaem de ***request*** ao novo líder, obtendo o histórico com sucesso.

Diagrama

Descrição gerada automaticamente com confiança média

Figura - Cenário com três servidores

## User

A aplicação **User** terá de abrir um Stub de forma a utilizar os serviços disponibilizados pelo **Server**, mais precisamente o serviço **VelocityQueries**. Durante a execução, deverá manter o stub aberto, para que o utilizador consiga realizar as ***queries*** que desejar.

## Logger

A aplicação Logger deverá obter os eventos da fila QUEUE\_VELOCITY\_LOG do servidor RabbitMQ, e escrevê-los para um ficheiro. Para escrever num ficheiro recorrer-se-á à class FileWriter.

# Teste

De forma a tornar os programas o mais versáteis possível, ao executá-los na janela de comandos, existem alguns parâmetros opcionais. A forma como os comandos de execução devem ser utilizados encontra-se [nesta](https://github.com/Alcachofraz/Velocity-Collecting-Distributed-System) página do **Github**, onde também se encontra o código fonte e os programas (**packages**) executáveis na pasta **/packages**.

Para testar o sistema, foram utilizadas quatro máquinas virtuais. A primeira servirá o propósito de hospedar o servidor **RabbitMQ**, onde serão configurados o exchange e as filas, com a aplicação **RabbitConfigurator**. As outras três estarã conectadas entre si para hospedar um servidor distribuído Daemon Spread, onde serão criados os grupos **Event Processing** e **Front-End**. Ou seja, nas três últimas três máquinas virtuais, hospedar-se-ão também **Servers** e **Consumers**. O **RabbitConfigurator**, o **Sensor**, o **User** e o **Logger** serão executados.

O primeiro passo é ligar a máquina em que será hospedado o RabbitMQ, e configurá-lo utilizando a aplicação **RabbitConfigurator**.

Texto

Descrição gerada automaticamenteNo máquina virtual com ***IP*** **35.197.247.130**:

Figura - Iniciar o docker RabbitMQ

Texto

Descrição gerada automaticamenteNa máquina local, executar RabbitConfigurator, especificando o **IP** da máquina virtual e o porto em que o **RabbitMQ** está hospedado.

Figura - Execução do RabbitConfigurator

Tabela

Descrição gerada automaticamenteNa interface web ([**http://35.197.247.130:15672**](http://35.197.247.130:15672)) verifica-se que o Exchange foi criado:

Figura – Confirmação da criação do Exchange

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamenteBem como as duas filas:

Figura - Confirmação da criação das filas

Na figura 11, verifica-se que as duas filas estão associadas ao Exchange, e que o próprio é do tipo Fanout como pretendido.

Interface gráfica do usuário

Descrição gerada automaticamente

Figura - Confirmação da associação das filas e do tipo de Exchange

O próximo passo é ligar as restantes máquinas virtuais, iniciar o **Daemon Spread** e ligar os **Servers** e **Consumers**. Note-se que dever-se-á ligar primeiro os servidores, para que estes detetem os eventos enviados pelos consumidores aquando da sua entrada (para atualizar o **Front-End** **Group** com o número de consumidores ativos). Se os Consumidores forem ligados primeiro, esses eventos não são recebidos por ninguém, e quando os servidores se ligarem, terão de esperar que algum consumidor saia ou entre para receberem atualizações acerca do número de consumidores ativos.

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamenteO ficheiro **newspread.conf**, partilhado entre as três máquians virtuais, apresenta o seguinte conteúdo:

Figura - Conteúdo do ficheiro newspread.conf

Texto

Descrição gerada automaticamenteLigando o **Daemon Spread** nas três máquinas virtuais:

Figura - Confirmação das treês máquinas virtuais no Sporead Daemon

Na **figura 13** encotnram-se os ***IPS*** itnernos das máquinas. Os ***IPS*** externos serão:

* Máquina 1: **34.89.26.148**
* Máquina 2: **35.242.139.84**
* Máquina 3: **35.242.139.84**

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamenteNa Máquina 1 hospedar-se-á um servidor no porto **5000**, especificando o ponto de acesso ao **Daemon Spread** da Máquina 2 (podia ser de qualquer uma das 3 máquinas):

Figura - Execução do servidor da Máquina 1

Desde já, verifica-se que o Servidor 1 é promovido a líder.

Texto

Descrição gerada automaticamenteNa Máquina 2 hospedar-se-á um servidor no porto **5000**, especificando o ponto de acesso ao **Daemon Spread** da Máquina 3:

Figura - Execução do servidor da Máquina 2

O Servidor 2 não é promovido a líder, pois existe no grupo um mais antigo (Servidor 1).

