

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA

ANDY RUIZ
CASSIANO JAEGER
GABRIEL STEPIEN
GUILHERME CHAVES

Relatório da parte 2 do trabalho prático da disciplina
de Sistemas Operacionais 2.

Professor: Weverton

Porto Alegre
2020

SUMÁRIO

SÚMULA	3
ARQUITETURA	4
VISÃO GERAL	5
ConnectionKeeper	5
ConnectionMonitor	5
Client	5
Front End (FE)	6
Server	8
Group, GroupManager e MessageManager	8
BÔNUS	9
Docker	9
Transparência perante falhas	9
Noção de mensagens dos clientes 2 e 3 ficando bloqueadas	10
INSTALAÇÃO	11
Como executar	11
QUESTÕES	12
Funcionamento da eleição de líder	12
Algoritmo de Eleição	14
Implementação da replicação passiva	15
Estrutura de replicação	15
Replicação na prática	16
Link para o vídeo da apresentação	17
Conclusão	17

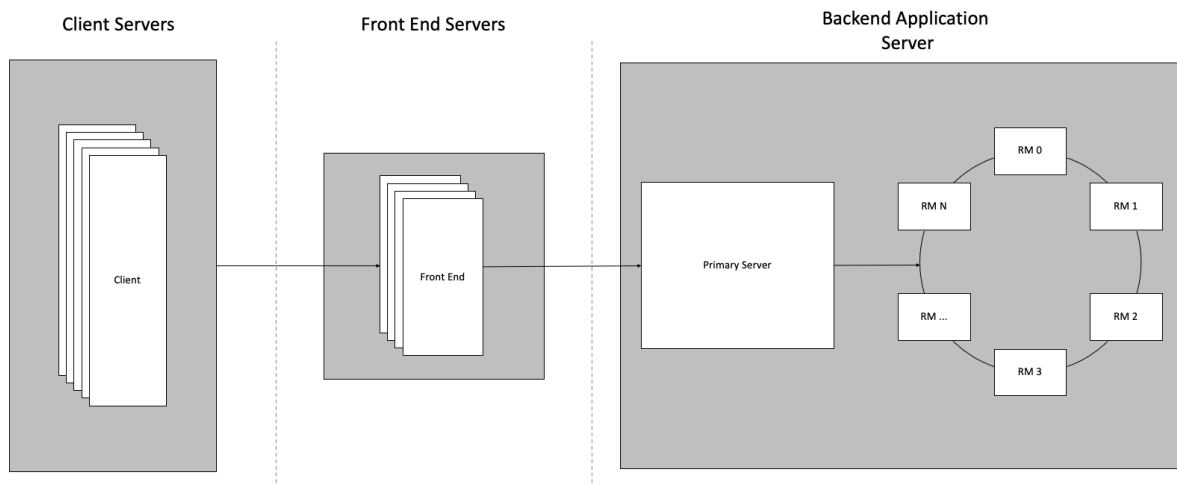
SÚMULA

Para esta etapa implementamos todos os requisitos listados na especificação do Trabalho Prático 2 e adicionamos algumas características extra em forma de bônus. A seguir listaremos e explicaremos os principais pontos da nossa arquitetura de forma semelhante a como foi feito no relatório da primeira entrega deste mesmo trabalho. Inclusive, assume-se que o leitor esteja ciente das explicações da primeira entrega pois usaremos muitos conceitos definidos lá.

Após a visão geral, explicaremos as características extras que implementamos e o que nos motivou a fazer isto, além de qual foi o ganho proporcionado por esses recursos. Por último, responderemos às questões levantadas na especificação do trabalho prático 2, havendo contextualizado com a nossa solução através das seções prévias.

ARQUITETURA

A seguir explicaremos os principais pontos de cada componentes da nossa arquitetura e como ele desenvolve sua responsabilidade. Parte-se do ponto em que foi entregue a primeira parte do trabalho prático e como funcionava o sistema naquele então.



Nova arquitetura

PS. essa imagem da arquitetura não considera a conexão entre todos os RM para a replicação, mais informações no fim do relatório.

VISÃO GERAL

A seguir explicaremos os principais pontos de cada componentes da nossa arquitetura e como ele desenvolve sua responsabilidade. Parte-se do ponto em que foi entregue a primeira parte do trabalho prático e como funcionava o sistema naquele então.

ConnectionKeeper

Criamos esta classe para garantir um keep alive dos nosso serviços. O seu funcionamento é bem simples e trivial: após estabelecer uma conexão via socket, se dispara uma thread com uma instância desta classe que ficará enviando mensagens do tipo keep alive de tempos em tempos para o socket ao que se conectou.

