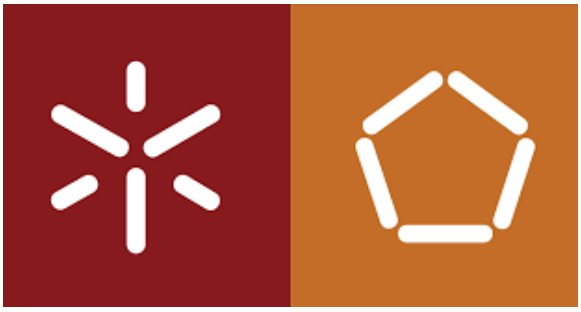
Universidade do Minho
Escola de Engenharia



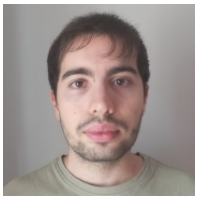
PSD/SDGE

Mestrado em Engenharia Informática

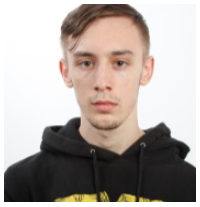
Relatório de Desenvolvimento da app "Partilha de Álbuns"

André Lucena Carlos Machado
Gonçalo Sousa Guilherme Outeiro

PG52672



PG52675



PG52682

A94984

18 de maio de 2024

1 Introdução

Nas unidades curriculares **PSD** e **SDGE** foi-nos proposto o desenvolvimento de uma *app* de partilha de álbuns. Este sistema permite a gestão colaborativa de álbuns. O servidor central, em *Erlang*, autentica utilizadores e mantém réplicas de metadados dos álbuns, enquanto os servidores de dados em *Java* garantem a persistência dos ficheiros. A comunicação entre clientes e servidor central é via *sockets TCP/IP*, com interação *peer-to-peer* entre clientes suportada pelo *ZeroMQ* para edição colaborativa dos álbuns e discussões em tempo real. Já a comunicação entre cliente e servidor de dados é via *reactive-grpc*.

2 Cliente

2.1 Tecnologia

O código-fonte dos Clientes foi escrito na linguagem *go*, utilizando a sua facilidade de criar *goroutines* e as suas bibliotecas de controlo de concorrência para que estas possam agir cooperativamente, nomeadamente primitivas atómicas e *locks* mutuamente exclusivos. A comunicação entre o Cliente e os restantes componentes utiliza *protobufs* para a sua estandardização.

2.2 Sessão de Edição de Álbum

Uma sessão de edição é definida por um conjunto de clientes a abrirem a edição de um álbum, que tenha sido criado anteriormente por qualquer cliente. Apenas os membros de um álbum podem editá-lo, e os membros de um álbum podem adicionar ou remover membros. Apenas quando a edição é fechada por um membro é que o estado é propagado para o Servidor Central, o que implica que é necessário um cliente fechar a edição para um novo cliente ter a disponibilidade de aceder ao álbum.

Quando um cliente se junta a uma sessão de um álbum (ou a cria, se for o primeiro), é-lhe atribuído um *id* que o irá representar junto do Servidor Central. Dois clientes nunca mantém o mesmo *id* simultaneamente, mas quando um cliente sai da sessão o seu *id* poderá ser reutilizado.

2.2.1 *Router Socket*

A comunicação entre os clientes de uma sessão é feita por *sockets* ZeroMQ, nomeadamente um único socket `Router` por cliente, que trata de receber as mensagens relativas tanto à sessão do álbum como ao *chat*. Escolhemos este

socket por um lado para poupar no número de *sockets* e por outro porque este *socket* garante fiabilidade na entrega das mensagens.

Quando se junta a uma sessão, cada cliente define um nome utilizado como referência do *Router Socket*, de acordo com o *id* que lhe foi atribuído, convertendo também todos os *ids* que recebe do servidor central dos seus *peers* nesse formato, para se poder usar no *Socket*.

2.2.2 Estado do Álbum

Para a transferência de dados de um álbum entre os diversos clientes, como se trata de comunicação que deve ser sincronizada *peer-to-peer*, definimos dois CRDTs distintos com a mesma semântica, **Observed-Remove Set**, um para os Ficheiros adicionados e outro para os Membros da Sessão adicionados. Efetivamente, os valores que irão representar estes objetos serão os **nomes dos ficheiros** e os **usernames**, guardados num mapa cujo *DotStore* utilizado foi um *DotSet*.

