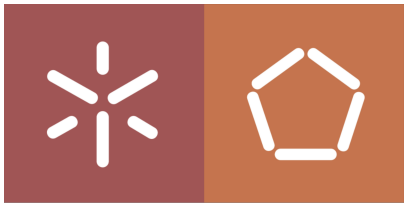

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Trabalho Prático

Relatório de Desenvolvimento

Mestrado em Engenharia Informática

Sistemas Distribuídos em Grande Escala &
Paradigmas de Sistemas Distribuídos

Desenvolvido por:

Eduardo Silva	**pg47167**	**PSD** & **SDGE**
Ivo Baixo	**pg47271**	**PSD** & **SDGE**
José Magalhães	**pg47355**	**PSD** & **SDGE**
Pedro Oliveira	**pg47570**	**SDGE**
Pedro Pereira	**pg47581**	**PSD** & **SDGE**

12 de junho de 2022

Conteúdo

Lista de Figuras

1. Introdução

No âmbito das unidades curriculares de Sistemas Distribuídos em Grande Escala (SDGE) e Paradigmas de Sistemas Distribuídos (PSD), enquadradas no perfil de Sistemas Distribuídos, foi proposto elaborar um protótipo de uma plataforma para suporte a recolha e agregação de dados, sendo estes enviados por diversos dispositivos IoT (e.g., drones, telemóveis, frigoríficos), em número potencialmente elevado.

A plataforma proposta é instanciada em vários nós, estando estes espalhados geograficamente em diferentes zonas, de forma a permitir escalabilidade e disponibilizar informação agregada. O protótipo é constituído essencialmente por dois componentes de software em cada zona, instanciados aos pares e que comunicam entre si: o Coletor e o Agregador. Por fim, a informação reunida é pública e publicada em tempo real, permitindo assim a consumidores arbitrários subscrever a mesma.

A solução proposta tem por base a aplicação prática das várias ferramentas lecionadas nas aulas de ambas as disciplinas, sendo que cada uma destas serve propósitos distintos. Acrescenta-se também que foram seguidas todas as sugestões de ferramentas presentes no enunciado.

Na componente relativa a PSD, foi utilizado **Erlang** como linguagem de desenvolvimento do Coletor; **ZeroMQ** em todos os componentes, incluindo os vários tipos de *Sockets* disponibilizados pela ferramenta, cuja utilidade e aplicação será explicada mais à frente e, por fim, **JAVA** em todos os restantes elementos, nomeadamente o Agregador - bem como em todos os serviços necessários ao funcionamento deste - e o Cliente.

Já a componente relativa a SDGE foi desenvolvida tendo em atenção não só aos pormenores e aplicação dos CRDT *(Conflict-free Replicated Data Type)*, mas também a todos os protocolos Epidémicos estudados, de forma a obter mecanismos eficientes para construir, representar, agregar e propagar informação, sendo o objetivo principal tornar estes mecanismos apropriados para grande escala, tanto em termos de dispositivos como dos eventos gerados por estes.

Acrescenta-se que a arquitetura final proposta, bem como todas as decisões tomadas pelo grupo para a construção desta, se encontram devidamente justificadas e documentadas no presente relatório.

2. Solução proposta

2.1 Cliente

Tendo em conta que, nesta entidade, havia alguma liberdade na linguagem usada para a sua implementação, optou-se por utilizar não só uma linguagem mais confortável ao grupo, mas também que já estivesse a ser usada noutros componentes (e.g. no Agregador) - JAVA.

Toda a sua interface é feita à base de Menus em formato de texto, sendo estes logicamente sequenciais e a sua interatividade resulta dos *inputs* do Cliente, diferindo estes consoante o tipo de operação pretendida (Consultar o estado global do sistema; Subscrever Notificações de uma Zona; Anular uma subscrição previamente efetuada).

De forma a controlar o fluxo do programa, são utilizadas duas *threads*:

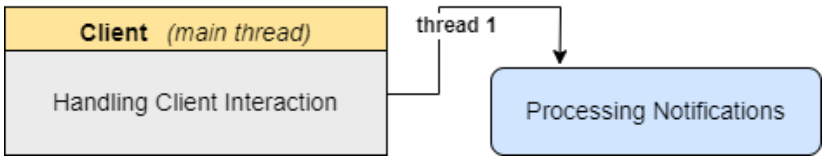

Figura 2.1: Estrutura do Cliente

- ***Main thread:*** Utilizada para toda a componente envolvente os Menus (apresentação, manipulação e processamento) e para os diversos pedidos e consequentes respostas às *"Queries"* do sistema;
- **Segunda *thread:*** Utilizada apenas para escuta e tratamento de Notificações.

