INF01151 Sistemas Operacionais II N
Relatório do Trabalho Prático 2

Augusto Falcão Flach e Nikolas Sousa Weige
03 de abril de 2023

1 Introdução:

O objetivo deste trabalho foi complementar uma aplicação similar ao Dropbox desenvolvida na primeira parte da disciplina, utilizando sockets e TCP(Transmission Control Protocol). A aplicação já permitia ao usuário o compartilhamento e a sincronização automática de arquivos entre diferentes dispositivos de um mesmo usuário, e após o desenvolvimento passou a permitir o uso de servidores backups transparentes ao usuário. O foco do trabalho foi a implementação de um **esquema de replicação passiva, com eleição de líder** através do **algoritmo do anel**.

A máquina utilizada para testes foi um Ubuntu 22.04.1 LTS, com 8GB RAM e um i7 7ª geração. O compilador utilizado foi o G++. O aplicativo possui um arquivo para compilação chamado comp.sh, para facilitar o processo.

2 Replicação Passiva:

A fim de aumentar a disponibilidade do sistema implementado na parte 1 do projeto, foi implementado um esquema de **replicação passiva**, onde um servidor primário de Réplica Manager (RM) é responsável por gerenciar uma ou mais instâncias secundárias de RM, também conhecidas como backups.

Com a introdução deste modelo, é possível garantir que a falha do servidor principal não levará à indisponibilidade de serviço para os clientes. A replicação passiva garante que as modificações de arquivos estarão disponíveis em um novo servidor, pois todas as modificações realizadas são informadas aos servidores secundários. Para que todo o processo fosse feito de forma transparente ao cliente, foi adicionado um **front-end** entre a comunicação do cliente e o conjunto de RMs.

A replicação passiva foi um processo mais simples de ser implementado. Ela consiste de três blocos majoritários, que são: (1) a sincronização inicial; (2) o monitoramento de falhas (por um servidor de “heartbeat”); (3) e a atualização de dados locais (arquivos e estado de conexões).

1) A sincronização inicial usou de boa parte das funções desenvolvidas na etapa um. Cada servidor de backup que conecta ao RM primário entra imediatamente nessa etapa. Ali, ele entra em uma fila para aguardar que todas trocas de arquivo (entre o RM primário e quaisquer clientes) sejam finalizadas. Novas trocas são impedidas de serem iniciadas.
   A partir disso, é feita a comunicação entre o RM primário e o de backup para que o backup crie todas as pastas existentes no primário, e todos os arquivos. Quando finalizado o processo de sincronização, as conexões são liberadas para voltarem a trocar arquivos com o RM primário, e o RM de backup entra nos estágios 2) e 3) (concorrentemente). As implementações

de bloqueios mencionadas acima são feitas com o uso de mecanismos de sincronização, pois senão não funcionariam corretamente.

2) O processo de RM primário possui um servidor específico para que cada RM de backup se conecte. Ao fazer isso, ambos iniciam uma thread para a troca de mensagens. Essa troca acontece de forma alternada, a cada X(=5) segundos. O objetivo da troca é identificar quaisquer erros em um dos processos (ao realizar a leitura, caso sejam lidos 0 bytes). Caso um dos RMs de backup seja comprometido, o servidor de RM primário os removerá da lista de RMs ativos. Caso o servidor primário seja comprometido, cada RM de backup que identificar o erro iniciará uma eleição. Ou seja, como o RM primário está conectado a todos RMs de backup, pelo menos alguém irá perceber o erro.

   Dificuldades: Uma dificuldade foi fazer com que a sincronização de dados de controle (vetores de conexões) entre o RM primário e os de backup fosse feito TANTO a cada ação dos clientes conectados, QUANTO a cada heartbeat. Isso seria ideal para que a sincronização não fosse feita somente por ações de clientes (como o atual), pois caso não existam clientes, os demais RMs de backup não teriam sincronizado e recebido a lista de seus "irmãos". A dificuldade se deu porque a sincronização e o heartbeat são feitos em threads diferentes, e implementar isso diretamente faria com que o envio de mensagens pudessem ser intercaladas de forma ruim, fazendo o servidor de destino se perder. Acredito que não seja difícil resolver isso com mecanismos de sincronização, mas eu preferi evitar fazer isso sem analisar mais profundamente a possível existência de deadlocks, e de garantir que as mensagens seriam sempre interpretadas na ordem correta.

