Trabalho Prático

PSD / SDGE

Guilherme Sampaio	Luís Pereira	Miguel Braga	Rui Oliveira
PG53851	PG54009	PG54095	PG54210

1. Introdução

O presente relatório é referente ao trabalho prático das Unidades Curriculares de Paradigmas de Sistemas Distribuídos e Sistemas Distribuídos em Grande Escala do Perfil de Sistemas Distribuídos do Mestrado em Engenharia Informática da Universidade do Minho.

Sucintamente, o projeto consiste em implementar um sistema de criação, edição e partilha de álbuns multimédia com grande escalabilidade, recorrendo a várias tecnologias e técnicas abordadas em ambas as Unidades Curriculares.

Neste documento, a Secção 2 discute a arquitetura do sistema e das suas diferentes partes em detalhe. De seguida, a análise experimental é discutida na Secção 3. Por fim, a Secção 4 discute as principais conclusões do projeto, assim como potencial trabalho futuro.

2. Arquitetura

2.1. Servidores de Dados

Os servidores de dados devem formar uma *hash table* distribuída (DHT), em que o mapeamento de chaves por servidores se faz através de *consistent hashing*. Formalmente, a *hash table* deve mapear *strings* para ficheiros, em que as chaves correspondem ao cálculo da *hash* SHA-1 do ficheiro.

Tratando-se de uma chave criptográfica de 160 bits, pode-se assumir que não existirão colisões de *hashes* entre ficheiros, o que simplifica bastante o sistema, visto que garante que dois *writes* para a mesma chave escrevem sempre o mesmo valor, não existindo problemas de *writes* concorrentes.

Os servidores não implementam qualquer tipo de autorização, sendo a chave suficiente para aceder ao ficheiro. Adicionalmente, cada servidor é responsável por várias regiões do anel, isto é, o servidor gera, ao arrancar, um número configurável de *tokens* - *hashes* SHA-1 de conteúdo gerado aleatoriamente, que determinam as posições no anel. As posições correspondem à *hash* módulo um valor configurável (por omissão igual a 2^{32}).

Quando um RPC de escrita ou leitura chega a um servidor, este primeiramente determina se é responsável pela *hash* em questão. Um servidor é responsável por uma *hash* se e só se, avançando no anel a partir da posição dessa *hash*, o primeiro *token* a ser encontrado for do servidor.

Em caso afirmativo, o servidor processa o pedido, senão devolve um erro. Os pedidos de um RPC de escrita chegam em *stream*, no entanto, estes são considerados independentemente, para não haver necessidade de manter estado entre pedidos. No entanto, isto implica abrir o ficheiro, realizar um *seek* para a posição correta, escrever os dados, e fechar o ficheiro a cada *chunk* do pedido, o que pode impactar o desempenho. A resposta a este RPC é única.

Contrariamente, um RPC de leitura consiste numa só mensagem que, em caso de sucesso, originará uma *stream* de respostas, cada uma com um *chunk* do ficheiro. Todos os ficheiros são armazenados em disco, numa diretoria configurável. O servidor de dados interpreta qualquer ficheiro nessa diretoria como sendo membro da *hash table*, em que a *hash* corresponde ao nome do ficheiro.

2.1.1. Entrada de Servidores

Um dos requisitos da DHT é suportar a entrada de novos nós, um de cada vez[1]. Por isso, quando um servidor se quer juntar à DHT, este envia uma mensagem ao servidor central que, se não houver nenhum outro servidor a entrar na DHT, devolve os endereços e *tokens* de todos os servidores da DHT.

Com essa informação, o novo nó envia um RPC de transferência de dados para os servidores que cobrem áreas do anel que passarão a ser sua responsabilidade. Os servidores determinam as *hashes* que devem transferir, e respondem com RPCs de escrita, como se um cliente normal se tratasse, de forma a reaproveitar essa funcionalidade. As *hashes* a transferir são todas as *hashes* que, sendo previamente responsabilidade do servidor antigo, com a entrada do novo passarão a ser responsabilidade deste. Quando todos os servidores terminam as suas respostas, o novo nó notifica o servidor central de que o seu processo de entrada terminou, e que este pode aceitar novas entradas.

Um aspeto importante a analisar é o que acontece a pedidos de clientes afetos a uma área que está a ser transferida para um novo servidor. A partir do momento em que um servidor pede para entrar na DHT ao servidor central, este passa a encaminhar todos os pedidos de escrita para o novo servidor. No entanto, isto implica que, por um lado, o servidor antigo deixa de ser capaz de responder a todas as *hashes* dessa área, mas; por outro lado, o servidor novo ainda não tem todas as *hashes* que necessita (senão já teria entrado) e, por isso, também não pode responder a qualquer pedido de leitura.