ConnectionMonitor

Esta classe coopera com a anterior para controlar um possível time-out da outra ponta da conexão. Para cada conexão aberta, se dispara uma thread com uma instância desta classe que ficará verificando de tempos em tempos quando foi recebido o último keep alive desta conexão.

É preciso que a aplicação que gerencia a leitura de Packets recebidos informe quando foi recebido um packet do tipo keep-alive. Assim, se atualiza o timestamp de quando foi recebido o último keep-alive para a conexão sendo monitorada. Se por algum motivo o último keep-alive que se tem constância aconteceu antes de um intervalo máximo de tempo definido, se considera que houve desconexão e se dá time-out na conexão fechando a mesma.

Os classes que a utilizam, precisam criar uma thread para inicializar o monitoramento. Elas devem dar join nesta thread para na sequência chamar as rotinas específicas para a desconexão devido ao time-out.

Client

O cliente apenas recebeu algumas modificações desde a sua última versão, as mudanças foram relacionadas à integração com a nova arquitetura: disparar uma thread que envie keep-alives para o FE e outra thread que monitore os keep-alives do FE para detectar se ele caiu. Também foi adaptado para usar um pseudo-protocolo na comunicação, semelhante ao SIP por exemplo, onde se envia um Packet do tipo INVITE para dar join em uma sala e espera por uma confirmação, uso de ACK's para envio de mensagens e coisas do gênero.

Agora que temos a noção de Front End na nossa arquitetura, o cliente se comporta da seguinte forma: um FE é aleatoriamente escolhido dentre uma lista de configuração onde constam todos os FE's existentes e se conecta a ele.

Vale ressaltar uma implementação nova que de fato aconteceu do lado do cliente, relacionada a uma tomada de decisão nossa para deixar a arquitetura mais interessante e o sistema, como um todo, mais transparente e tolerante a falhas: foi implementada uma fila de mensagens. De forma a como acontecia na classe Group no trabalho prático 1, agora existe uma fila de mensagens e uma noção de produtor e consumidor nesta fila. A motivação e melhor funcionamento serão explicados na seção bônus deste relatório, mas de forma geral a fila de mensagens é consumida apenas quando se recebe um ACK para cada mensagem lida. Desta forma, a mensagem $n+1$ só será processada e enviada quando a mensagem n for enviada e se recebeu seu ACK.

Front End (FE)

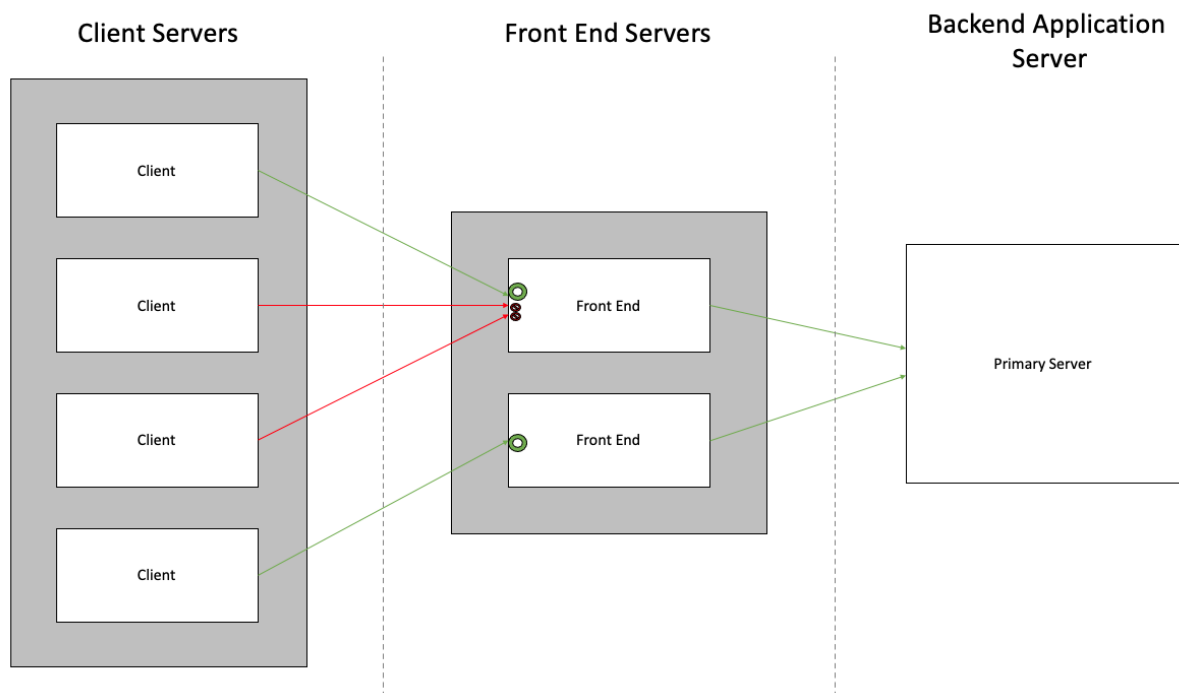
O Front End pode ser visto como um certo tipo de proxy da nossa aplicação: os clientes se conectam nele e suas mensagens são encaminhadas para o servidor. A mesma coisa no sentido inverso (Servidor - FE). Para gerenciar isto, o FE abre um socket em uma determinada porta para conexões dos clientes, e outro socket para conexão do servidor RM principal. Foi decidido abrir 2 portas para deixar as conexões separadas e uma melhor organização de código e da aplicação, mas poderia abrir apenas um socket e ouvir tudo na mesma porta.