Para a comunicação entre Clientes e Agregadores, e tendo em conta que esta é efetuada de duas formas distintas, com diferentes propósitos, foram usados dois tipos de *sockets* **ZeroMQ**:

- Um *socket* do tipo **SUB**: desta forma, dá-se a oportunidade a um cliente arbitrário de registar o interesse em ser notificado, em tempo real, sobre as diferentes ocorrências disponíveis na plataforma. Uma subscrição pode também ser anulada, sendo utilizado, para esse efeito, o mesmo *socket*;
- Um *socket* do tipo **REQ**: necessário para obter resposta a um determinado pedido (referente ao estado do sistema) pois, para esta resposta, é necessária uma comunicação com o Agregador, garantido assim que todos os pedidos, por parte do Cliente, são respondidos.

2.2 Bootstrapper

De forma a que cada Agregador contenha informação relativa a todo o sistema, é necessário que estes comuniquem entre si, formando assim uma rede sobreposta, pelo que esta comunicação é efetuada apenas entre vizinhos. Para implementar esta funcionalidade, decidiu-se criar uma entidade independente, em JAVA - *Bootstrapper*, com conhecimento de toda a rede - cuja função passa essencialmente por, quando solicitado, informar o Agregador em questão dos seus vizinhos podendo este iniciar a comunicação.

O funcionamento normal desta entidade passa, então, pelo seguinte:

- Partindo de um ficheiro JSON (ver A.1 para detalhes) que corresponde à topologia da nossa rede (ver A.2), procede-se então para o seu processamento e as suas diversas informações são colocadas em memória;
- Quando um Agregador se liga, o *Bootstrapper* recebe um pedido para enviar, ao Agregador em questão, as informações referentes aos seus vizinhos. Após a receção da mensagem, o *Bootstrapper* infere então quais os vizinhos do Agregador em questão e procede ao envio desta informação;
- Por fim, a informação é sintetizada numa estrutura criada para o efeito, sendo que o Agregador fica então informado não só do **ID** de todos os seus vizinhos, mas também das respetivas portas reservadas ao *socket* **PULL**.

Acrescenta-se ainda que a comunicação entre o *Bootstrapper* e os Agregadores é feita através de um *socket* do tipo **REP** devido à sua "propriedade" bloqueante, visto que, o Agregador, para estar *online*, necessita do conhecimento dos seus vizinhos.

2.3 Agregadores

Conforme sugerido pelo enunciado, toda esta entidade, bem como todas as outras criadas para completar todos os requisitos, foi implementada em JAVA. Tendo em conta o número de operações complexas e distintas que o Agregador tem que suportar, decidiu-se então utilizar, para todo este fluxo, três *threads*, cada uma com funções distintas. Além disto, foi ainda necessária a utilização do conceito de *ThreadPool*, que será visto de forma detalhada em 2.3.1.

De seguida, é possível ver um esquema da distribuição destas *threads*, no Agregador:

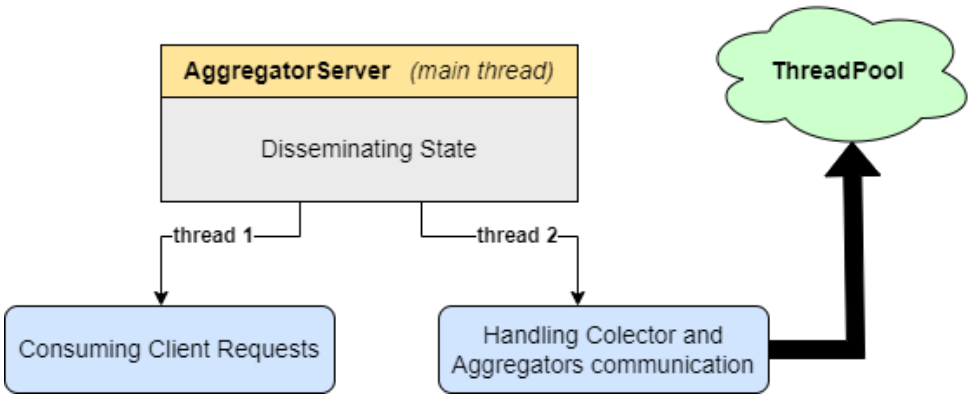

Figura 2.2: Estrutura do Agregador.

