Universidade do Minho

Departamento de Informática

Sistemas Distribuídos em Grande Escala
Paradigmas de Sistemas Distribuídos

Agregação para dispositivos IoT

Ana Rita Peixoto	PG46988	PSD e SDGE
André Gonçalves	PG46525	PSD e SDGE
Henrique Neto	PG47238	PSD e SDGE
Márcia Teixeira	A80943	SDGE

12 de junho de 2022

Índice

1 Introdução

O presente trabalho prático insere-se no perfil de Sistemas Distribuídos, e foi realizado no âmbito das UCs de Paradigmas de Sistemas Distribuídos e Sistemas Distribuídos em Grande Escala.

O objetivo deste trabalho é elaborar um protótipo para recolher e agregar dados enviados por dispositivos, sendo necessário considerar dois componentes de *software*: coletor e agregador, que atuam em conjunto para dar resposta ao problema. Estes componentes recorrem a ***Erlang***, no caso do coletor, dada a sua capacidade de lidar com grandes volumes de clientes (dispositivos) e **Java**, no caso do agregador. Além disso, as comunicações entre eles funcionam através de *sockets* **ZeroMQ**.

2 Coletor e dispositivos

Nesta secção será abordado como foram implementados o coletor e os dispositivos, sendo que o coletor denota parte do protótipo desenvolvido e os dispositivos correspondem aos clientes que injetam carga no sistema.

2.1 Coletor

O coletor é um dos componentes do protótipo de agregação, e interage com os dispositivos da sua zona, podendo efetuar a sua autenticação, receção de eventos (podem ser alarme, avaria ou travagem) e a monitorização do estado da sua sessão. Para implementar o coletor, recorreu-se à linguagem *Erlang*, que utiliza programação orientada por atores para gerir as mensagens recebidas e os estados dos dispositivos, recorrendo à abordagem processo-por-cliente. A utilização desta linguagem deve-se à sua simplicidade e facilidade de gerir inúmeros processos em grande escala, permitindo construir um sistema mais eficiente e robusto. Relativamente à comunicação entre os elementos do sistema, considerou-se conveniente utilizar ***sockets*** **TCP** entre os dispositivos e o coletor para permitir que exista uma sessão para cada dispositivo conectado, e ***sockets*** **ZeroMQ** para efetuar a comunicação entre o coletor e o agregador, nomeadamente um *socket* do tipo **PUSH** no coletor (biblioteca *chumak* do *erlang*). A decisão de implementar este padrão **PUSH-PULL** deve-se ao facto de apenas ser necessário enviar mensagens numa única direção, do coletor para o agregador. Desta forma, o coletor irá receber todo o tipo de mensagens dos dispositivos e filtrar a informação relevante para o agregador da sua zona. Na figura abaixo podemos observar de forma representativa a comunicação entre o coletor e outros elementos do sistema.

Figura 2.1: *Sockets* presentes no coletor

Quando o coletor inicia a sua execução, efetua a leitura de um ficheiro que contém informações sobre os dispositivos da sua zona, tais como os nomes de utilizador, palavras-passe e tipo de dispositivo. Na conexão de um dispositivo o coletor cria um novo processo/ator que gere esse mesmo dispositivo e, nessa primeira interação, o dispositivo tenta se autenticar, enviando o seu nome de utilizador e palavra-passe. O processo responsável pelo dispositivo verifica a autenticidade das suas credenciais e, caso não seja um dispositivo conhecido, fecha o seu *socket* e este ator termina a sua execução. Caso contrário, o ator efetua a autenticação do dispositivo e assume que se encontra ativo, começando a monitorizar a chegada de novas

mensagens do dispositivo durante 1 minuto e, caso não receba, o ator responsável por este dispositivo assume que ele está inativo, ficando a aguardar a chegada de novas mensagens que iriam denotar que voltou a estar ativo. Adicionalmente, o coletor também é capaz de detetar caso a conexão com cada um dos seus dispositivos se perca ou seja desconectada.