No caso do servidor RM primário cair, o FE detectará a queda pelo ConnectionMonitor e parará de enviar mensagens para o servidor. Quando o FE detectar que um novo servidor RM conectou nele no socket aberto para este tipo de conexões, será reconhecido como o novo servidor RM primário e se voltará a enviar mensagens para ele e receber dele.

O FE, graças à nossa classe Semaphore, consegue ler 1 mensagem por vez dos clientes e respeitar a ordem de chegada. Sendo assim, se o cliente 2 e 3 enviarem mensagens enquanto estava sendo processada a mensagem do cliente 1, elas serão processadas por ordem de chegada pois cada thread que ouve dos clientes dá um `.wait` no nosso Semaphore.

Vale ressaltar que a ordem se preserva apenas dentre as mensagens do mesmo FE seguindo o princípio de precedência happened-before: as mensagens do FE1 serão processadas na ordem que chegaram assim como em FE2. Porém, as mensagens dos clientes que estão em diferentes FEs serão processadas concorrentemente e não há nenhum mecanismo empregado para garantir a ordem entre eles.

Um ponto onde tomamos a liberdade para implementar de forma um pouco diferente da sugerida pelo professor foi na questão de leitura e processamento de mensagens. Após receber uma mensagem de um cliente e encaminhá-la para o servidor RM primário, não consumimos a próxima mensagem que chegar no FE mas sim esperamos pelo ACK da mensagem enviada. Se o servidor primário cair, não se tentará consumir a próxima mensagem recebida de um cliente. Ao invés disso, acontecerão *retries* no envio da mensagem a cada X segundos. Quando for eleito um novo servidor primário e este se conectar ao FE, eventualmente o retry irá acontecer e conseguirá enviar a mensagem. Quando se receber o ACK, se lê a próxima mensagem recebida de algum cliente, lembrando que se preserva a ordem. Vale notar que isto serve tanto para mensagem do tipo texto, quanto para mensagens de controle do tipo INVITE/JOIN. A motivação e mais detalhes sobre isto serão explicados na seção bônus.



Se acontecer de um FE cair, o servidor RM detecta a queda e removerá todas as sessões de usuários que estavam naquele FE. Além do mais, os clientes que tentarem se conectar depois da queda de algum ou alguns FE, estabelecerão uma conexão com um dos FE disponíveis. Vale ressaltar que não implementamos uma lógica de reconexão automática a FE's ou deleção de endereços de FE's válidos no arquivo de configuração que o cliente lê. Sendo assim, pode acontecer do cliente sortear para conectar em um FE que caiu. Neste caso,

o programa dará erro e o usuário deverá rodar de novo para tentar conectar em algum que ainda esteja de pé.

Server

O servidor teve 3 principais mudanças: enxergar FE's e ter noção de Front Ends sockets ao invés de sockets, replicação e ter uma classe ServersRing dentro dele responsável pelo anel e eleição de líder.

Agora, um socket não representa um usuário/sessão, mas sim um FE com diversas sessões de diferentes usuários nele. Dessa forma, só adaptamos nossas estruturas de sessões para tuplas, pois um usuário tem uma sessão X num FE Y, precisamos salvar ambos dados juntos ao invés de apenas uma sessão X. Como o socket tem valor semântico apenas no FE em que ele foi aberto, precisamos guardar um outro identificador para distinguir os FEs (juntamente com o identificador do usuário). Para isso, utilizamos a string "ip:porta" que é um endereço que irá mapear para um único processo. Com isso, podemos replicar essa estrutura de sessões abertas em um determinado RM para as réplicas e teremos como identificar os mesmos FEs quando um RM backup assumir.