- ***Main thread:*** Responsável essencialmente por:
  - Requisitar os respetivos vizinhos ao *Bootstrapper*, sendo obrigatório para inicialização;
  - Gerar e processar os vários tipos de *sockets*, que podem ser visto com detalhe em 2.3.2;
  - Por fim, procede ainda à disseminação do seu estado pelos seus vizinhos. Esta disseminação é feita a cada segundo, de forma a que todos os Agregadores possuam informação correta uns dos outros, mas sem um exagerado fluxo de mensagens na rede.
- ***Thread 1:*** É nesta *thread* que, depois de preparada a resposta a um determinado pedido de consulta de estado, a mesma é devidamente enviada ao Cliente que a requisitou;
- ***Thread 2:*** Responsável por "Aggregation Network", esta *thread* recebe mensagens do *Socket PULL*, que podem vir do Coletor da zona ou de outro Agregador, e insere essas mensagens numa *ThreadPool* dinâmica que as processa de acordo com o seu tipo. As mensagens que chegam ao *Socket PULL* podem ser de 3 tipos:
  - Tipo **'A'** se têm um Agregador como origem;
  - Tipo **'C_Event'** se se referem a um conjunto de eventos enviado por um Coletor;
  - Tipo **'C_Device'** se representam informação sobre o estado de um dispositivo enviado por um Coletor.

2.3.1 ThreadPool

De forma a distribuir a carga das receções das mensagens que chegam através do *Socket PULL* e paralelizar o seu processamento, foi criada uma *ThreadPool* dinâmica, que começa com duas *threads* e, conforme a necessidade, pode utilizar até um máximo de quatro.

Esta *threadpool*, para além de reduzir o tempo de processamento das mensagens, permite a reutilização de *threads*, não sendo por isso necessário criar uma nova *threads* por mensagem. Dado que a criação de *threads* é um processo bastante custoso a nível de recursos, a utilização de uma *threadpool* previne um aumento deste custo, devido à reutilização das mesmas.

2.3.2 *Sockets* ZeroMQ

Tendo em conta que existem vários tipos de comunicações no nosso sistema, variando estes consoante as entidades, foram usados cinco tipos de *sockets* **ZeroMQ**:

- Para comunicação entre Agregadores e Clientes:
  - Um *socket* do tipo **PUB**: necessário para publicar eventos relativos à sua zona, a fim de serem subscritos por clientes arbitrários;
  - Um *socket* do tipo **REP**: necessário para, após processar um pedido de consulta do estado do sistema (proveniente do Cliente), enviar a resposta a esse mesmo pedido;

- Para comunicação entre Agregadores, e entre Agregador e Coletor (da mesma zona):
  - Um *socket* do tipo **PULL**: escolhido pela sua capacidade de se conseguir conectar a múltiplo *sockets* do tipo PUSH. É, então, utilizado para dois fins distintos: por um lado, recebe mensagens provenientes dos seus Agregadores vizinhos; por outro, recebe também mensagens do Coletor da sua zona. Acrescenta-se também que mais uma das vantagens deste tipo passa pela receção das mensagens em *fair-queuing*;
  - Um conjunto de *sockets* do tipo **PUSH**: cada Agregador terá um *socket* deste tipo para cada um dos seus vizinhos. Mais uma vez, é aproveitada a propriedade deste *socket* se conseguir conectar a vários **PULL** vizinhos.
- Para a comunicação entre Agregadores e *Bootstrapper*, é utilizado um *socket* do tipo **REQ**, no momento em que o primeiro pede os seus vizinhos ao segundo.

2.3.3 Disseminação epidémica

Como referido anteriormente, de forma a propagar o estado entre Agregadores, utilizou-se uma estratégia com disseminação epidémica simples que consiste no envio, por parte de um Agregador, do seu estado aos seus vizinhos. Quando um Agregador recebe um estado de um dos seus vizinhos, vai fazer a junção (*merge*) do seu próprio estado com o que recebeu (visto que os estados se tratam de *State Based CRDT's*).