3) Essa foi a parte mais simples. Existem duas sincronizações: as de arquivos; e as de dados de controle. As de arquivos acontecem a cada troca de arquivos entre RM primário e seus clientes. Quando um arquivo é alterado, ele também é propagado para todos os RMs de backup, e somente então a alteração é confirmada para o cliente (literalmente, com a função send_OK_packet/wait_for_OK_packet). Novamente, isso acontece a cada troca de arquivos, e somente sobre os arquivos alterados. Isso é viável devido à sincronização inicial, pois sabemos que o RM primário e os backups possuem os mesmos arquivos.

   O segundo tipo de sincronização é de dados de controle, o qual foi brevemente comentado na dificuldade em (2). Isso engloba os vetores de conexões do RM primário com todos clientes, e do RM primário com todos os RMs de backup, cada qual com seus dados de conexão. Isso acontece a cada loop de troca de mensagens, pois uma dessas mensagens pode ser justamente um cliente se desconectando.

   Dificuldade: Aqui, a maior dificuldade foi esse fator de somente sincronizar os dados mediante interação com clientes. Isso faz com que, caso não haja nenhum cliente, e o RM primário seja comprometido, a eleição não aconteça (visto que nenhum RM de backup conhece os demais). Uma ideia de solução para isso foi comentada na dificuldade em (2).

3 Eleição de Líder:

Esta parte teve como objetivo a implementação de um algoritmo de eleição de líder em um sistema distribuído, no qual diferentes backups podem assumir o papel de servidor primário. Na primeira parte do trabalho, assumiu-se a existência de um único servidor Dropbox, porém agora, é necessário selecionar o próximo servidor primário por meio de um dos algoritmos de eleição, onde o implementado foi o **algoritmo do anel**. Será garantido que o mecanismo atualize as informações sobre o novo líder nos front-ends dos clientes, a fim de garantir que o serviço seja transparente para os usuários e que os arquivos estejam disponíveis mesmo após a falha do servidor principal.

O algoritmo escolhido foi o **algoritmo do anel** por possibilitar a eleição de um líder de forma simples e com um baixo número de mensagens, além de que o ambiente de testes (descrição do trabalho) não contemplava falhas em outros RMs.

A aplicação do algoritmo se deu pela troca de uma estrutura de “voto”, que contém outros dados além do voto de cada nodo em si. Isso foi feito para que o servidor de “eleição” de cada nodo saiba sempre o tamanho da mensagem a ser recebida, pois ela não irá variar. Essa estrutura carrega o maior **id** até o momento, e carrega dados de “vitória” da eleição. Também carrega um estado de “eleição finalizada”, para que, ao receber um objeto dessa estrutura, o nodo saiba como interpretá-lo (como eleição em andamento, ou como o resultado da eleição finalizada).

Isso poderia ser feito através da troca de mais mensagens, mas essa foi uma simplificação que deixou a implementação bastante mais simples e direta, por padronizar o número de bytes a serem lidos.

Também, o “próximo nodo” do anel é dado após uma ordenação de todos os RMs por um critério bem-definido, para que seja consenso quem é o “próximo nodo” de cada RM, e que eles se organizem de forma unânime.

Assim, cada RM possui um servidor de “eleição” aguardando conexões e mensagens, e também cada RM que detectar a falha no sistema inicia uma “votação”.

