Local-First Shopping List Application

Desenvolvido por:
- João Miguel da Silva Lourenço (up202108863)
- Tiago Ribeiro de Sá Cruz (up202108810)
- Tomás Filipe Fernandes Xavier (up202108759)

Índice

Descrição do Problema

O objetivo deste projeto é desenvolver uma lista de compras partilhada, onde diferentes utilizadores podem contribuir ao adicionar ou remover itens.

Esta lista deverá ser local-first de modo que, mesmo que o utilizador não tenha internet ou, por alguma razão, a comunicação cliente-servidor esteja em baixo, ele continuará a poder utilizar o sistema.

Utilização CLI

- python Client.py <create|use listid|view listid|delete listid> userid [operations]"
- Operações: Add / Remove / Delete

Exemplo:

- python Client.py create user1 add banana 2
- python Client.py use e2a2d7d6f97c94feb12b183393e690da user1 delete banana
- python Client.py use e2a2d7d6f97c94feb12b183393e690da user2 add banana 3 remove bandaid 1
- python Client.py view e2a2d7d6f97c94feb12b183393e690da user1
- python Client.py delete e2a2d7d6f97c94feb12b183393e690da user1

Implementação da CRDT

A **CRDT** utilizada é baseada numa **Positive-Negative CRDT**. Possui 2 contadores: um **positive counter**, para incrementação de quantidades a um item; e um **negative counter**, para decremento de quantidades a um item.

```
# Separate increment and decrement counters for each item
self.p_counter: Dict[str, Dict[str, int]] = {}
self.n_counter: Dict[str, Dict[str, int]] = {}
```

Cada contador é representado da seguinte forma: {'item': {'node':quantidade,...},...}.

Para lidar com deleções de itens é adicionado ao **n_counter** o valor da diferença da quantidade desse item no **p_counter** com o **n_counter**, ou seja, o valor efetivo da quantidade desse item.

Este tipo de deleção seria considerado *soft-deletion*, pois, embora para o user seja equivalente a uma deleção, internamente a **key** desse item não é eliminada dos **counters**.

Implementação da CRDT (cont.)

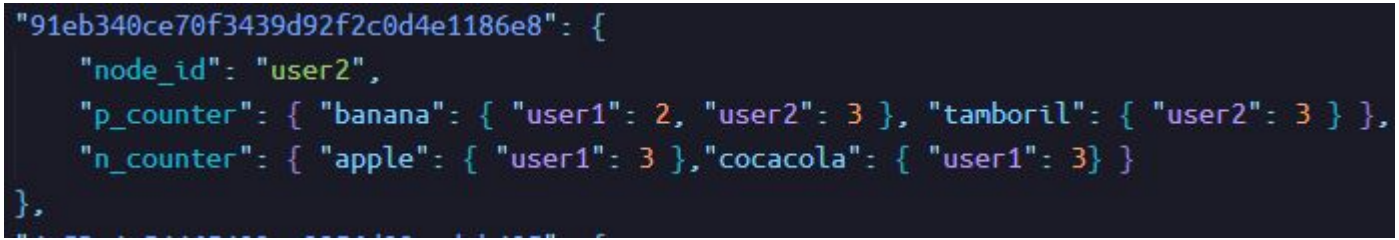
- O merge utiliza a estratégia de valor **máximo,** o que permite garantir idempotência e que todos os nodes eventualmente convirjam para um estado equivalente.
- Para retrieval de informação da quantidade dos itens é utilizada a função **get_state()**, que retorna um dicionário com cada item e a sua quantidade. Este cálculo é realizado somando o valor da quantidade desse item em cada node no **p_counter** e subtraindo o resultado da mesma operação no **n_counter.**

Manutenção do Estado

Para manter o estado, tanto no cliente como nos servidores, utilizamos ficheiros **JSON** que contém informação das **CRDTs**, cada **CRDT** representando uma shopping list.

Em baixo está parte do ficheiro **data.json**, onde é mantido o estado do cliente.

Contém o ID da lista de compras, o ID do node (a máquina do cliente) e os positive e negative counters.