O facto de se propagar esta informação a cada segundo, e não sempre que o estado atualiza com nova informação, permite que o fluxo de informação não seja um fator limitativo, evitando um congestionamento da rede. Posto isto, alcança-se um equilíbrio no *trade-off* entre eficiência (congestão na rede), e consistência de informação relativamente equilibrado.

2.3.4 Estado do Agregador - *State Based* CRDT

De forma a permitir não só que todos os Agregadores tivessem o mesmo estado, mas que o sistema desenvolvido tivesse grande disponibilidade, a estrutura (que guarda o estado de cada Agregador em memória) que se decidiu utilizar foi um **State Based CRDT**. Foi escolhido este tipo de CRDT's, ao invés de *Operation Based CRDT*, essencialmente por dois motivos:

- Não necessita da propagação de todas as operações por toda a rede, o que não seria viável para a quantidade de eventos pretendida;
- A abordagem baseada em estados funciona corretamente mesmo havendo partições de rede e/ou mensagens repetidas;
- O estado de um dado Agregador é relativamente pequeno, uma vez que apenas é preciso guardar alguns inteiros para cada zona, o que permite que os estados sejam enviados entre Agregadores sem grande custo para o sistema e para a rede.

O Estado de um Agregador consiste, então, num *Map* pelo que, para cada zona, existem os quatro *Maps* seguintes:

- *Map* que, para cada tipo de eventos, guarda o número de eventos ocorridos desse mesmo tipo (*PCounter*);
- *Map* que, para cada tipo de dispositivos, guarda o recorde do número de dispositivos *online* desse mesmo tipo (*PCounter*);

- *Map* que contém o número de dispositivos *online* para cada tipo de dispositivos (*PNCounter*);
- *Map* que guarda, para cada dispositivo, se este se encontra *online* ou *offline*.

Por fim, e para complementar o Estado do Agregador, criou-se a estrutura **Pair**, que atua como um **PNCounter** e é a partir dela que se obtém informação correta sobre cada dispositivo no estado. Neste sentido, um dos inteiros $(\mathrm{p1})$ é incrementado sempre que o dispositivo fica *online*, e o segundo inteiro $(\mathrm{p2})$ é incrementado sempre que o dispositivo fica *offline*. A partir destas duas variáveis, é possível inferir o estado de um dispositivo através da operação lógica $\mathrm{p1} > \mathrm{p2}$, sendo que o estado é *online* caso a operação seja *True* e *offline* se o contrário.

2.4 Coletores

Com o intuito de ser capaz de lidar com uma grande quantidade de dispositivos, o Coletor (implementado em *Erlang*), cria, para cada dispositivo que a ele se conecta, um novo processo. A autenticação dos dispositivos é realizada através de um *login manager* que contém no seu estado a lista de todos os dispositivos existentes (obtida através de um ficheiro *JSON* - ver A.3) e que, de cada vez que um dispositivo se quer autenticar, comunica com este módulo que verifica se os dados do dispositivo são válidos.

Para cada dispositivo autenticado, o Coletor está continuamente a receber os eventos deste - que podem ter 3 tipos diferentes: **Alarm**, **Error** e **Accident** - que estes enviam e periodicamente, com um fator aleatório para evitar um excesso de carga no Agregador que não seria representativo de um cenário normal, envia as mensagens que armazenou para o seu Agregador e limpa o seu estado.

O Coletor, para cada dispositivo mantém um registo de quando é que foi a última mensagem que este lhe enviou e se o dispositivo estiver inativo por demasiado tempo o Coletor "marca-o"como *offline*, enviando o respetivo aviso ao Agregador.

Com o objetivo que o cliente tenha uma noção de que dispositivos estão *online/offline* o mais realista possível, quando o Coletor descobre que um dispositivo terminou a sua execução (através do fecho do *socket* por onde comunicava com o dispositivo) envia logo esse *update* ao seu Agregador.

A comunicação entre um Coletor e o respetivo Agregador foi feita utilizando a biblioteca **Chumak**, que é uma implementação de **ZeroMQ** para *Erlang*.

2.4.1 Dispositivos

Com o intuito de simular uma grande quantidade de dispositivos foi utilizada a linguagem *Erlang* para o desenvolvimento de um simples módulo para este efeito. Os dispositivos possuem essencialmente três propósitos: autenticação, envio de eventos de vários tipos e alteração entre as várias zonas da rede. Neste sentido, para cada dispositivo é criado um novo processo *Erlang* que envia de forma periódica - com um certo grau de aleatoriedade - mensagens para um Coletor. No que diz respeito à alteração entre zonas, essa decisão é feita pelo processo do dispositivo, de forma aleatória e periódica.