Dificuldade: Foi encontrada uma dificuldade no momento de conexão com o “próximo nodo”, devido ao fato de que o valor propagado (desde a inicialização do RM até seu fim) para a “porta” de comunicação do servidor do nodo, foi na verdade a porta utilizada pelo RM Primário para conexão com o mesmo, e não a porta em que seu servidor de backup estava rodando. Dito isso, seria necessária uma refatoração mais profunda do código para fazer isso de forma correta. Nada muito difícil, mas com inúmeros testes para evitar a existência de “corner cases” não previstos. Também não terminamos de implementar a parte onde o servidor vencedor envia um sinal para que os front-ends se conectem consigo. Também fizemos uma solução para ressincronização entre os clientes e as threads do RM, que envolve exceções e tratamento de exceções, que consideramos interessante e criativa, mas que no entanto não chegou a ser utilizada (pois não finalizamos a parte de reconexão ao RM eleito).

4 Dificuldades:

Nas descrições acima, comentamos dificuldades pontuais, pois acreditamos que façam mais sentido em seus contextos. Outras dificuldades foram a falta de expertise na

linguagem, a dificuldade de um uso de ponteiros que contemple todos os “corner cases”, e a dificuldade de testar tudo em “local”. Sabemos que temos o laboratório do INF, mas querendo ou não é difícil programar em um computador e ter que levar a todos computadores a cada teste. Ainda mais debugar. Por isso, coisas que funcionavam perfeitamente, na hora do teste não funcionaram por razões bizarras (mas que não encontramos antes pela falta de testes prévios em sistemas diferentes e distribuídos).

O maior problema, encontrado no laboratório, e que acontecia **às vezes** (a pior parte), é esse tipo de código abaixo:

```
CLI_CONNECTION_DATA cli_conn_data;
cli_conn_data.sock_fd = sock_fd;
strcpy(cli_conn_data.username, username);

pthread_create(&cli_connection_thread, NULL, cli_connection_loop, (void *)&cli_conn_data);
```

Esse erro aconteceu em algumas partes do sistema, algumas vezes, e é difícil de encontrar (pois é muito silencioso, e não gera erros ou exceções). Em nosso ambiente de testes, como o computador é mais rápido que no INF, geralmente não acontecia.

O erro é que a variável ***cli_conn_data*** é inicializada em uma função local. E seu endereço é passado para a thread **cli_connection_loop**. Como a variável é local, algumas vezes, ao iniciar o processamento da thread, como a função já terminou, o antigo endereço da variável já está sendo usado para algo diferente, e possui valores diferentes, e completamente arbitrários. As outras vezes, não.

E ele era ainda mais escondido pois as primeiras 4 mensagens com o RM primário (e o front-end) eram trocadas normalmente! Somente após elas a thread era iniciada, e a possibilidade de erro existia. Inicialmente, acabamos culpando o front-end, mas o problema se encontrava nesse tipo de código.

Dessa forma, o código correto para resolver esse problema seria:

```
CLI_CONNECTION_DATA *cli_conn_data = (CLI_CONNECTION_DATA *)malloc(sizeof(CLI_CONNECTION_DATA));
cli_conn_data->sock_fd = sock_fd;
strcpy(cli_conn_data->username, username);

pthread_create(&cli_connection_thread, NULL, cli_connection_loop, (void *)cli_conn_data);
```

Pois faz uso de malloc para alocar o espaço dinamicamente. Isso resolve o problema sempre (pelo menos, até agora).

Esse tipo de problema apareceu pelo menos 4 vezes ao longo do código, e até identificar qual a causa do mesmo, levou pelo menos 8h de *debugging* somado. Não é o fim do mundo, mas é apenas um dos tipos de problemas que apareceram. Explicitando assim, a sensação é de que é simples, que faz sentido, e que é só não fazer de novo, mas em nossa experiência (e por não dominarmos absolutamente a linguagem), esse tipo de situações “escondidas” acontecem com frequência. Que parecem corretas, mas no fundo não são.

Como esse foi um dos problemas mais estressantes, e que nos orgulhamos de encontrar a solução, achamos interessante mencionar!

UPDATE: Removi e atualizei a entrega porque a capivara aqui esqueceu de mandar o código. Coloquei ele no ZIP, mas se quiser aqui também tem o repositório, porque fica mais fácil de gerenciar.

https://github.com/aufgfua/dropbox-sisop2