A parte de replicação não tem muita complexidade pois, na verdade, são apenas condições (if's) e envio de mensagem do RM primário para todos os RM backup aos quais ele estiver ligado. Todo o funcionamento da replicação será explicado na última seção.

Por último, o servidor tem a classe ServersRing, responsável por conectar aos outros RM em forma de anel e processar eleição de novo líder na queda do servidor primário. A implementação mais detalhada será apresentada, também, na última seção deste relatório.

Group, GroupManager e MessageManager

Essas classes não tiveram mudanças muito significativas. No máximo, se adicionou alguma lógica de que, caso não fosse o servidor primário, não enviar nenhuma mensagem. Assim, elas processavam toda a lógica de usuários e sessões, além de salvar histórico, mas sem enviar nenhum Packet, pois apenas o primário se comunica com os FE. Outra adaptação necessária se refere aos métodos que guardavam as sessões dos usuários e enviavam mensagens para determinadas sessões, eles agora utilizam o pair identificador de sessão mencionado anteriormente.

BÔNUS

Docker

O primeiro recurso extra que disponibilizamos foi a “dockerização” de todo o nosso sistema usando a ferramenta docker e docker-compose. Desta forma, deixamos empacotadas todas partes da nossa app em containers. Assim, nosso sistema se tornou multiplataforma e dependente apenas de ter docker no ambiente onde for rodar, pois por baixo dos panos todos os serviços rodarão em um Ubuntu virtualizado compatível com nosso sistema. Para rodar, basta apenas executar o comando `make services-up`.

Caso se deseje, pode-se utilizar os seguintes comandos para ver todos os logs dos serviços de FE ou servidores RM respectivamente:

```
make logs-fe  
make logs-rm
```

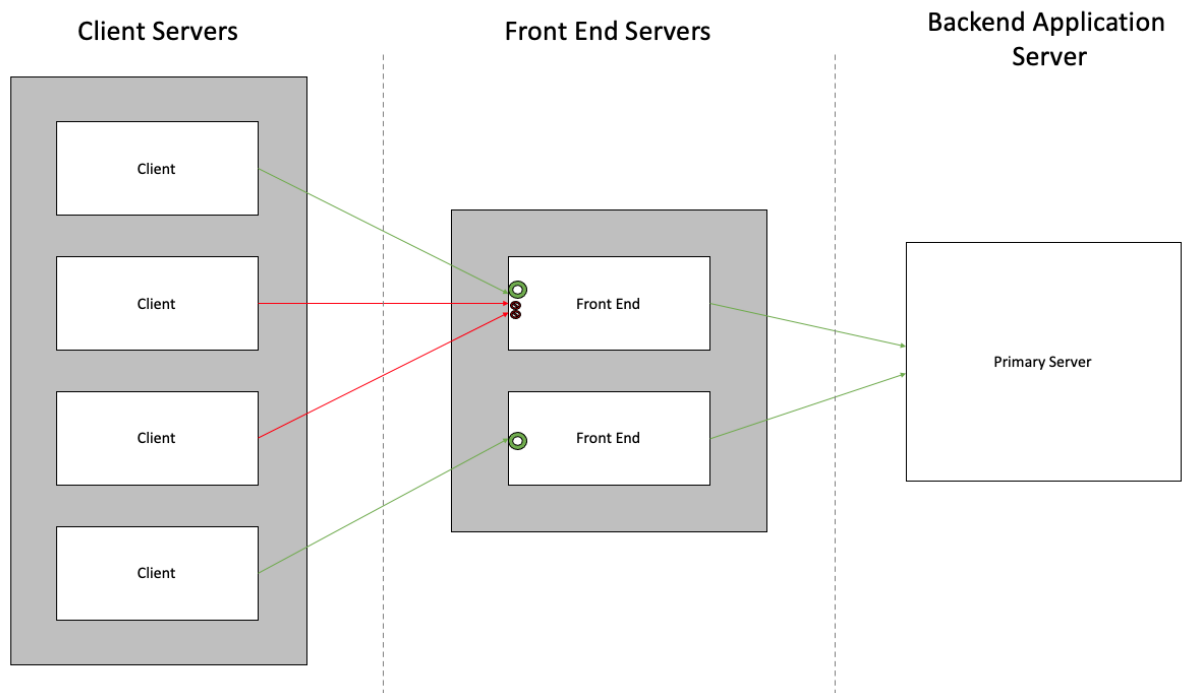
Assim sendo, e utilizando os comandos `docker ps` e `docker rm -f <id>`, pode-se derrubar um container e ver como a aplicação se comporta: derrubando um FE e vendo os servidores RM lidando com isso, ou derrubando o servidor RM primário e vendo a eleição de um novo líder acontecendo.