Para resolver este problema, o servidor central encaminha o cliente para ambos os servidores. Se o primeiro a ser contactado tiver a *hash*, responde ao pedido normalmente. Senão, o cliente contacta o outro servidor, que é garantido ter a *hash* em questão.

2.2. Servidor Central

O servidor central, concebido como uma aplicação Rebar3 para ser mais facilmente utilizado, desempenha um papel crucial na gestão de informação dos álbuns, informação dos clientes e como Tracker para a DHT.

Visto que a comunicação entre os clientes e este servidor deve ser feita via Sockets TCP foi criado um protocolo de comunicação, mediante `Protobuf`, para a serialização e desserialização das mensagens entre estes. Como tal, foi necessário utilizar uma dependência externa sob a forma de um compilador `Protobuf` para `erlang` chamado `gpb`.

Para gerir estes dados, o servidor encontra-se dividido em 4 processos-base:

- Receiver Client
- Account Manager
- DHT Manager
- Albums Manager

Um esquema de todos os processos pode ser visualizado na Figura 1.

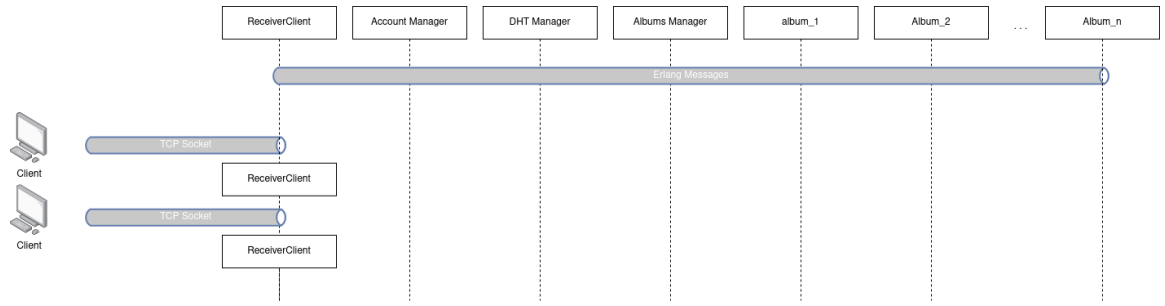

Figura 1: Processos Servidor Central

[1]Esta restrição é necessária para garantir que um nó não entra para uma região do anel que já esteja a ser transferida para outro servidor, simplificando todo o processo.

2.2.1. Receiver Client

Este processo é responsável por receber e manter conexões via Sockets TCP e processar os pedidos recebidos posteriormente contactando os outros processos conforme as mensagens recebidas.

Por cada nova conexão irá ser criado um processo para atender o próximo cliente, havendo sempre um total de "Número_de_clientes_ativos" + 1 processos deste tipo. No entanto, estes estarão maioritariamente bloqueados a espera de novas mensagens.

2.2.2. Account Manager

Este processo guarda o estado de todos os utilizadores do sistema guardando quais são as suas credenciais assim como quais álbuns cada cliente tem acesso e por fim qual o Endereço e Porta para outros clientes comunicarem. Sendo assim possível verificar permissões de um utilizador e as suas credenciais.

2.2.3. DHT Manager

Este processo é utilizado como um Tracker para a DHT dos servidores de dados, guardando o par $(\text{porta}, \text{endereço})$ do nó responsável por cada secção do anel. Este processo permite a entrada de novos nós na DHT bloqueando a entrada caso outro nó esteja no processo de se juntar. Por fim, calcula também qual o nó para os quais os clientes devem enviar ou obter um ficheiro.

2.2.4. Albums Manager

Este processo gere a criação de novos Álbuns assim como qual processo está associado a este, ou seja, sempre que um novo álbum é criado um processo também o é, guardando o respetivo PID. Apenas existe um processo Albums Manager, no entanto, este cria processos Album Manager, um por álbum existente.

2.2.5. Album Manager

Este processo gere a informação dos metadados de cada um dos álbuns, quem tem permissão para o editar e que utilizadores o estão a editar de momento, assim como qual a posição e valores do *vector clocks* que novos utilizadores podem utilizar.

Devido à gestão de uso de posições dos *vector clocks* serem feitas no servidor central, é possível a reutilização de posições, levando a um uso linear no número de utilizadores que editam o álbum concorrentemente.

2.3. Cliente

Os clientes dentro de uma determinada sessão de edição formam uma rede peer-to-peer, realizando a edição de forma descentralizada com CRDTs. Optamos pelo uso de Operation-Based CRDTs de forma a tornar a comunicação entre clientes mais eficiente, dado o menor volume de dados trocados comparativamente com a abordagem State-Based.