Cada processo correspondente a um dispositivo comunica com apenas um Coletor num dado instante, sendo esta comunicação feita através de um *socket* com comunicação orientada à linha - **TCP**.

Acrescenta-se ainda que estes dispositivos podem ser de quatro tipos diferentes: **Drone**, **Car**, **Phone** e **Fridge**, sendo gerados, para fins de simulação a partir de uma *script* em *python* gerando o respetivo ficheiro *JSON*.

3. Arquitetura Final

Na figura 3.1 está presente o diagrama que representa na generalidade a arquitetura da solução criada. Nela estão presentes o tipo de *sockets* utilizados para a comunicação entre os diferentes componentes, assim como as relações entre os mesmos. Note-se também que é evidenciada a noção de "Zonas"do nosso sistema.

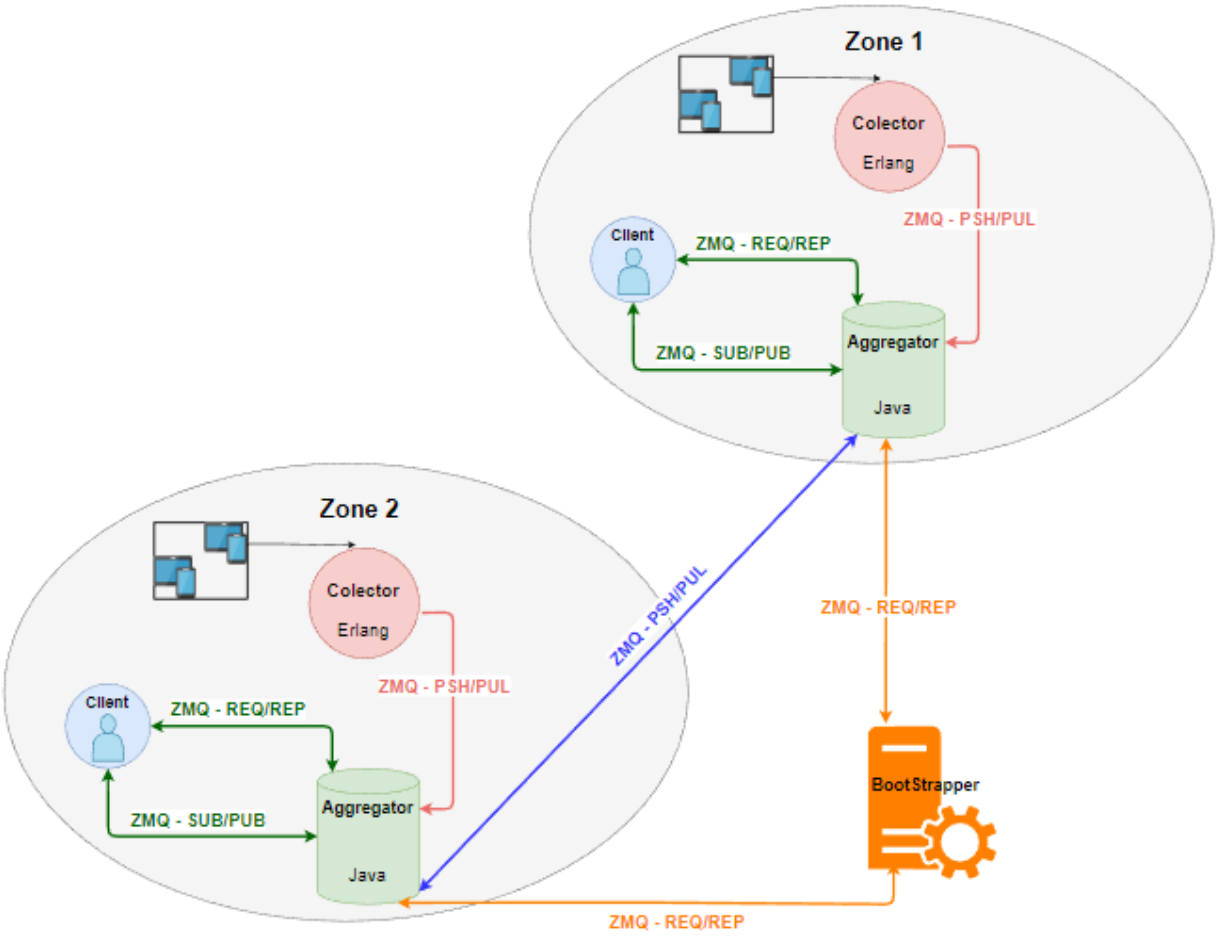

Figura 3.1: Diagrama com a arquitetura final.