2.2 Dispositivos

Estes dispositivos denotam os clientes que comunicam com o coletor, sendo identificados por um nome de utilizador e palavra-passe. Tal como referido na secção anterior, os dispositivos cliente efetuam autenticação com o coletor para posteriormente enviar eventos (alarme, avaria, travagem).

A implementação destes clientes, de modo a testar as funcionalidades do programa, foi efetuada com recurso a *Erlang* devido à sua eficiência em simular e gerir inúmeros processos ao mesmo tempo.

3 Agregador

A outra componente principal do programa é o agregador, que tem como objetivo processar as informações provenientes do coletor e de as armazenar, para que possam ser fornecidas aos clientes. A sua implementação foi efetuada recorrendo à linguagem Java e a comunicação com outros elementos do sistema é efetuada com recurso ao ZeroMQ (biblioteca *JeroMQ* do Java), utilizando os padrões mais convenientes para cada efeito. Além disso, para armazenar os dados foi necessário recorrer a CRDTs *state-based*, de forma a garantir a consistência entre os diferentes agregadores através da disseminação periódica do estado dos mesmos.

3.1 Armazenamento dos dados

Os dados do programa são armazenados recorrendo a *Conflict-free replicated data types* (CRDTs) de estado, nomeadamente a PNCounters (*Positive-Negative Counter*), PCounters (*Positive Counter*) e ORSets (*Observed-Remove Set*), de modo a dar resposta às funcionalidades propostas. Os PNCounters representam um contador, e são utilizados para manter registo dos dispositivos *ativos*, visto que o valor pode aumentar e diminuir; Os PCounters são utilizados para contabilizar os eventos recebidos por parte dos clientes, que são encaminhados pelos coletores e, por fim, o ORSet implementado pretende armazenar o nome e respetivo tipo dos dispositivos *online* no sistema, num dado momento. A utilização destes CRDTs teve como objetivo resolver o problema de sincronização de dados neste ambiente de agregadores distribuídos.

As estruturas de dados que suportam o programa guardam informações relativamente a todos os agregadores, tendo informação global do sistema. Esta informação pode ser atualizada quando o coletor envia novas informações ao agregador da sua zona (informação local), ou quando um agregador recebe a disseminação de estado de outro agregador.

Tal como supramencionado, o agregador recebe as mensagens do coletor com o auxílio de um *socket* ZeroMQ do tipo PULL, para posteriormente as processar e integrar a informação relevante nas suas estruturas de dados. No que toca à comunicação com outros agregadores, possui também um *socket* PULL do ZeroMQ para receber o estado dos mesmos.

3.1.1 Counters

Como referido anteriormente, dois dos CRDTs desenvolvidos implementam contadores. Ambos armazenam os dados de forma semelhante, sendo cada um deles um conjunto de modificações associados a cada replica que podem ser agregados no valor escalar do contador ou

reunidos com um estado para obter um estado mais recente deste. A diferença entre estes baseia-se nas funcionalidades que possuem e na memória que ocupam.

O **PCounter** implementa um contador que apenas se incrementa, o que o torna ideal para lidar com os eventos. O seu conjunto contém pares que associam a cada replica o incremento que esta contribui para o contador, sendo o valor do contador obtido pela soma de todos os incrementos registados.

O **PNCounter** consiste na expansão do contador anterior podendo este ser incrementado e decrementado. Para tal, é associado a cada réplica um valor adicional que corresponde aos decrementos que esta provocou, passando o valor do contador a ser obtido com subtração da soma dos incrementos com a soma dos decrementos. Este contador é usado para contabilizar o número de utilizadores ativos que apresenta-se constantemente a subir e a descer devido ao ambiente inferido pelos dispositivos.

3.1.2 ORSets

De forma a poder existir escrita concorrente, e consistência eventual em todo o sistema relativamente a informações cruciais para o funcionamento do mesmo, foram utilizados ORSets especificamente para lidar com o fluxo de informação relativo aos dispositivos online.