Transparência perante falhas

O segundo recurso bônus, de certa forma, que implementamos foi a total transparência de quedas de servidor RM primário para o cliente. O motivo pelo qual tanto o cliente tem um buffer de mensagens quanto o FE só processa uma próxima mensagem após ter recebido um ACK da atual é para manter a maior consistência e transparência da aplicação. Se o servidor RM cair e haviam 5 mensagens para serem processadas, quando um novo servidor primário assumir, todas as mensagens serão enviadas na ordem que foram recebidos pelo FE e tudo continuará sem o cliente ter percebido nada. Assim, se o cliente 1 enviou a mensagem “oi” e “tudo bem?” enquanto o servidor RM estava caído, assim que um novo for eleito estas mesmas mensagens serão processadas e o cliente nunca nem saberá que internamente algo caiu, além do fato de que não se perde nenhuma mensagem a nenhum momento.

Com isto, nossa aplicação não só se torna tolerante à falhas como também consegue manter uma aparência de que tudo está normal e funcionando para o cliente. Acreditamos que,

sempre que possível, não tem motivo para deixar um cliente notar que algo estranho aconteceu perdendo sua mensagem, por exemplo.



Noção de mensagens dos clientes 2 e 3 ficando bloqueadas

INSTALAÇÃO

Descrição do ambiente de testes: versão do sistema operacional e distribuição, configuração da máquina (processador(es) e memória) e compiladores utilizados (versões)

- Linux Mint 19.3 Cinnamon
- Linux Kernel 5.3.0-45-generic
- Processador: Intel Core i7-8550U
- Memória: 16GB
- Thread model: posix
- gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)

Recomenda-se utilizar o deploy via Docker e rodar na máquina local apenas o cliente. Desta forma, o sistema se torna multi-plataforma e independente de ambiente.

Os endereços dos FE existentes encontram-se no arquivo `addresses.txt` e o cliente escolherá aleatoriamente um deles para se conectar.

Como executar

Tenha instalado docker e docker-compose: [Get Docker | Docker Documentation](#)

Rode os comandos:

- `make deploy,`
- `make build-client e`
- `./build/client.app <nome_usuario> <nome_usuario>.`

Um exemplo, tendo Docker e os serviços já rodando, seria rodar `./build/client.app Johnny Room2.` para conectar o usuário "Johnny" no grupo "Room2". Recomendamos assistir o vídeo onde é executado na prática.

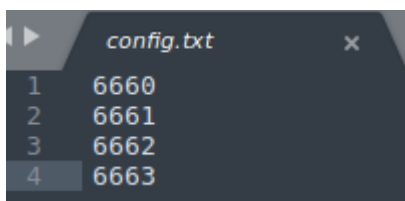
Na hora de levantar os containers de RM via docker-compose, para cada container que se levanta, se copia automaticamente o repositório inteiro para dentro. Desta forma, existem N cópias. Com isto, conseguimos garantir que cada RM tenha seu próprio arquivo de histórico de mensagens e verificar que a replicação acontece de fato.

QUESTÕES

Funcionamento da eleição de líder

Com o intuito de trazer mais segurança e estabilidade para o sistema, foram criados servidores de backup para o caso do servidor primário vir a falhar. E para esse caso é preciso escolher um dentre os vários servidores de backup criados para a aplicação. Com o propósito de escolher o servidor que substituirá o principal, em caso de falhas, foi desenvolvido um algoritmo de eleição conhecido como algoritmo do anel, pela sua simplicidade.

Para o algoritmo do anel poder ser executado, antes é preciso criar a sua própria estrutura de anel. Para a criação desta estrutura foi criado o arquivo “config.txt”, que precisa ser preenchido, por quem for inicializar a aplicação, antes da execução dos servidores. Este arquivo consiste na escrita das portas em que se deseja executar as aplicações, sendo que neste trabalho estamos considerando que todos os servidores rodarão no mesmo endereço de IP (endereço local da máquina). A seguir se encontra um exemplo de como este arquivo deve ser escrito:



```
config.txt
1 6660
2 6661
3 6662
4 6663
```

No exemplo acima, se deseja executar quatro servidores. Um servidor primário, e outros três que servirão de backup. Todos os servidores rodam na mesma máquina local, sendo que no exemplo se deseja rodar os 4 servidores nas portas 6660, 6661, 6662, 6663, respectivamente.