Distinguem-se 3 tipos de informação a armezanar na sessão de edição:
- Metadados dos Ficheiros (nome e *hash*);
- Informação dos Utilizadores com acesso ao álbum;
- Classificação de cada ficheiro;

Quanto aos ficheiros e utilizadores, uma vez que deve ser dado suporte a inserções e remoções, optamos pelo uso de *orsets* (*observed-removed sets*). Quanto às classificações, e de modo a garantir eficiência quer no cálculo da classificação média quer na verificação da votação de dado utilizador, recorremos a 2 estruturas diferentes: *grow-only set* para armazenar as classificações de cada cliente para cada ficheiro e *p-counter* para acumular os totais de cada ficheiro.

Um componente essencial da nossa aplicação é o *middleware* de envio de mensagens que implementa o algoritmo *causal broadcast*. Este é um dos requisitos para a implementação dos *orsets* e também já nos garante entrega causal de mensagens para o serviço de chat.

2.3.1. Entrada e Saída de Clientes

Apesar de tornar o processo de edição mais eficiente, a opção pelos CRDTs *operation-based* dificultou o processo de entrada e saída de clientes, uma vez que é necessário assegurar que nenhuma mensagem é perdida e, portanto, o estado mantém-se consistente entre todos os clientes. Se tivéssemos optado por uma abordagem *state-based* esse processo seria simplificado, uma vez que o estado dos vários nós poderia ser conciliado através do *merge* dos dois estados, dada a garantia que os dois nós em conjunto receberam todas as mensagens.

2.3.1.1. Entrada de um novo cliente

O processo de entrada inicia-se com o contacto ao servidor central, que devolve a réplica do álbum no caso do cliente ser o primeiro da sessão ou o conjunto dos clientes ativos em sessão, no caso de já não ser o primeiro. O caso mais interessante é o segundo, uma vez que o cliente que se junta não sabe do estado da sessão. Será necessário, por isso, contactar um dos clientes da sessão.

Este processo inicia-se com o envio de uma mensagem de *join* a identificar o cliente que entra e a pedir o envio do estado. O cliente que recebe essa mensagem vai então enviar o seu estado atual para esse novo cliente, juntamente com as mensagem que já recebeu mas não fez *deliver*. Paralelamente, envia também um pedido a cada outro cliente a informar a entrada de um novo elemento na sessão.

Contudo, não é garantido que todas as mensagens enviadas antes do ponto de contacto do novo cliente sejam contemplados nesse estado. Por isso, o cliente que envia o estado terá que assumir o papel de encaminhador de mensagens até receber de todos os outros clientes uma mensagem com um *timestamp* superior ao do envio da mensagem que informa a entrada de um novo cliente.

2.3.1.2. Saída de um cliente

O processo de saída tem algumas parecenças com o processo de entrada, na medida em que não queremos que o último cliente a sair não seja possuidor de toda a informação enviada durante a sessão. Para isso, quando um cliente sai, envia a todos os outros a informação que quer sair, o que permite aos outros clientes saber que este não enviará mais mensagens. Esse cliente espera por confirmações de todos os outros, antes de efetivar essa decisão, enviando mensagem de saída para o servidor central. Assim, garantimos que o último cliente a sair é possuidor de todas as mensagens, uma vez que tem de ter recebido mensagens de confirmação de saída por parte de todos os outros clientes da sessão. Uma vez que temos garantias FIFO, esse processo não precisa de esperar por mais nenhuma mensagem, uma vez que após um cliente indicar a sua saída, não envia mais mensagens.

É necessário, ainda assim, evitar situações que 2 ou mais clientes decidem sair de forma concurrente, havendo cruzamento das mensagens com pedidos de *leave* de ambos os processos, com o potencial de ocorrer uma situação de *deadlock*. A forma de evitar isto é dar prioridade ao processo com *id* mais alto, abortando a saída do outro processo.

2.3.2. Otimização nos *Vector Clocks*

Um dos requisitos do enunciado era tornar certas partes de estruturas de dados da uma réplica lineares no número de utilizadores que editaram o álbum concorrentemente, e não no total de utilizadores do álbum. Esta otimização foi aplicada aos *Vector Clocks* enviados entre clientes através da primitiva *Causal Broadcast*.

A forma encontrada foi delegar ao servidor central a responsabilidade de atribuir a cada cliente novo que se junta um índice e um valor inicial para o vetor. É necessário, por isso, que cada cliente indique

ao servidor central qual o valor final do seu índice aquando da sua saída, por forma a permitir a reutilização do mesmo noutro cliente.

2.4. Comunicação entre clientes

Tal como é pedido no enunciado, a comunicação entre clientes é feita através de ZeroMQ, neste caso recorrendo a *sockets* do tipo *ROUTER*. A opção por este tipo de *sockets* deveu-se à facilidade de criar com estes *sockets* uma rede *peer-to-peer* em que todos podem comunicar com todos, tenha havido o *connect* entre quaisquer dois pontos da rede. Em relação a este ponto, uma vez que a operação *connect* é assíncrona[2], foi necessário esperar pelo estabelecimento da conexão antes de começar a enviar mensagens. Para enviar uma mensagem para outro *socket*, o *socket* emissor envia uma mensagem *multipart* (identidade do destino, dados), em que a identidade do destino é um valor inteiro distinto atribuído a cada processo.