Num *Observed Remove Set* cada elemento está associado a um conjunto (*set*) de tags únicas. Estas tags são um par composto pelo identificador do agregador, e um contador local incremental. Este formato é utilizado para garantir a unicidade da informação armazenada pelos agregadores, uma vez que a utilização de um inteiro incremental para identificar esta informação em todos os nodos levaria à existência de conflitos relacionados com a concorrência. Desta forma, cada operação recebida por um nodo fica por si identificada, em adição ao contador incremental mantendo-se esta tag quando estes disseminam a sua informação pelos restantes nodos. No momento de remoção de um dispositivo, esta informação é facilmente passada e compreendida pelos restantes nodos. Em adição, dado que é assumida a entrega causal de informação, nunca há uma sequência de operações a chegar a algum nodo na ordem errada, ou seja, não existem remoções de elementos que não existem (por exemplo).

As operações possíveis num ORSet são a adição - *add*, remoção - *remove* e consulta de elementos - *elemens*. Uma operação *add* local cria uma nova tag para o elemento, um *remove* local efetua a remoção de todas as tags do elemento e uma operação de consulta dos elementos *elements* apresenta todos os elementos existentes.

3.2 Rede *Overlay*

Um dos elementos do agregador de elevada importância é a componente da rede, que foi construída com recurso ao algoritmo **SCAMP**, oferecendo a cada agregador uma visão local do sistema, e não global. Desta forma, cada agregador apenas conhece alguns vizinhos, e a rede está estruturada de forma que exista sempre um caminho entre cada 2 nodos, garantindo disseminação atómica.

A implementação deste algoritmo passou por utilizar *sockets* ZeroMQ do tipo **PUSH** e **PULL**, permitindo implementar este padrão, de forma a simplificar a comunicação entre os agregadores. Assim, cada agregador possui o *socket* PULL para receber informações de outros nodos, e possui um *socket* PUSH por cada um dos seus vizinhos, de modo a poder enviar lhes mensagens.

No que toca à disseminação do estado do agregador, esta é feita periódicamente com recurso à rede, em que cada agregador envia todo o seu estado para os seus vizinhos, e assim sucessivamente. Conseguimos garantir atomicidade dado que a rede representa um grafo fortemente ligado.

Na figura abaixo podemos observar uma representação esquemática da rede e dos *sockets* utilizados em cada agregador. A título de exemplo, consideremos o agregador central da imagem, e podemos observar que existe um *socket* PULL que recebe mensagens de todos os nodos que o tem como vizinho (lado direito da imagem) e também possui vários *sockets* [PUSH]

que correspondem aos seus vizinhos, para os quais pode enviar mensagens (lado esquerdo da imagem).