Na formação do anel inicial, nós consideramos que o servidor referente à primeira porta fornecida pelo usuário no arquivo é o servidor primário, enquanto os outros servem de backup.

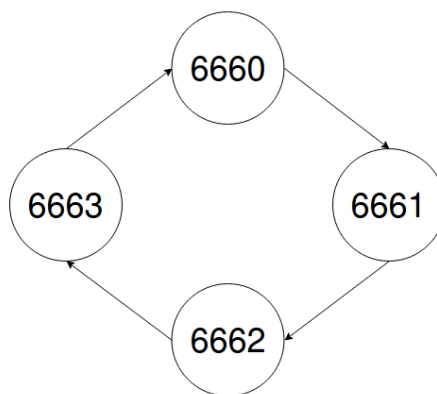
A formação do anel se dá na seguinte forma:

Procuramos a primeira porta, fornecida no arquivo, que se encontra disponível para podermos realizar o bind como servidor, através de um loop while. Uma vez realizado o bind nós chamamos as seguintes funções no código:

acceptServerConnection → chama a função accept, da biblioteca socket, e fica aguardando, como servidor, a realização de uma conexão TCP.

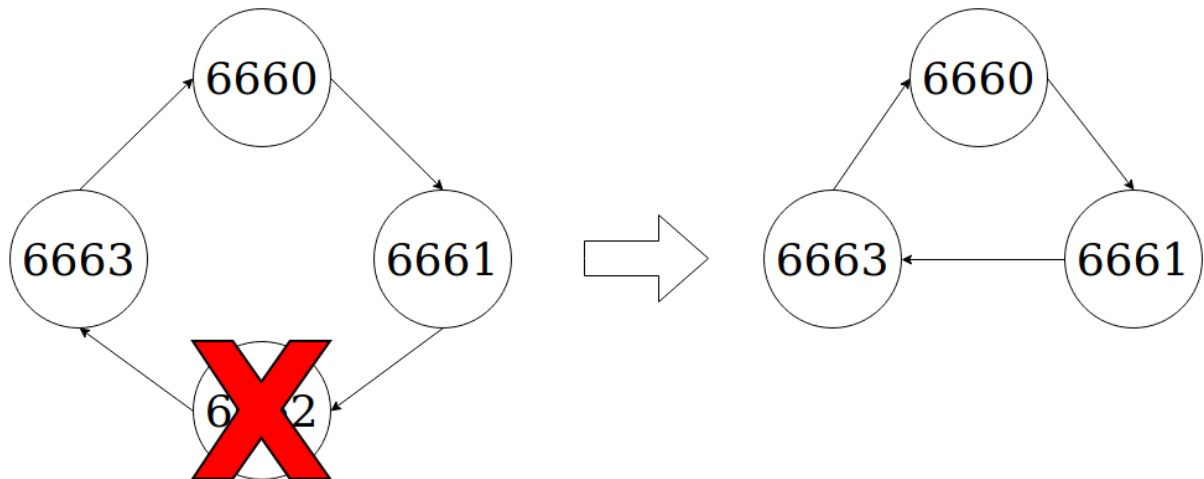
connectToServer → chama a função connect, da biblioteca socket, e fica tentando realizar uma conexão TCP, como cliente. O método ConnectToServer funciona com um loop while, que fica tentando realizar uma conexão com a porta vizinha fornecida pelo usuário. No exemplo acima, o servidor de porta 6660 tentaria se conectar apenas com a porta 6661, o referente a porta 6661 tentaria se conectar a porta 6662, e assim por diante. Como não se sabe quando um servidor será executado e se conectará com outro, esta função virou uma thread separada no sistema.

O resultado do exemplo acima deve gerar a seguinte estrutura após a inicialização de todos os servidores:



Se houver a desconexão de um dos servidores após a formação da estrutura de anel, os servidores vizinhos que estavam conectados ao que falhou, após perceberem a falha, tentam realizar uma nova conexão entre eles com o objetivo de manter uma estrutura de anel mesmo após o ocorrido.

A seguir encontra-se um exemplo de como a estrutura se reorganiza após a falha de um dos servidores:



Algoritmo de Eleição

Para fim exclusivo do algoritmo de eleição, nós escolhemos a porta de acesso fornecida no arquivo “config.txt” como sendo o ID do processo, uma vez que a porta será um valor único para cada aplicação. Uma informação importante que é definida no início da inicialização dos servers é o ID do servidor primário atual. Ou seja, no exemplo mostrado acima o servidor 6660 seria o primário, e todos os outros, que servem de backup, possuem o conhecimento de que o servidor 6660 é o líder atual. Desta forma, independente das desconexões dos servidores de backup que ocorram antes do primário falhar, os servidores que estiverem ligado ao principal vão estar cientes desta informação. Sendo assim, foi decidido que o servidor que estiver conectado como cliente do primário será aquele que iniciará a eleição, ao perceber que ocorreu uma falha na comunicação com o server.