4. Conclusão

De um modo geral, o sentimento que passa é que este trabalho prático, além de extremamente desafiante, ajudou bastante a consolidar a matéria lecionada em ambas as disciplinas. Reconhecemos que foi muito enriquecedor para o nosso coletivo, pois permitiu que cimentássemos todas as competências não só de *Erlang* e ZeroMQ, mas também da parte relativa aos complexos CRDTs e aos protocolos epidémicos.

Assumimos que houve alguma dificuldade na ligação de todos os conceitos, pois, além de ser uma solução que envolve um grande número de componentes, devido aos extensos requisitos, também foi necessário ter sempre presente a maior escalabilidade possível do sistema.

Todas as decisões tomadas pelo grupo, desde as estratégias escolhidas até traços fundamentais da implementação, foram sempre discutidas entre todos, pelo que o grupo acha que a solução final pode ser considerada bastante válida.

Em suma, o esforço foi grande com o intuito de garantir boas soluções para o enunciado proposto, deixando no grupo um sentimento de grande satisfação por ter cumprido todos os objetivos.

A. Anexos

A.1 Ficheiro JSON relativo à Rede *Overlay* - Exemplo

```
[
  {
    "node": "1",
    "neighbors": [2],
    "port": "8301"
  },
  {
    "node": "2",
    "neighbors": [1, 3, 4],
    "port": "8302"
  },
  {
    "node": "3",
    "neighbors": [2,5],
    "port": "8303"
  },
  {
    "node": "4",
    "neighbors": [2],
    "port": "8304"
  },
  {
    "node": "5",
    "neighbors": [3],
    "port": "8305"
  }
]
```

A.2 Representação da Rede *Overlay* - Exemplo

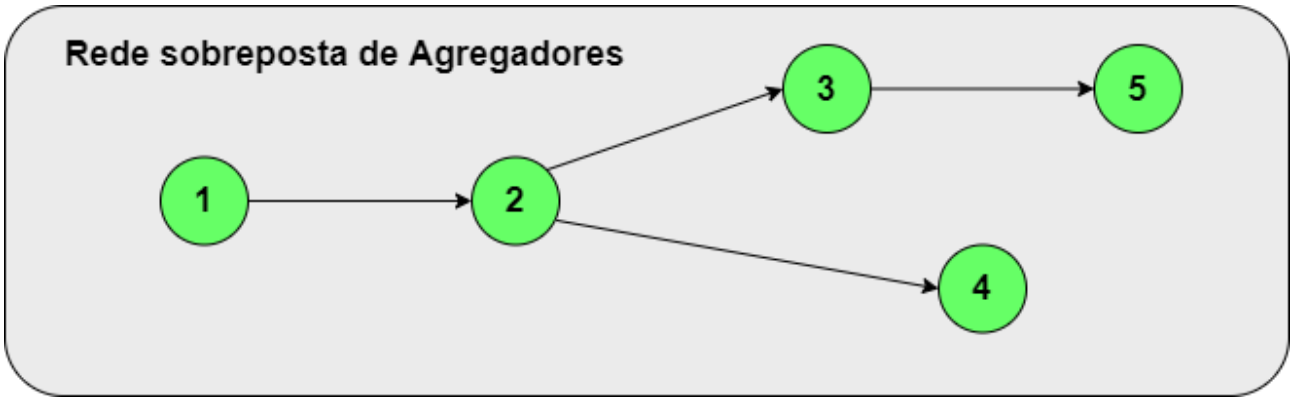

Figura A.1: Rede *Overlay* criada.

A.3 Ficheiro JSON relativo aos Dispositivos - Exemplo

```
[
    {
      "id": 1,
      "password": "pw_1",
      "type": "phone"
    },
    {
      "id": 2,
      "password": "pw_2",
      "type": "fridge"
    },
    {
      "id": 3,
      "password": "pw_3",
      "type": "phone"
    }

]
```