2.5. Funcionamento geral

A aplicação cliente assenta em duas *threads* que comunicam por *sockets* ZeroMQ. Uma *thread* lê do *Standard Input* os comandos digitados pelo utilizador e envia esses comandos para o *socket* do tipo *ROUTER* onde a segunda *thread* espera por novos pedidos. Ao receber um pedido, a *thread* que está à escuta no *socket* verifica se esse é um pedido do próprio cliente ou é uma mensagem vinda de outro cliente (e.g adicionar um ficheiro). Ao receber uma mensagem, é invocado o *Controller* correspondente, mediante o tipo de mensagem: operação no servidor central, operação nos CRDTs ou gestão de sessão (*leave* e *join*).

3. Análise Experimental

3.1. Teste de causalidade e convergência do lado do Cliente

Para a entrega de mensagens ser causal, se uma mensagem A ocorrer no passado causal de uma mensagem B, então a mensagem B não pode ser entregue até que a mensagem A tenha sido entregue em todos os nós do sistema. Isto quer dizer que, apesar de um determinado nó C poder receber primeiro a mensagem B, a sua entrega deverá ser adiada até ser recebida a mensagem A.

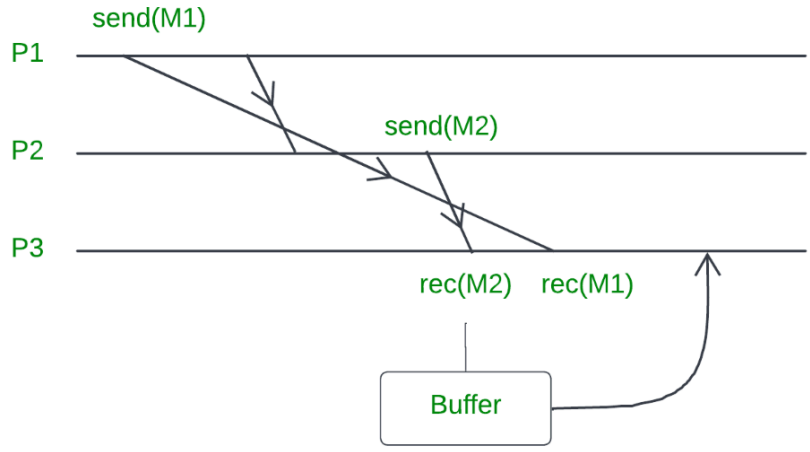

Figura 2: Causal Delivery (imagem retirada de Geeks For Geeks).

[2] A presença de um *slow joiner* pode levar à perda de mensagens, impactando negativamente a comunicação entre clientes.

No nosso sistema, isso foi testado provocando *delays* no envio de mensagens de um determinado nó para outro, mas mantendo o tempo de envio das restantes mensagens. Por exemplo, aumentando o tempo de envio de uma mensagem do nó 1 para o nó 3, é possível que o nó 2 receba a mensagem do nó 1 e envie de seguida uma mensagem para o nó 3, chegando antes da mensagem enviada pelo nó 1. O comportamento correto é a entrega da mensagem do nó 2 só seja entregue quando for entregue a mensagem do nó, algo que foi observado em análise experimental.

De forma semelhante, foi testada a convergência do estado dos clientes, com particular foco nos momentos de entrada e saída de nós.

4. Conclusão

Ao longo deste trabalho, desenvolveu-se um sistema de edição colaborativo de álbuns multimédia, recorrendo a um conjunto de tecnologias e paradigmas abordados nas Unidades Curriculares. Com isto, o grupo considera ter cumprido com todos os objetivos propostos, tendo este sido um projeto bastante bem sucedido.

4.1. Trabalho Futuro

No futuro, poder-se-ia fazer uma análise experimental mais detalhada, incluindo uma avaliação profunda do desempenho e escalabilidade do sistema e seus subcomponentes, em função do número de clientes, e nós da DHT. Adicionalmente, poder-se-ia ter implementado mais testes automáticos para validar informalmente que o sistema funciona corretamente. Para complementar estes testes, embora fora do âmbito do perfil, seria interessante fazer uma prova formal da correção dos diversos algoritmos implementados.

Relativamente a arquitetura, Poder-se-ia ter criado um arquivo com todos os álbuns que não têm nenhum utilizador a editá-lo no processo `Albuns Manager`, baixando o número de processos concorrentes. Em geral, poder-se-ia reduzir a dependência do sistema do servidor central.