Para lhe adicionar ou remover valores, os clientes invocam um comando que altera o seu próprio CRDT. Essa alteração será propagada a todos os outros membros da sessão através de um método `Heartbeat` inicializado ao mesmo tempo que a Sessão, e é ouvido por outra *goroutine* `PeerListen`, que executa respostas *handler* para as mensagens recebidas.

A outra alteração de estado é invocada pelo Servidor Central quando um novo cliente se junta ou abandona a sessão de edição. Estas mensagens são lidas por outra *thread* que está responsável pela comunicação TCP, e que tem de controlar a concorrência com os restantes manipuladores do estado da sessão.

FileInfo

No caso dos ficheiros, outros dados foram guardados junto com o mapa físico que representa o CRDT, nomeadamente o *hash* calculado para o ficheiro, `FileHash`, e o mapa com os votos dos clientes, `Votes`.

O `FileHash` tem de ser concordado entre as réplicas, quando dois ficheiros com **o mesmo nome** são adicionados concorrentemente em dois clientes diferentes. Nesse caso, a ordem de desempate define-se pelo *id* dado pelo Servidor Central, onde o menor *id* tem maior prioridade e, então, manterá o seu ficheiro em relação aos restantes. Este valor é determinado de acordo com o *dot* registado no CRDT com menor *id*, quando se faz a união do mesmo, independentemente do *id* do cliente em questão, perpetuando assim as decisões dos outros.

Votes

O `Votes` é um mapa que associa a cada *id* dos clientes dois valores, um `Count` que conta o número de avaliações feitas àquele ficheiro por aquele *id*, e um `Sum` que acumula todas as avaliações feitas. A partir destes valores, para cada *id*, é possível calcular uma média das avaliações feitas para cada ficheiro. Estes valores usufruem um **Single-Writer Principle** já que cada *id* é único para cada cliente na Sessão, então não irá haver conflitos necessários de resolver por mecanismos mais poderosos.

Note-se que dois clientes que partilhem, ao longo do tempo, um *id* irão escrever no mesmo local do CRDT os seus votos, apenas acumulando ao seu anterior. Isto garante o pedido no enunciado, de que o espaço do CRDT cresce apenas até ao número máximo de utilizadores simultâneos a editar o álbum.

VoteMap

Para garantir que um cliente não vota duas vezes no mesmo ficheiro, é guardado um mapa local de todos os ficheiros em que um cliente já votou. Este mapa é temporário e existe apenas enquanto está registado numa sessão. Quando um cliente sai de uma sessão, o mapa é enviado para o Servidor Central com o CRDT e registado àquele álbum. Sempre que um cliente se junta a uma edição de um álbum, recebe o mapa com os seus votos e impõe em si próprio a restrição dos ficheiros nos quais já tinha votado.

2.2.3 *Chat*

O *chat* de comunicação entre os clientes é construído ao redor da primitiva de comunicação *causal broadcast*, de modo a entregar as mensagens por uma ordem causal. O *chat* é criado ao mesmo tempo que a sessão, e utiliza o mesmo *Router Socket*. No entanto, o CRDT inicial não é controlado pelo Servidor Central, mas pelos outros *peers* que já se encontram no *chat*. Para isso, um cliente pode encontrar-se em um de dois estados: à espera de um `Version Vector` ou na posse de um. Quando um cliente se junta ao *chat*, e não for o primeiro que se juntou, pede a um que já esteja no *chat*, isto é, a um dos seus *peers* iniciais, o seu CRDT.

O momento definido pelo CRDT é considerado como o "presente inicial"do novo cliente. Quaisquer mensagens entretanto enviadas pelos seus *peers* são guardadas, mas apenas entregues após a receção do CRDT e se estas forem no futuro deste mesmo CRDT, as restantes mensagens que tiveram origem antes são descartadas já que não há forma de saber a sua

ordem.