Texto

Descrição gerada automaticamentePor equanto não se executará um terceiro servidor. Na Máquina 1, executa-se um consumidor com o ponto de acesso ao **Daemon** **Spread** da Máquina 2, e com o ***IP*** e porto do broker **RabbitMQ**:

Figura - Execução de um servidor na Máquina 1

Note-se que o consumidor imediatamente envia uma atualização do número de consumidores ativos para o grupo **Front-End**, visto que é o líder.

Tela preta com letras brancas

Descrição gerada automaticamenteNa Máquina 2, executa-se um consumidor com o ponto de acesso ao **Daemon** **Spread** da Máquina 3, e com o ***IP*** e porto do broker **RabbitMQ**.

Figura - Execução de um servidor na Máquina 2

Assim que este segundo consumidor se junta, o primeiro, percebendo que se mantém o líder, volta a atualizar o **Front-End** com o número atual de consumidores ativos.

Texto

Descrição gerada automaticamentea Máquina 3, executa-se um consumidor com o ponto de acesso ao **Daemon** **Spread** da Máquina 1, e com o ***IP*** e porto do broker **RabbitMQ**.

Figura - Execução de um servidor na Máquina 3

Mais uma vez, o primeiro consumidor atualiza o grupo **Front-End** com o número atual de consumidores ativos, que agora são **3**.

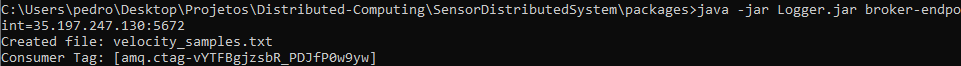
O próximo passo é iniciar a aplicação **Logger** na máquina local:

Figura - Execução da aplicação Logger

Verifica-se que criou o ficheiro velocity\_samples.txt, onde armazenará todos os valores de velocidade publicado na fila a que se subscreveu (**VELOCITY\_QUEUE\_LOG**).

Uma imagem contendo Interface gráfica do usuário

Descrição gerada automaticamenteUma imagem contendo Texto

Descrição gerada automaticamenteInterface gráfica do usuário

Descrição gerada automaticamente com confiança médiaInterface gráfica do usuário

Descrição gerada automaticamente com confiança médiaResta apenas iniciar alguns aplicações **Sensor** e utilizar a aplicação **User** para questionar os servidores acerca dos dados coletados. Executando alguns sensores:

Figura - Sensor 3, em Madrid, há 6 dias

Figura - Sensor 1, em Lisboa, hoje

Figura - Sensor 2, em Madrid, hoje

Figura - Sensor 1, em Lisboa, há 3 dias

Ainda antes de executar a aplicação **User**, excutar-se-á uma nova aplicação **Server** na Máquina 3, com acesso ao **Daemon Spread** pelo endpoint da Máquina 1. Texto

Descrição gerada automaticamente

Figura - Execução de um servidor na Máquina 3

O servidor líder (o servidor da Máquina 1, visto que ainda é o mais antigo), é reponsável pro enviar o histórico corrente (composto por uma lista de velocidades e pelo número de consumidores ativos) para este novo servidor da Máquina 3.

Texto

Descrição gerada automaticamentePara testar se esse envio foi bem sucedido, executar-se-á a aplicação **User** com o ***endpoint*** deste novo servidor:

Figura - Teste do número de consumidores ativos

Verifica-se que o servidor retorna o número de consumidores ativos atualizado (**3**). Se a mensagem de atualização dos histórico tivesse falhado, a resposta seria **0.** Se se disconectarem dois dois consumidores ativos, e utilizar-se o mesmo comando:

**Texto

Descrição gerada automaticamente**

Figura - Disconexão de dois consumidores

Texto

Descrição gerada automaticamenteTexto

Descrição gerada automaticamenteTexto

Descrição gerada automaticamenteTestam-se os restantes comandos:

Figura - Maior e menor velocidade coletada

Figura - Média numa data

Figura - Média numa cidade

Tabela

Descrição gerada automaticamentePor fim, confirma-se que a aplicação **Logger** funcionou corretamente, e que o ficheiro **velocity\_samples.txt** contém os dados publicados pelos sensores:

Figura - Ficheiro velocity\_samples.txt

# Conclusão

O projeto consistiu num sistema com várias camadas. A camada da aquisição de dados, em que se simularam sensores a obter valores de velocidade a um determinado período de tempo. Na camada Edge/Fog, estes dados são publicados para um mecanismo de Publish/Subscribe muito útil disponibilizado por um servidor RabbitMQ. Por sua vez, na camada da Cloud, as entidades que se subscreveram a esse mecanismo, recebem os eventos com novos dados e partilham-nos (filtrando-os) com o grupo Spread de servidores Front-End que estão abertos a aplicações User por gRPC. Por fim, as aplicações User permitem inquerir os dados armazenados nestes servidores. Todas estas tecnologias contribuem para a sincronização, fluxo e bom funcionamento do sistema.