Quando ocorrer uma desconexão por parte do servidor primário, primeiramente, a estrutura de anel será reorganizada, e logo em seguida o servidor (cliente) que estava conectado ao que caiu irá iniciar o algoritmo de eleição chamando a função *startElection()* da classe *Election*.

A eleição funciona seguindo os seguintes passos:

- 1) O servidor que inicia o algoritmo transmite o seu próprio ID para o servidor vizinho, com uma mensagem de “ELECTION”.
- 2) O servidor que recebe o ID compara ele com o seu próprio, e transmite o maior dentre os IDs para o próximo servidor, com uma mensagem de “ELECTION”.

- 3) O passo 2 acontece com todos os servidores até que o ID recebido por um servidor seja igual ao seu próprio.
- 4) O servidor cujo ID recebido é igual ao seu próprio, envia o seu próprio ID novamente para o servidor vizinho, mas desta vez com uma mensagem de “ELECTED”.
- 5) O servidor que recebe o ID com a mensagem de “ELECTED” guarda a informação de quem é o novo líder, e transmite a mesma mensagem para o servidor seguinte.
- 6) O passo 5 acontece com todos os servidores até que o ID recebido por um servidor seja igual ao seu próprio. Este é nomeado o novo líder.

Implementação da replicação passiva

O modelo de replicação passiva construído faz uso de identificação numérica de cada servidor para a conexão entre eles. Essa estrutura será detalhada a seguir. Além disso, estruturas de controle e dados foram adicionados ao servidor para gerenciar processos e métodos que devem ou não ocorrer em servidores réplica e primário.

Estrutura de replicação

O Modelo de replicação passiva empregado faz uso de sockets de listening e conexões diretas para diminuir a quantidade de sockets abertos, facilitando assim a manutenção e gerenciamento dos mesmos a medida em que mais servidores de replicação são criados. O que acontece na prática, descrito graficamente na figura 3, é que cada servidor possui um número que o representa na estrutura de replicação. Servidores de número $X > 0$ criam sockets de listening que ficam esperando conexões de servidores de número $X-1$, $X-2 \dots X-N$ sendo N a quantidade de servidores com número inferior a X . Foi criada uma estrutura de lista onde todos os sockets de replicação do servidor são gerenciados. Caso um servidor caia, essa estrutura é atualizada de forma a evitar envio de mensagens para servidores que não estejam funcionando corretamente.