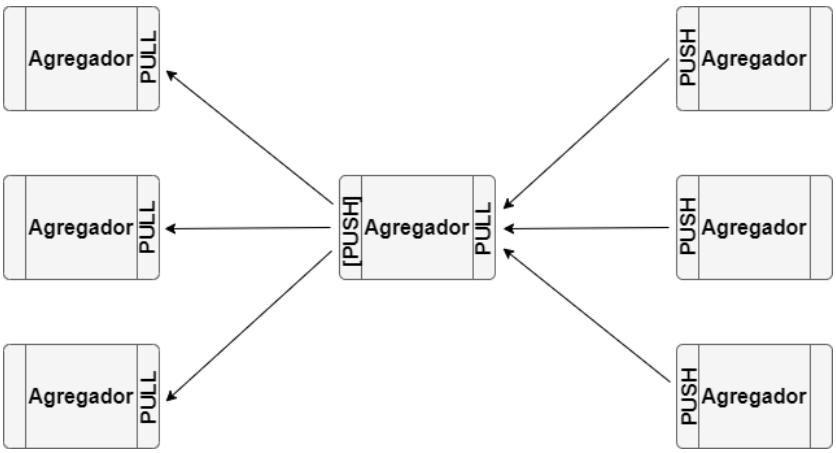

Figura 3.1: Implementação do PUSH-PULL na rede

3.3 Serviço

O agregador possui uma componente serviço que tem como propósito responder às *queries* do programa, solicitadas pelos clientes. Esta componente tem um *socket* do ZeroMQ do tipo **ROUTER** que permite receber vários pedidos em simultâneo resultando assim na capacidade de servir vários clientes de forma concorrente. A implementação desta concorrência baseia-se em *threadpools*, sendo que após a sua receção cada pedido é submetido como uma tarefa no serviço de execução do componente, na qual este irá inevitavelmente ser recolhido por uma das *threads* presentes e processado. No final de cada processamento, é enviada a resposta do pedido ao respetivo cliente.

Os pedidos enviados pela aplicação do cliente são compostos pelo tipo de procura, que pode ser global ou local, opção escolhida e possivelmente um argumento. O agregador responde aos pedidos com uma mensagem, que a aplicação do cliente imprime.

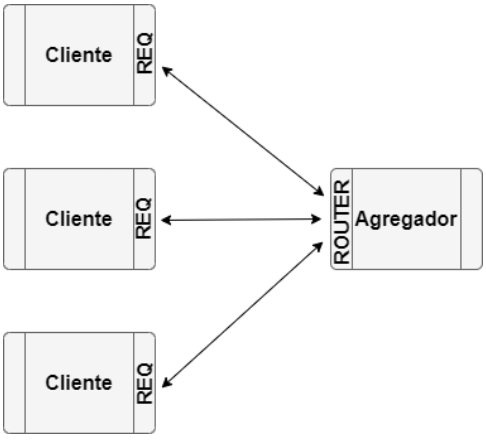

Figura 3.2: Implementação do REQ-ROUTER

3.4 Notificações

Para implementar as notificações recorreu-se ao padrão *publisher-subscriber*, que se traduz no ZeroMQ por *sockets* PUB e SUB. Desta forma, o agregador possui um componente para lidar com a publicação de eventos para os clientes que estão subscritos a um dado tópico. A forma de resolver este problema passou pela utilização de uma *thread* que monitorizara a ocorrência dos eventos necessários, ficando esta adormecida num *await* enquanto não houver novas informações a publicar para os clientes. Quando as estruturas de dados são alteradas e podem originar uma notificação, a *thread* é sinalizada e acorda, e notifica os clientes associados, caso seja o caso.

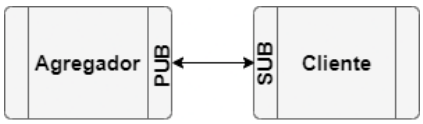

Figura 3.3: Padrão PUB-SUB agregador

3.5 Clientes

Para testar as funcionalidades e *queries* do programa, foi necessário implementar um programa cliente que tem como propósito efetuar pedidos ou subscrever a notificações.

Este cliente tem conhecimento acerca do endereço do agregador com o qual pretende comunicar. Através do seu interface no terminal, pode optar por efetuar subscrição a uma dada notificação ou enviar pedidos ao agregador. A aplicação simplesmente imprime as respostas e notificações do agregador no terminal.

A aplicação do cliente possui um *socket* ZeroMQ do tipo REQ (figura 3.2) para realizar pedidos ao agregador e uma *socket* ZeroMQ do tipo SUB (figura 3.3) para subscrever a notificações.

4 Conclusão

Dado por concluído este trabalho prático, consideramos que se tratou de uma boa oportunidade para colocar em prática os conceitos aprendidos nas aulas de Paradigmas de Sistemas Distribuídos, como a programação por atores (*Erlang*) e os *sockets* ZeroMQ, assim como os diferentes padrões suportados e a sua adequação a cada problema. No âmbito de Sistemas Distribuídos em Grande Escala conseguimos por em prática os CRDTs de estado, mais concretamente os contadores e conjuntos. Além disso, também foi utilizado o algoritmo SCAMP para a construção da rede *overlay*.