2.3 Servidores de Dados

As últimas funcionalidades dos clientes são fornecidas pelos Servidores de Dados. Quando um cliente entra no serviço, o Servidor Central avisa-o de todos os Servidores de Dados atualmente a correr. Sempre que um novo servidor é adicionado, o Servidor Central avisa todos os atuais membros dessa atualização. Para colocar o Servidor de Dados no local correto, o cliente executa uma **procura binária** na lista ordenada pelos *hashs* máximos atribuídos aos servidores. Este algoritmo é reutilizado noutros locais da solução.

As duas funcionalidades são o *download* e o *upload* de ficheiros. O cliente executa uma procura binária na sua lista de Servidores de Dados e encontra o servidor mais adequado, requerendo/enviando por **gRPC** uma *stream* dos dados, sem nunca esforçar demais a sua memória local. Este processo também é utilizado quando se prentede calcular o *hash*.

Sempre que um ficheiro é adicionado ao CRDT, antes o cliente determina o seu *hash* e envia-o para o servidor de dados correspondente, para se certificar que no CRDT existem apenas ficheiros que existem também no Servidor de Dados.

2.4 Comandos

Os seguintes são os comandos disponíveis a um utilizador para interagir com a sua interface do cliente.

- `register name password` - Registo do utilizador;
- `login name password` - Autenticação do utilizador;
- `createAlbum name` - Criação do álbum;
- `getAlbum name` - Pedido de início de sessão;
- `putAlbum` - Pedido de término de sessão;
- `listReplica` - Apresentar todo o CRDT;
- `listFiles` - Apresentar a média dos votos para todos os ficheiros;
- `addUser/removeUser name` - Adicionar/remover um utilizador ao álbum;
- `addFile/removeFile name` - Adicionar/remover um ficheiro ao álbum;

- `downloadFile` - Descarregar o Ficheiro dos servidores de dados;

3 Servidor Central

3.1 Adição de Servidores de dados ao Cluster

Para incorporar uma adição dinâmica de servidores ao *cluster*, criamos um processo *erlang* que fica responsável pela coordenação da adição desse novo servidor e por avisar os clientes da nova alteração, a lógica reside no módulo *data_loop*, quando um novo servidor é adicionado, usamos os mesmos algoritmos para aceder e adicionar servidores que os clientes e buscamos o servidor que terá possivelmente parte dos dados que serão, agora, corespondentes ao novo, e notificamos o novo servidor com o IP e Porta do mesmo. Além disso notificamos o cliente sempre que é adicionado um novo servidor (para os que estão online) e ao logar, com a lista de servidores disponíveis.

3.2 Álbuns

Para guardar cada um dos álbuns, guarda-se a mesma estrutura CRDT que cada um dos clientes. Esta é recebida no final da edição de cada cliente e o servidor executa o mesmo *Join* nos CRDTs como os clientes. Notavelmente, o estado do álbum é limpo quando a sessão termina, colocando apenas o par $(0, 0)$ em cada *DotSet*, apenas o *dot* $0 \rightarrow 0$ e colapsando todos os votos num único utilizador, já que não é esta a estrutura que verifica quais os ficheiros em que um utilizador já votou. Estas reduções permitem a que a próxima sessão criada sobre o álbum recomece com todos os *ids* dos clientes e diminuindo substancialmente o tamanho do CRDT. Não existe conflito porque a primeira ação do primeiro cliente é sempre marcada com a versão a 1. Os metadados dos álbuns são persistentes em ficheiro sempre que são propagados para o servidor.

3.3 Comunicação com Cliente

Para comunicarmos com o cliente estruturamos as mensagens com o auxílio de *protocol buffers*. Essa lógica reside no ficheiro *user_logic*, cada cliente terá um processo leve correspondente que terá os seguintes estados:

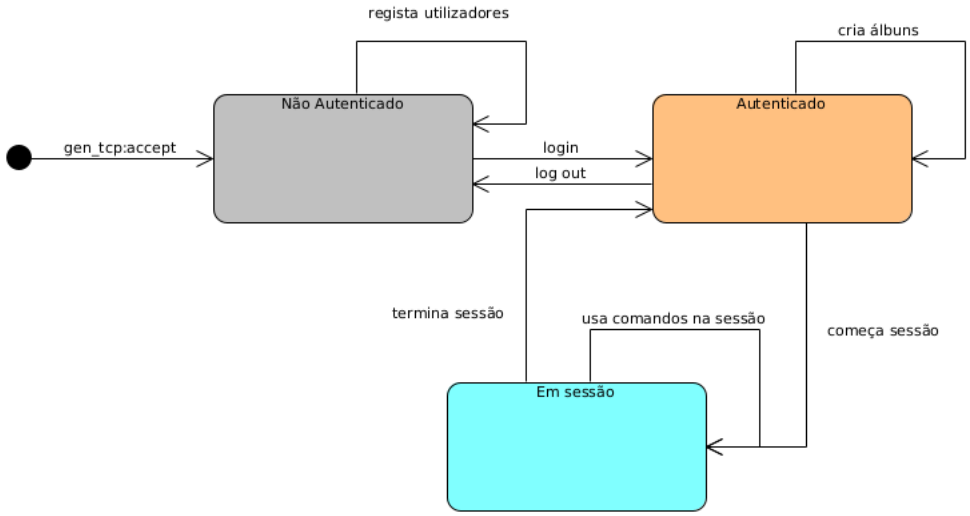


Figura 1: Possíveis estados do Cliente

3.4 MainLoop e Sessões

O cerne do servidor concentra-se no ficheiro *mainLoop*, é ele que guarda todos os utilizadores e faz a gestão dos registos, *logins*, notificar adição de servidores, criar álbuns e, por fim, criar sessões. Cada sessão é um processo leve diferente para tirar proveito da escalabilidade e do isolamento que o *erlang* oferece, desse modo, se uma sessão falhar, apenas essa sessão é afetada. Para simplificar ainda mais a lógica de sessões e guardar metadados, cada sessão fica responsável por cuidar dos metadados do seu álbum. Uma possível melhoria futura seria ter *Supervision Trees* para reiniciar as sessões na ocorrência de falhas.

4 Servidor de Dados

O servidor de dados implementa um serviço *grpc* reativo com 3 *rpc's* diferentes:

- *Download* - Permite os clientes fazerem *download* dos ficheiros de um servidor sob forma de *stream*, é aberto um ficheiro através do *Flowable.generate* que ao contrário de um *Flowable.create* com um *while loop* o que faria com que a produção dos *chunks* não fosse controlada, o *generate* vai emitindo de forma controlada, o client ao aceder faz

subscribeOn no *IO scheduler* uma vez que as operações de leitura de ficheiros são *IO-bound*, por fim convertemos numa mensagem *FileMessage* e é enviado para o cliente;

- *Upload* - Permite os clientes fazerem *upload* de ficheiros para um servidor sob a forma de *stream*, usamos uma classe *fileWriter* que funciona como descritor para a stream e também dá lógica extra de se existir aquele ficheiro no servidor de dados, devolve logo ao invés de voltar a escrever por cima do ficheiro existente, senão vai a escrever em stream;

- *Transfer* - o *rpc Transfer* serve para os servidores trocam informação entre si aquando da adição de novos servidores de modo que os ficheiros com *Hash* menor que o Novo servidor, sejam transferidos para o mesmo. A lógica de é semelhante à do *Download* só que com certificações extra de que os ficheiros pertencem ao novo servidor ou não, e contar com o facto do *hashing* ser circular nos ponteiros do relógio.

5 Trabalho Futuro e Conclusão

Em suma, neste projeto foram criados um cliente e dois servidores, um central e um de dados de modo a criar um sistema distribuído de edição de álbuns, para isso foi implementado *causal broadcast*, para mensagens *peer to peer* e um mecanismo de *crdt* para manter um estado partilhado entre os utilizadores. Em termos de dificuldades, manter a comunicação coerente entre o *go* e o *erlang* tornou-se laborioso com o crescimento do programa, principalmente porque a segunda linguagem não tinha forma de comunicar erros de tipagem que eram comuns por alterações subtis nos *protobufs*. Propomos como possível trabalho futuro a extensão do algoritmo de *causal broadcast* com utilização de *broadcast* epidémico, para reduzir o volume de mensagens a passar pela rede. Além disso, poderia ser implementado um serviço de *middle ware* entre os servidores de dados e os clientes de modo que os endereços destes estivessem guardados apenas nesse serviço, este passo aumentaria ao realismo do projeto e aumentaria a transparência da aplicação para o cliente.