RM Replication Socket Structure

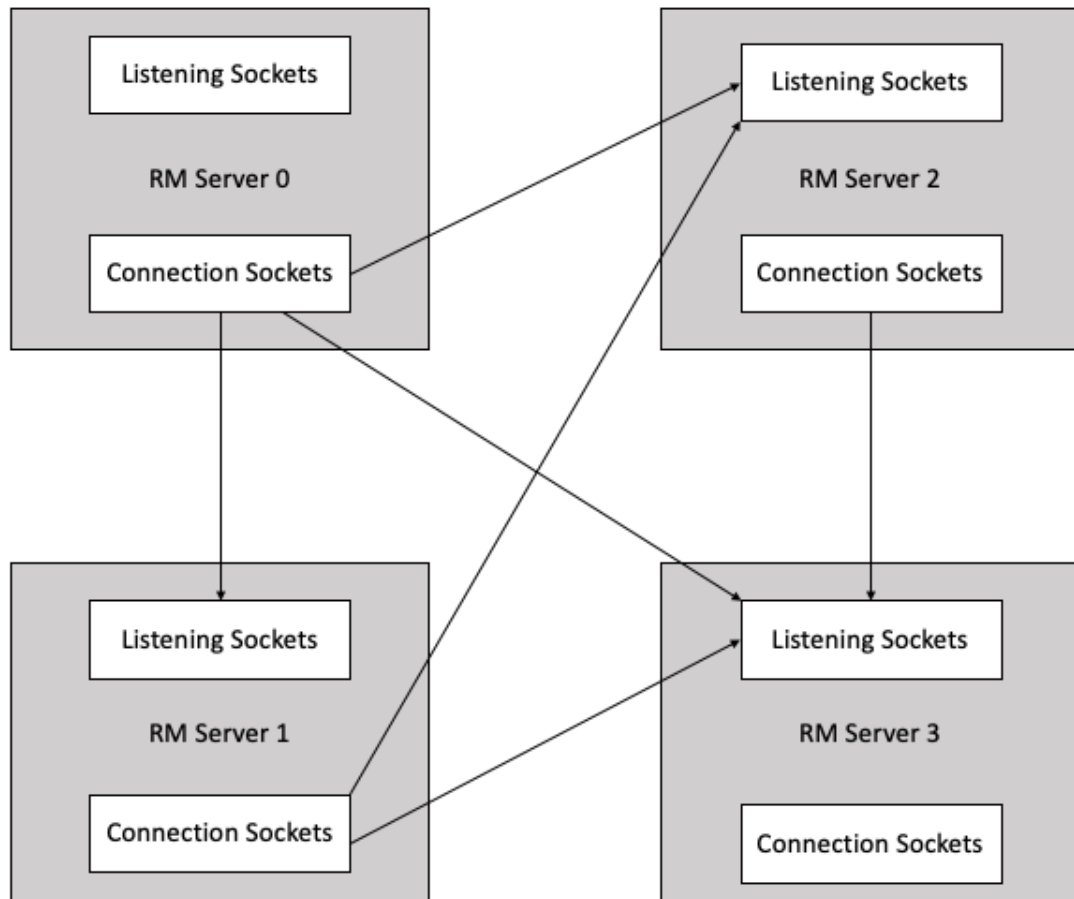


Figura 3: Modelo de replicação passiva

No exemplo acima vemos que o servidor RM 3 recebe conexões em seu socket de listening dos servidores RM 2, RM 1 e RM 0. Assim sucessivamente até o servidor RM 0 que não recebe nenhuma conexão em seu listening já que não existe nenhum servidor RM de menor número. Já o servidor RM 3 não se conecta em ninguém, pois não existem servidores RM de número maior que o dele.

Replicação na prática

Após o servidor primário receber uma requisição do servidor de FE, o mesmo dispara réplicas do pacote recebido para os servidores de replicação. Ele então espera que cada servidor retorne uma mensagem de confirmação que demonstra que pacote foi efetivamente

processado pelas réplicas. Apenas após isso a mensagem recebida pelo servidor principal é processada. Os servidores réplicas não enviam qualquer tipo de mensagem aos FE's ou cliente, pois apenas o servidor primário é responsável por isso.

Três tipos de pacotes são replicados para os servidores de backup. São eles os pacotes de JOIN, MESSAGE e DISCONNECT. Esses três tipos de pacotes são responsáveis por garantir que todos os RM's e servidor primário estejam consistentes entre si. Assim, após a eleição de um novo líder, é possível saber quais são os usuários logados e quais grupos existem atualmente.

LINK PARA O VÍDEO DA APRESENTAÇÃO

<https://youtu.be/vDEWKnPD0bE>

CONCLUSÃO

Na implementação do trabalho tivemos a oportunidade de nos aprofundarmos em conceitos aprendidos nas aulas da disciplina. Nessa etapa, para a construção de um sistema capaz de tolerar falhas nos servidores RMs, o mecanismo de replicação foi explorado e a arquitetura do nosso servidor foi estendida. Para implementarmos tais mudanças, o grupo trabalhou de forma distribuída e na semana anterior à entrega, começamos a integrar os desenvolvimentos.

Cada membro do trabalho foi responsável por um dos seguintes tópicos: criação do front-end e integração do mesmo com cliente/RMs, refactor do servidor para ele comunicar com os front-ends e não mais com os clientes diretamente, módulo de eleição de líder implementando o algoritmo de anel e, por fim, a replicação das mensagens que chegavam no servidor principal.

Podemos falar que, além dos conceitos principais desta etapa (replicação e algoritmo de eleição), também foi necessário atentar, com muito cuidado, aos conceitos aprendidos na primeira área da disciplina (seções críticas), principalmente dentro dos servidores RMs. Como o nosso trabalho não possui nenhum recurso diretamente acessado por diferentes processos de forma concorrente (sempre temos o intermédio do server que emprega os mecanismos de proteção interna para seções críticas - semáforos), não foi necessário utilizar nenhum

algoritmo de exclusão mútua para processos distribuídos (como tokens de permissão ou filas distribuídas, por exemplo).

Concluimos dizendo que apesar das dificuldades que passamos pela situação atípica que estamos vivendo neste semestre, saímos com uma sensação de satisfação com os resultados atingidos. Isso tudo foi possível pelo empenho dos colegas - que estavam sempre prontos para discutir questões referentes às decisões arquiteturais e resoluções de problemas em reuniões que iam até tarde da noite - e também pelo empenho do professor que resolvia as dúvidas de forma rápida e precisa.