ZMQ - Client

- Socket **REQ** ligada à socket **Router** do Broker, simplificando a implementação do Client e permitindo o Broker lidar com vários clientes ao mesmo tempo e identificar quem enviou a mensagem, de modo a enviar ao cliente correto.
- É enviado um **JSON** com toda a informação do pedido (type, id, node, p_counter, n_counter)
- De modo a permitir o envio de vários pedidos sem receber resposta usamos: socket.setsockopt(zmq.**REQ_RELAXED**, 1).
- **3 tentativas** por pedido, **timeout** de **5 segundos.**
- **Poller** para receber a resposta do Broker, de modo a ser facilmente escalável.
- Em caso de erro usamos socket.setsockopt(zmq.**LINGER**, **0**) de modo a mensagens pendentes serem descartadas após a chamada à função de fechar a socket.

ZMQ - Broker

- 2 sockets **Router**, uma ligada para os Clients e outra para os Workers.
- De modo a garantir que o Broker envia a resposta do Worker ao Client correto, adicionamos um **request_id** usando um UUID a cada mensagem do Client.
- Sistema **Heartbeat** com os Workers, de modo a que, caso esteja algum em baixo, o Broker não irá enviar pedidos do Client a esses Workers, evitando assim congestão.
- **2 Pollers** diferentes para receber pedidos dos Clients e dos Workers de forma assíncrona e não congestionar o sistema.

ZMQ - Worker

- Socket **REQ** ligada à socket **Router** do Broker, do mesmo modo que no cliente, simplifica a implementação do Worker e permite o Broker lidar com vários clientes ao mesmo tempo e identificar quem enviou a mensagem, de modo a enviar as mensagens ao Worker correto.
- Socket **REP** para receber pedidos de outros Workers e socket **REQ** para enviar pedidos para outros Workers, de modo a não ocupar a socket do Client e simplificar o código.
- **Timeout** inter-workers, mas **sem retries**, de modo a **priorizar disponibilidade** em vez de consistência.
- Utiliza o sistema de **Heartbeat** para detetar se o Broker está em baixo. Se estiver, começa a tentar reconectar-se ao Broker.

Dynamo

- A implementação do Dynamo resume-se num **anel** de **consistent hashing**.
- Cada servidor é um **node** do anel.
- Cada list id ao ser hashed é associado um servidor, de forma que, se o anel não mudar, cada id vai ser associado ao mesmo servidor respetivo.
- Cada lista é replicada nos dois servidores que seguem o servidor a que foi associada, portanto, no código, muitas vezes referimos os **“Three servers”** porque essencialmente o ring associa a cada lista 3 servidores consecutivos.
- Ao adicionar um servidor novo, este é adicionado ao anel o que significa que os hashes vão mudar entre o último servidor do final e do início do anel. Portanto é feita uma adaptação, de forma aos servidores ficarem corretamente integrados no anel depois de uma adição, com a informação corretamente propagada entre eles.

Dynamo (cont.)

- Outro aspeto do Dynamo que replicamos é que quando o cliente manda um request de save a um servidor, este começa uma operação em que tenta dar merge dessa lista que recebeu com os outros 2 servidores que a partilham, assim mantendo um nível de consistência. No entanto, só forçamos que um desses servidores receba a lista atualizada, mantendo também um nível de disponibilidade.

A Faltar

- É impossível remover um servidor. Caso um servidor seja desligado, o **Broker** irá assumir que ele foi abaixo e não o retira da lista de servidores existentes. Isto poderia ser resolvido com um **Servidor Admin** e um **Cliente Admin**.
- **Hard-delete** de keys nas **CRDTs**. Desta forma, só itens realmente na lista de compras fariam parte dos positive e negative **counters**, o que poderia poupar memória no long-term.
- Da forma atual, para criar um servidor novo, o **port** atribuído a esse servidor é diretamente relacionado ao número atribuído a esse servidor na sua criação, ou seja, ao criar um “server1”, o número 1 vai ser acrescentado ao **port**. Isto é uma forma muito rudimentar, e poderia ser resolvido com a criação de um **Servidor Admin**, que funcionasse como um **Router** que atribui automaticamente **ports**, independentemente do nome atribuído ao server na sua criação.